Data Structures and Algorithms

(ESO207)

Lecture 26

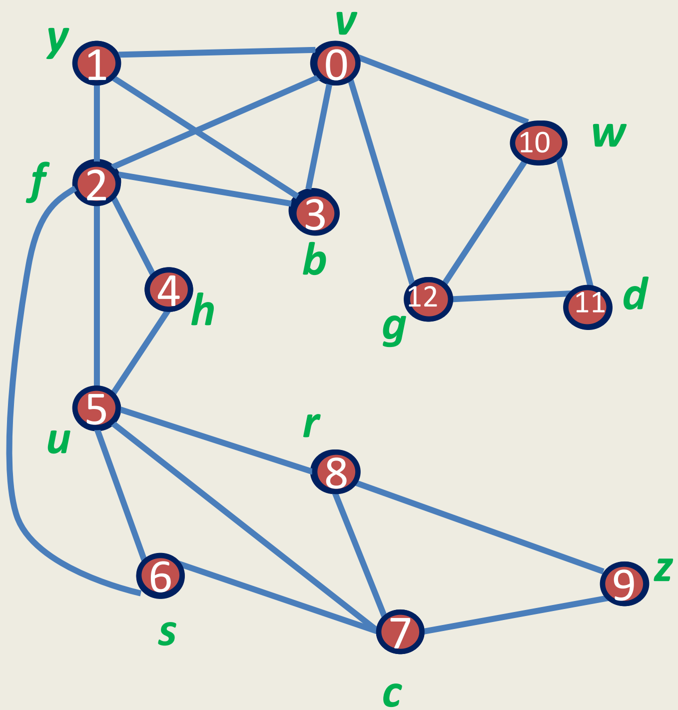
- Depth First Search (DFS) Traversal
- DFS Tree
- Novel application: computing biconnected components of a graph

DFS traversal of *G*

```
DFS(v)
{  Visited(v) ← true;  DFN[v] ← dfn ++;
   For each neighbor w of v
   {        if (Visited(w)  = false)
            {   DFS(w) ;
                 ........;
            }
            ........;
   }
}
```

```
DFS-traversal(G)
{  dfn ← 0;
   For each vertex vϵ V  {      Visited(v)← false                 }
   For each vertex v ϵ V {      If (Visited(v ) = false)    DFS(v)   }
}
```

DFN number

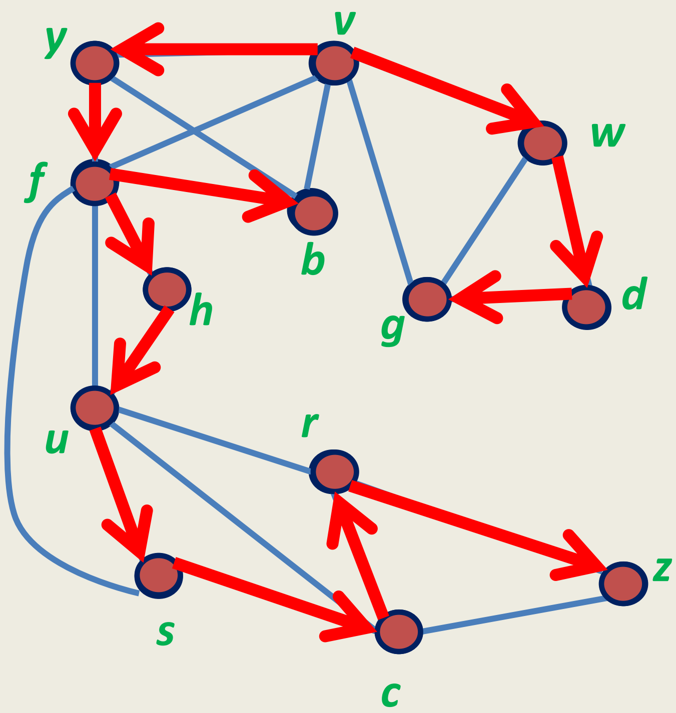


DFN[***x***] :

The number at which ***x*** gets visited during DFS traversal.

