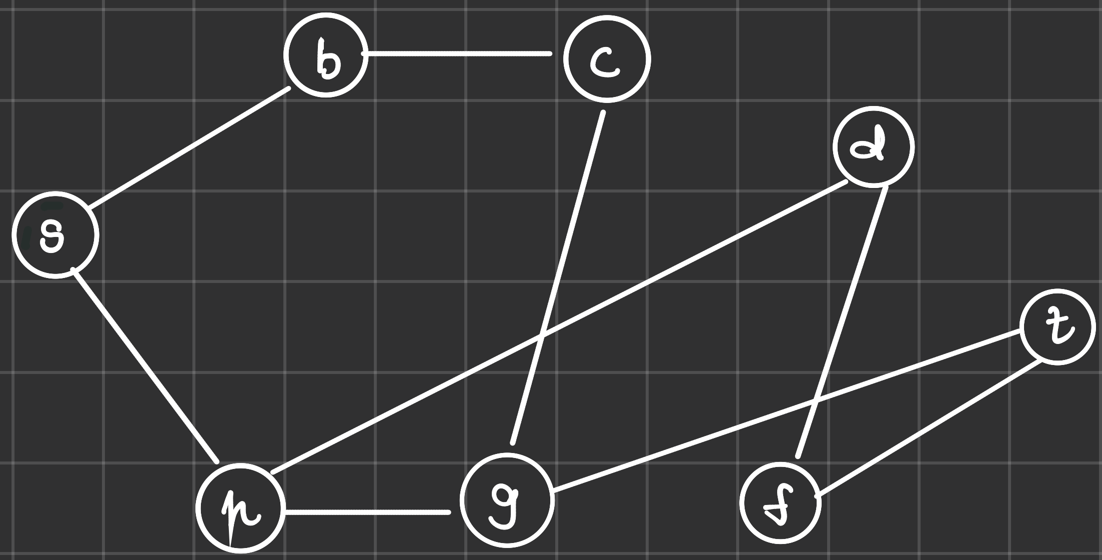
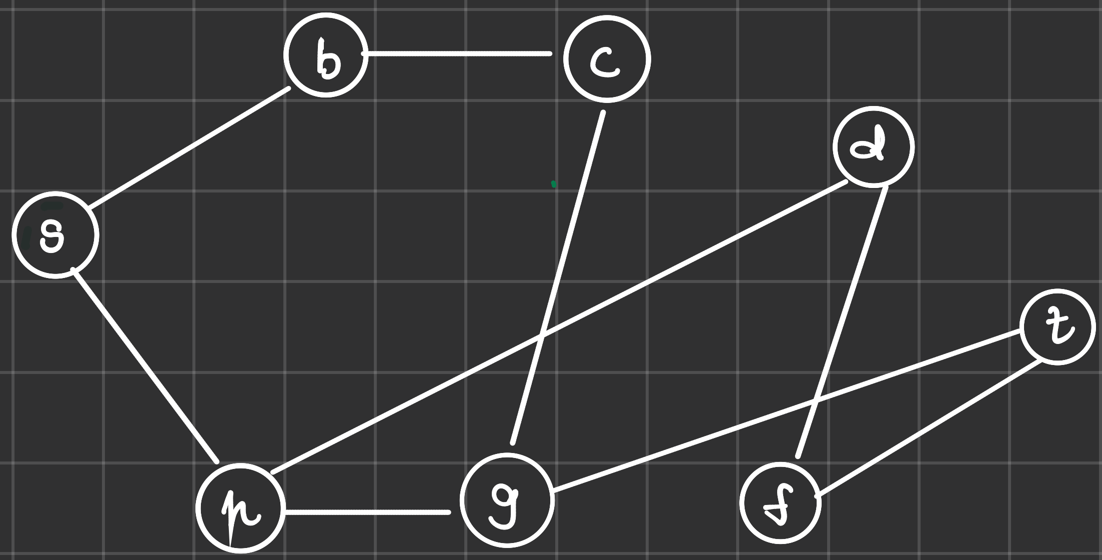
Graphs



A Graph has a set of nodes or vertices and edges between them.

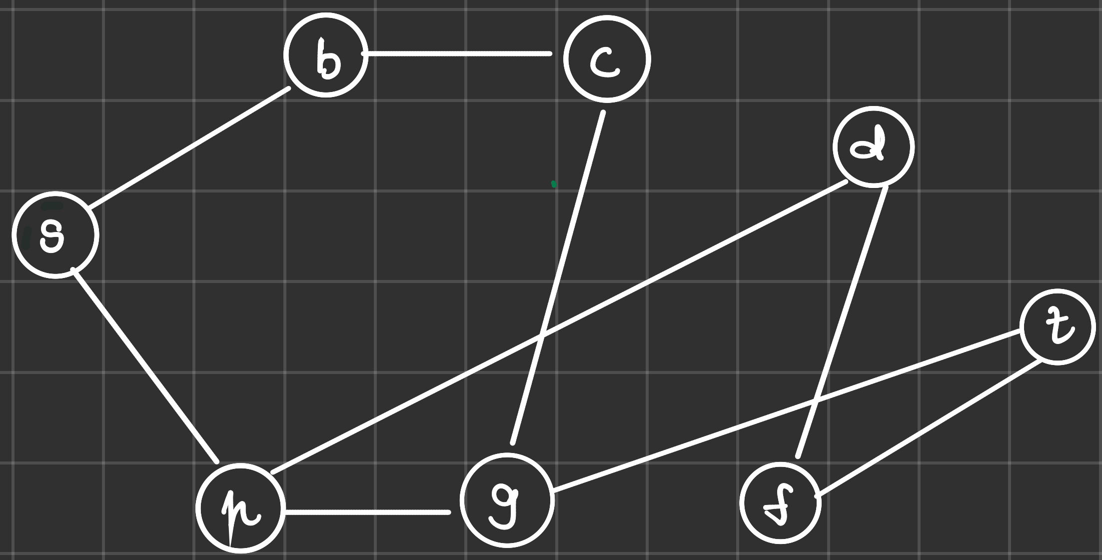
Graphs



A Graph has a set of nodes or vertices and edges between them.

Formally: Graph G is a tuple (V, E)
where V is a set of nodes
and $E \subseteq V \times V$

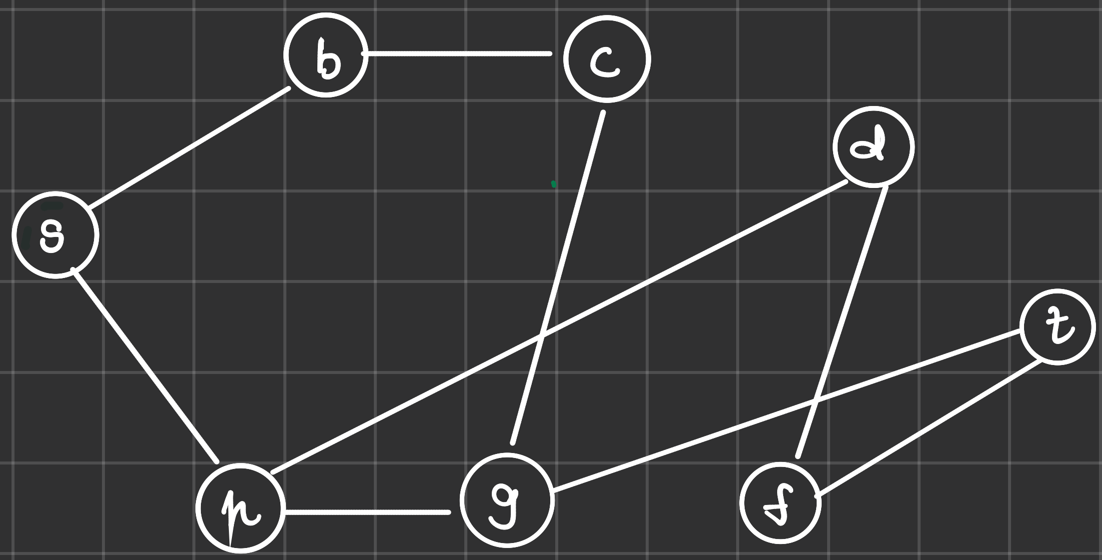
Graphs



A Graph has a set of nodes or vertices and edges between them.

Formally: Graph G is a tuple (V, E)
where V is a set of nodes
and $E \subseteq V \times V$

Graphs



A Graph has a set of nodes or vertices and edges between them.

Formally: Graph G is a tuple (V, E)
where V is a set of nodes
and $E \subseteq V \times V$

$G(\{s, b, c, d, f, g, h, t\}, \{(s,b), (b,c), (s,h), (h,g), (c,g), (h,d), (d,f), (g,t), (f,t)\}$

Notation :

$m \leftarrow$ number of edges in the graph

$n \leftarrow$ number of vertices in the graph

$d(v)$ or $\deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

Notation :

$m \leftarrow$ number of edges in the graph

$n \leftarrow$ number of vertices in the graph

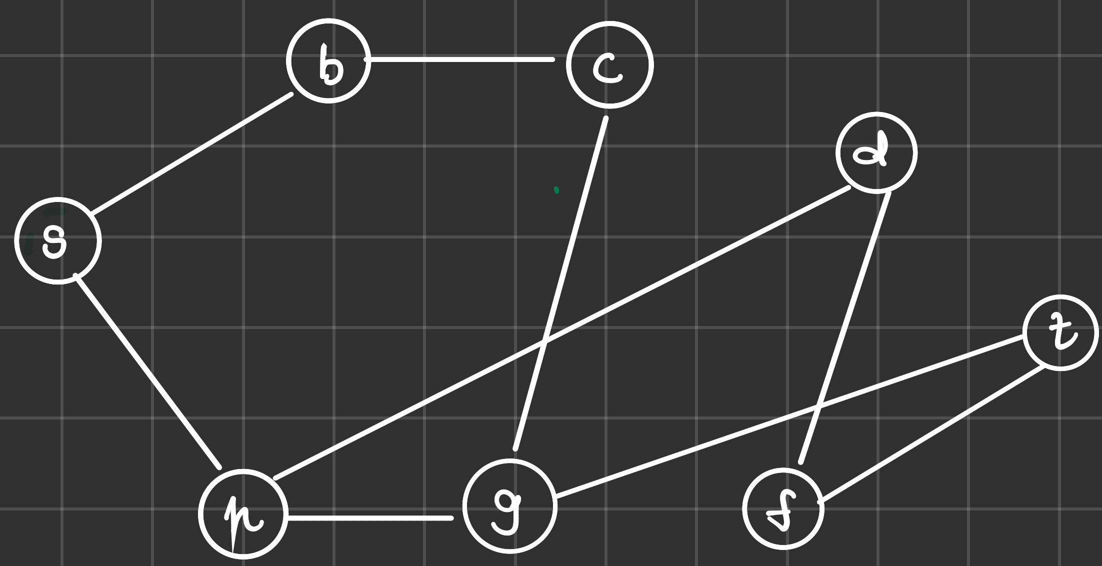
$d(v)$ or $\deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

Lemma : $\sum_{v \in V} d_v =$

Notation :

$m \leftarrow$ number of edges in the graph
$n \leftarrow$ number of vertices in the graph
$d(v)$ or $deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

Lemma : $\sum_{v \in V} d_v = 2m$



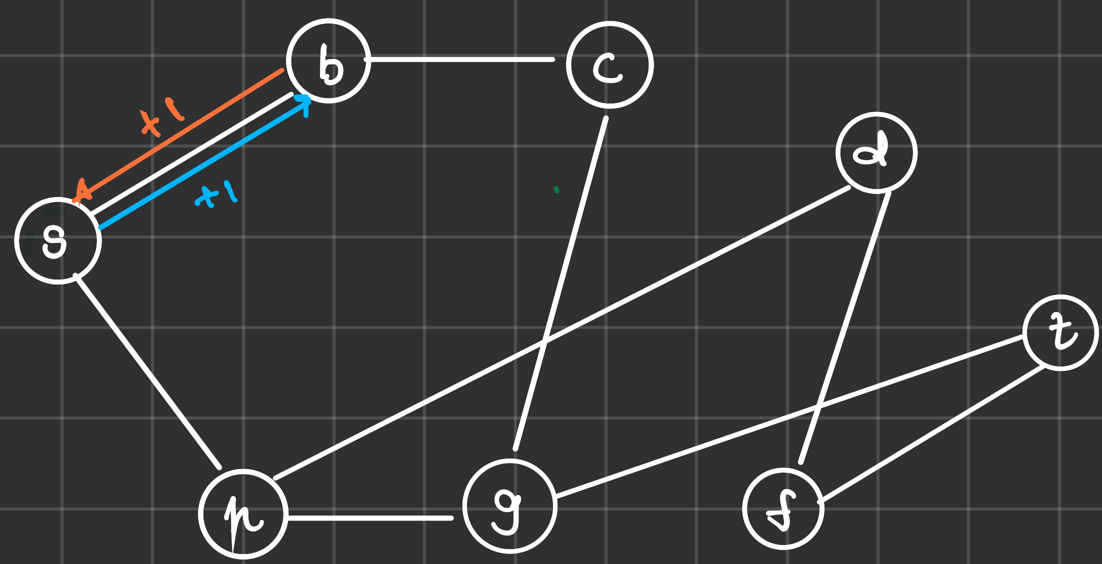
Notation :

$m \leftarrow$ number of edges in the graph

$n \leftarrow$ number of vertices in the graph

$d(v)$ or $deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

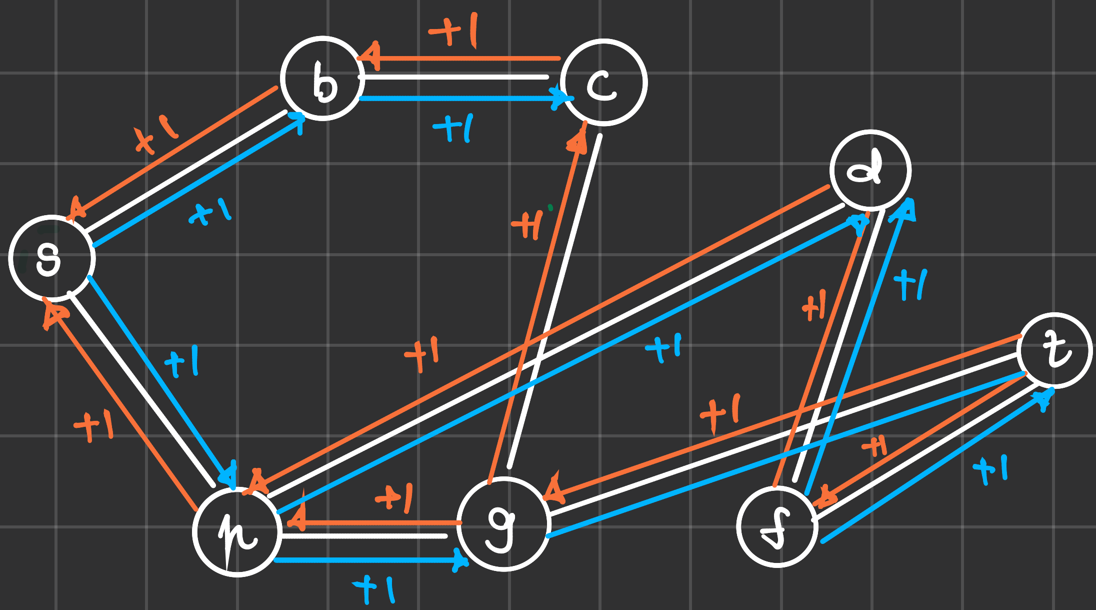
Lemma : $\sum_{v \in V} d_v = 2m$



Notation :

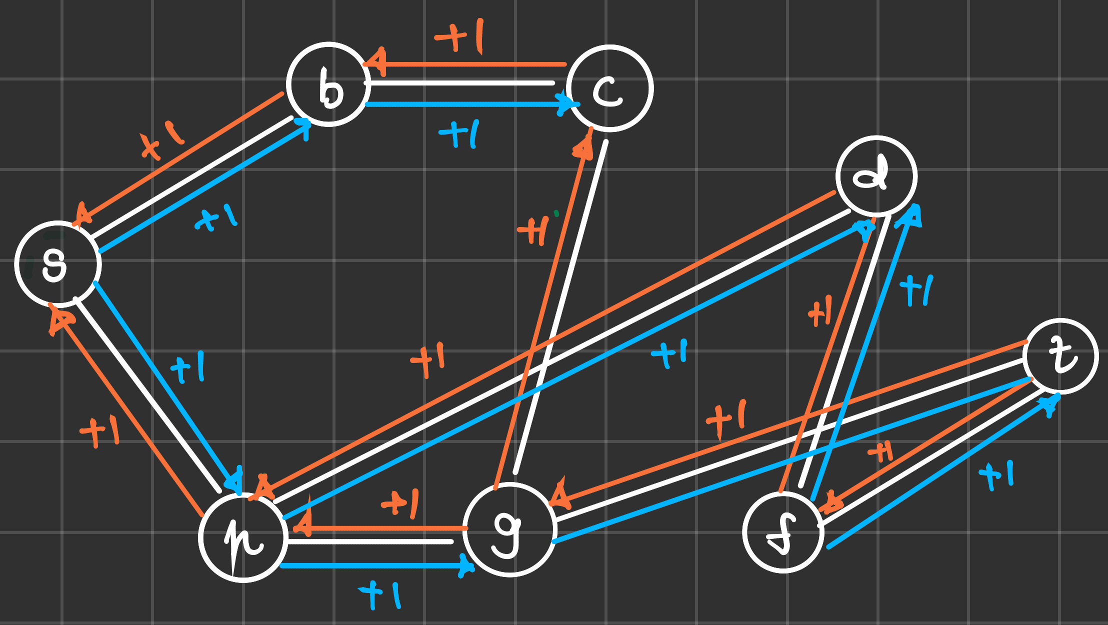
$m \leftarrow$ number of edges in the graph
$n \leftarrow$ number of vertices in the graph
$d(v)$ or $\deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

Lemma : $\displaystyle\sum_{v \in V} d_v = 2m$



Notation : $m \leftarrow$ number of edges in the graph
$n \leftarrow$ number of vertices in the graph
$d(v)$ or $\deg(v) \leftarrow$ degree of v or number of edges adjacent to v.

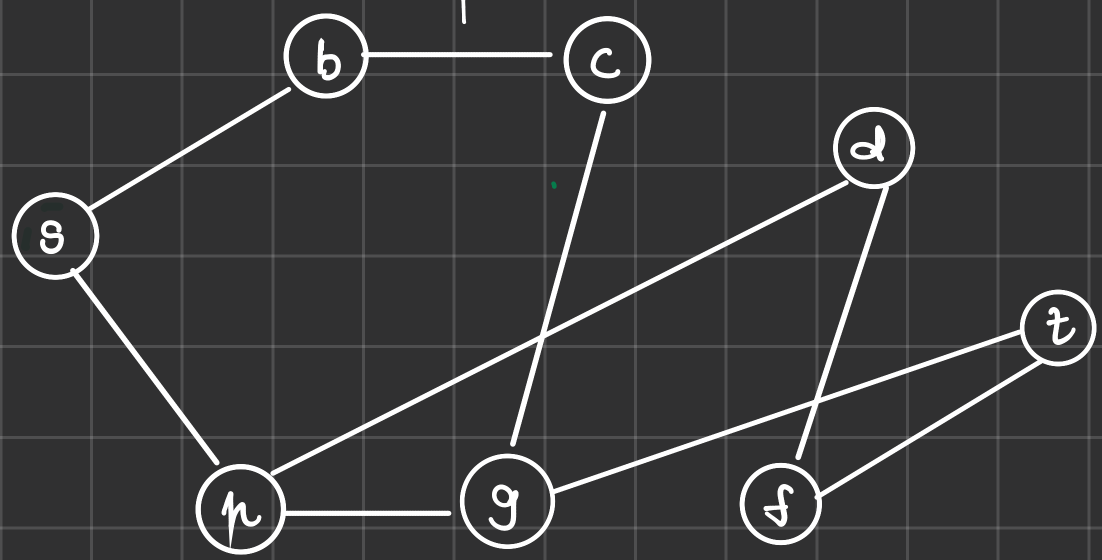
Lemma : $\sum_{v \in V} d_v = 2m$



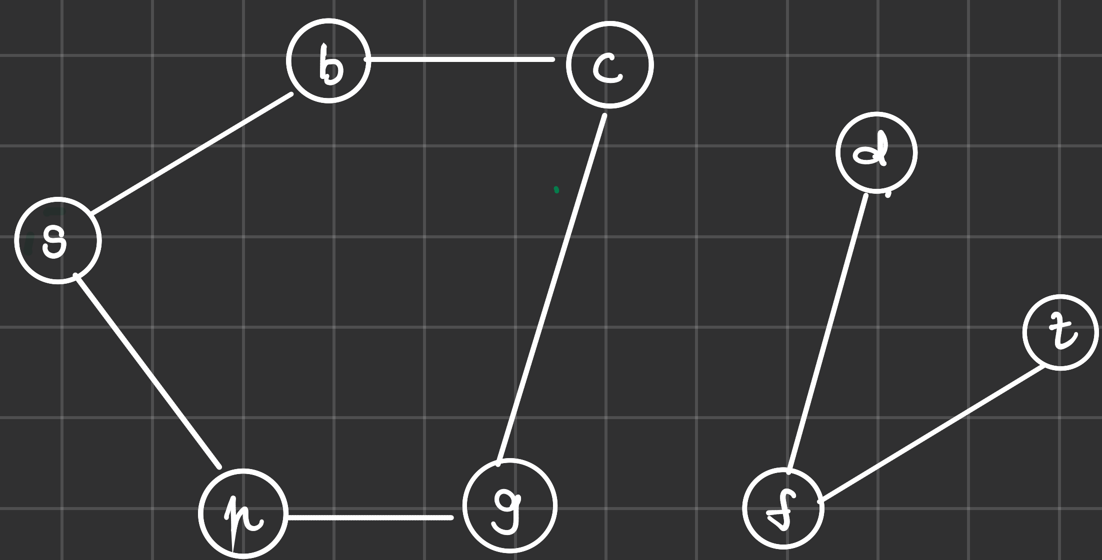
In $\sum_{v \in V} d_v$ each edge is added exactly twice.

$$\Rightarrow \sum_{v \in V} d(v) = 2m.$$

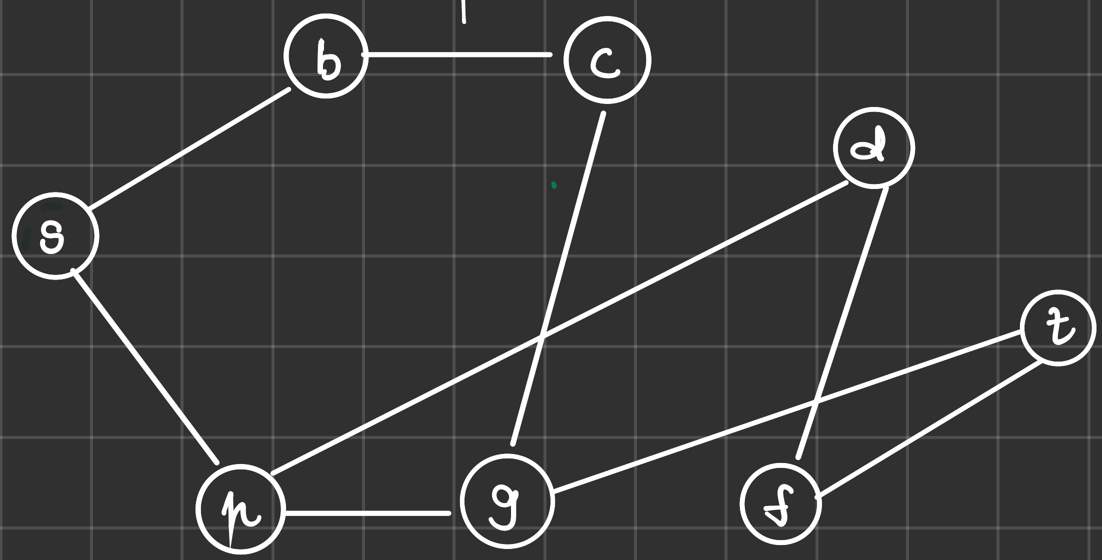
Connected Graph.



