# Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

**Lecture 26**

- **Quick revision of Depth First Search (DFS) Traversal**
- **An O($m + n$):algorithm for biconnected components of a graph**
- **Quick Sort: Average time complexity analysis**

# Quick revision of
# Depth First Search (DFS) Traversal

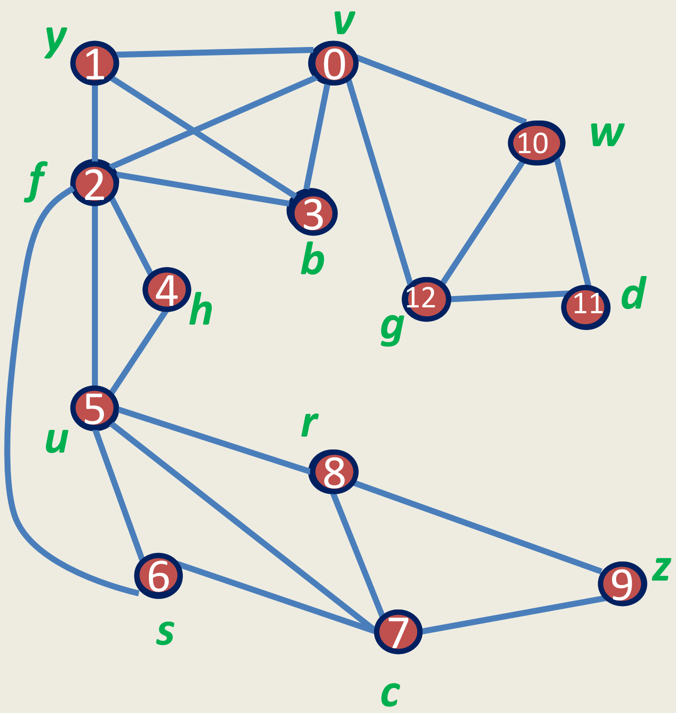
# DFS traversal of *G*

```
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)   }
}
```

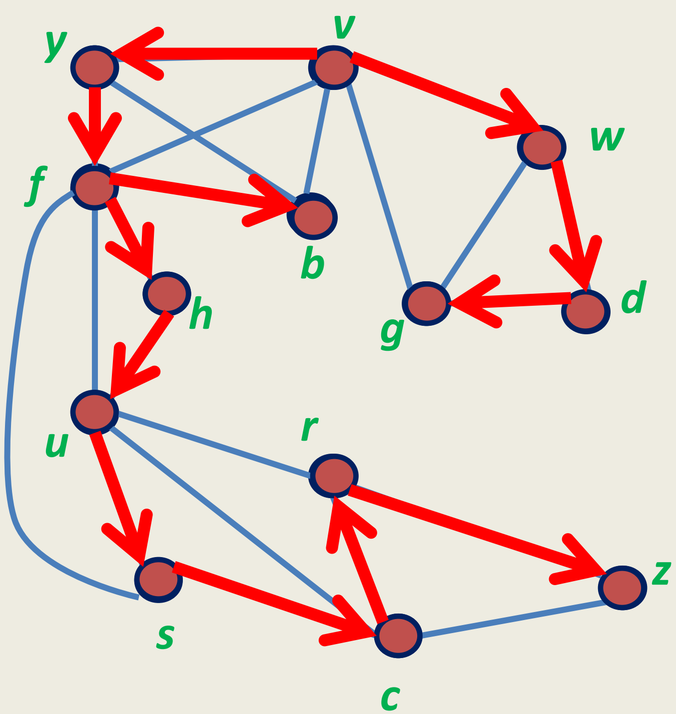
# DFN number




DFN[*x*] :

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

# DFS(v) computes a tree rooted at v




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

If **x** is ancestor of **y** then

DFN[*x*] < DFN[*y*]