DFS tree

DFS(v) computes a tree rooted at v

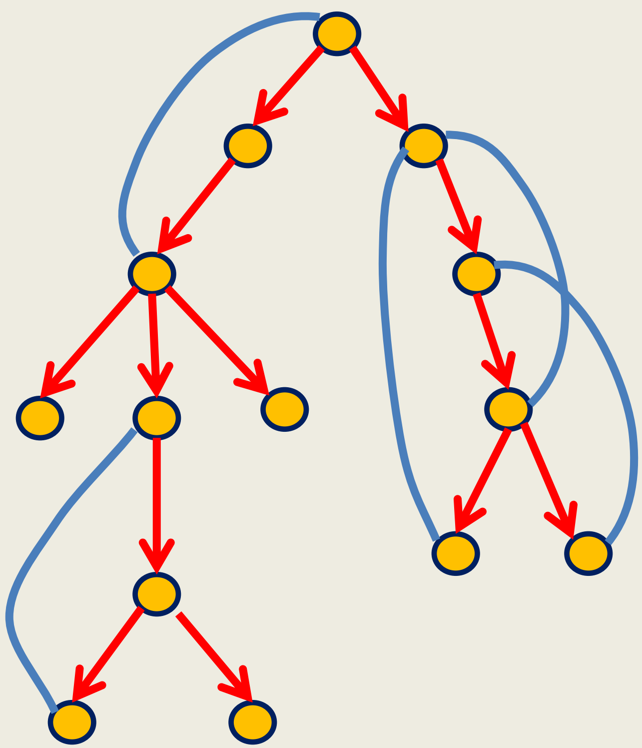


A **DFS** tree rooted at ***v***

Can any rooted tree be obtained through DFS ?

No

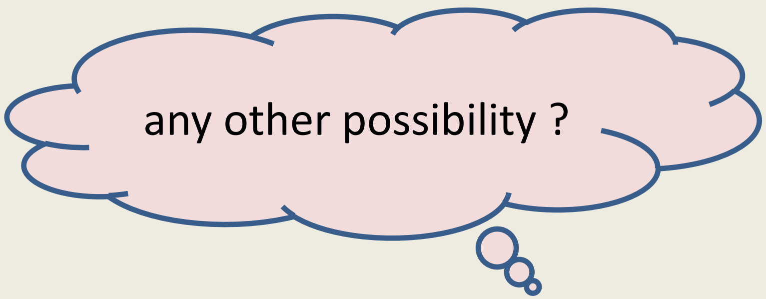
How will an edge appear in DFS traversal ?



- as a **tree-edge**.

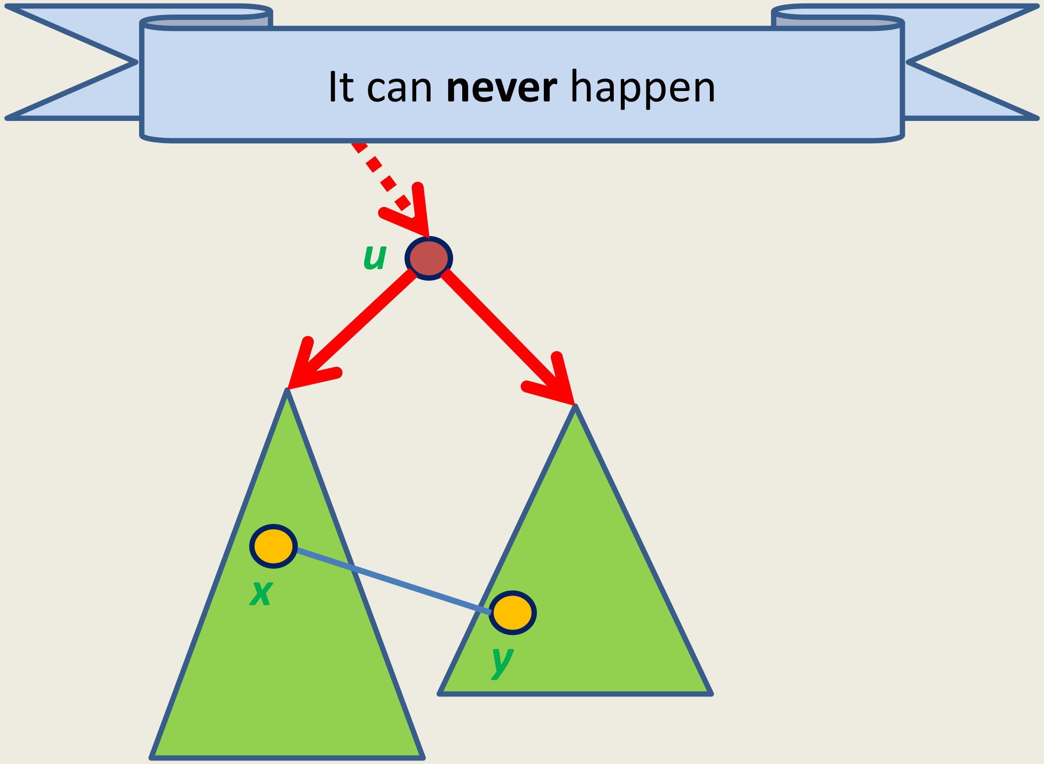
If the edge is a **non-tree** edge :

- Edge between **ancestor** and **descendant** in **DFS** tree.