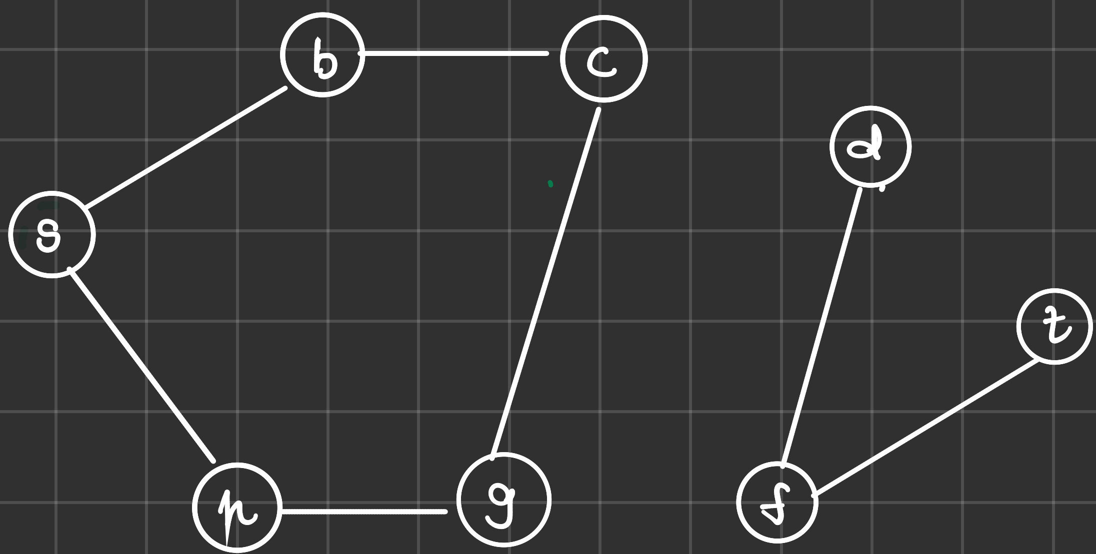
Disconnected Graph



Connected Graph.

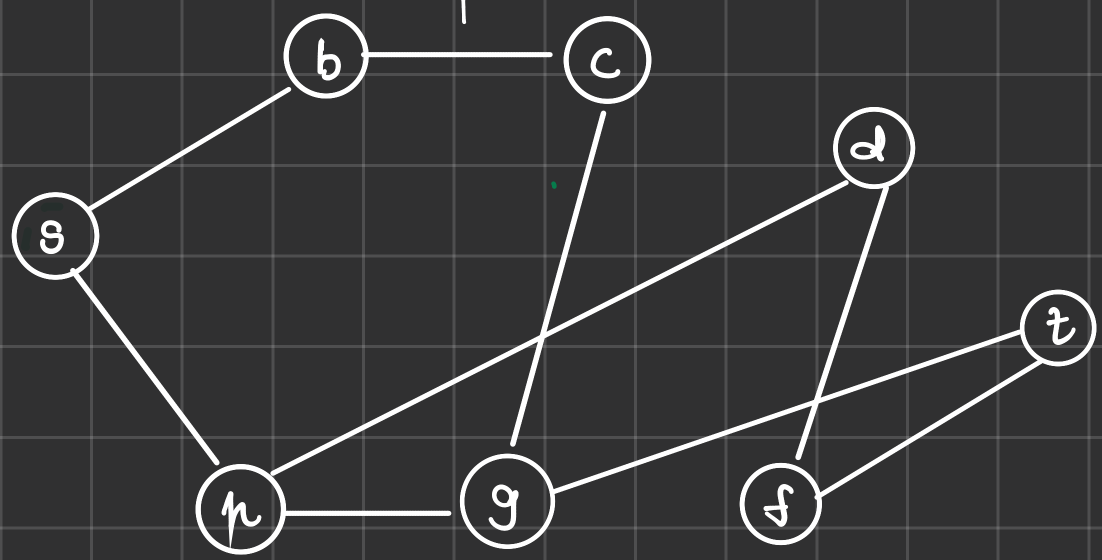


Disconnected Graph

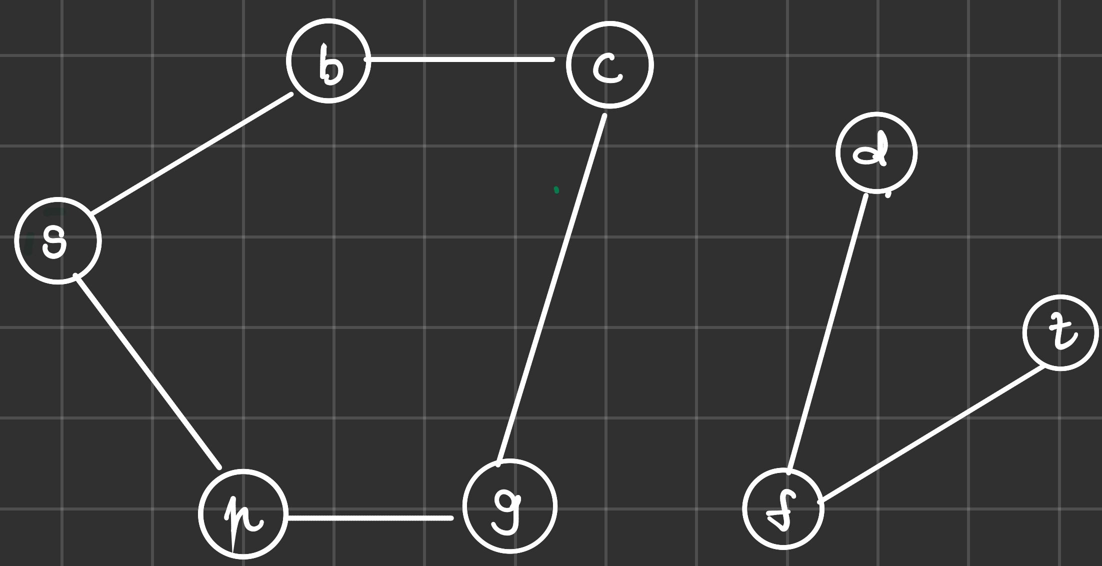


Q : What is the maximum number of edges in a connected graph?

Connected Graph.



Disconnected Graph



Q: What is the maximum number of edges in a connected graph?

A: $\frac{n \cdot (n-1)}{2}$

Q: What is the minimum number of edges in a connected graph?

Q: What is the minimum number of edges in a connected graph?

A: $n-1$

$\uparrow$

This graph is also called a tree.

Q: What is the minimum number of edges in a connected graph?

A: $n-1$

This graph is also called a tree.

Def: A tree is a connected graph with no cycle.

Lemma: The number of edges in a tree is $n-1$

Q: What is the minimum number of edges in a connected graph?

A: $n-1$

This graph is also called a tree.

Def: A tree is a connected graph with no cycle.

Lemma: The number of edges in a tree is $n-1$

Proof: By induction on n

Base case $n=1$

Trivially true.

Q: What is the minimum number of edges in a connected graph?

A: $n-1$

This graph is also called a tree.

Def: A tree is a connected graph with no cycle.

Lemma: The number of edges in a tree is $n-1$

<u>Proof</u>: By induction on n

Base case $n = 1$

Trivially true.

<u>Induction Hypothesis</u>

Assume that the statement is true for all values in range $[1, n-1]$

Prove the statement when there are n vertices

Q: What is the minimum number of edges in a connected graph?

A: $n-1$

This graph is also called a tree.

Def: A tree is a connected graph with no cycle.

Lemma: The number of edges in a tree is $n-1$

Proof: By induction on n

Base case $n=1$

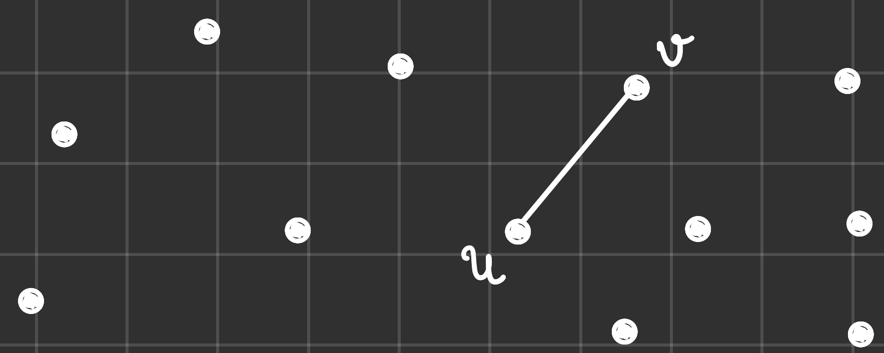
Trivially true.

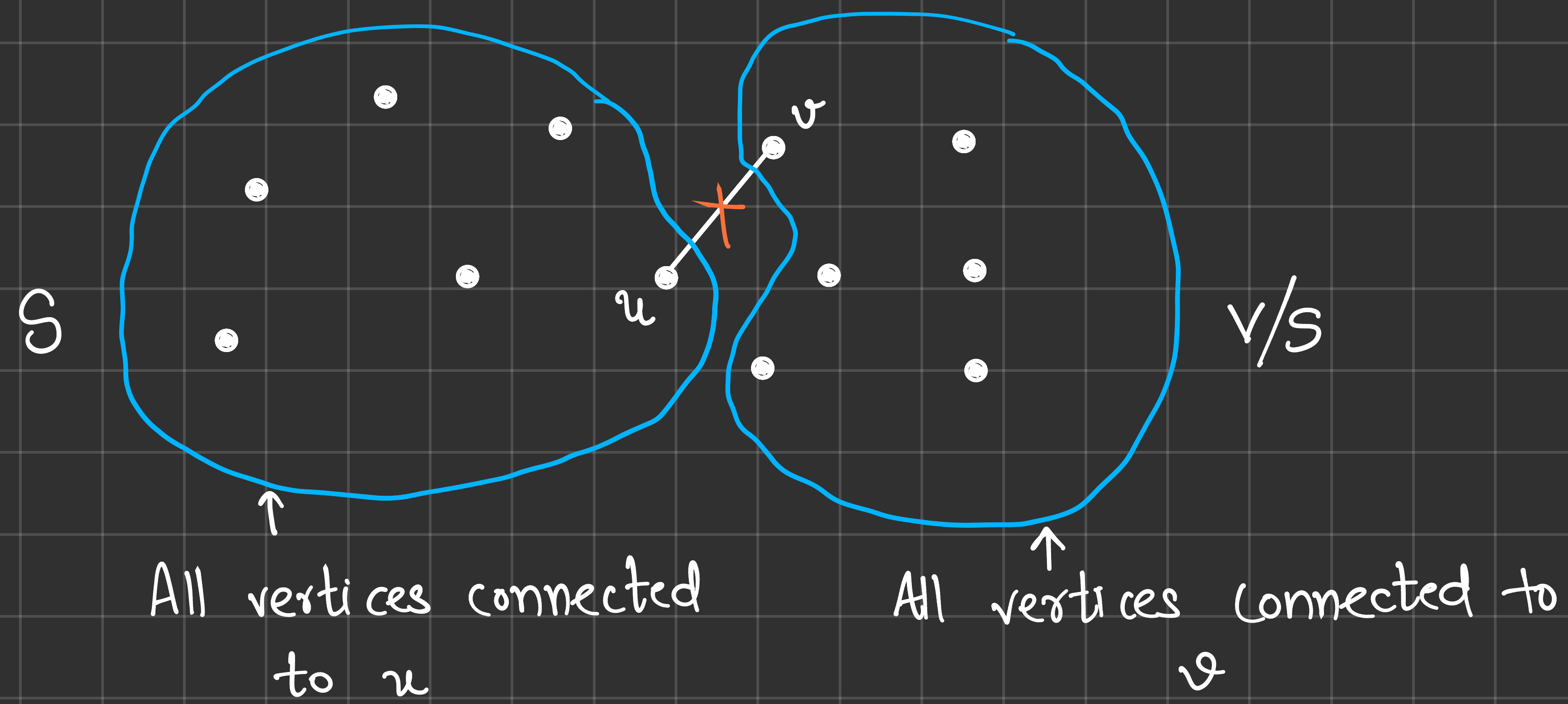
<u>Induction Hypothesis</u>

Assume that the statement is true for all values in range $[1, n-1]$

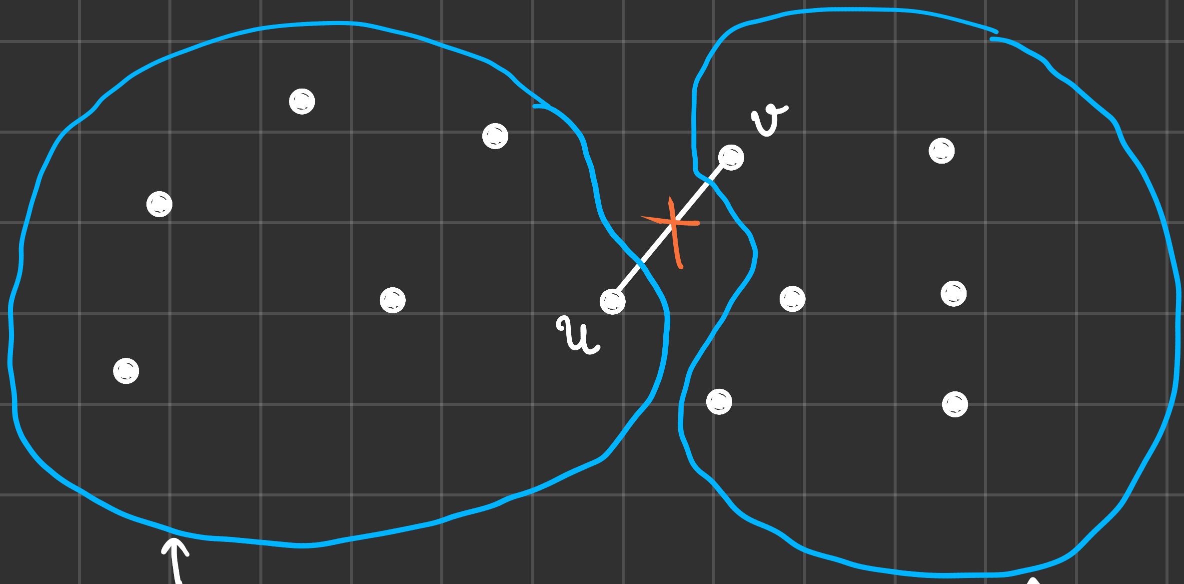
Prove the statement when there are n vertices

Assume that there exists a tree on n vertices

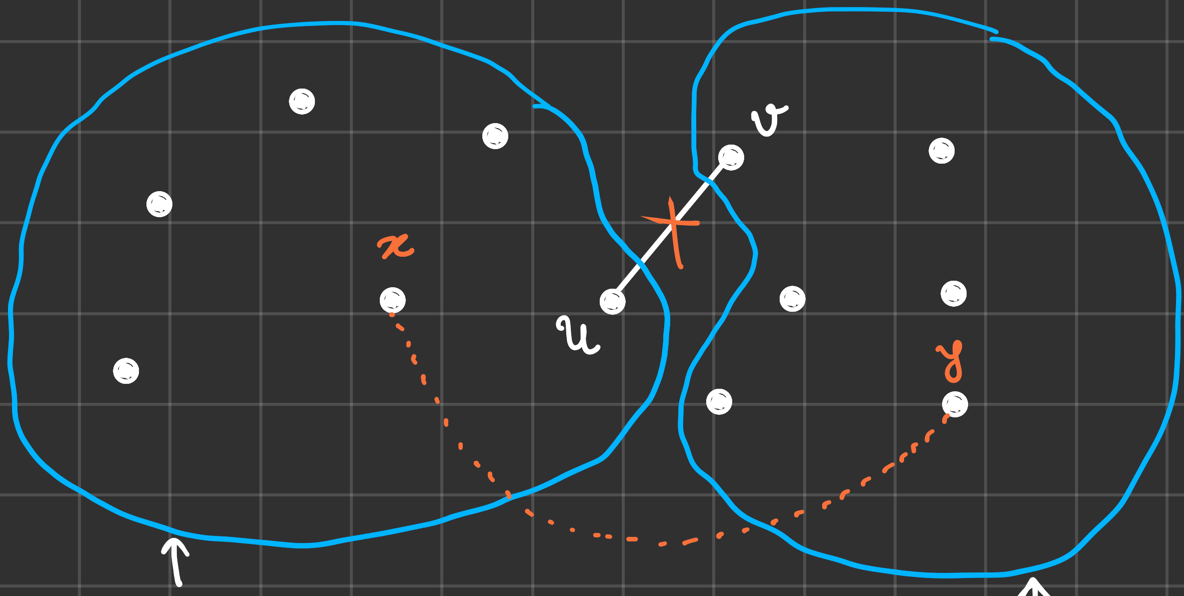




We have divided the graph into two connected components.



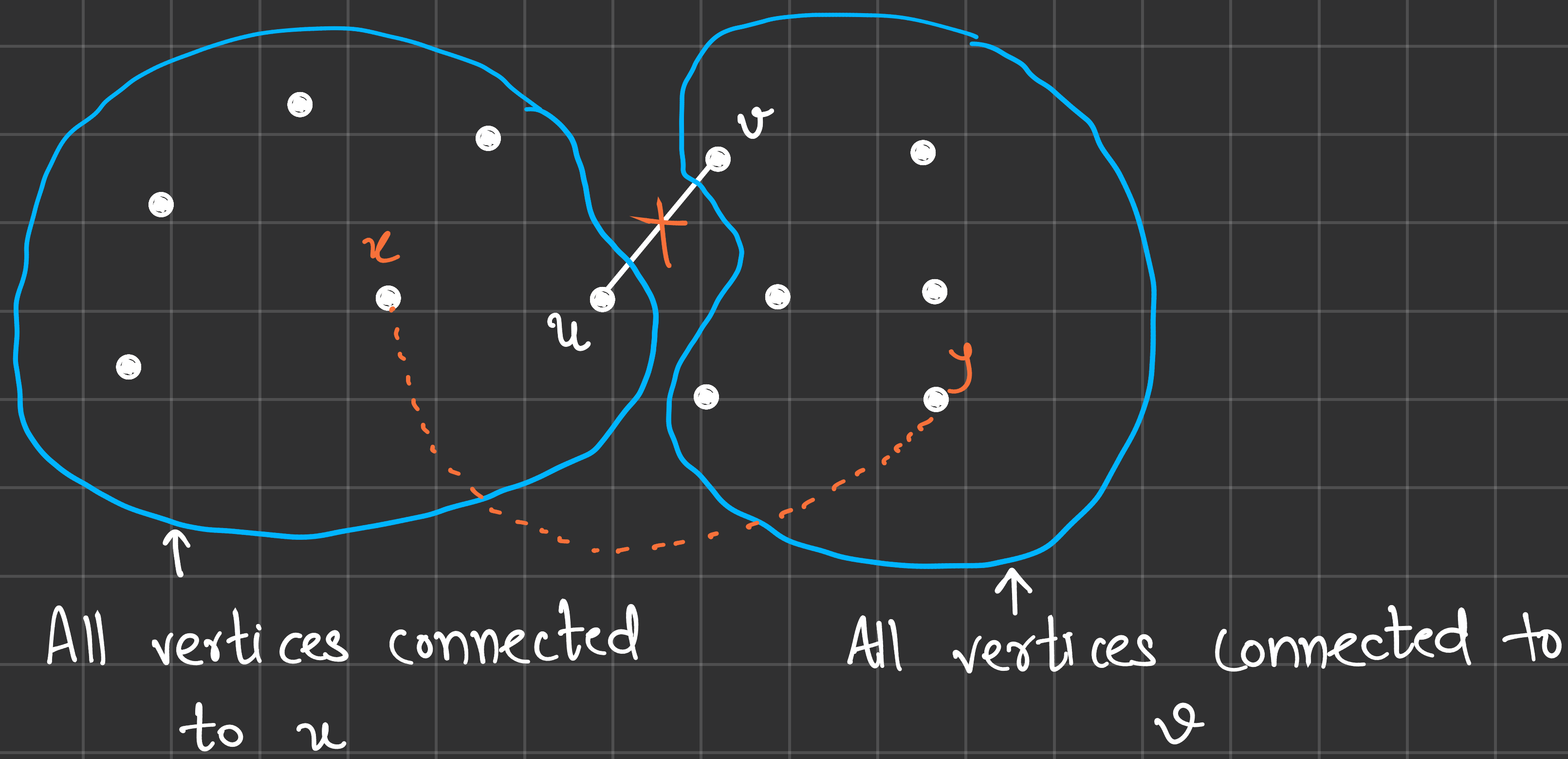
We have divided the graph into two connected components. or have we ?!



All vertices connected to u

All vertices connected to v

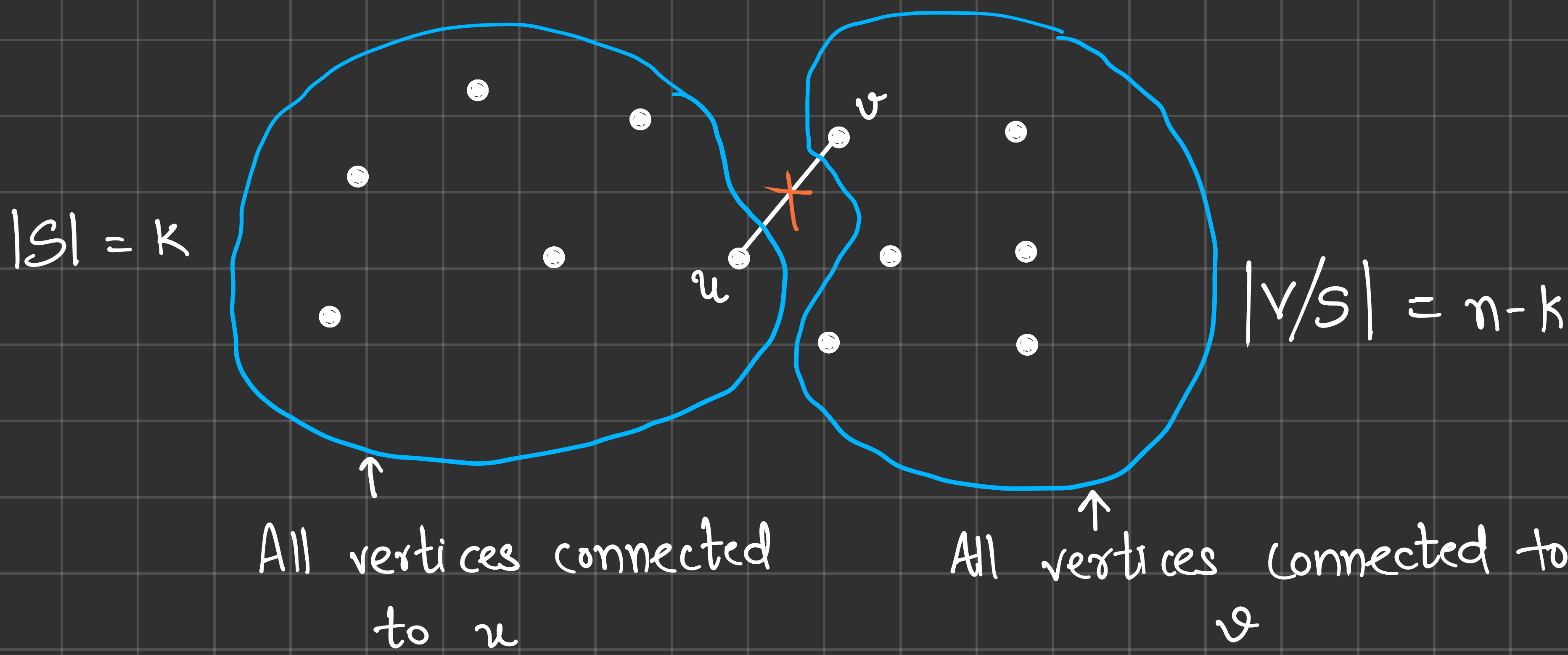
We have divided the graph into two connected components. or have we ?!