**Question**: Is a **DFS** tree unique ?

**Answer**: No.

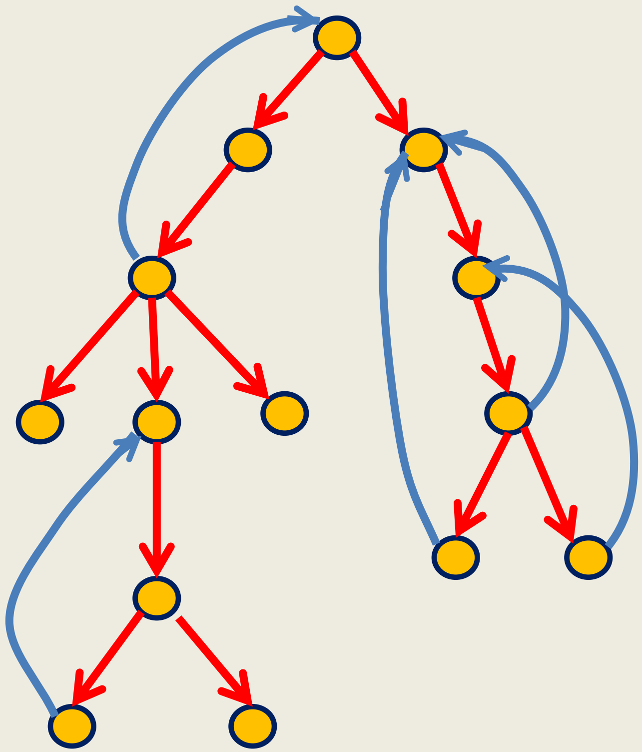
**Question**:

Can any rooted tree be obtained through DFS ?