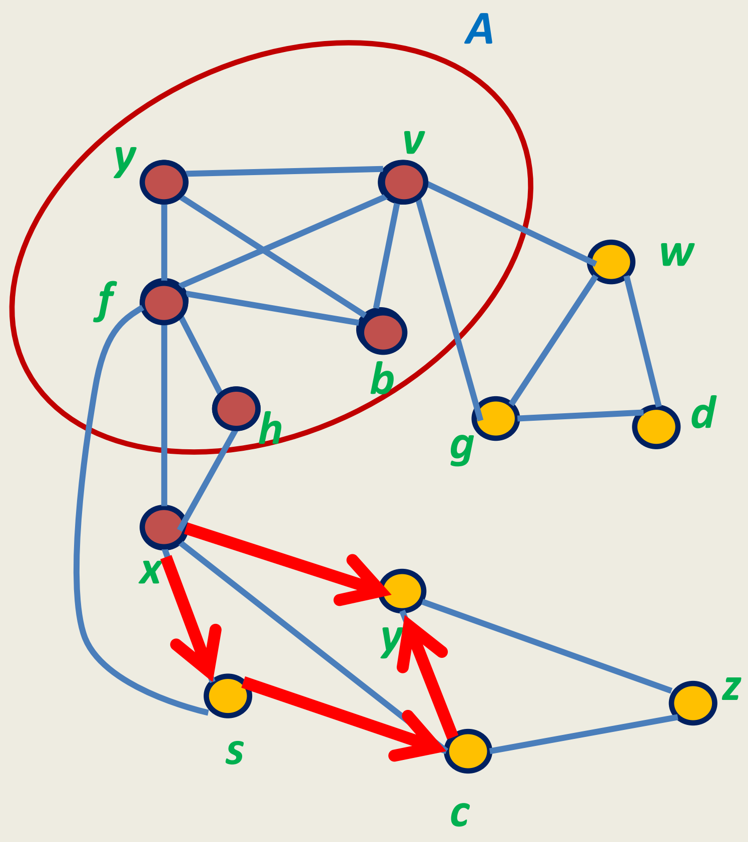
No

How will an edge appear in DFS traversal ?



How will an edge appear in DFS traversal ?

How will an edge appear in DFS traversal ?



How will an edge appear in DFS traversal ?

A short proof:

Let (***x***,***y***) be a non-tree edge.

Let ***x*** get visited before ***y***.

Question:

If we remove all vertices visited prior to ***x***, does ***y*** still lie in the connected component of ***x*** ?

Answer: yes.

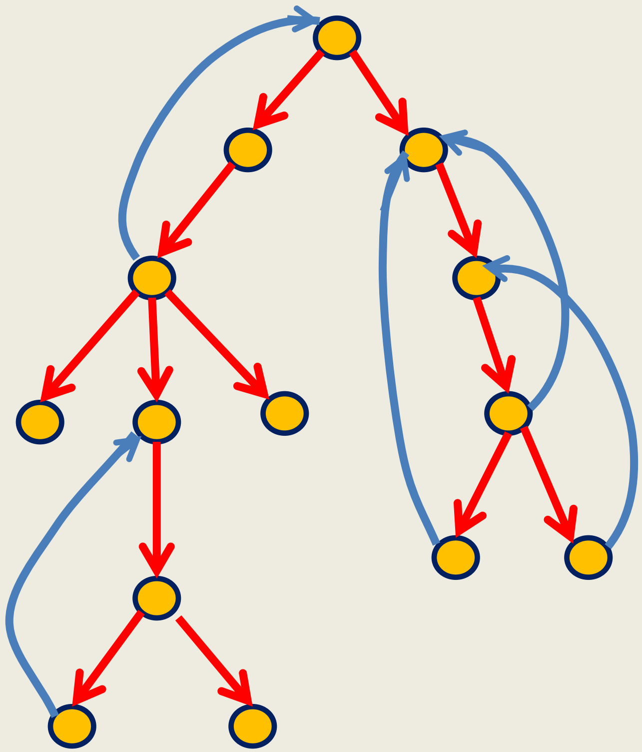
➔

DFS pursued from ***x*** will have a path to ***y*** in **DFS** tree.

Hence ***x*** must be ancestor of ***y*** in the **DFS** tree.

Always remember

the following picture for DFS traversal



non-tree edge ➔ **back** edge

This is called **DFS representation of the graph**.

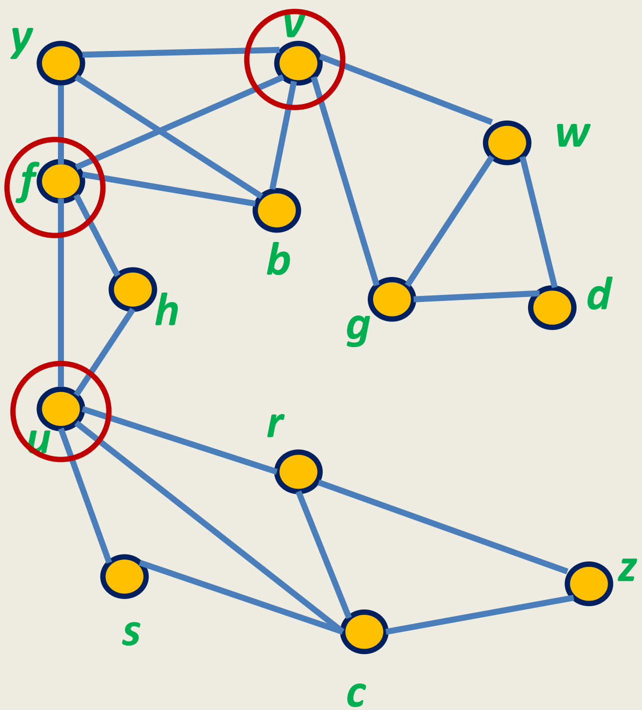
It plays a key role in the design of every efficient algorithm.