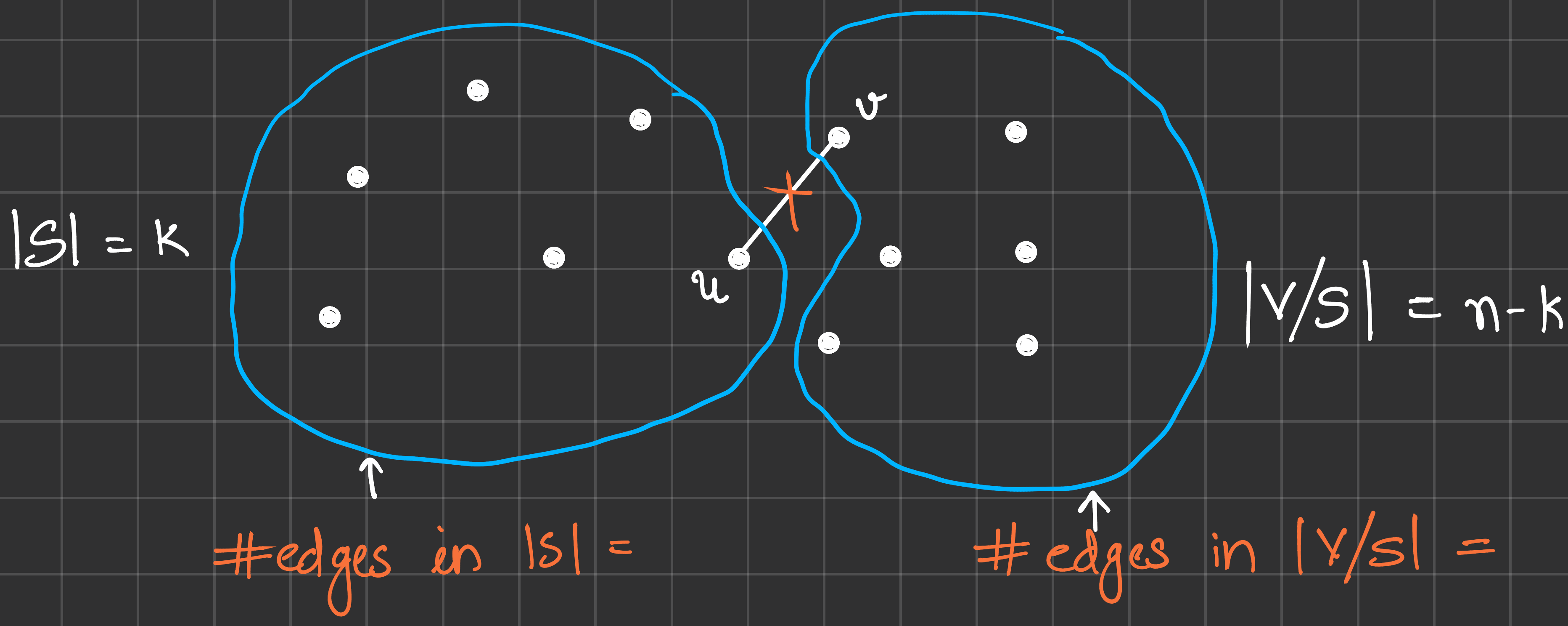
We have divided the graph into two connected components. or have we ?!

$\Rightarrow$ $u \rightsquigarrow x + xy + y \rightsquigarrow v + vu$ is a cycle.
But we started out with a tree

$\Rightarrow$ Edge xy cannot exist

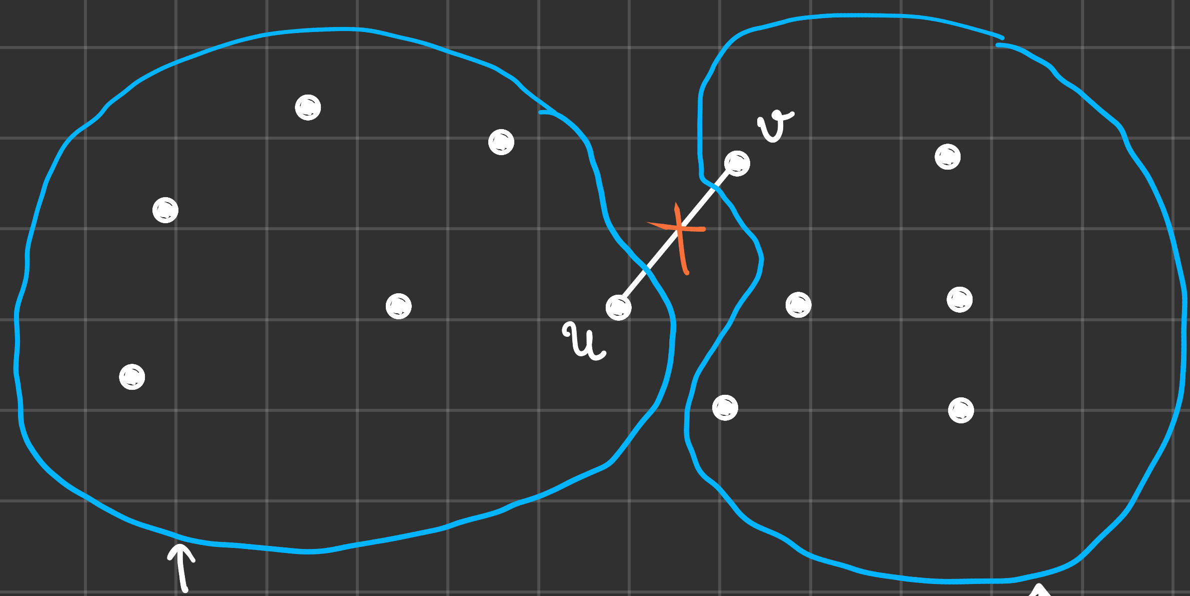


We have divided the graph into two connected components.



We have divided the graph into two connected components.

$|S| = K$



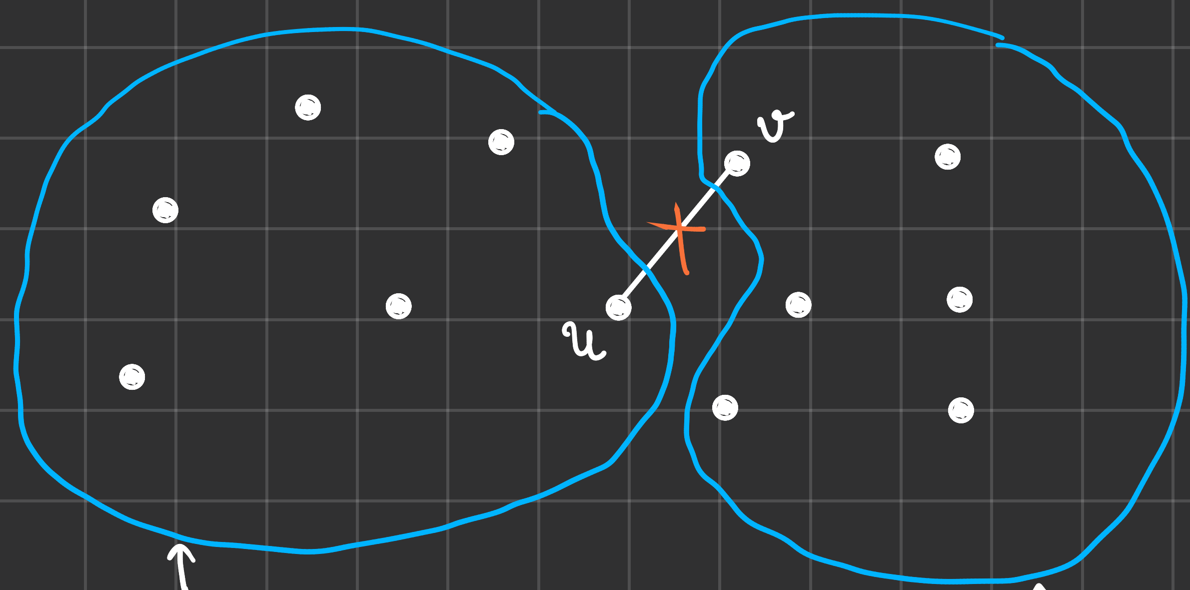
$|V/S| = n-k$

$\#$edges in $|S| = K-1$

$\#$edges in $|V/S| = n-K-1$

We have divided the graph into two connected components.

$|S| = k$



$|V/S| = n-k$

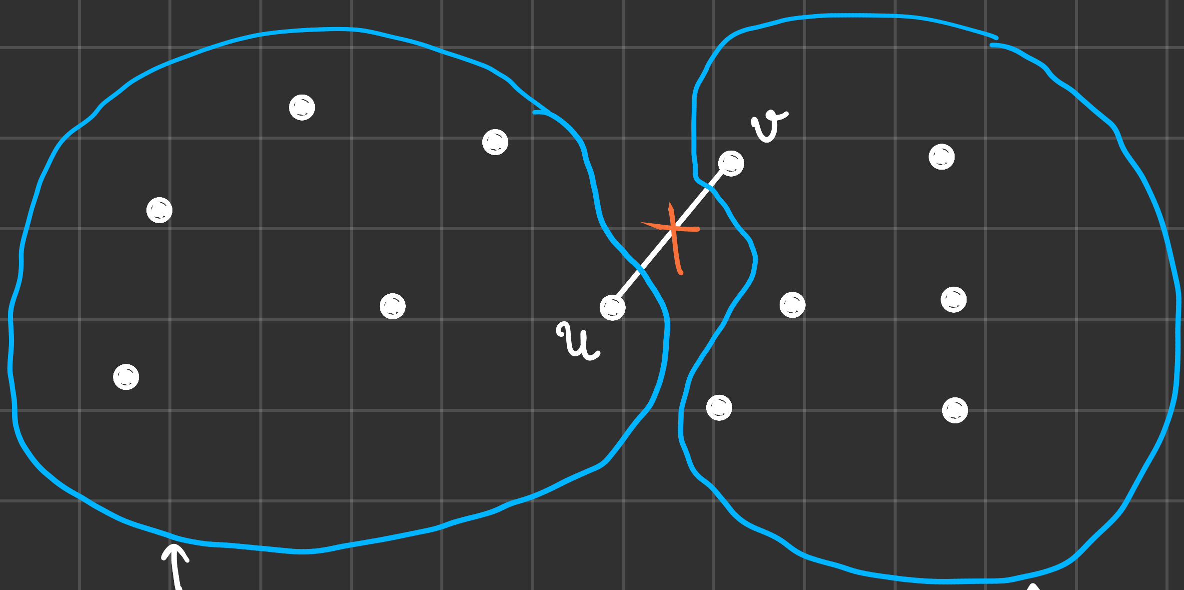
#edges in $|S| = k-1$

#edges in $|V/S| = n-k-1$

We have divided the graph into two connected components.

#edges in our tree $= k-1 + n-k-1 + 1$

$|S| = k$



$|V/S| = n-k$

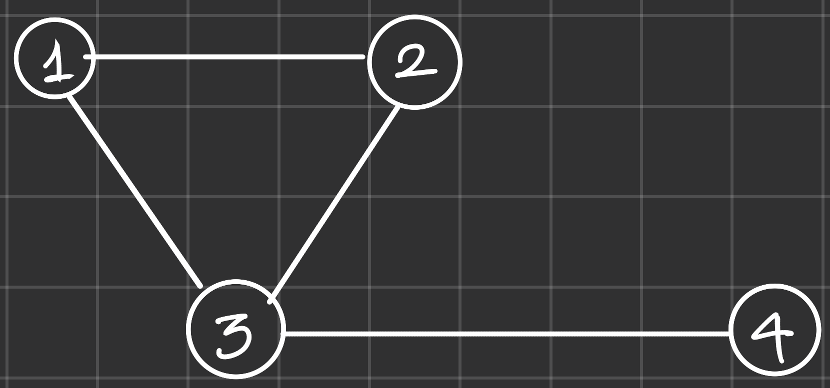
edges in $|S| = k-1$

edges in $|V/S| = n-k-1$

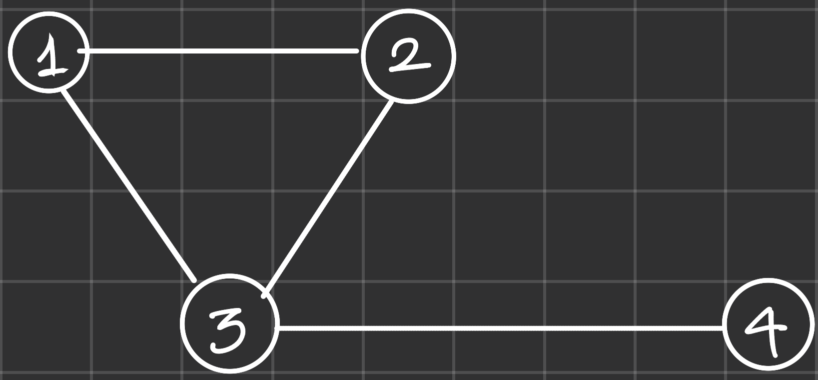
We have divided the graph into two connected components.

$$\#\text{edges in our tree} = k-1 + n-k-1 + 1$$
$$= n-1$$

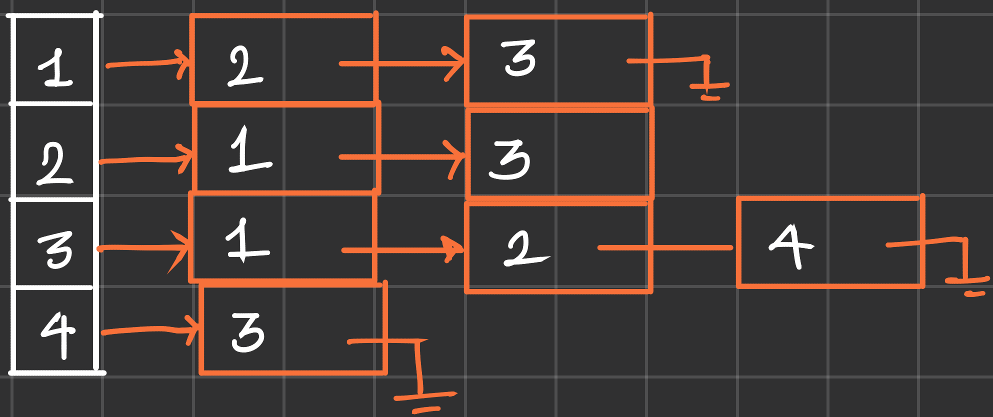
Data structure for graph representation



Data structure for graph representation



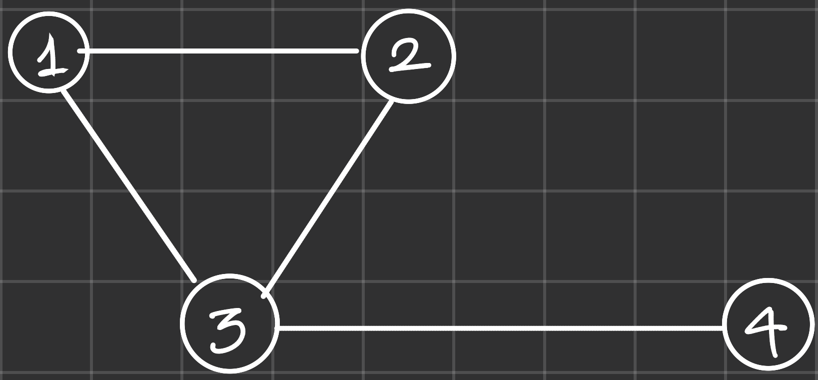
Adjacency list



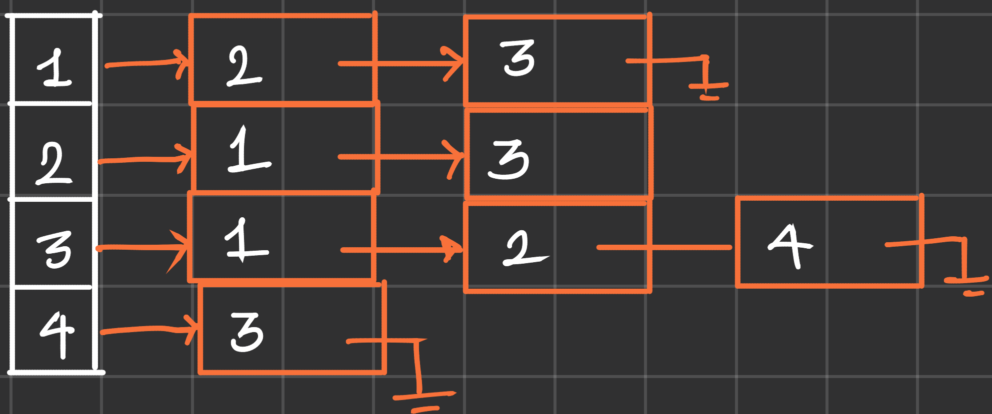
Adjacency matrix

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
$$

Data structure for graph representation



Adjacency list

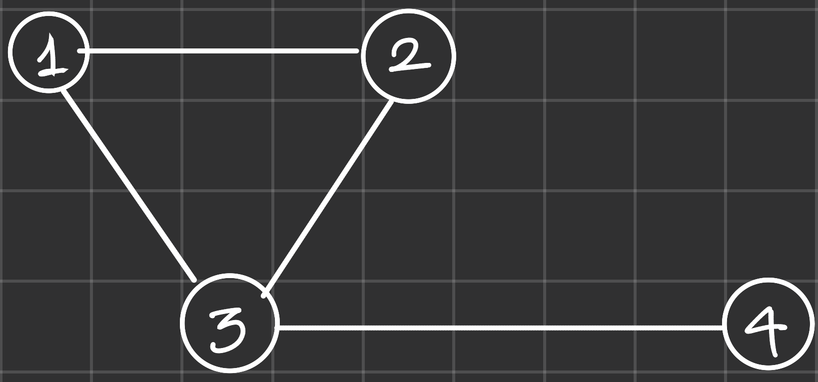


Adjacency matrix

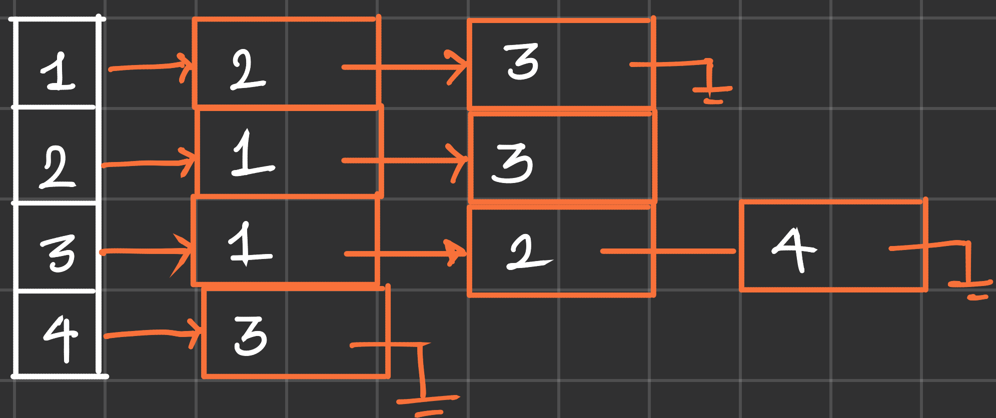
$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0
\end{array}
$$

Space:

Data structure for graph representation



Adjacency list

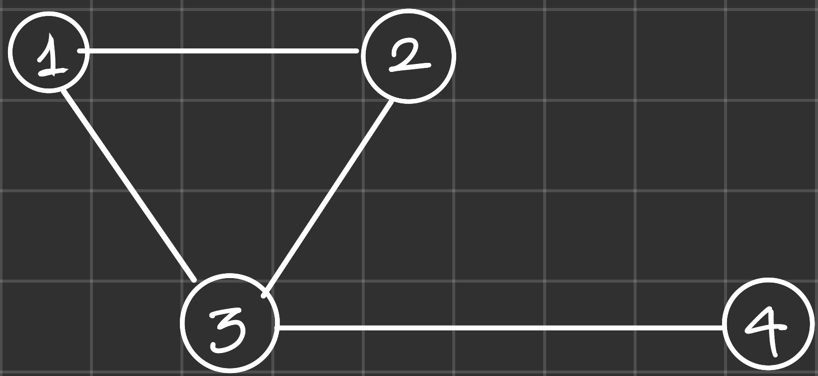


Adjacency matrix

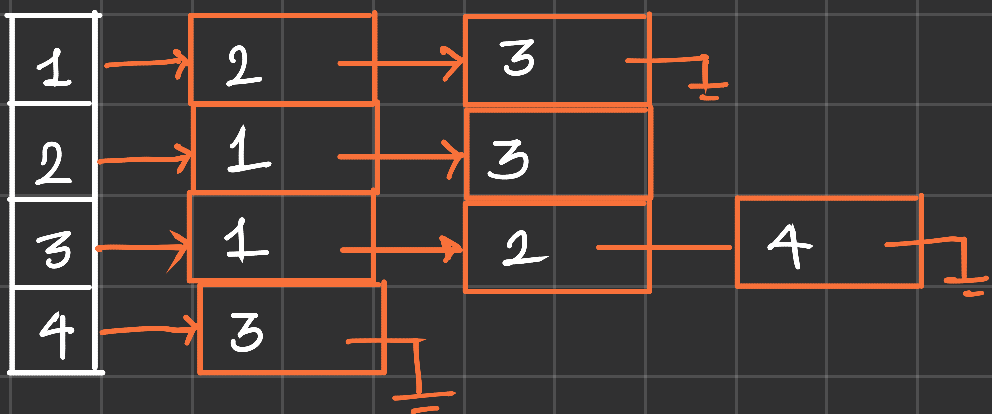
$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

Space: $O(m+n)$ $O(n^2)$

Data structure for graph representation



Adjacency list



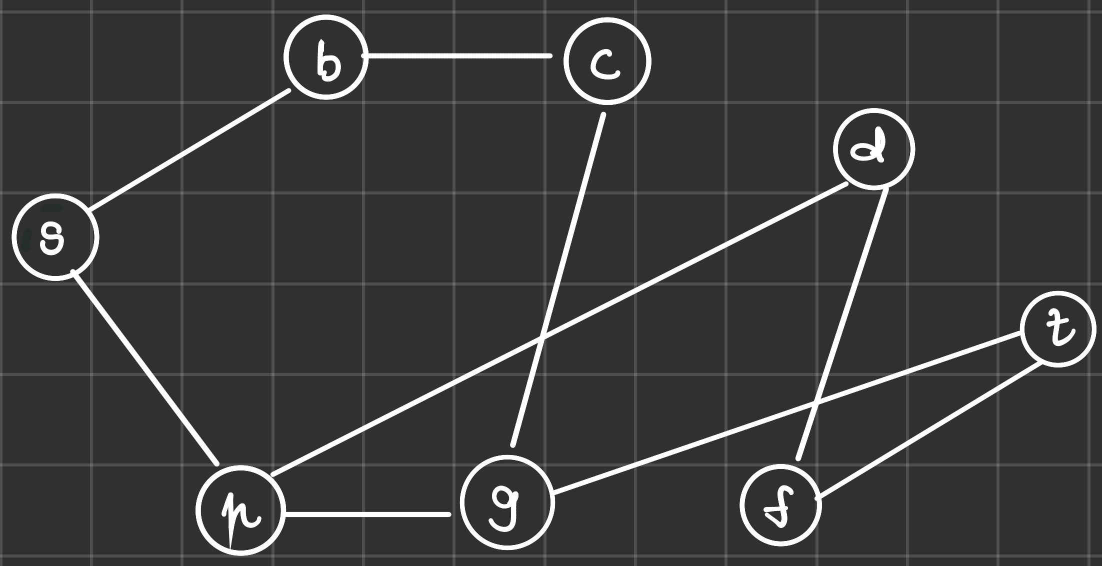
Adjacency matrix

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
$$

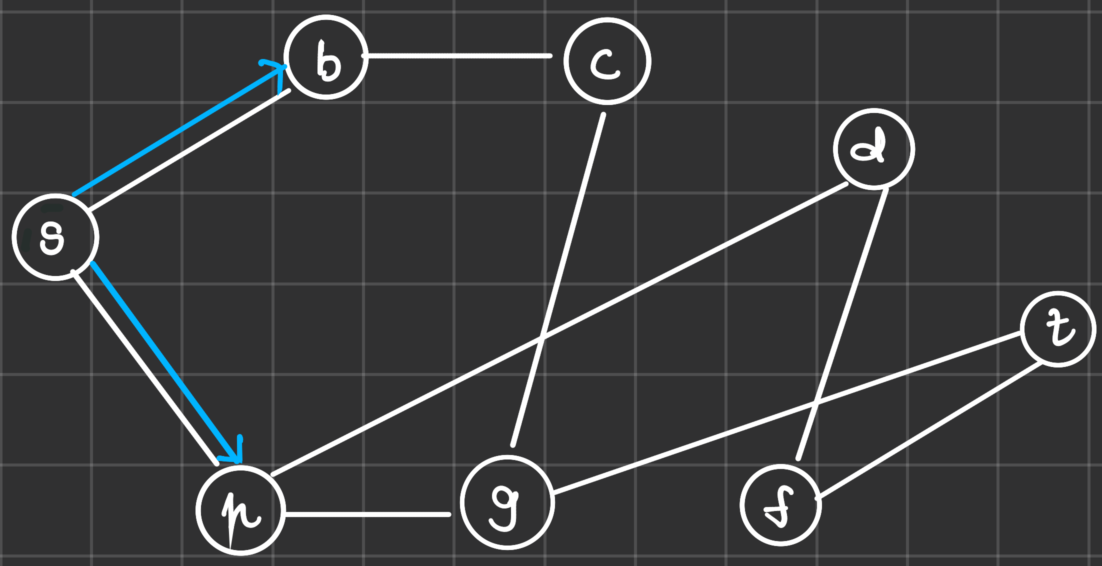
Space: $O(m+n)$ $O(n^2)$

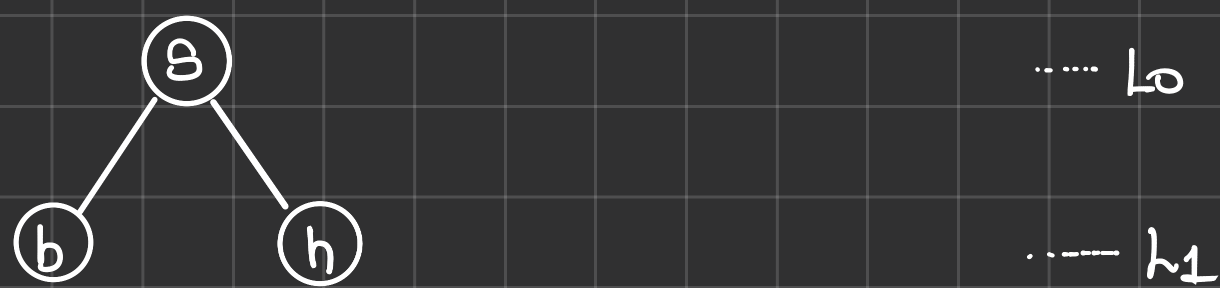
Unless stated otherwise, we will assume that the graph is given in adjacency form list.