**Answer**: No.

# Always remember

**Instead of looking at a graph,**
**look at its picture in terms of any of its DFS traversal**



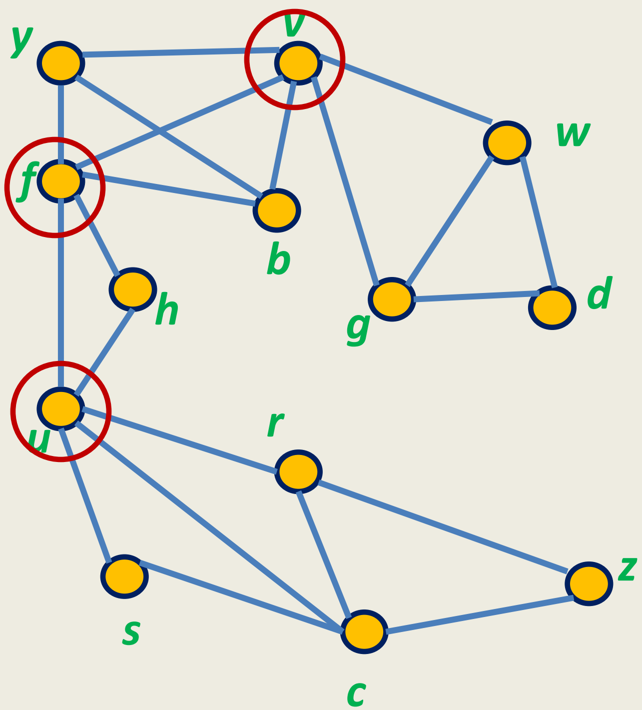
non-tree edge ➔ **back** edge

# 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 exit 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

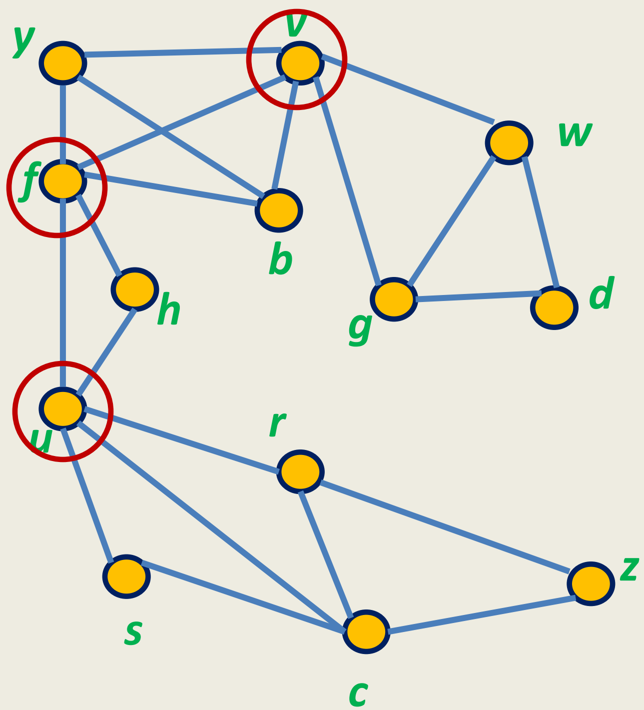
# An O$(m+n)$ time algorithm

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

  (**articulation points**)

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

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



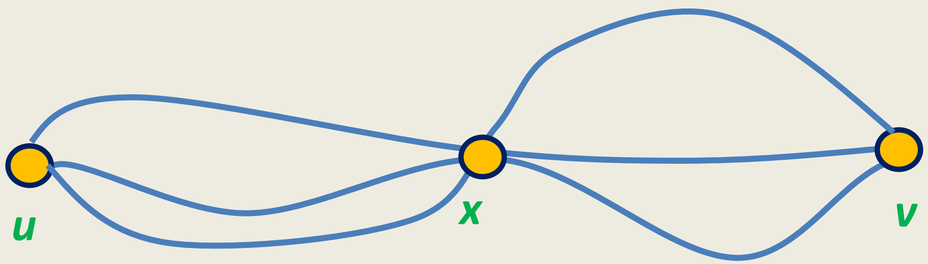

This graph is NOT biconnected

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 there exist two distinct vertices ***u*** and ***v*** such that every path between ***u*** and ***v*** passes through ***x***.