A novel application of DFS traversal

Determining if a graph **G** is **biconnected**

Definition: A connected graph is said to be **biconnected** if there does not exist any vertex whose removal disconnects the graph.

Motivation: To design **robust** networks
(immune to any single node failure).



Is this graph **biconnected ?**

No.

A trivial algorithms for checking **bi-connectedness** of a graph

- For each vertex ***v***, determine if **G\{*v*}** is connected

  (One may use either **BFS** or **DFS** traversal here)

Time complexity of the trivial algorithm : O(mn)

An **O**$(\boldsymbol{m} + \boldsymbol{n})$ time algorithm

A <u>single</u> **DFS** traversal

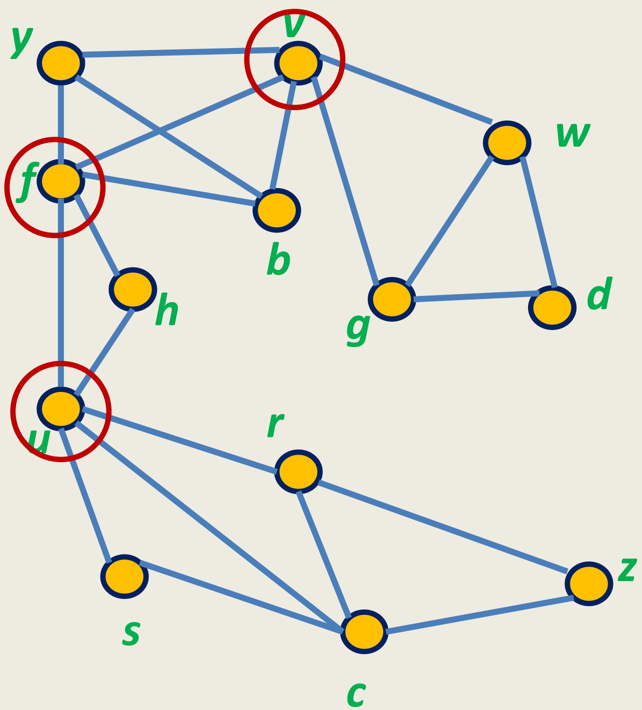
An **O**$(m + n)$ time algorithm

- A formal **characterization** of the problem.

  (**articulation points**)

- Exploring **relationship** between articulation point & DFS tree.

- Using the relation **cleverly** to design an efficient algorithm.



This graph is NOT biconnected

The removal of any of {***v***,***f***,***u***} can destroy connectivity.

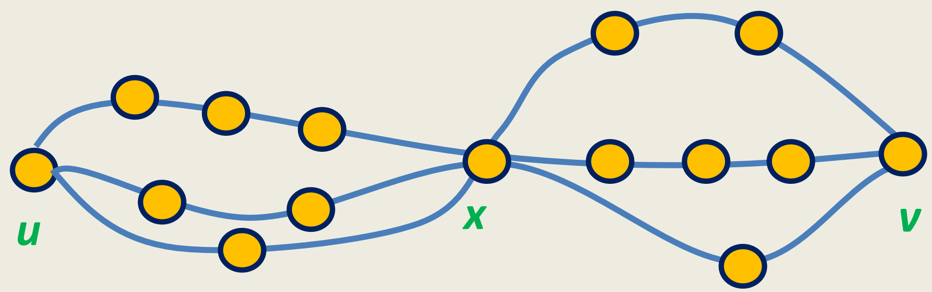
v,***f***,***u*** are called the **articulation points** of ***G***.

A formal definition of articulaton point

Definition: A vertex x is said to be **articulation point** if

$\exists$ u,v different from x

such that every path between u and v passes through x.



Observation: A graph is biconnected if none of its vertices is an articulation point.

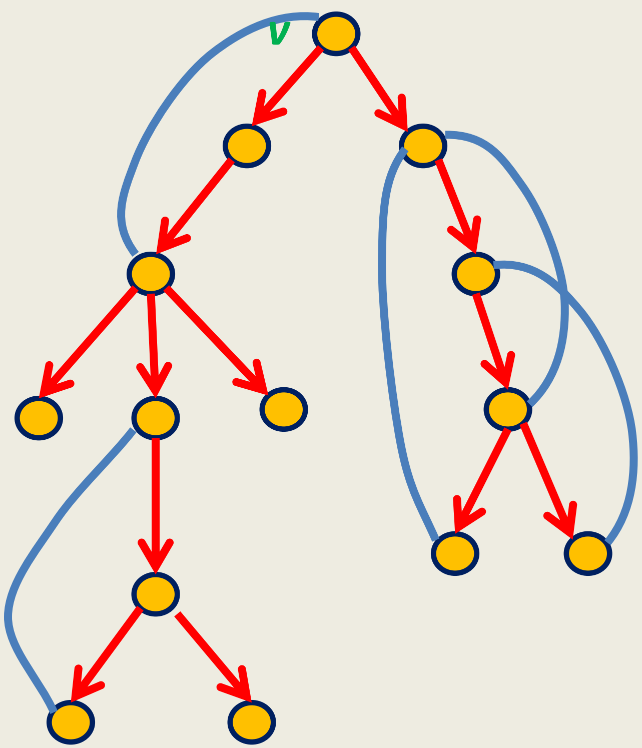
AIM:

Design an **algorithm** to compute all **articulation points** in a given graph.

Articulation points and DFS traversal

Don't Focus on the graph. Instead,
focus on the **DFS tree representation** of the graph.

Some observations

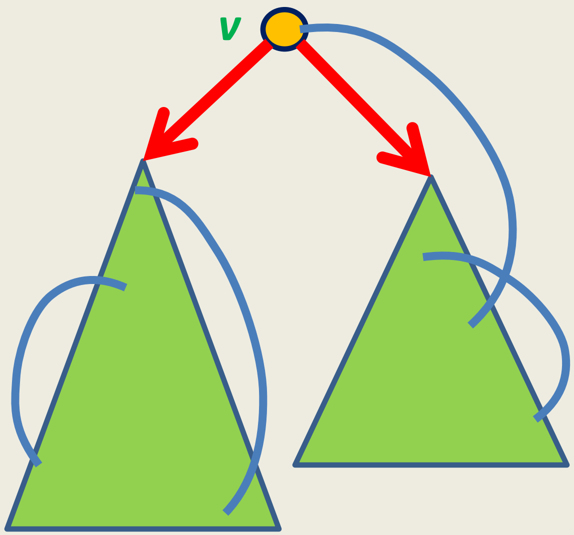


Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Some observations

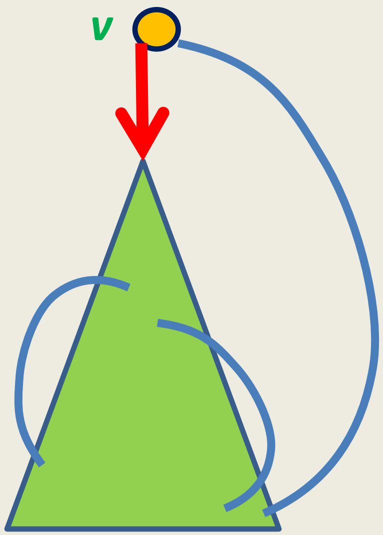


Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Some observations



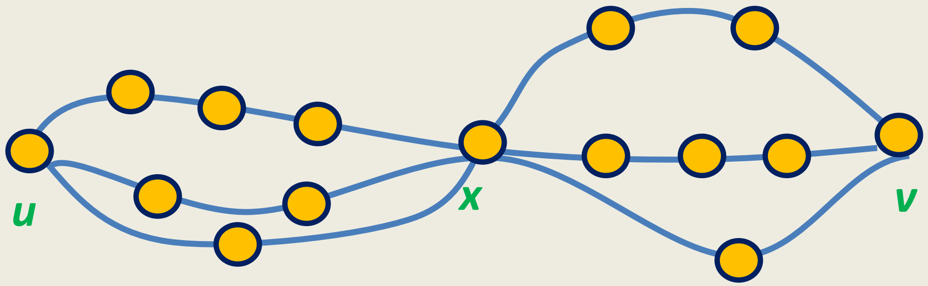
Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Answer: Iff it has <u>**two or more**</u> children.

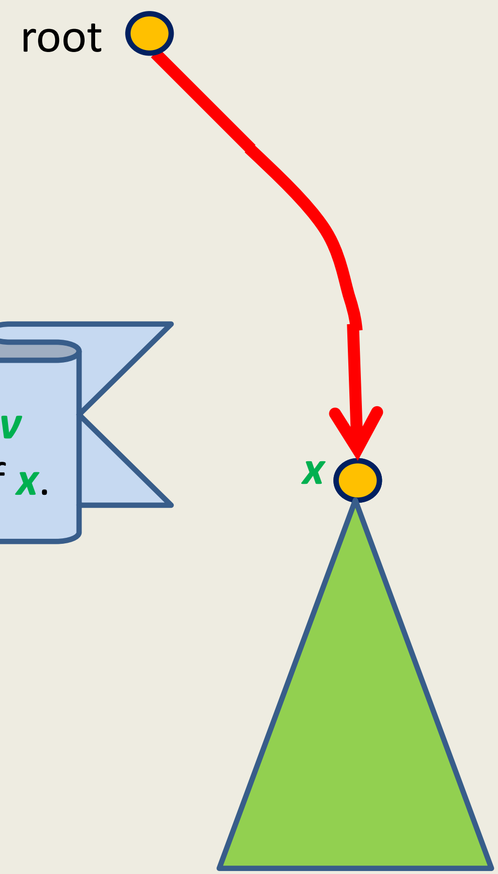
Some observations



AIM:

To find **necessary** and **sufficient conditions** for an **internal node** to be **articulation point**.