Q: Given a graph G and two vertices s & t, find if t is reachable from s.

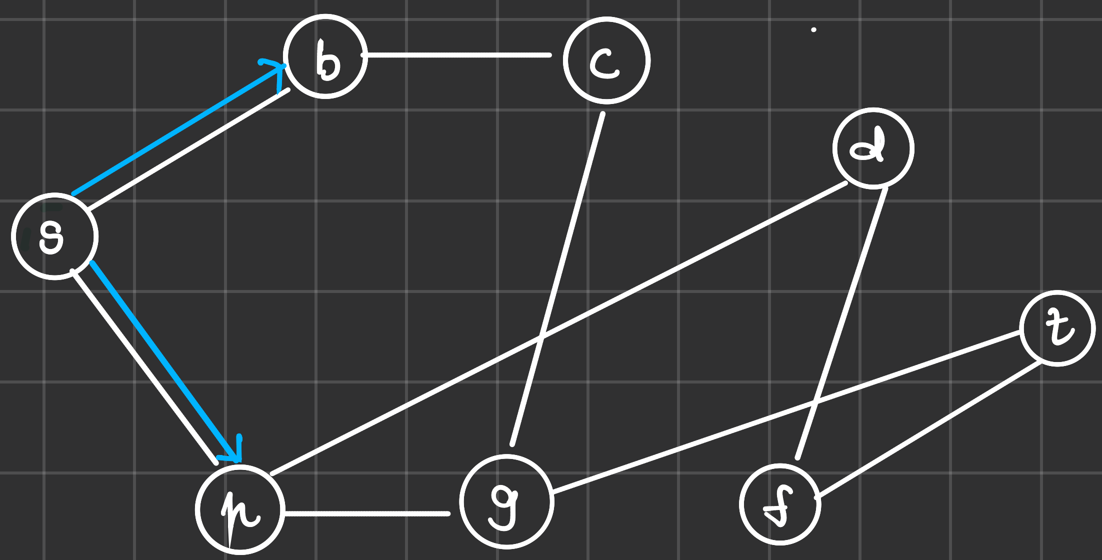


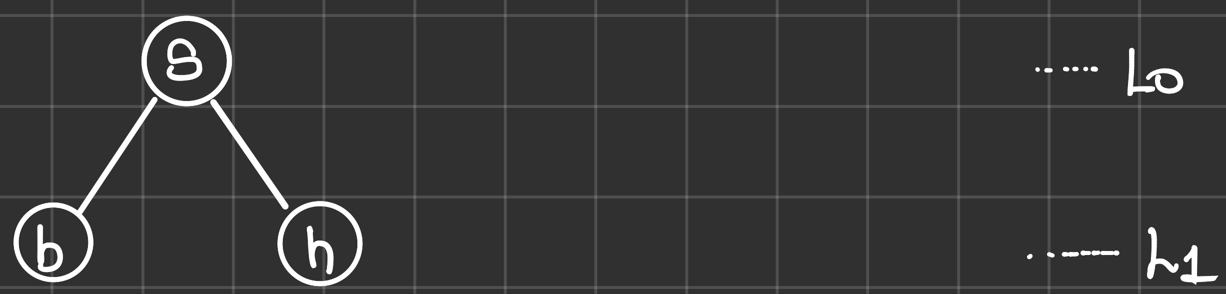
Q: Given a graph G and two vertices s & t,
find if t is reachable from s.





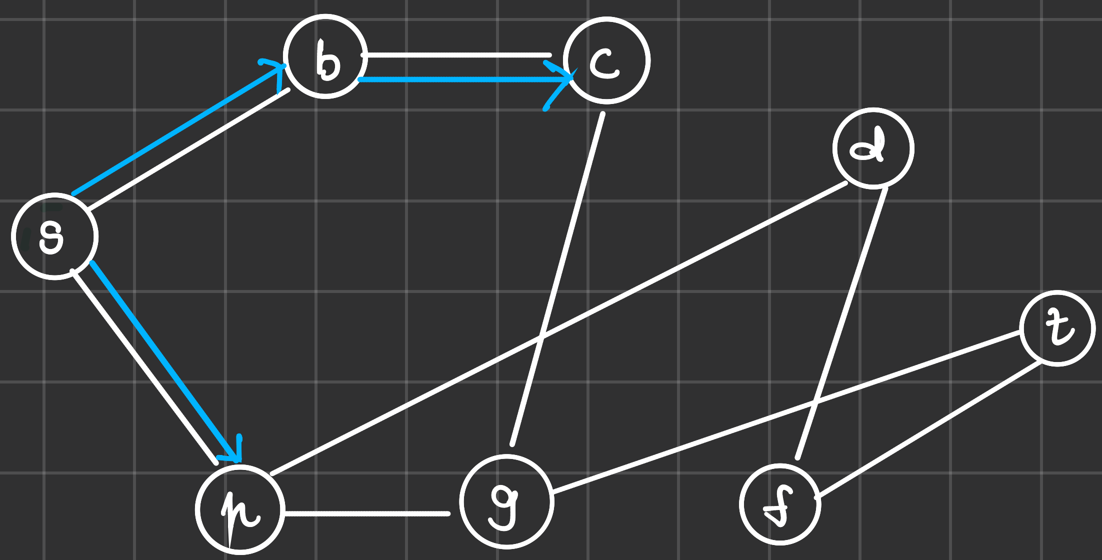
Q: Given a graph G and two vertices s & t, find if t is reachable from s.



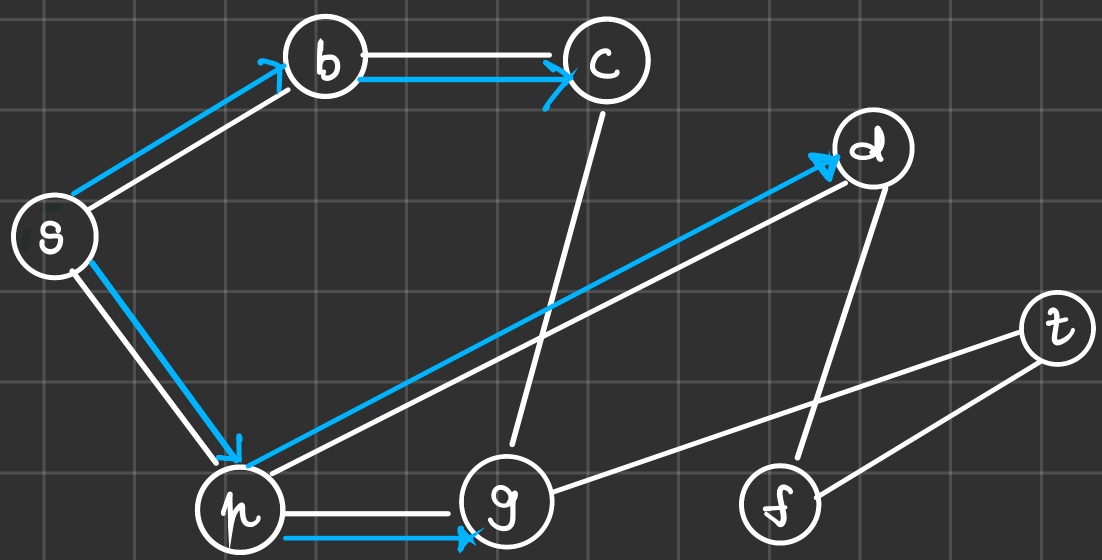


1) consider nodes in order
2) discover new vertices

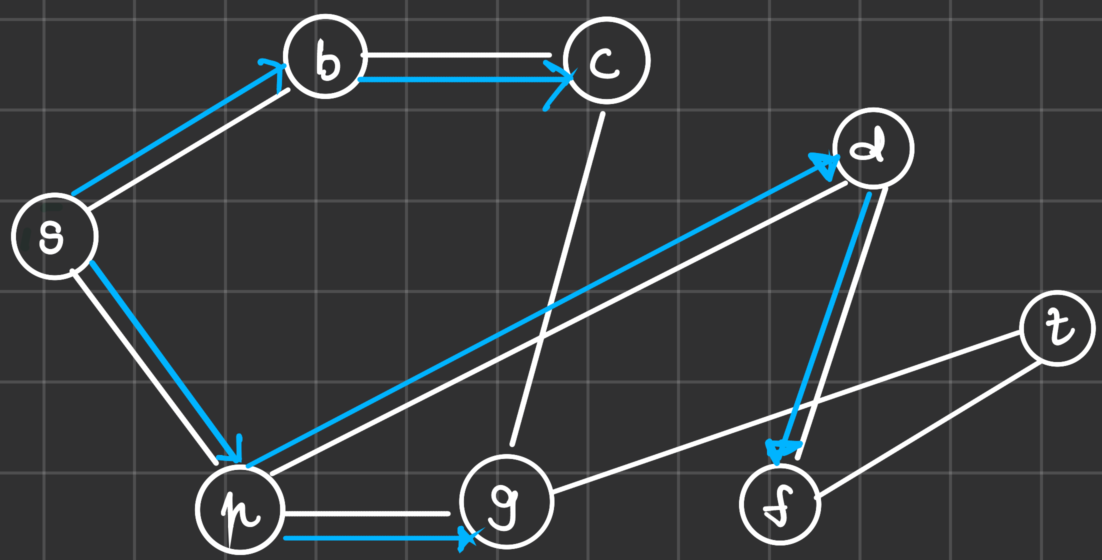
Q: Given a graph G and two vertices s & t, find if t is reachable from s.

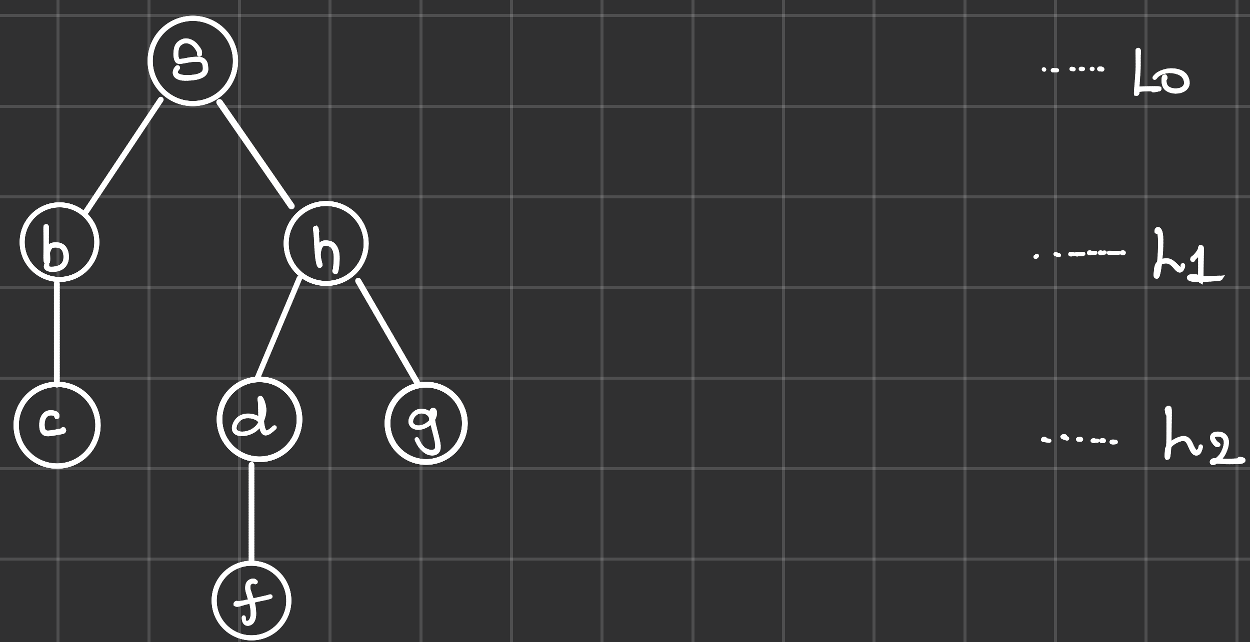


Q: Given a graph G and two vertices s & t, find if t is reachable from s.

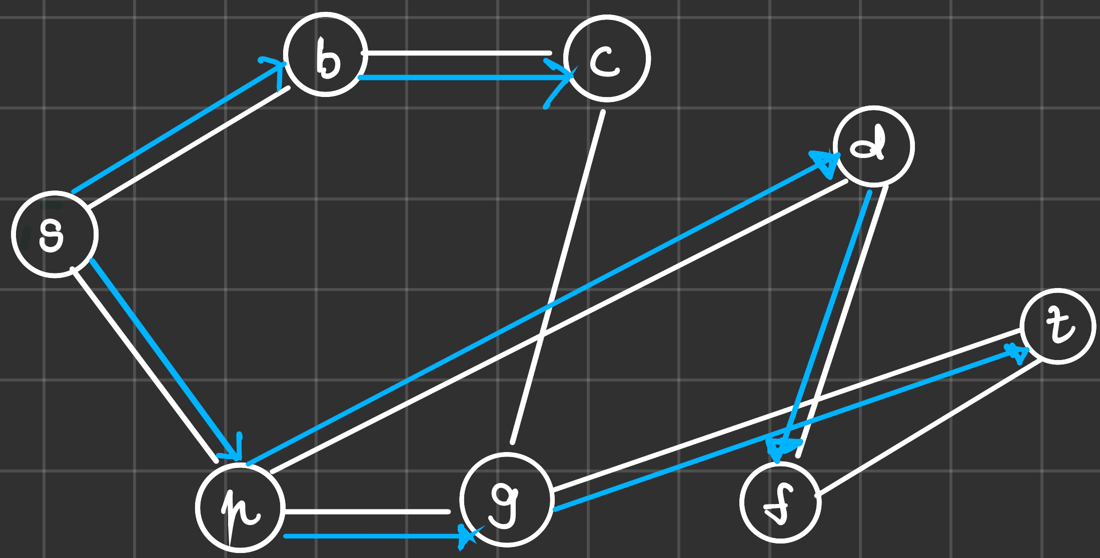


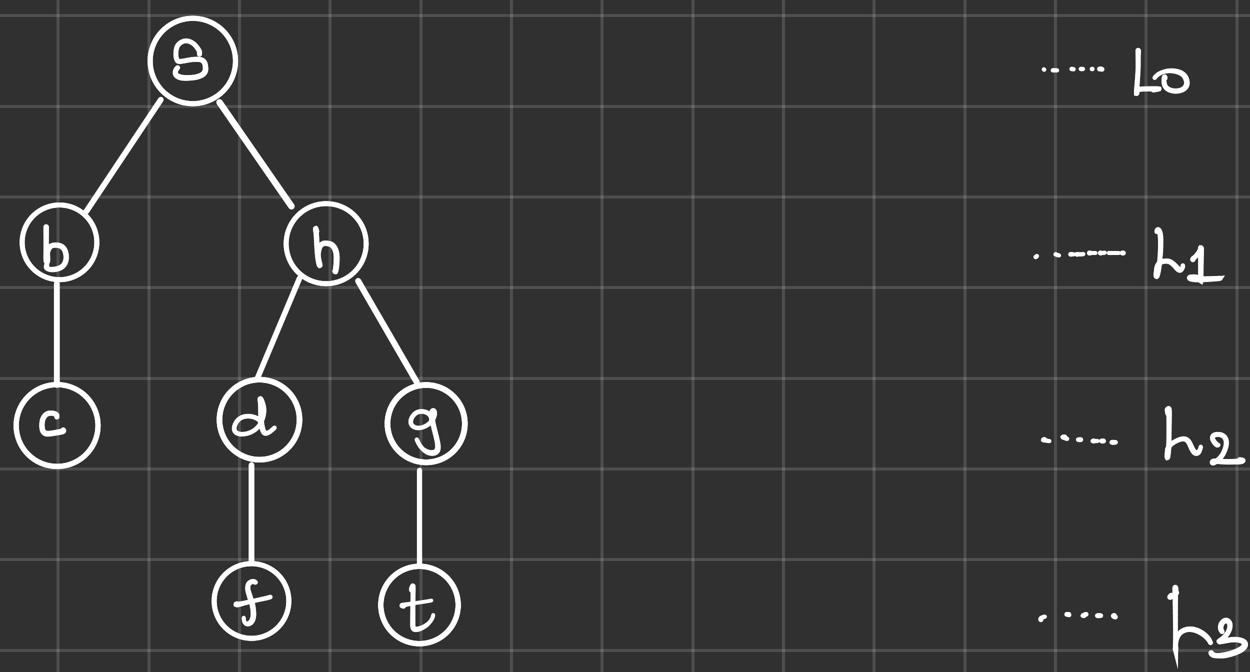
Q: Given a graph G and two vertices s & t, find if t is reachable from s.



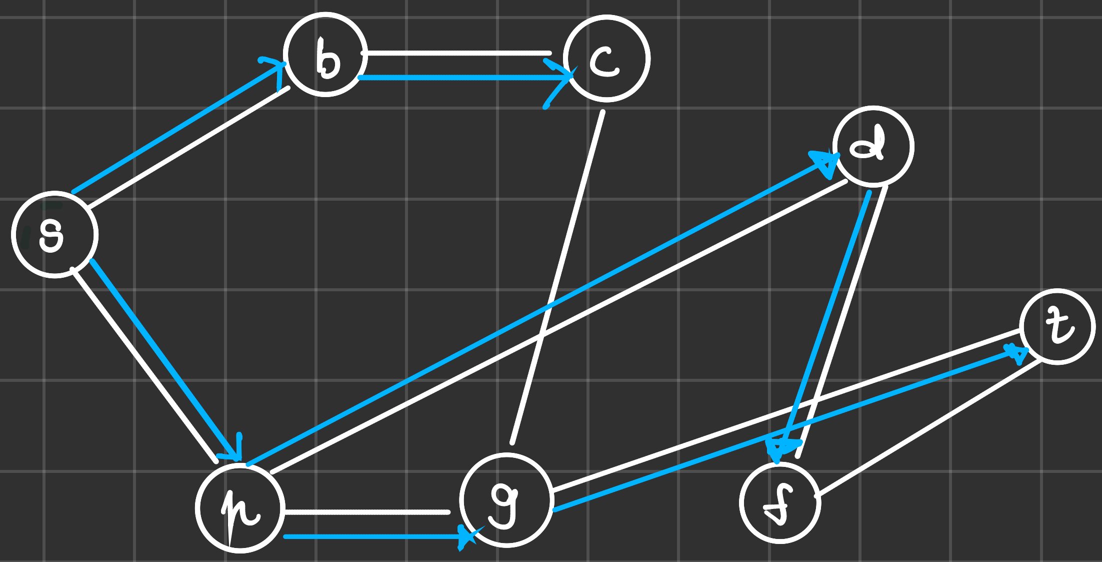


Q: Given a graph G and two vertices s & t, find if t is reachable from s.

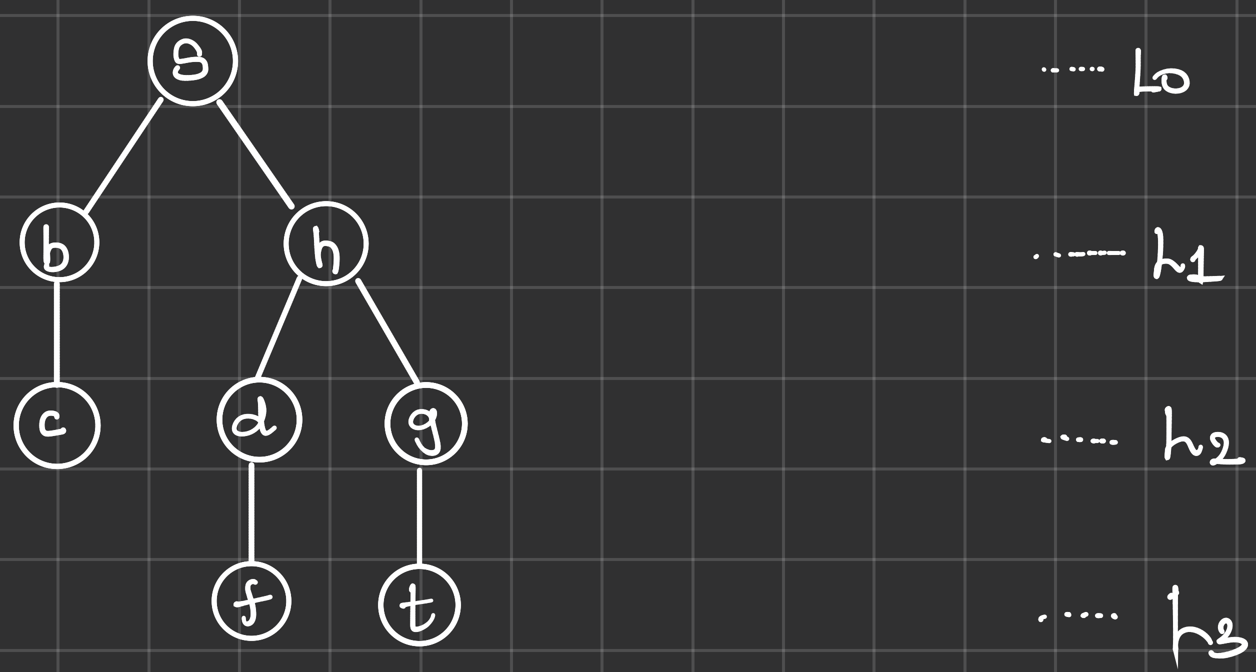




Q: Given a graph G and two vertices s & t, find if t is reachable from s.