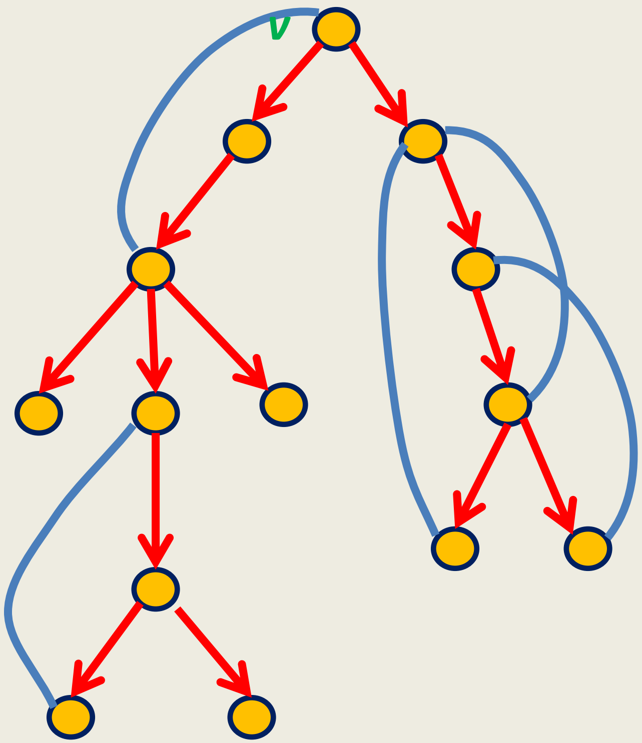



**Observation:** A graph is biconnected if none of its vertices is an articulation point.

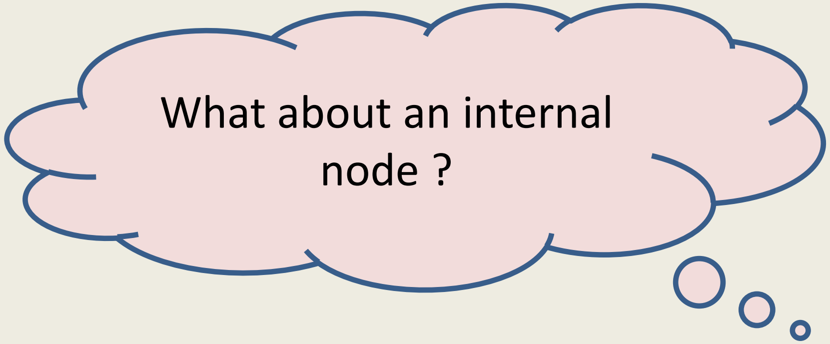
**AIM:**

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

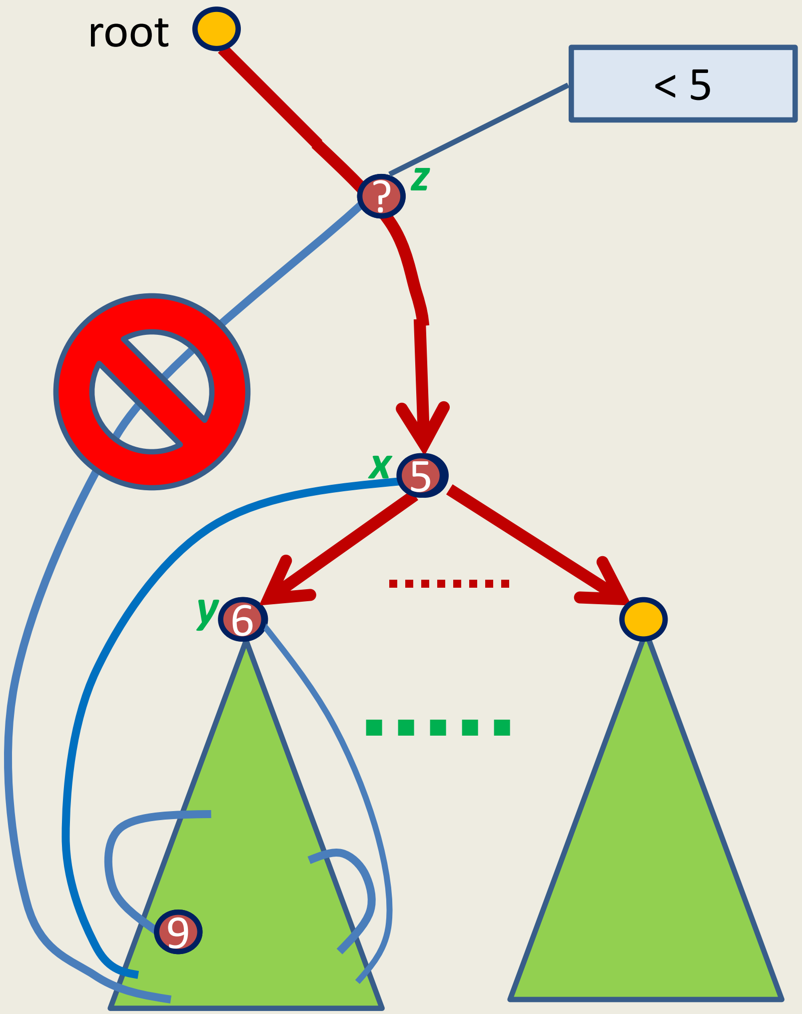
# Some observations




- **A leaf node** can never be an **a.p.** ?
- **Root** is an **a.p.** iff it has two or more children.