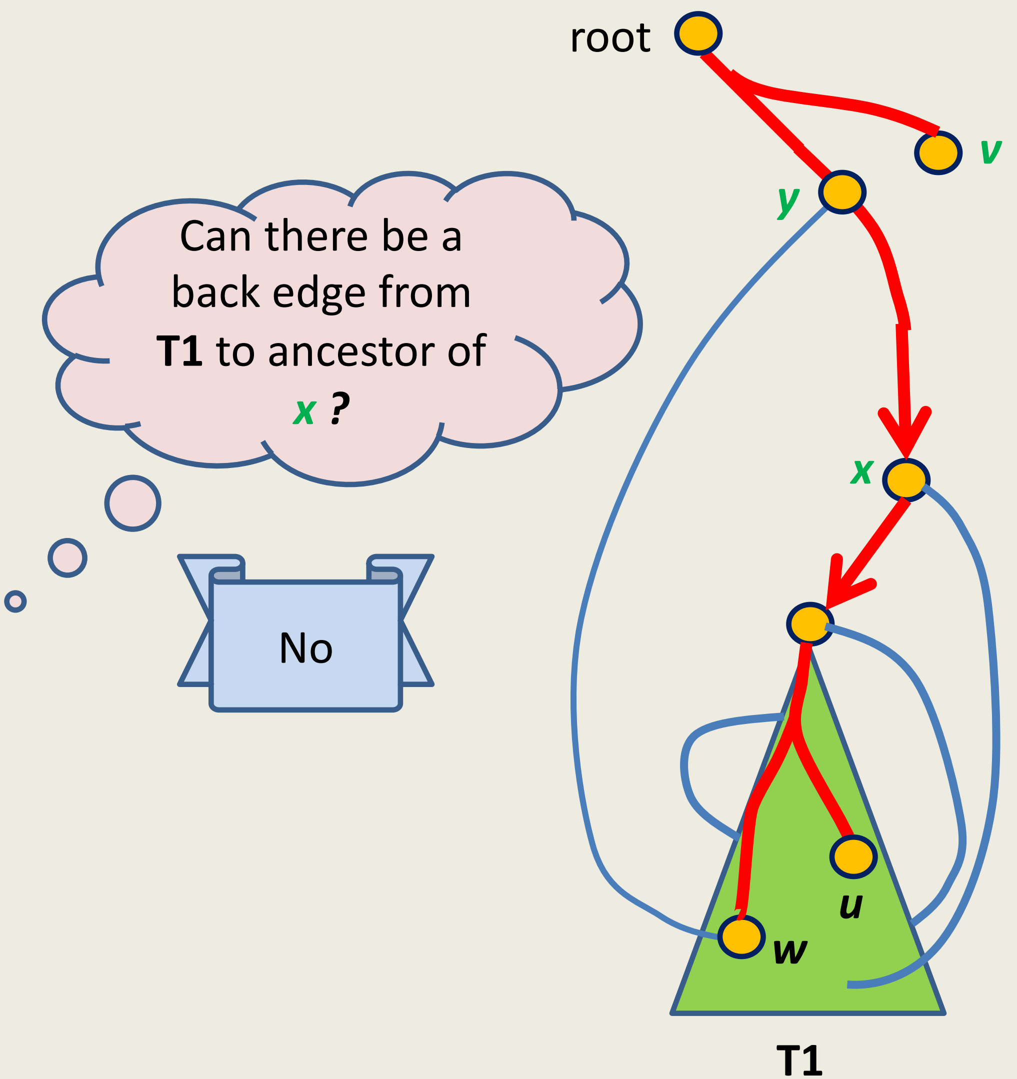
conditions for an **internal node** to be **articulation point**.



At least one of ***u*** and ***v*** must be **descendant** of ***x***.

