Data Structures and Algorithms

(CS210A)

Lecture 27

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

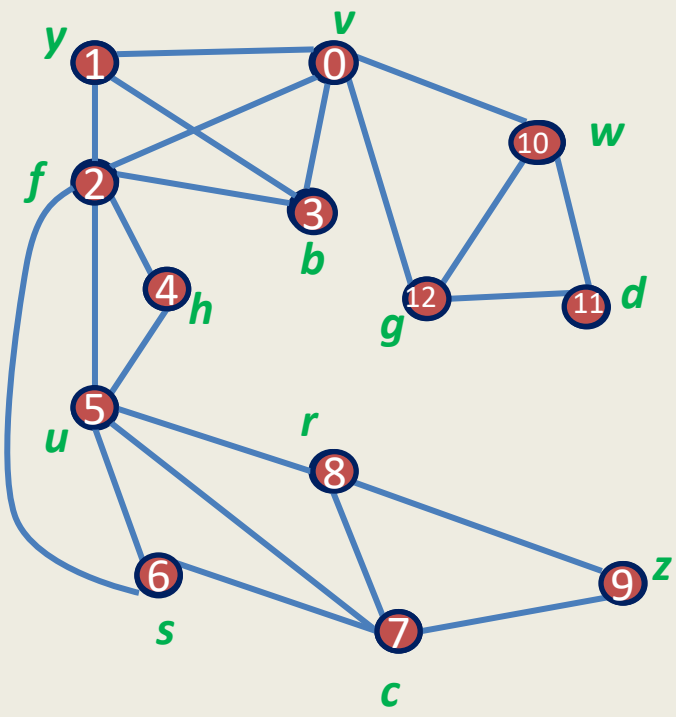
Quick revision of
Depth First Search (DFS) Traversal