# **Necessary** and **Sufficient** condition for *x* to be articulation point




**Theorem1**:

An internal node ***x*** is **articulation point** iff

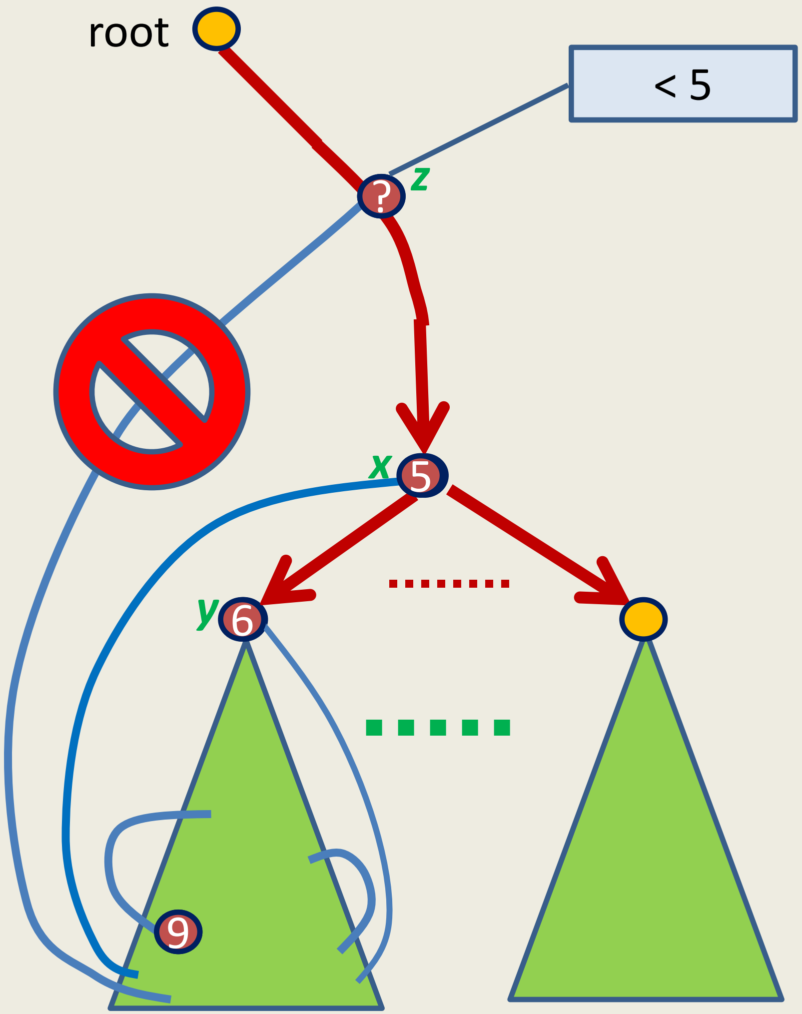
***x*** has **<u>at least</u>** one child ***y*** s.t. there is **no** back edge from **subtree**(y) to **ancestor** of ***x***.

➔ No back edge from **subtree**(y) going to a vertex "**higher**" than ***x***.

How to define the notion "**higher**" than ***x*** ?

Use **DFN** numbering

# Necessary and Sufficient condition for *x* to be articulation point




**Theorem1**:

An internal node ***x*** is **articulation point** iff

***x*** has **at least** one child ***y*** s.t. there is **no** back edge from **subtree**(**y**) to **ancestor** of ***x***.

Invent a new function

**High_pt**(***v***):

**DFN** of the *highest ancestor* of ***v*** to which there is a back edge from **subtree**(***v***).

**Theorem2**:

An internal node ***x*** is **articulation point iff** it has a child, say **y**, in **DFS** tree such that

**High_pt**(**y**) ≥ **DFN**(***x***).

**Theorem2**:

An internal node ***x*** is **articulation point** **iff** it has a child, say **y**, in **DFS** tree such that

**High_pt**(**y**) ≥ **DFN**(***x***).

Good ☺!
But how to transform this Theorem
into an efficient algorithm for
articulation points ?