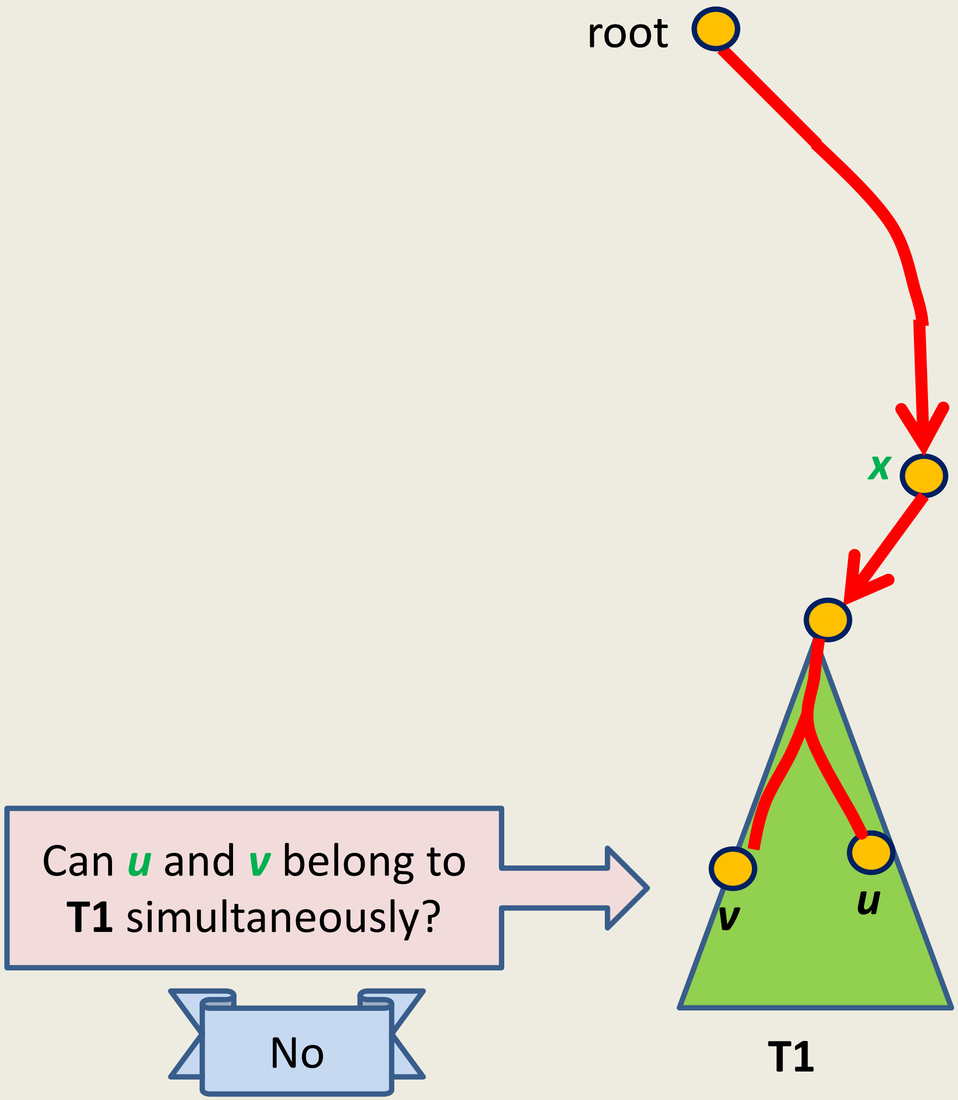
Where will ***u*** and ***v*** be relative to ***x*** ?