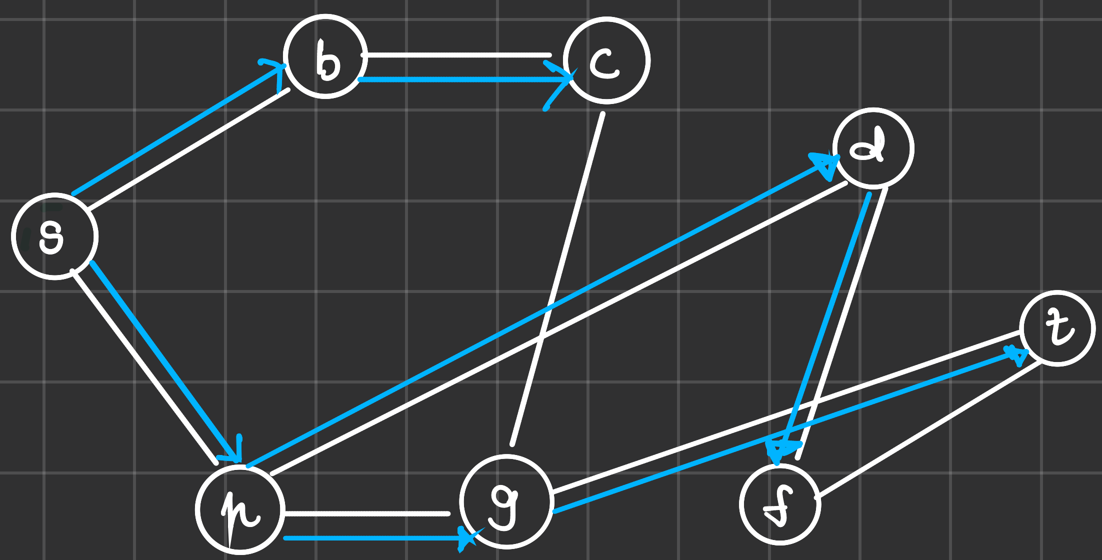


BFS tree

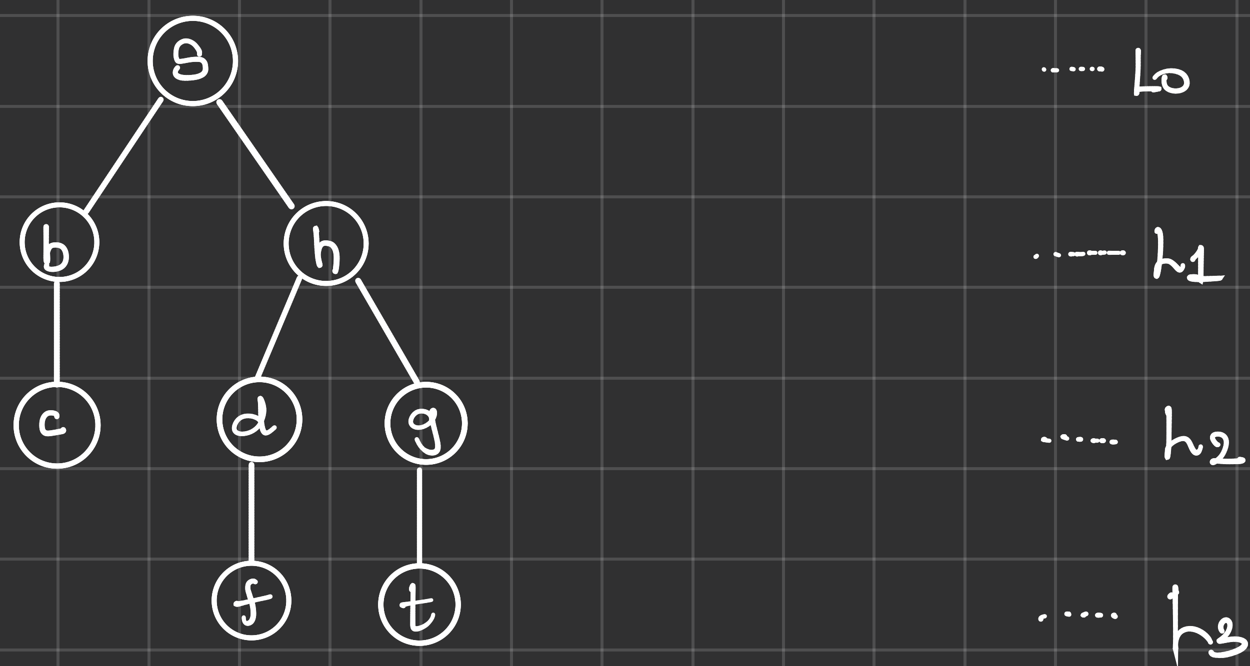


Tree Edges : all the edges in BFS tree (or all blue edges).

Q: Given a graph G and two vertices s & t, find if t is reachable from s.



BFS tree



Tree Edges : all the edges in BFS tree (or all blue edges).

Non-tree edges: edges not present in the BFS tree

{ (c,g) , (f,t) }

<u>Observation</u>: If distance from s to x, $d_G(s,x) = \ell$, then x will be part of layer i:

<u>Observation</u>: If distance from s to x, $d_G(s,x) = \ell$, then x will be part of layer : L_ℓ

$\Rightarrow$ If s & t are connected, then $d(s,t) = c$

$\Rightarrow$ t will lie in the BFS-tree

Observation: If distance from s to x, $d_G(s,x) = l$, then x will be part of layer : L_l

$\Rightarrow$ If s & t are connected, then $d(s,t) = c$

$\Rightarrow$ t will lie in the BFS - tree

Some more properties of Breadth First Search.

Lemma: Let xy be an edge in the graph. Let x lie in layer L_i & y lie in layer L_j then $|i-j| \leq 1$.

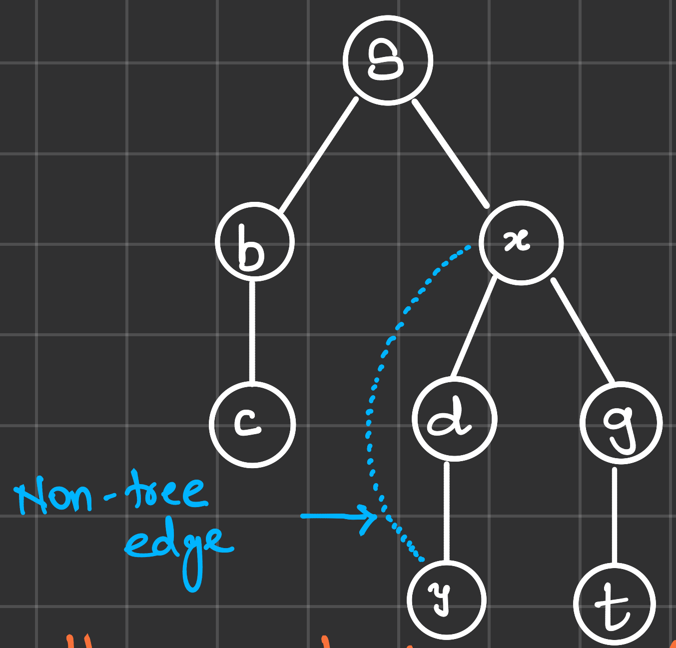
Observation: If distance from s to x, $d_G(s,x) = \ell$, then x will be part of layer : L_ℓ

$\Rightarrow$ If s & t are connected, then $d(s,t) = c$

$\Rightarrow$ t will lie in the BFS - tree

Some more properties of Breadth First Search.

Lemma: Let xy be an edge in the graph. Let x lie in layer L_i & y lie in layer L_j then $|i-j| \leq 1$.



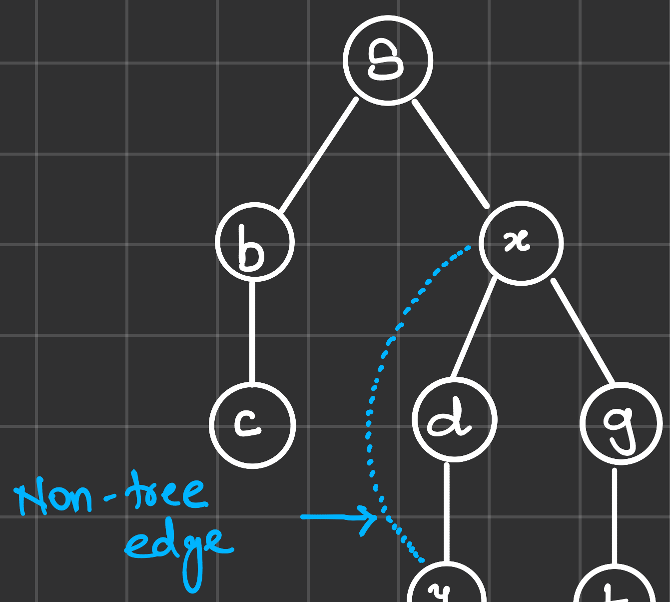
Why can this not happen !

Observation: If distance from s to x, $d_G(s,x) = \ell$, then x will be part of layer: L_ℓ

$\Rightarrow$ If s & t are connected, then $d(s,t) = c$
$\Rightarrow$ t will lie in the BFS-tree

Some more properties of Breadth First Search.

Lemma: Let xy be an edge in the graph. Let x lie in layer L_i & y lie in layer L_j then $|i-j| \leq 1$.



When x is processed, it should have found y. So y should be the child of x.

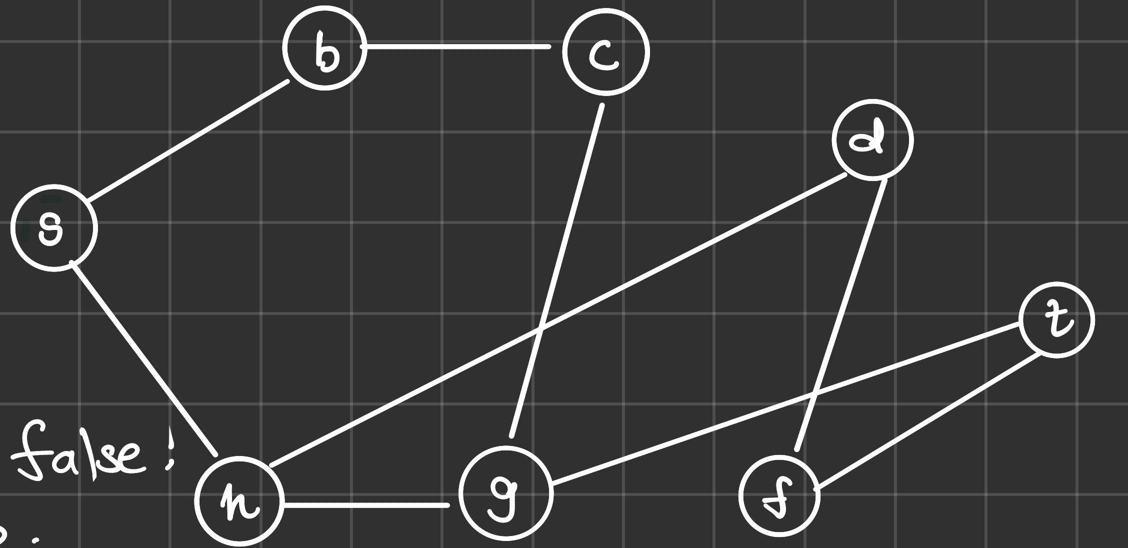
Implementing BFS

```
BFS (s)
{   for each v ∈ V
    {
        discovered[v] ← false;
    }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {


            }
        }
    }
}
```

Implementing BFS

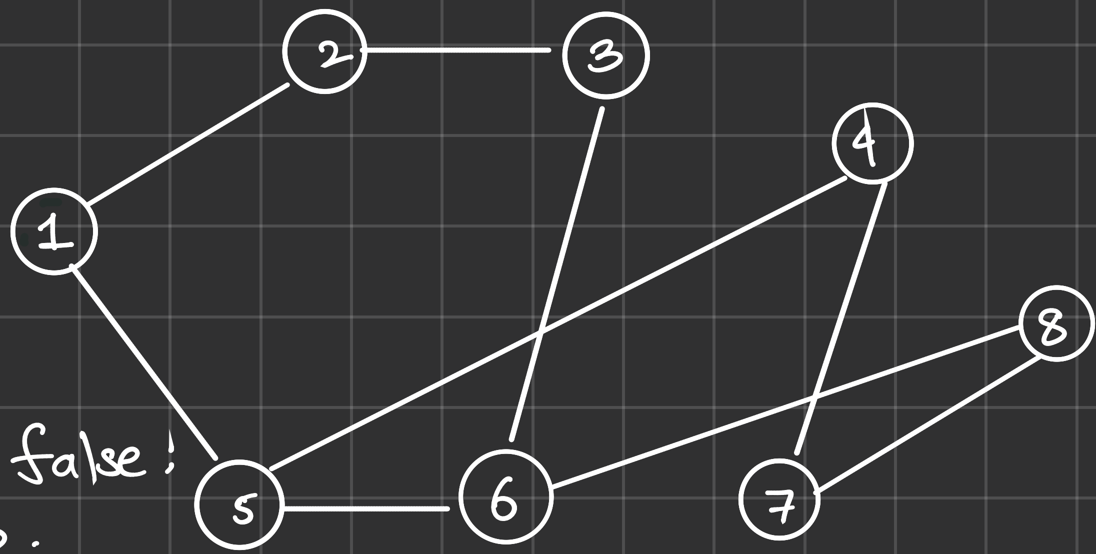
```
BFS (s)
{   for each v ∈ V
    {
        discovered [v] ← false ;
    }
    discovered [s] ← true ;

    Q. enqueue (s)
    while ( Q is not empty)
    {   v ← Q. deque () ;
        for each neighbor w of v
        {   if ( discovered [w] = false)
            {
                discovered [w] = true  ;   Q. enque(w) ;
                add  (v,w)  to  the  BFS  tree;
            }
        }
    }
}
```

Implementing BFS



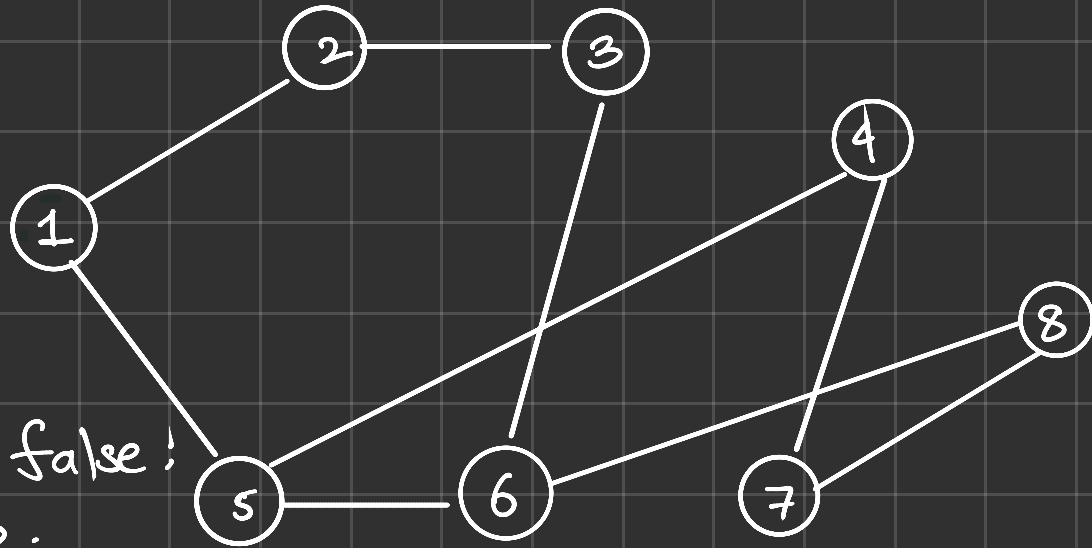

```
BFS (s)
{   for each v ∈ V
    {   discovered [v] ← false;
    }
    discovered [s] ← true;
    Q. enqueue (s)
    while ( Q is not empty)
    {   v ← Q. deque();
        for each neighbor w of v
        {   if ( discovered [w] = false)
            {   discovered [w] = true ;  Q. enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Implementing BFS




```
BFS (s)
{ → for each v ∈ V
  → {      discovered [v] ← false ;
  → }
    discovered [s] ← true ;

    Q. enqueue (s)
    while ( Q is not empty)
    {   v ← Q. deque() ;
        for each neighbor w of v
        {   if ( discovered[w] = false)
            {
                discovered [w] = true ;   Q. enque(w) ;
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Implementing BFS

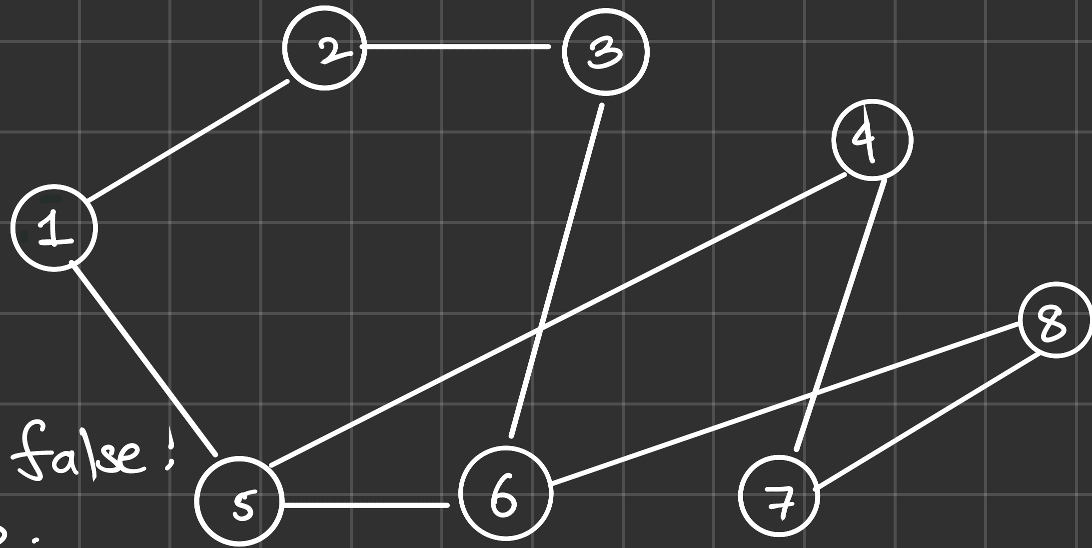



```
BFS(s)
{ → for each v ∈ V
  → {     discovered[v] ← false;
  → }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;   Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
f	f	f	f	f	f	f	f

discovered

Implementing BFS

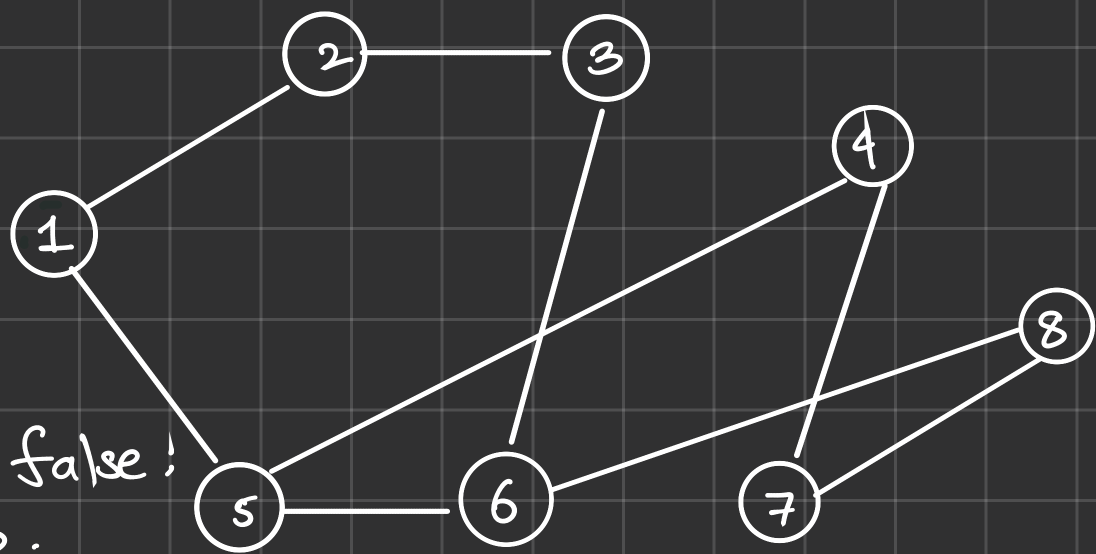



```
BFS(s)
{   for each v ∈ V
    {     discovered[v] ← false;
    }
  → discovered[s] ← true;
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {     discovered[w] = true ;  Q.enque(w);
                  add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	f	f	f	f	f	f	f

discovered

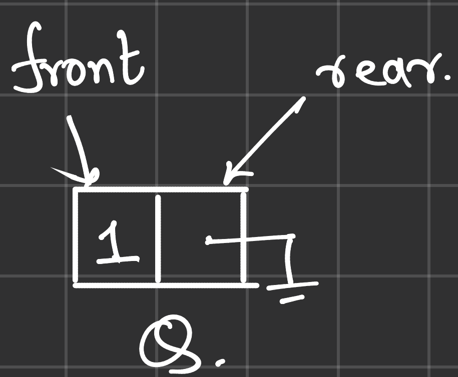
Implementing BFS



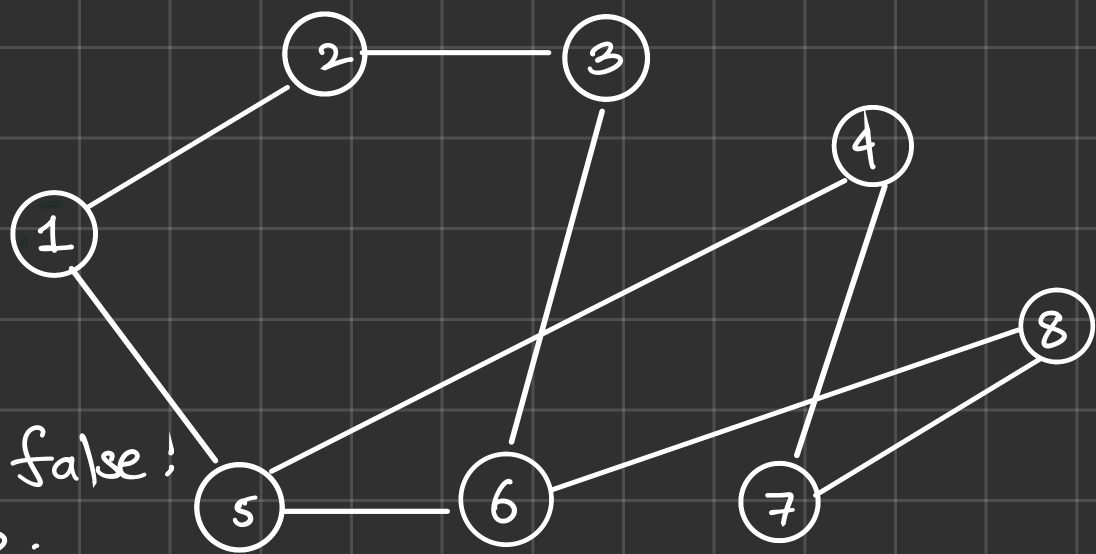

```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;
→   Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true ;  Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	f	f	f	f	f	f	f

discovered



Implementing BFS




```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true ; Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	f	f	f	f	f	f	f

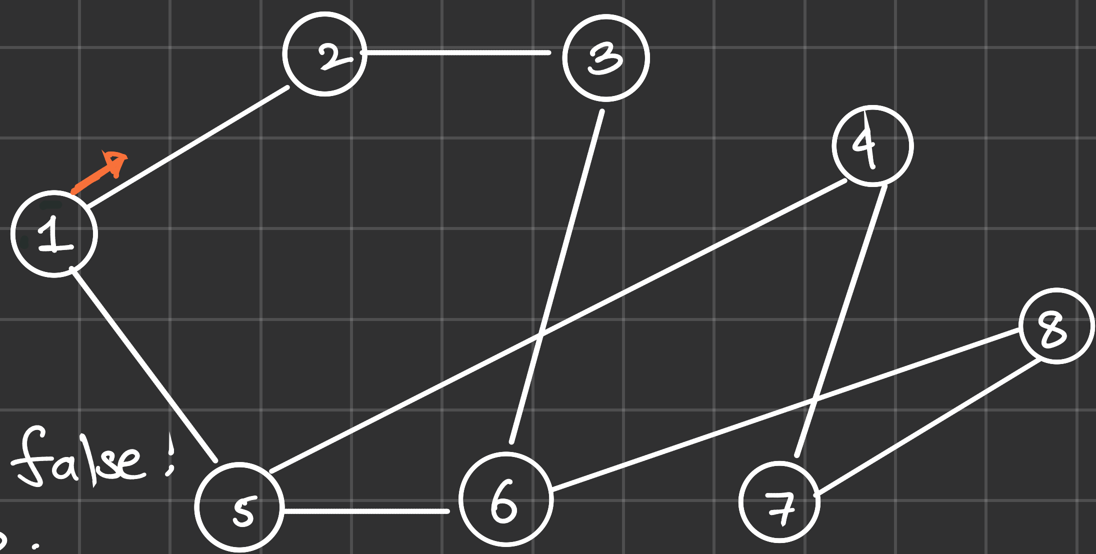
discovered

null

Q.