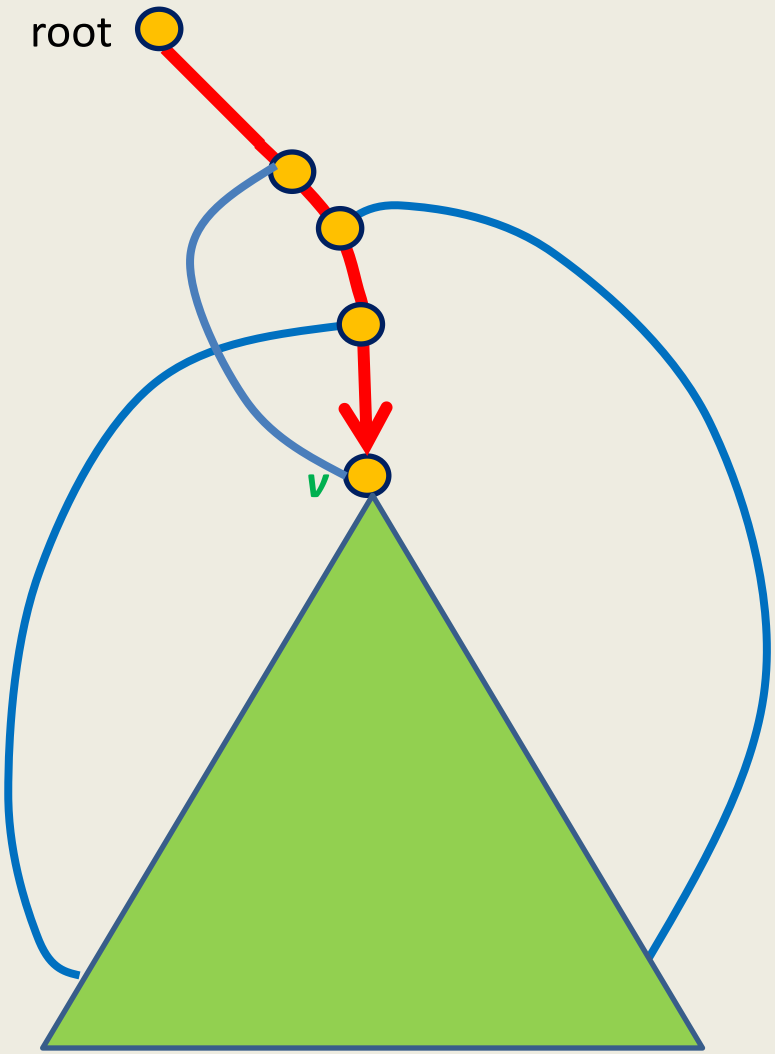
In order to compute **High_pt**(**v**) of a vertex **v**,

we have to traverse the adjacency lists of all vertices of subtree ***T***(**v**).

➔ **O**($m$) time in the worst case to compute **High_pt**(**v**) of a vertex **v**.

➔ **O**($mn$) time algorithm ☹

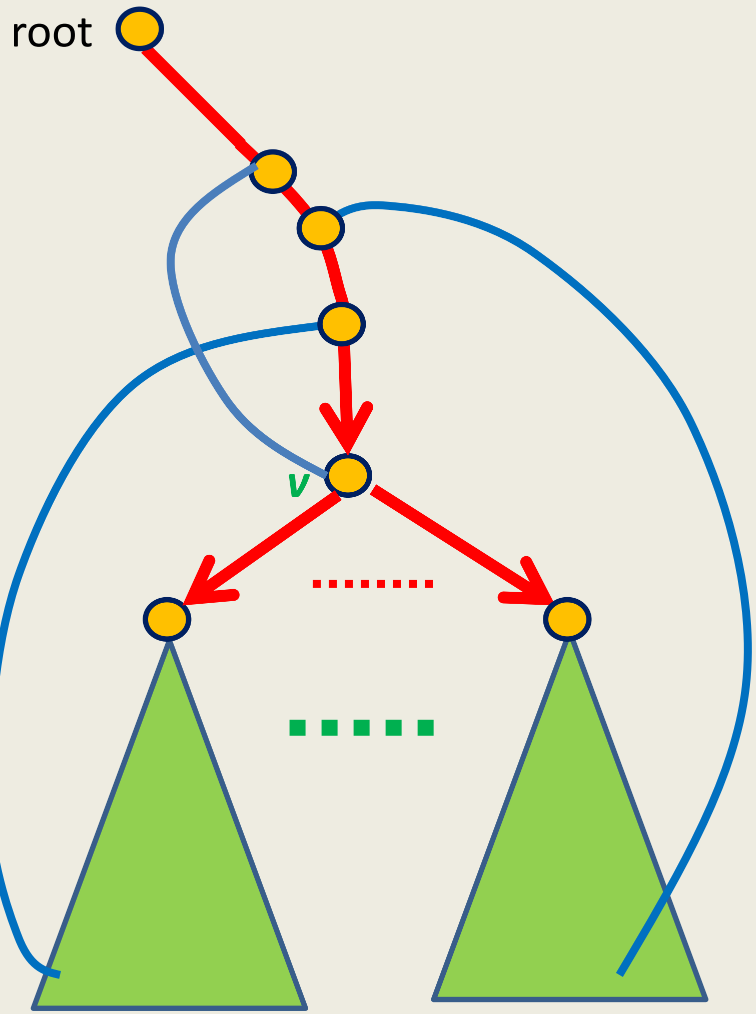
# How to compute High_pt(v) efficiently ?