Case 1: Exactly one of u and v is a descendant of x in DFS tree

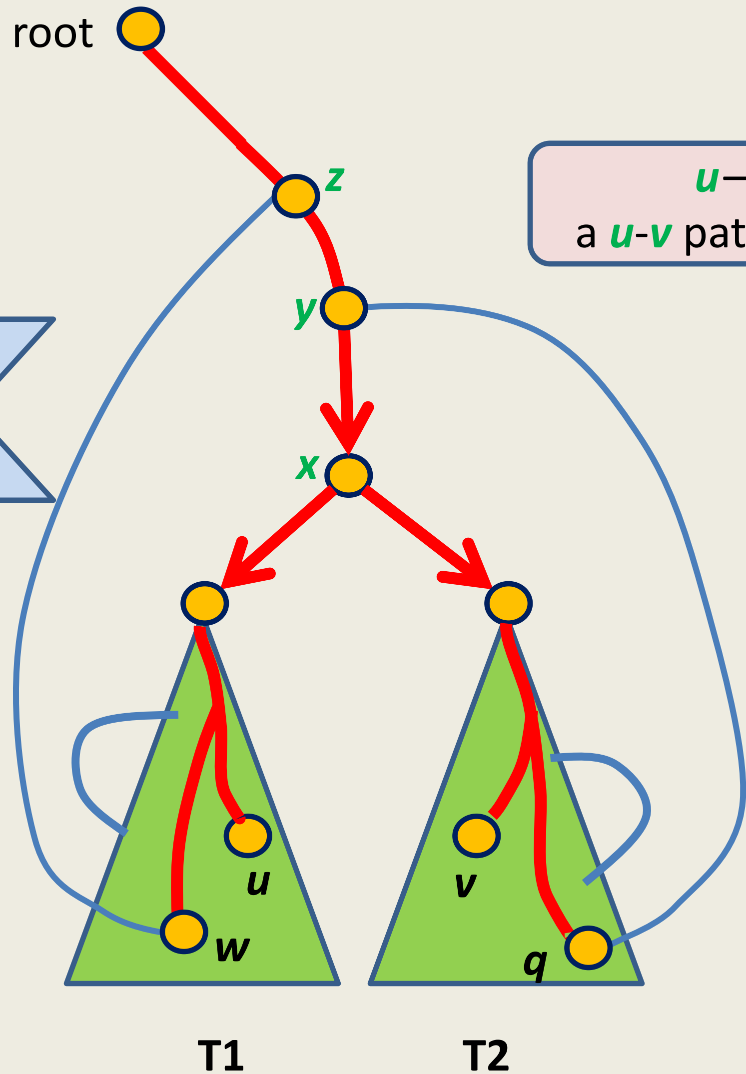


u—*w*—*y*—*v* :
a *u*-*v* path not passing through ***x***

Case 2: both u and v are descendants of x in DFS tree



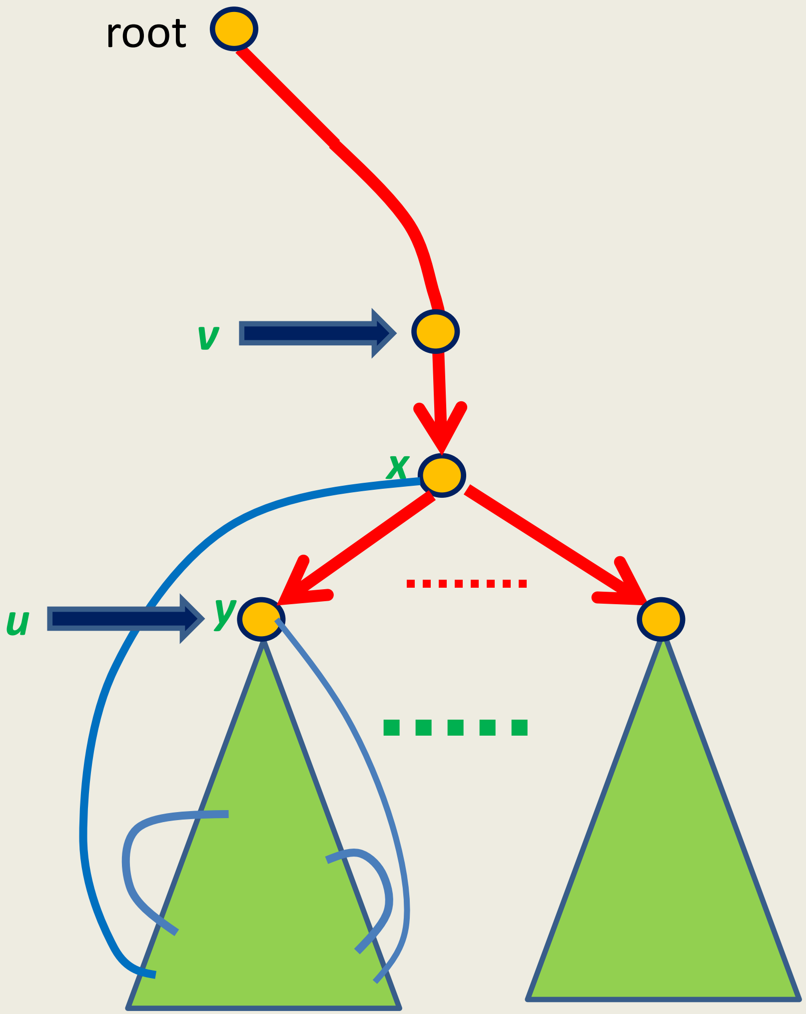
Case 2: both u and v are descendants of x in DFS tree



u—*w*—*z*—*y*—*q*—*v* :
a *u*-*v* path not passing through *x*

At least one of **T1** and **T2** have no back edge to ancestor of *x*

Necessary condition for *x* to be articulation point

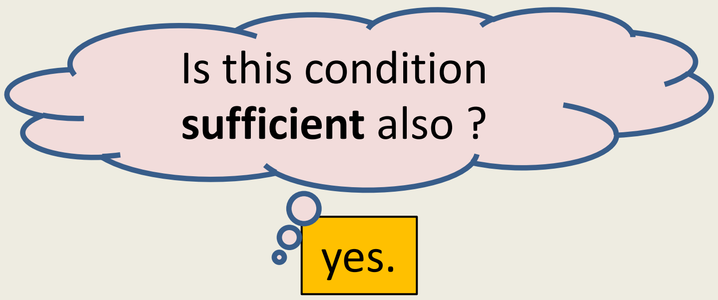


Necessary condition:

x has **at least** one child *y* s.t.

there is **no** back edge

from **subtree**(y) to **ancestor** of *x*.



Articulation points and DFS

Let G=(V,E) be a connected graph.

Perform **DFS** traversal from any graph and get a DFS tree T.

- No leaf of T is an **articulation point**.
- root of T is an **articulation point** if and only if it has more than one child.
- For any internal node … ??

Theorem1 : An internal node x is **articulation point** **if and only if**
it has a child y such that
there is **no** back edge
from **subtree**(y) to any ancestor of x.

Efficient algorithm for Articulation points

Use Theorem 1

Exploit recursive nature of DFS

Ponder over it before next lecture ☺