$v \leftarrow 1$

Implementing BFS




```
BFS(s)
{   for each v ∈ V
    {
        discovered[v] ← false;
    }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;  Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	f	f	f	f	f	f	f

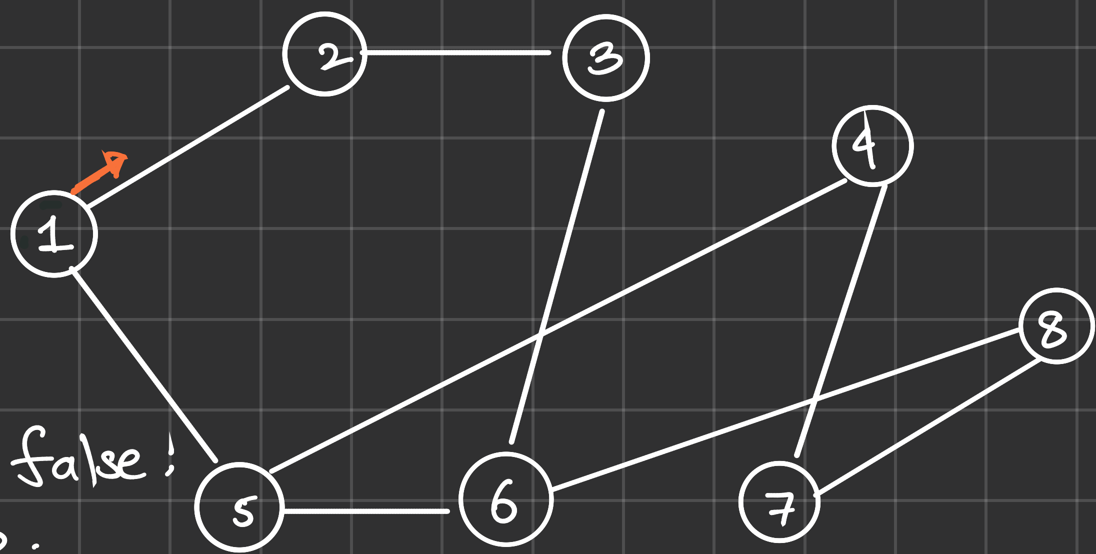
discovered

null.

Q.

$v \leftarrow 1$

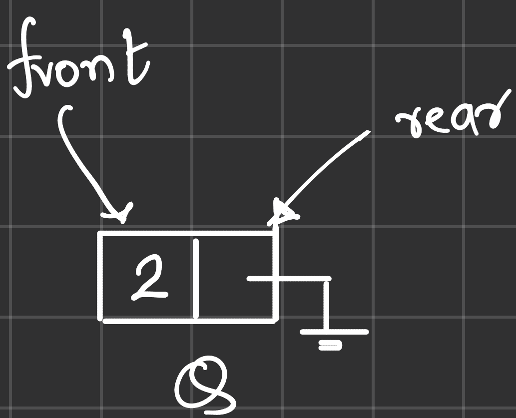
Implementing BFS




```
BFS(s)
{   for each v ∈ V
    {
        discovered[v] ← false;
    }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;  Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

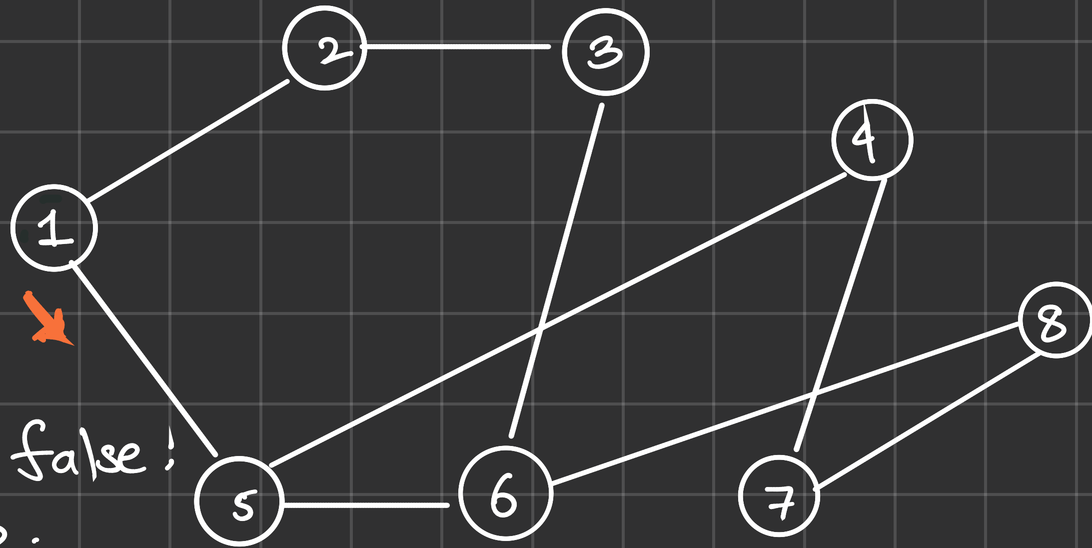
1	2	3	4	5	6	7	8
t	t	f	f	f	f	f	f

discovered



$v \leftarrow 1$

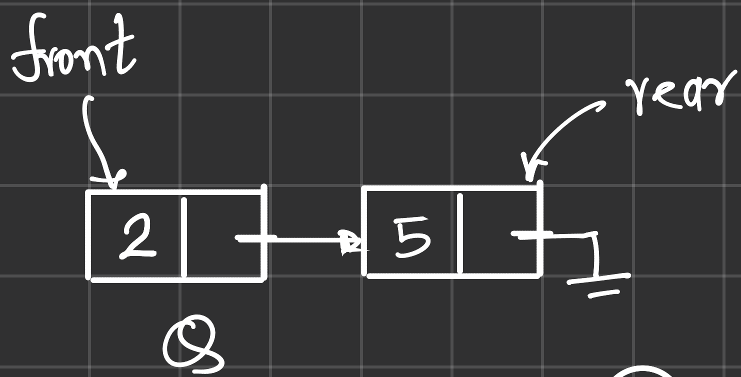
Implementing BFS



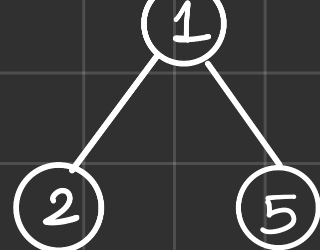

```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;

    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;  Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	t	f	f	t	f	f	f

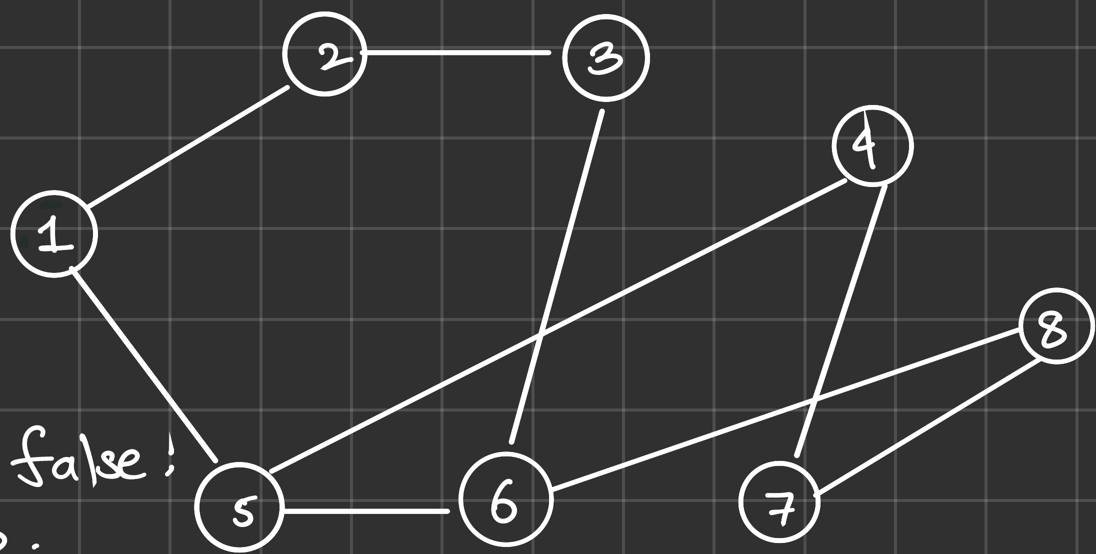
discovered



$v \leftarrow 1$



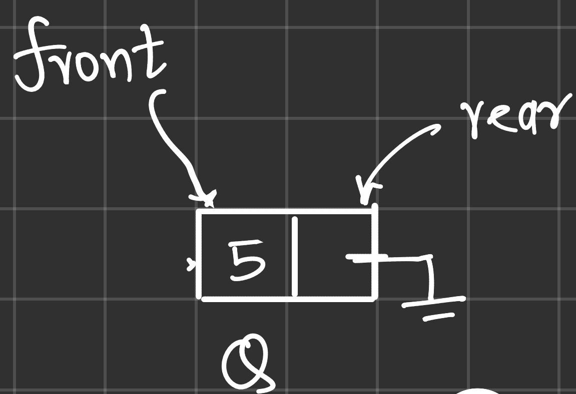
Implementing BFS

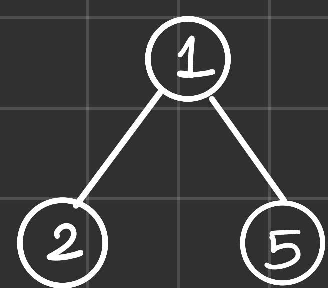


```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;
    Q.enqueue(s)
    while (Q is not empty)
→   {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true ; Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	t	f	f	t	f	f	f

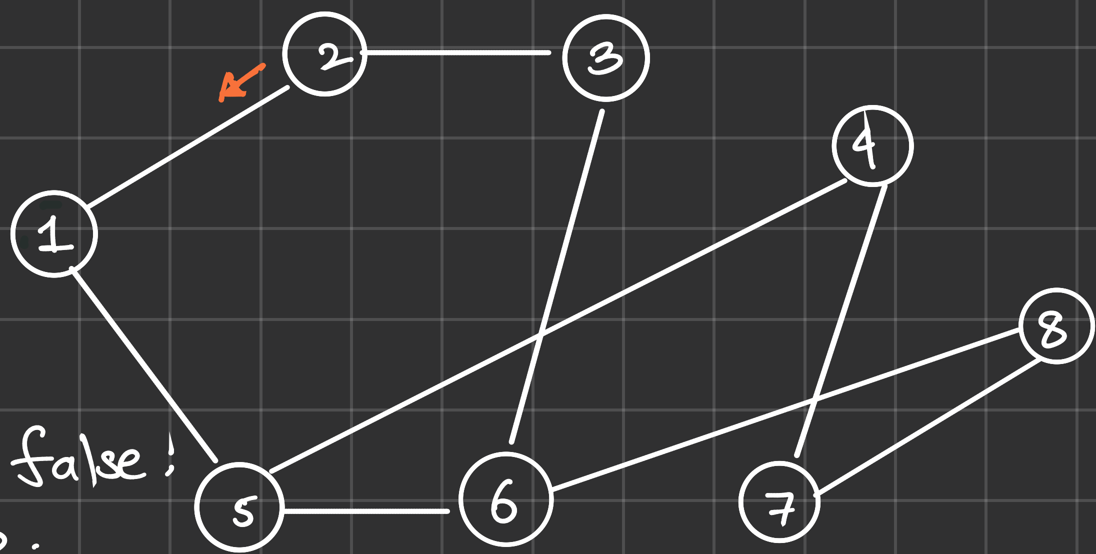
discovered





v ← 2

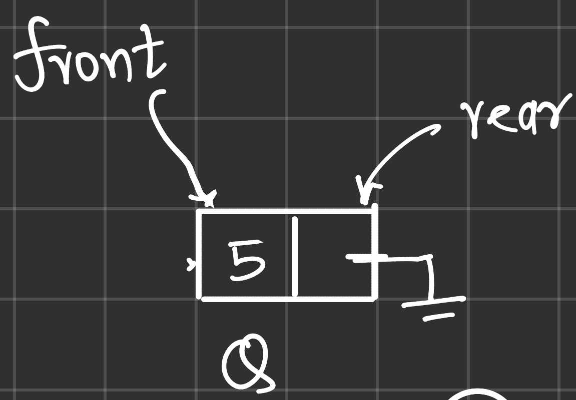
Implementing BFS

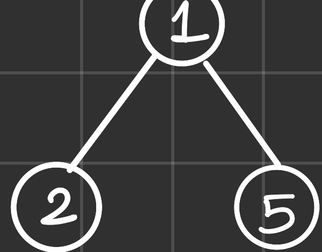



```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true ; Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	t	f	f	t	f	f	f

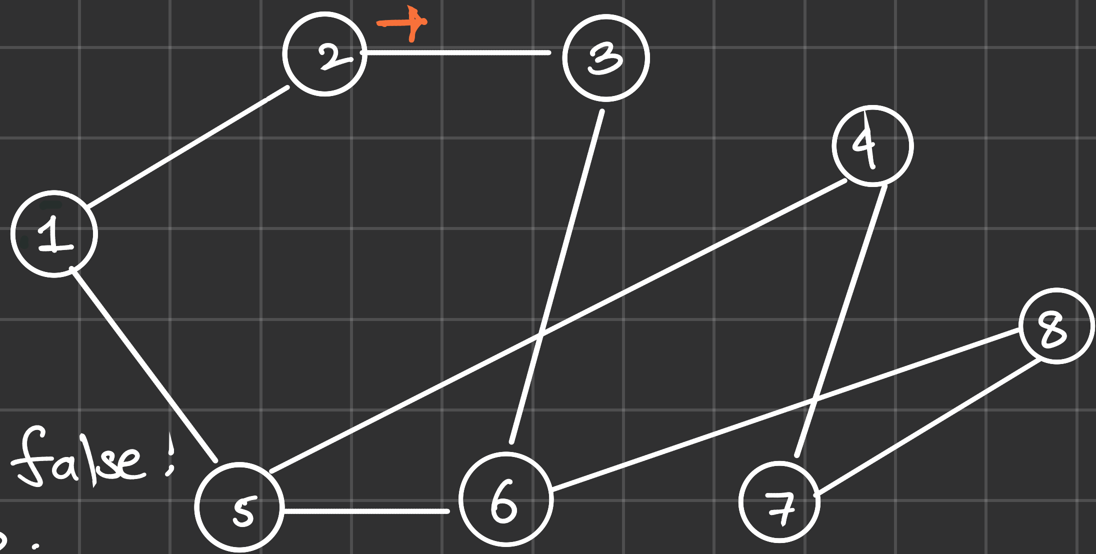
discovered





$v \leftarrow 2$

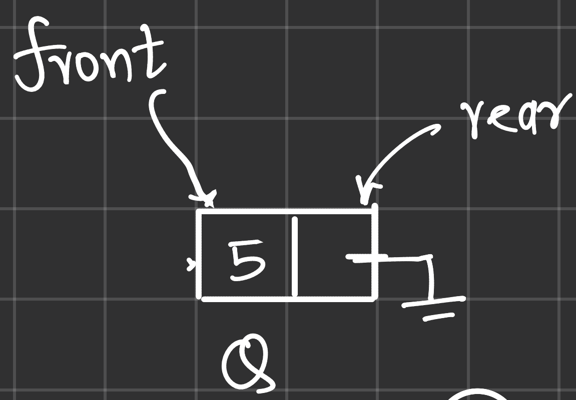
Implementing BFS

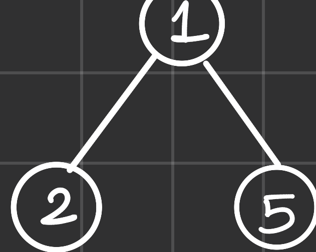



```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;
    }
    discovered[s] ← true;
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
    →   {   if (discovered[w] = false)
            {
                discovered[w] = true;   Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

1	2	3	4	5	6	7	8
t	t	f	f	t	f	f	f

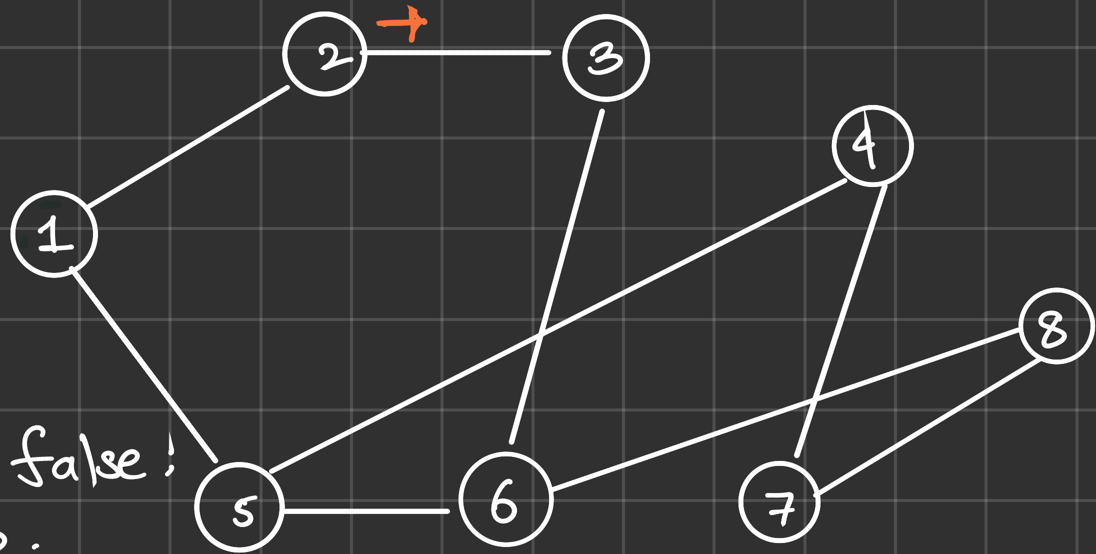
discovered





v ← 2

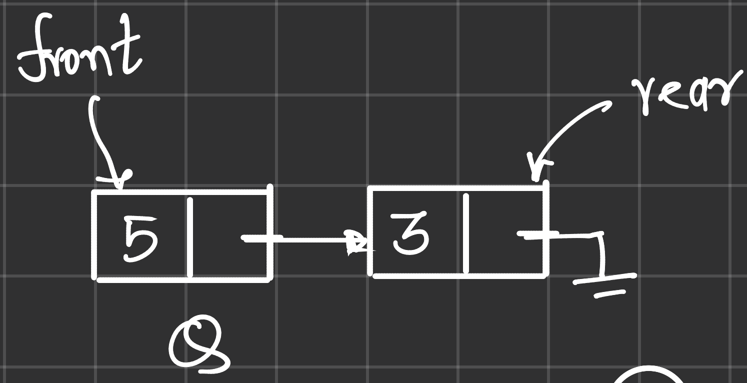
Implementing BFS



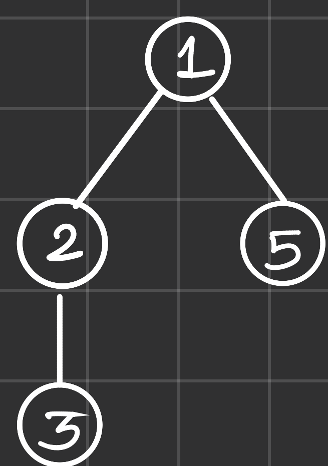

```
BFS (s)
{   for each v ∈ V
    {     discovered[v] ← false ;
    }
    discovered[s] ← true ;

    Q.enqueue (s)
    while ( Q is not empty)
    {    v ← Q.deque() ;
         for each neighbor w of v
         {   if ( discovered[w] = false)
             {        discovered [w] = true ;   Q.enque(w) ;
                      add (v,w) to the BFS tree;
             }
         }
    }
}
```

1	2	3	4	5	6	7	8
t	t	t	f	t	f	f	f

discovered



v ← 2



Implementing BFS

```
BFS(s)
{   for each v ∈ V
    {
        discovered[v] ← false;
    }
    discovered[s] ← true;
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true;   Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time :

Implementing BFS

```
BFS(s)
{   for each v ∈ V
    {   discovered[v] ← false;              } O(n)
    }
    discovered[s] ← true;                   } O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                      } O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true;  Q.enque(w);
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time :

Implementing BFS

```
BFS(s)
{   for each v ∈ V
    {
        discovered[v] ← false;                              O(n)
    }
    discovered[s] ← true;                                   O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                                      O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ; Q.enque(w);          O(d(v))
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time :

Implementing BFS

```
BFS(s)
{   for each v ∈ V
    {      discovered[v] ← false;             O(n)
    }
    discovered[s] ← true;                      O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                         O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {   discovered[w] = true ;  Q.enque(w);        O(d(v))
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time :

Main Observation : Each vertex is discovered only once
$\Rightarrow$ Each vertex enters the queue only once.

Implementing BFS

```
BFS (s)
{   for each v ∈ V
    {
        discovered[v] ← false;                          } O(n)
    }
    discovered[s] ← true;                               } O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                                  } O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;  Q.enque(w);     } O(d(v))
                add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time : $O(n) + O(1) + \sum_{v \in V} O(1 + d(v))$

Main Observation : Each vertex is discovered only once
$\Rightarrow$ Each vertex enters the queue only once.

Implementing BFS

```
BFS(s)
{   for each v ∈ V                                   } O(n)
    {     discovered[v] ← false;
    }
    discovered[s] ← true;                            } O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                               } O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {     discovered[w] = true ;  Q.enque(w);     } O(d(v))
                  add (v,w) to the BFS tree;
            }
        }
    }
}
```

Running time : $O(n) + O(1) + \sum_{v \in V} O(1 + d(v))$

$= O(n) + O(1) + O(n+m)$

$= O(n+m)$

Implementing BFS

```
BFS(s)
{   for each v ∈ V
    {
        discovered[v] ← false;
    }                                          } O(n)
    discovered[s] ← true;
                                               } O(1)
    Q.enqueue(s)
    while (Q is not empty)
    {   v ← Q.deque();                         } O(1)
        for each neighbor w of v
        {   if (discovered[w] = false)
            {
                discovered[w] = true ;  Q.enque(w);
                add (v,w) to the BFS tree;     } O(d(v))
            }
        }
    }
}
```

Running time : $c_1 n + c_2 + \sum_{v \in V} c_3 + c_4 d(v)$

$= c_1 n + c_2 + c_3 \cdot n + c_4 \cdot 2m$

$= O(n+m)$

Our problem: Find if t is reachable from s.

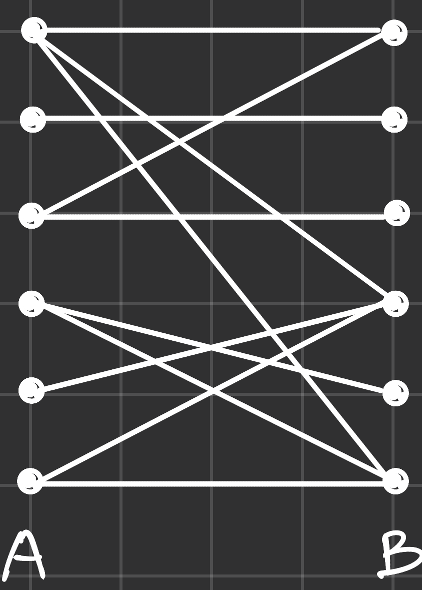
```
Reachable(s,t)
{
```

Our problem : Find if t is reachable from s.