**Question:** Can we express **High_pt(v)** in terms of its **children** and **proper ancestors**?

Exploit **recursive structure** of DFS tree.

# How to compute High_pt(v) efficiently ?




**Question:** Can we express **High_pt**(*v*) in terms of its **children** and **proper ancestors**?

**High_pt**(*v*) =

$$\min_{(v,w) \in E} \begin{cases} \text{High\_pt}(w) & \text{If } w = \text{child}(v) \\ \text{DFN}(w) & \text{If } w = \text{proper ancestor of } v \end{cases}$$

# The novel algorithm

The algorithm will output an array **AP[]** such that **AP**[***v***]= **true** if and only if ***v*** is an articulation point.

# Algorithm for articulation points in a graph *G*

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

                 ……..;

                 ……..;
            }

            ……..;
   }
}
```

---

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

# Algorithm for articulation points in a graph *G*

```
DFS(v)
{  Visited(v) ← true; DFN[v] ← dfn ++;  High_pt[v]←∞ ;
   For each neighbor w of v
   {        if (Visited(w)  = false)
            {    Parent(w)← v;   DFS(w);
                 High_pt(v) ← min(High_pt(v), High_pt(w));
                 If  High_pt(w) ≥ DFN[v]   AP[v] ← true
            }
         Else if (Parent(v) ≠  w)
                          High_pt(v) ← min(DFN(w), High_pt(v))
   }
}
```

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

# Conclusion

**Theorem2 :** For a given graph $G$=($V$,$E$), all **articulation points** can be computed in **O**($m + n$) time.

# QuickSort

**Average time complexity = O($n \log n$)**

# Pseudocode for QuickSort($S$)

```
QuickSort(S)
{       If (|S|>1)
                Pick and remove an element x from S;
                (S_<x, S_>x)← Partition(S,x);
                return( Concatenate(QuickSort(S_<x), x, QuickSort(S_>x))
}
```

# Pseudocode for QuickSort($S$)

When the input $S$ is stored in an array

```
QuickSort(A,l, r)
{       If (l < r)
                i← Partition(A,l,r);
                QuickSort(A,l, i − 1);
                QuickSort(A,i + 1, r)
}
```

**Partition** :

$x$← $A[l]$ as a pivot element,

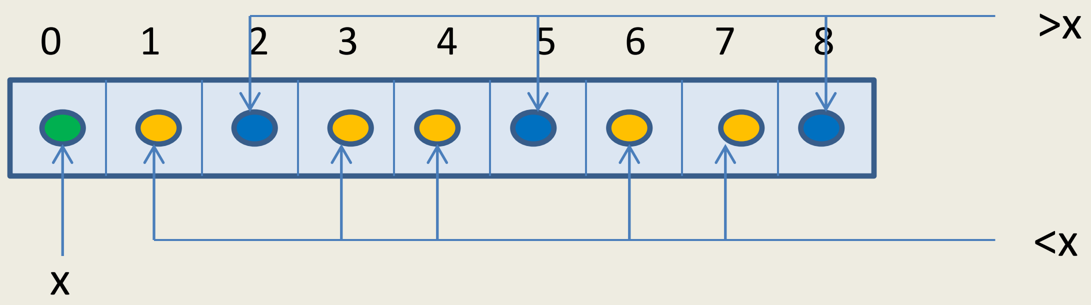
permutes the subarray $A[l \dots r]$ such that

elements preceding $x$ are smaller than $x$,

$A[i]= x$,

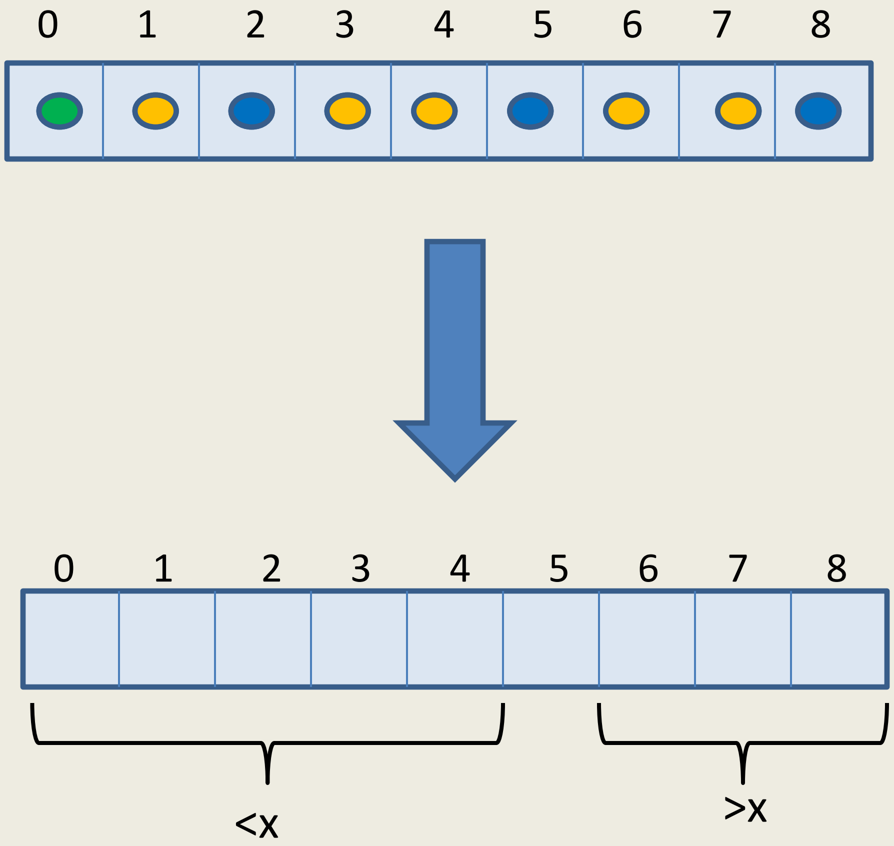
and elements succeeding $x$ are greater than $x$.

# Example: Partition($A$,0,8)




What happens after
**Partition**($A$,0,8)

# Example: Partition($A$,0,8)



# Analyzing average time complexity of QuickSort

## Part 1

## Deriving the recurrence

# Analyzing average time complexity of QuickSort

Let $e_i$ : $i$th **smallest** element of $\boldsymbol{A}$.

**Observation:** the running time of **Quick sort** depends upon the permutation of $e_i$'s and not on the values taken by $e_i$'s.

$\boldsymbol{T}(\boldsymbol{n})$ : Average running time for **Quick sort** on input of size $\boldsymbol{n}$.

**Question:** average over what ?

**Answer:** average over all possible permutations of $\{e_1, e_2, \dots, e_n\}$

Hence, $\boldsymbol{T}(\boldsymbol{n}) = \frac{1}{\boldsymbol{n}!}\sum_{\pi} \boldsymbol{Q}(\pi)$,

where $\boldsymbol{Q}(\pi)$ :the time complexity (or no. of comparisons) when the input is permutation $\pi$.

Think for at least 5-10 minutes to find
**alternate** way to calculate $\boldsymbol{T}(\boldsymbol{n})$.
We shall give discuss two such ways tomorrow.