```
Reachable(s,t)
{
    BFS (s) ;
    if ( discovered[t] = true)
        t is reachable from s
    else
        t is not reachable from s
}
```

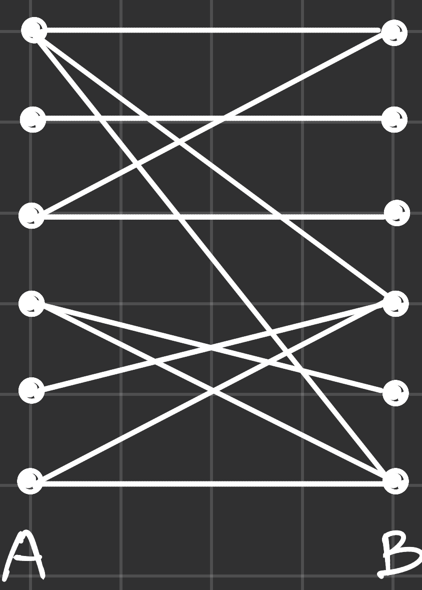
Problem: Given a connected graph G, find if it is a bipartite graph?

Problem : Given a connected graph G, find if it is a bipartite graph?



For each edge (x,y) in a bipartite graph,
$(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

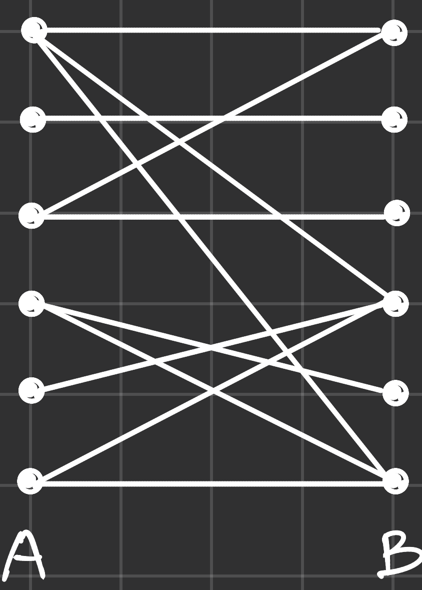
Problem : Given a connected graph G, find if it is a bipartite graph?



For each edge (x,y) in a bipartite graph, $(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

Property : If graph G is bipartite, it cannot contain odd cycle

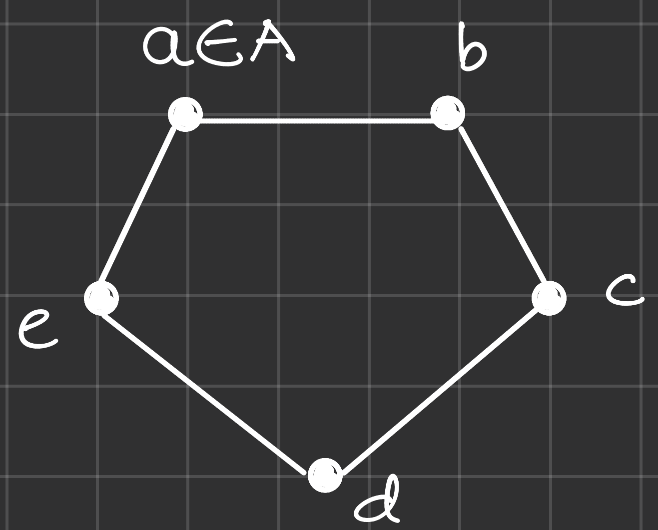
Problem : Given a connected graph G, find if it is a bipartite graph ?



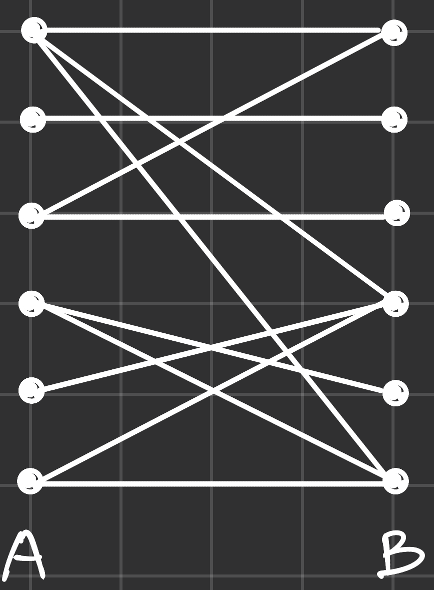
For each edge (x,y) in a bipartite graph, $(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

Property : If graph G is bipartite, it cannot contain odd cycle

Proof : By contradiction



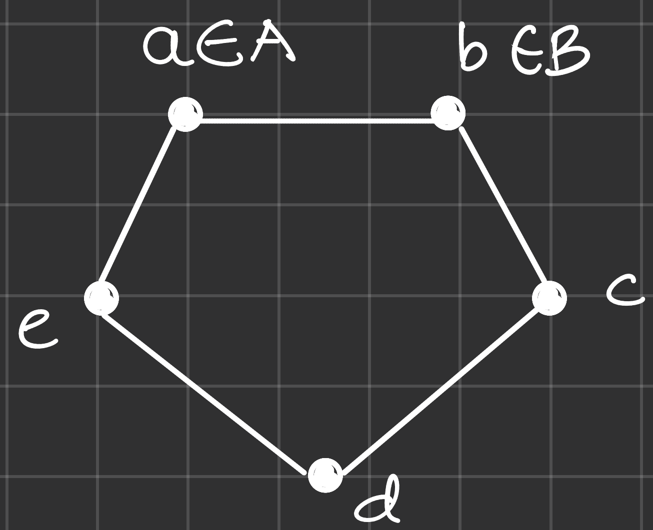
Problem: Given a connected graph G, find if it is a bipartite graph?



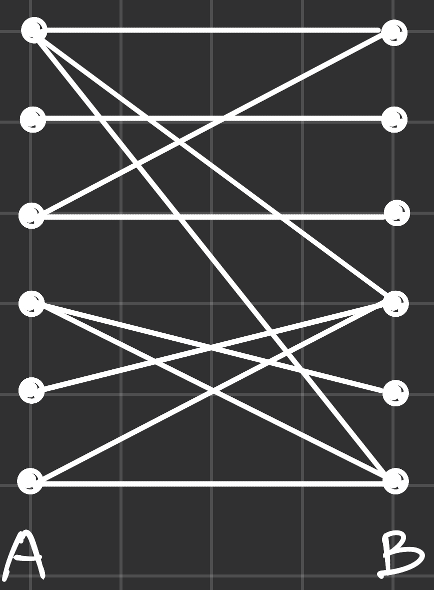
For each edge (x, y) in a bipartite graph, $(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

Property: If graph G is bipartite, it cannot contain odd cycle

Proof: By contradiction



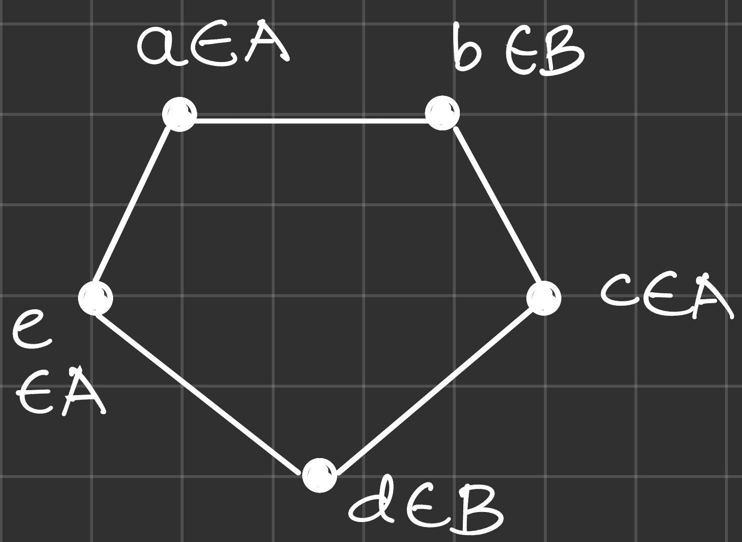
Problem: Given a connected graph G, find if it is a bipartite graph?



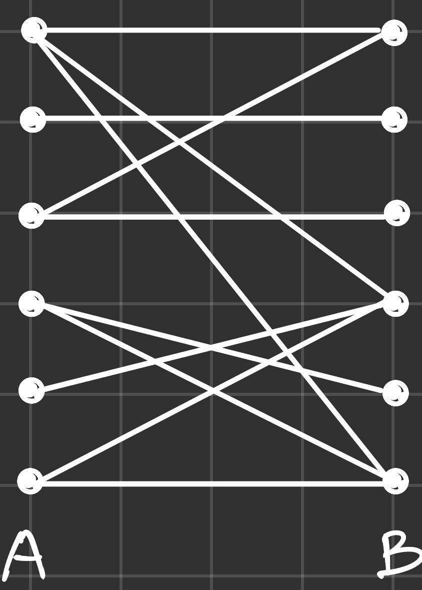
For each edge (x, y) in a bipartite graph, $(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

Property: If graph G is bipartite, it cannot contain odd cycle

Proof: By contradiction



Problem : Given a connected graph G, find if it is a bipartite graph?

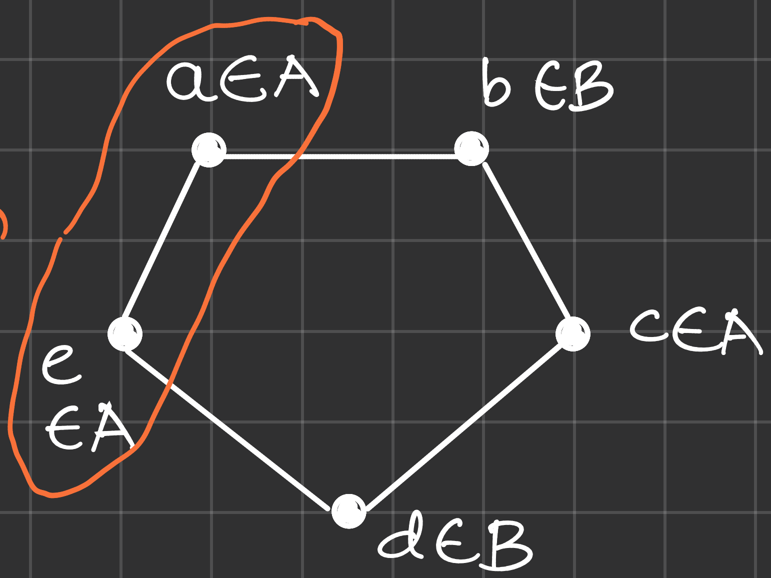


For each edge (x,y) in a bipartite graph, $(x \in A$ and $y \in B)$ or $(x \in B$ and $y \in A)$

Property : If graph G is bipartite, it cannot contain odd cycle

Proof : By contradiction

a contradiction



<u>Property</u> : If graph G is bipartite, it cannot contain odd cycle

Is this property is true?

If a graph has no odd cycle, then it is bipartite.

Property : If graph G is bipartite, it cannot contain odd cycle

Is this property is true?
If a graph has no odd cycle, then it is bipartite.

To show the above property, given G you should partition the vertex set into A & B s.t each edge has one endpoint in A and other endpoint in B.

Property : If graph G is bipartite, it cannot contain odd cycle

Is this property is true?
If a graph has no odd cycle, then it is bipartite.

To show the above property, given G you should partition the vertex set into A & B s.t each edge has one endpoint in A and other endpoint in B.

We will use an algorithm to find this partition

An algorithm is used as the proof.

Property : If graph G is bipartite, it cannot contain odd cycle

Is this property is true ?
If a graph has no odd cycle, then it is bipartite.

To show the above property, given G you should partition the vertex set into A & B s.t each edge has one endpoint in A and other endpoint in B.

We will use an algorithm to find this partition

An algorithm is used as the proof.

BFS algorithm

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

} $\Rightarrow$ G is not bipartite

else

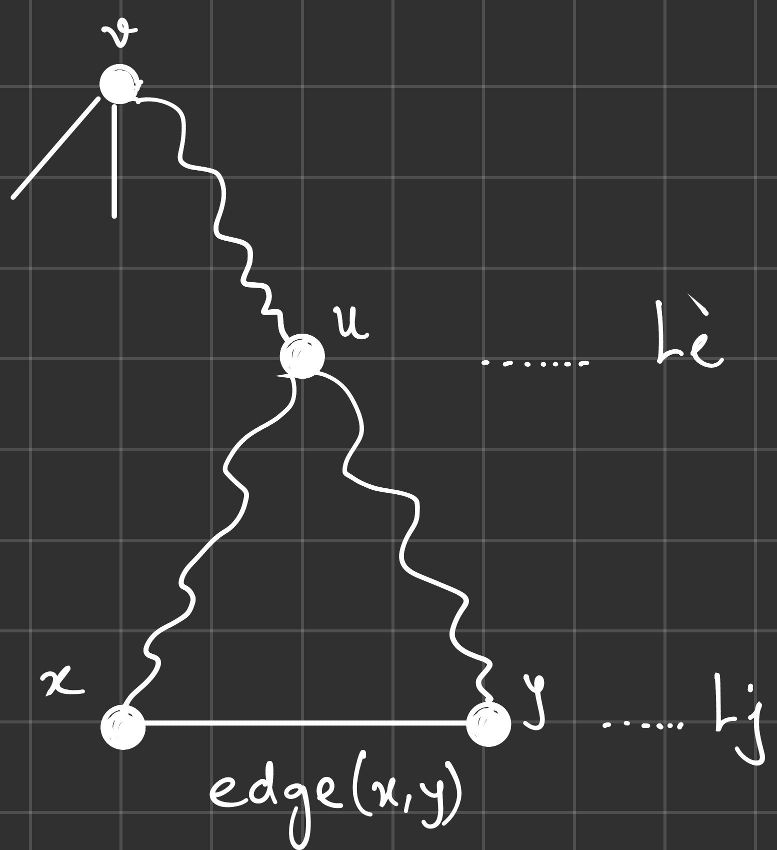
{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}



Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are

{ at same level in the BFS tree

We have found an odd cycle

} $\Rightarrow$ G is not bipartite

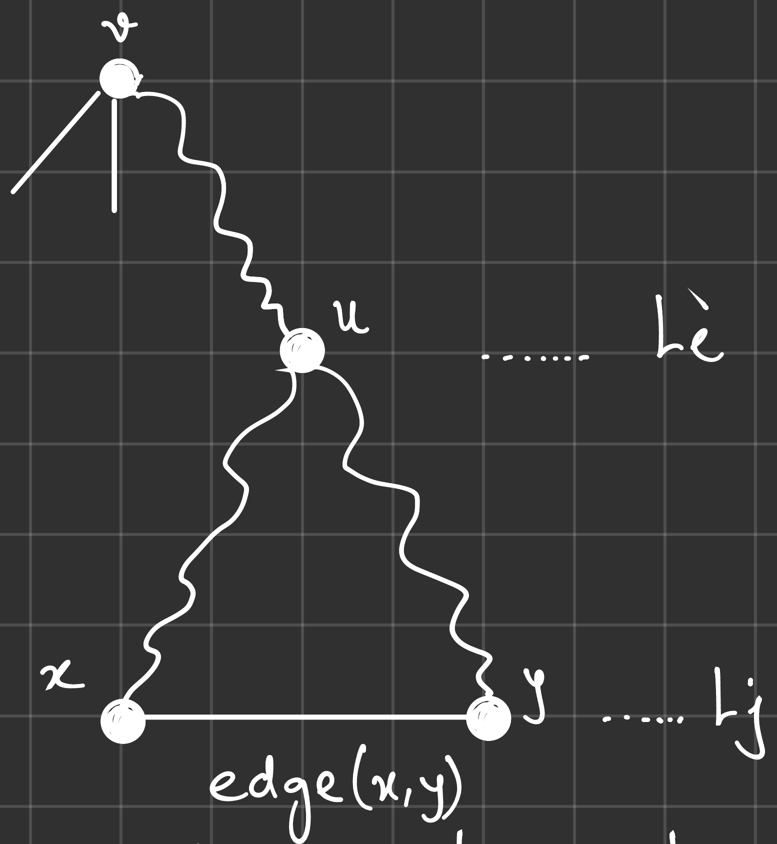
else

{ We have found a bipartite graph

$A = \{ L_0, L_2, L_4, \ldots \}$ even layer vertices

$B = \{ L_1, L_3, L_5, \ldots \}$ odd layer vertices

}



Length of the cycle $= |u \rightsquigarrow y| + 1 + |x \rightsquigarrow u|$

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

} $\Rightarrow$ G is not bipartite

else

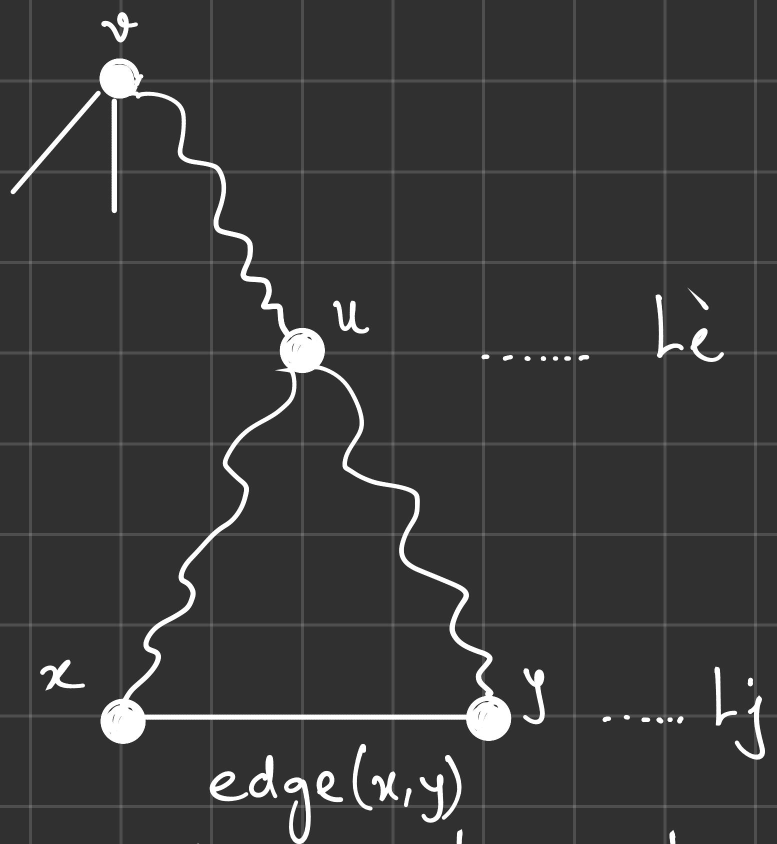
{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \dots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \dots\}$ odd layer vertices

}



$$
\begin{aligned}
\text{Length of the cycle} &= |u \rightsquigarrow y| + 1 + |x \rightsquigarrow u| \\
&= (j-i) + 1 + (j-i) \\
&= 2(j-i) + 1
\end{aligned}
$$

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

} $\Rightarrow$ G is not bipartite

else

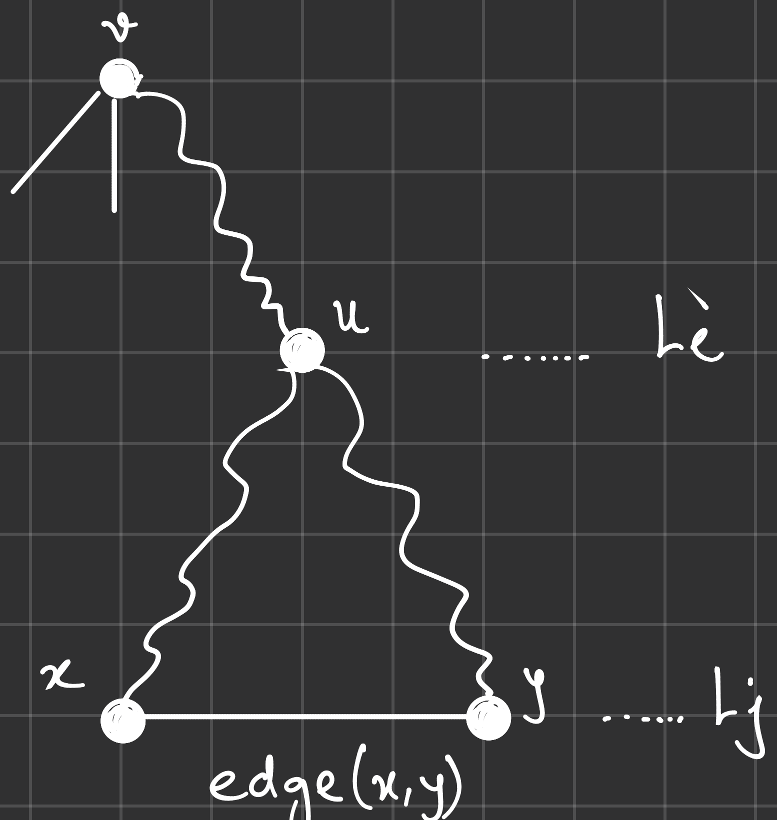
{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}



Length of the cycle $= |u \rightsquigarrow y| + 1 + |x \rightsquigarrow u|$

$= (j-i) + 1 + (j-i)$

$= 2(j-i) + 1$

odd

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

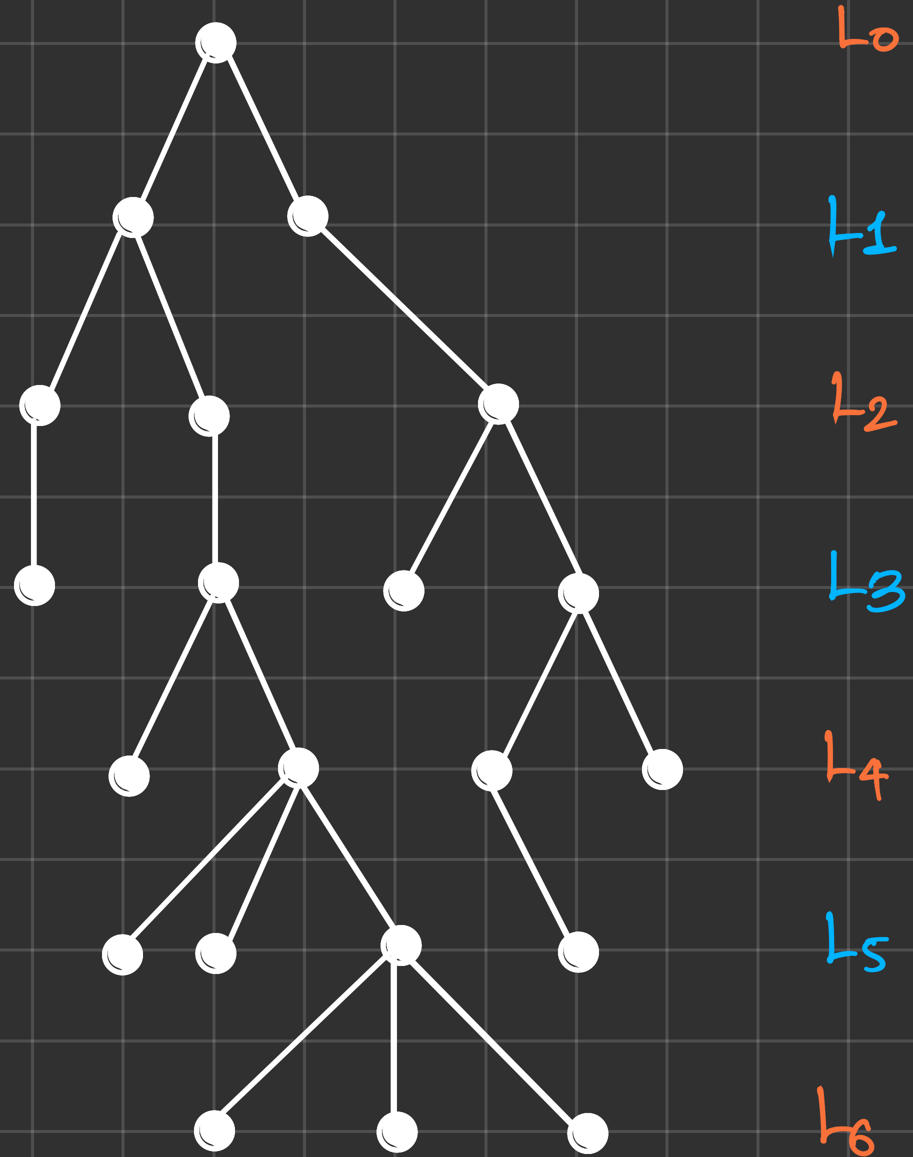
{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \dots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \dots\}$ odd layer vertices

}



Algo

Pick an arbitrary vertex v ;

BFS(v) ;

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

We have found a bipartite graph

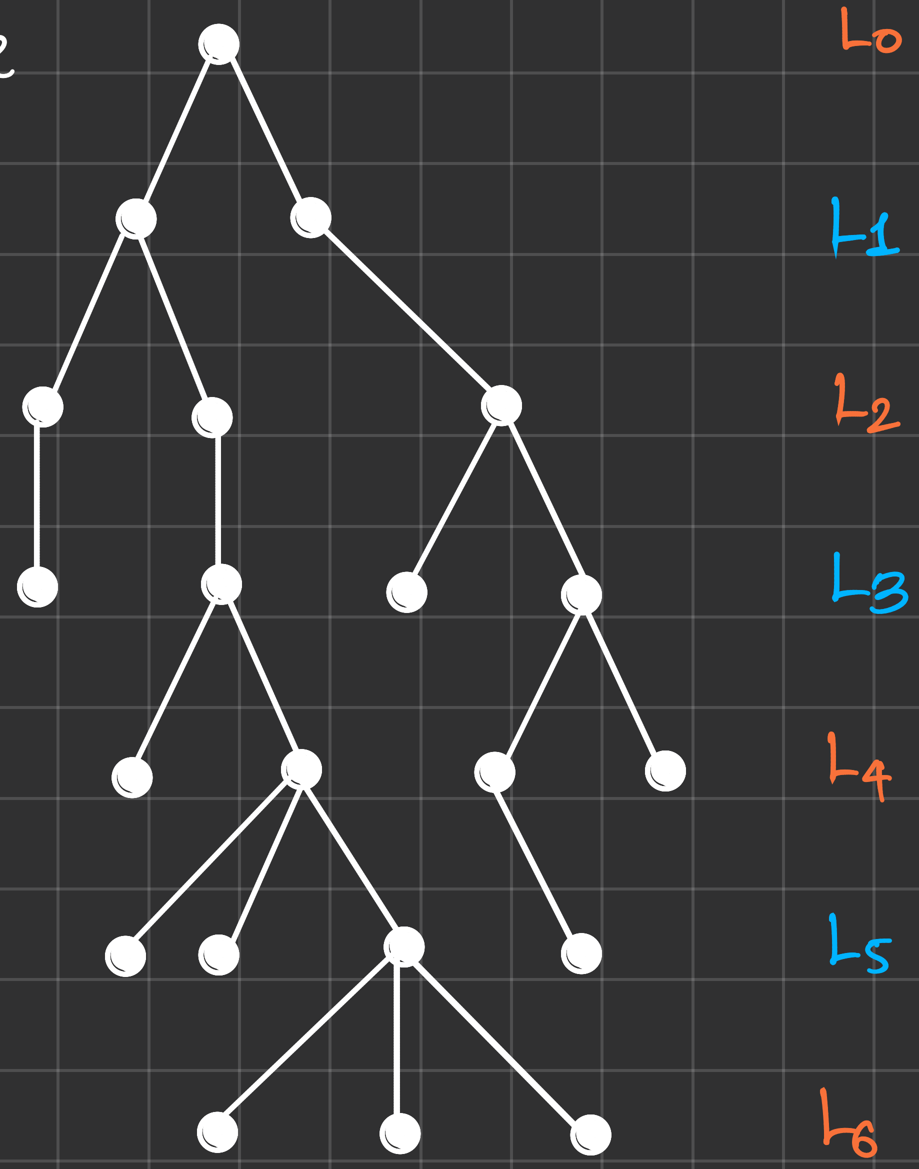
$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or

$x \in B$ & $y \in A$



Algo

Pick an arbitrary vertex v ;

BFS(v) ;

If there exists an edge xy s.t x & y are

{ at same level in the BFS tree

We have found an odd cycle

} $\Rightarrow$ G is not bipartite

else

{ We have found a bipartite graph

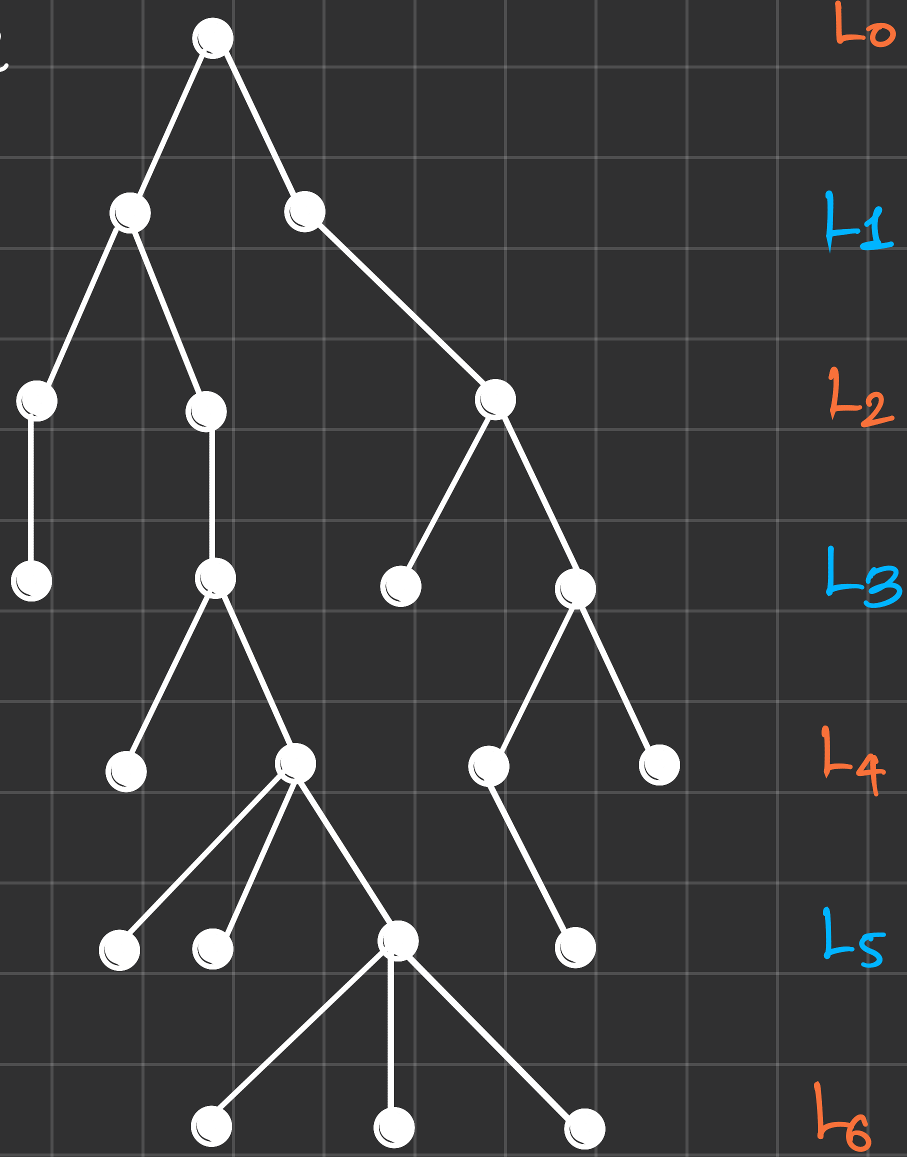
$A = \{L_0, L_2, L_4, \dots\}$ even layer vertices

} $B = \{L_1, L_3, L_5, \dots\}$ odd layer vertices

Show that for each edge (x,y) $x \in A$ & $y \in B$ or

$x \in B$ & $y \in A$

(1) If (x,y) is a tree edge



Algo

Pick an arbitrary vertex v ;

BFS(v) ;

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

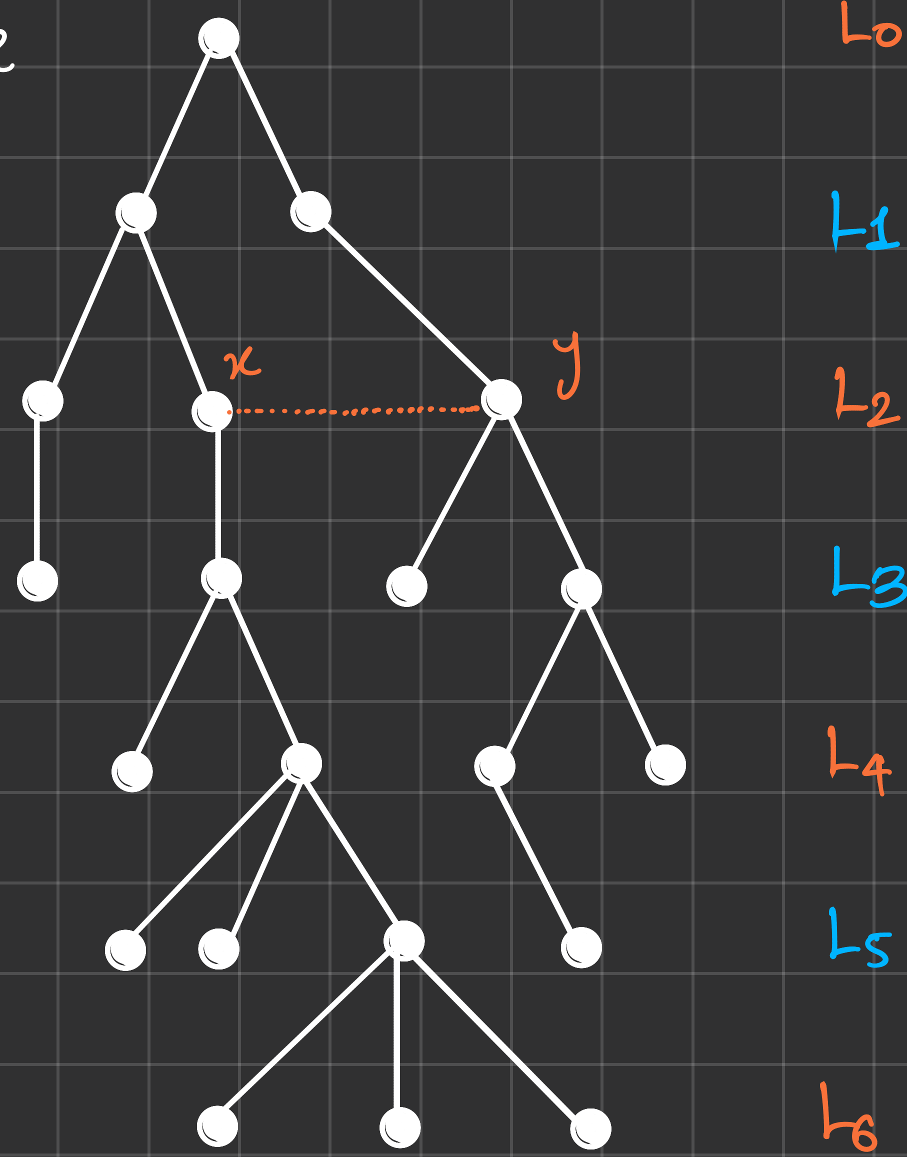
We have found a bipartite graph

$A = \{ L_0, L_2, L_4, \ldots \}$ even layer vertices

$B = \{ L_1, L_3, L_5, \ldots \}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$



(1) If (x,y) is a tree edge
trivially true

(2) If (x,y) is a non-tree edge

(a) edge at same level

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow G$ is not bipartite

}

else

{

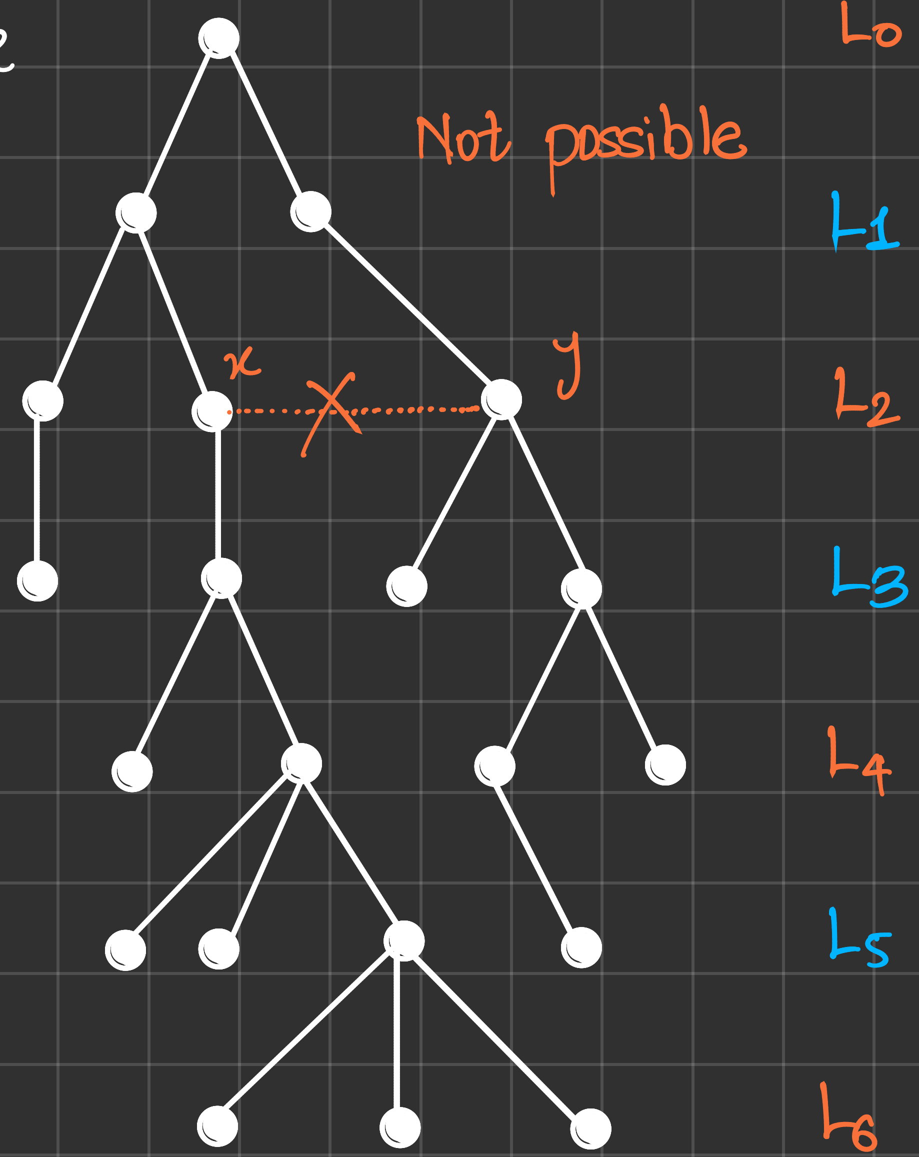
We have found a bipartite graph

$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$



(1) If (x,y) is a tree edge trivially true

(2) If (x,y) is a non-tree edge

~~(a) edge at same level~~

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

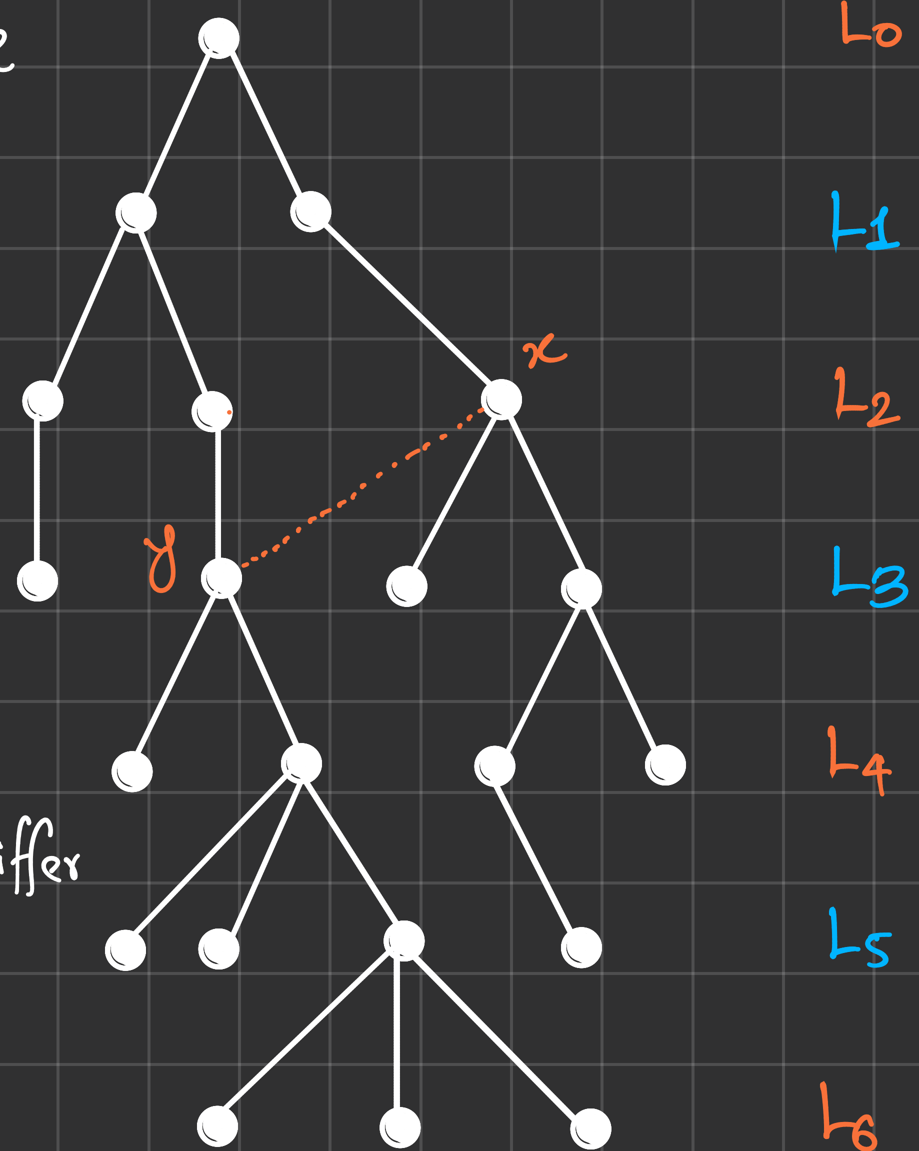
We have found a bipartite graph

$A = \{ L_0, L_2, L_4, \dots \}$ even layer vertices

$B = \{ L_1, L_3, L_5, \dots \}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$



(1) If (x,y) is a tree edge trivially true

(2) If (x,y) is a non-tree edge

~~(a) edge at same level~~

(b) edge whose endpoint differ by 1 level

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

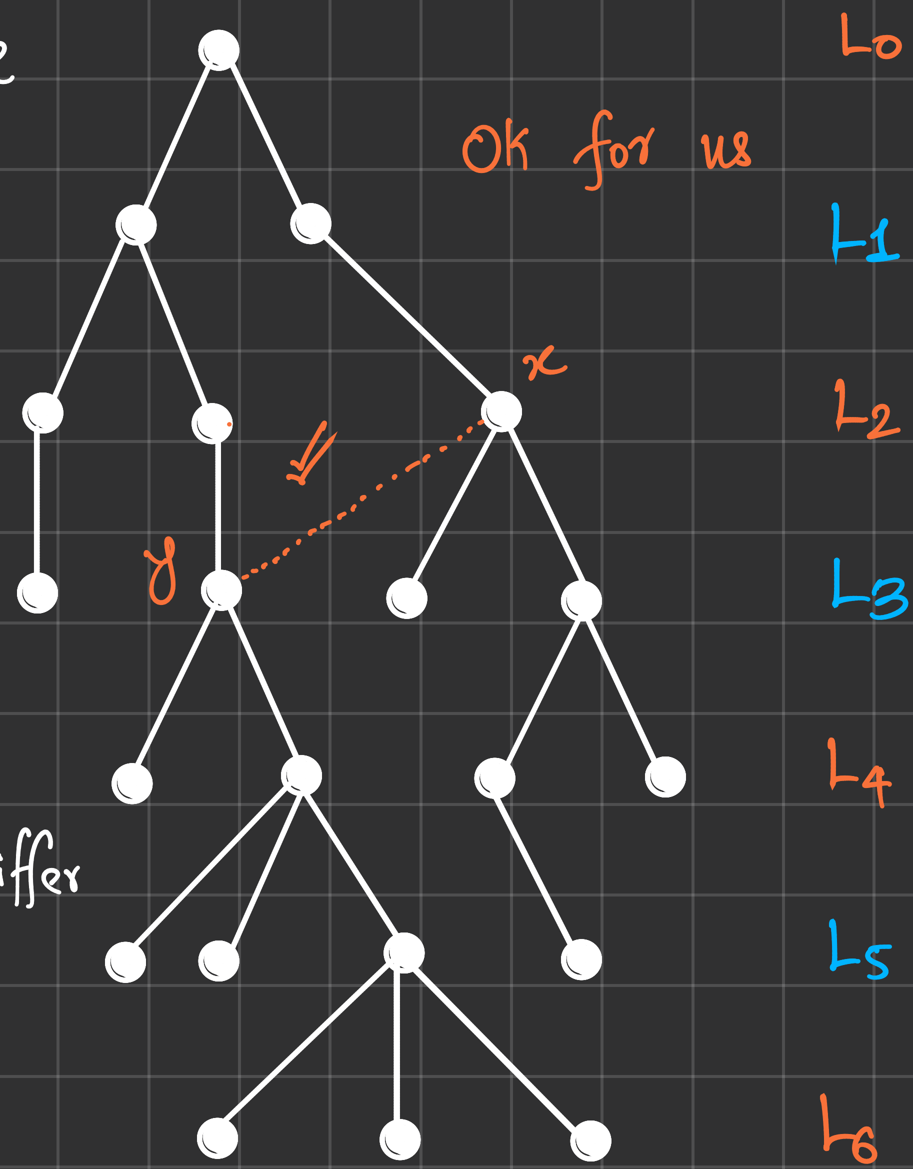
We have found a bipartite graph

$A = \{ L_0, L_2, L_4, \dots \}$ even layer vertices

$B = \{ L_1, L_3, L_5, \dots \}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$



(1) If (x,y) is a tree edge trivially true

(2) If (x,y) is a non-tree edge

~~(a) edge at same level~~

(b) edge whose endpoint differ by 1 level

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow$ G is not bipartite

}

else

{

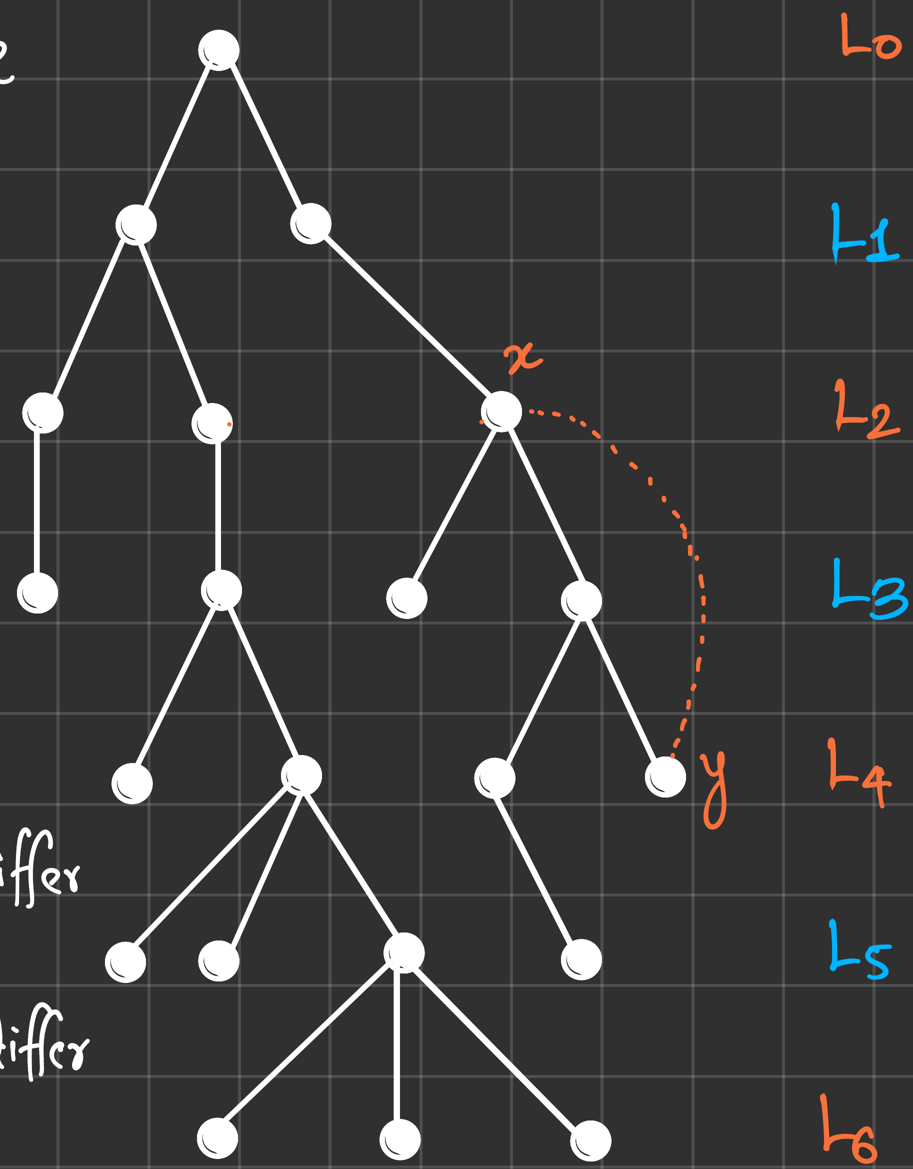
We have found a bipartite graph

$A = \{L_0, L_2, L_4, \ldots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \ldots\}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$



(1) If (x,y) is a tree edge trivially true

(2) If (x,y) is a non-tree edge

~~(a) edge at same level~~

(b) edge whose endpoint differ by 1 level

(c) edge whose endpoint differ by >1 level

Algo

Pick an arbitrary vertex v;

BFS(v);

If there exists an edge xy s.t x & y are at same level in the BFS tree

{

We have found an odd cycle

$\Rightarrow G$ is not bipartite

}

else

{

We have found a bipartite graph

$A = \{L_0, L_2, L_4, \dots\}$ even layer vertices

$B = \{L_1, L_3, L_5, \dots\}$ odd layer vertices

}

Show that for each edge (x,y) $x \in A$ & $y \in B$ or $x \in B$ & $y \in A$

(1) If (x,y) is a tree edge trivially true

(2) If (x,y) is a non-tree edge

~~(a) edge at same level~~

(b) edge whose endpoint differ by 1 level

~~(c) edge whose endpoint differ by >1 level~~