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 v∈ V  {     Visited(v)←  false                }
   For each vertex v ∈ V {     If (Visited(v ) = false)   DFS(v)  }
}
```

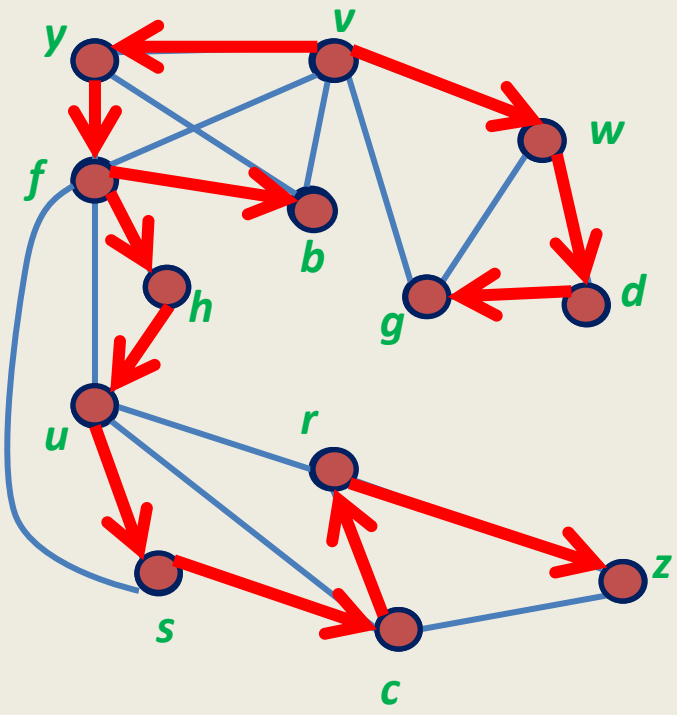
DFN number



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 ?

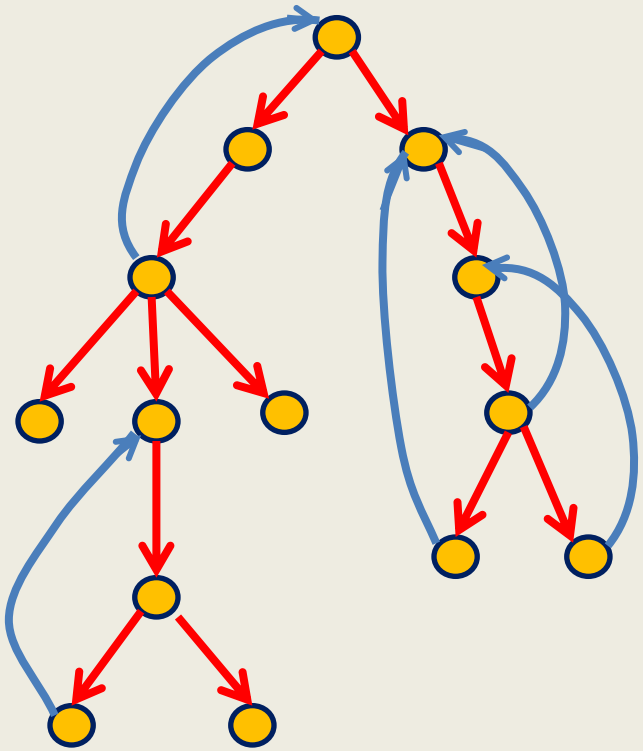
Answer: No.

Question:

Can any rooted tree be obtained through DFS ?

Answer: No.

Always remember this picture



non-tree edge ➔ **back** edge

A **DFS** representation of the graph

Verifying bi-connectivity of a graph

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

A single **DFS** traversal

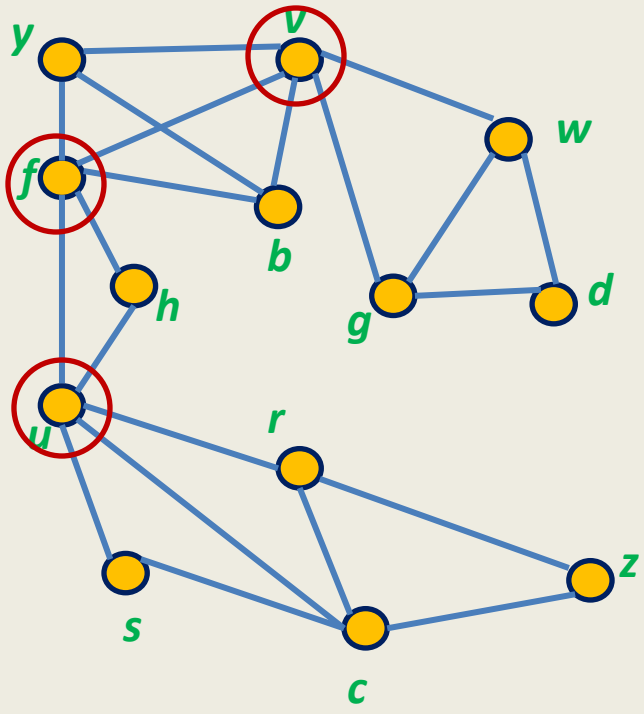
An **O**($m + n$) time algorithm

- 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.

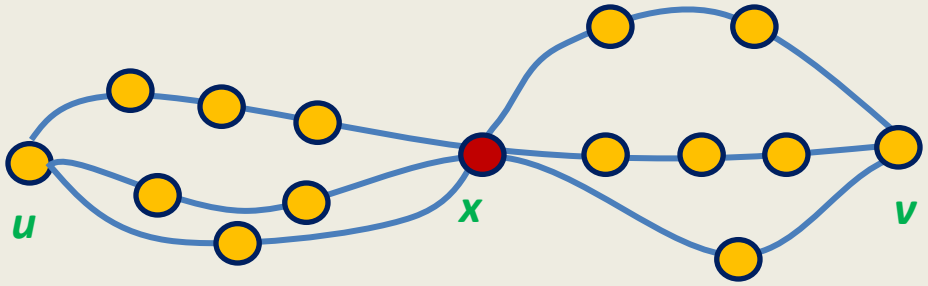
v,*f*,*u* are called the **articulation points** of *G*.

A formal definition of articulaton point

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

∃ ***u***,***v*** different from ***x***

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

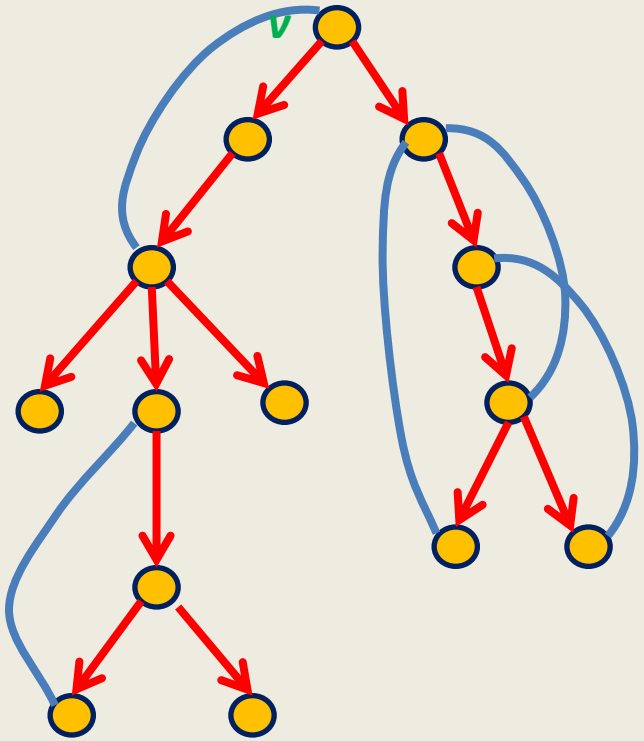


Observation: A graph is biconnected if none of its vertices is an articulation point.

AIM:

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

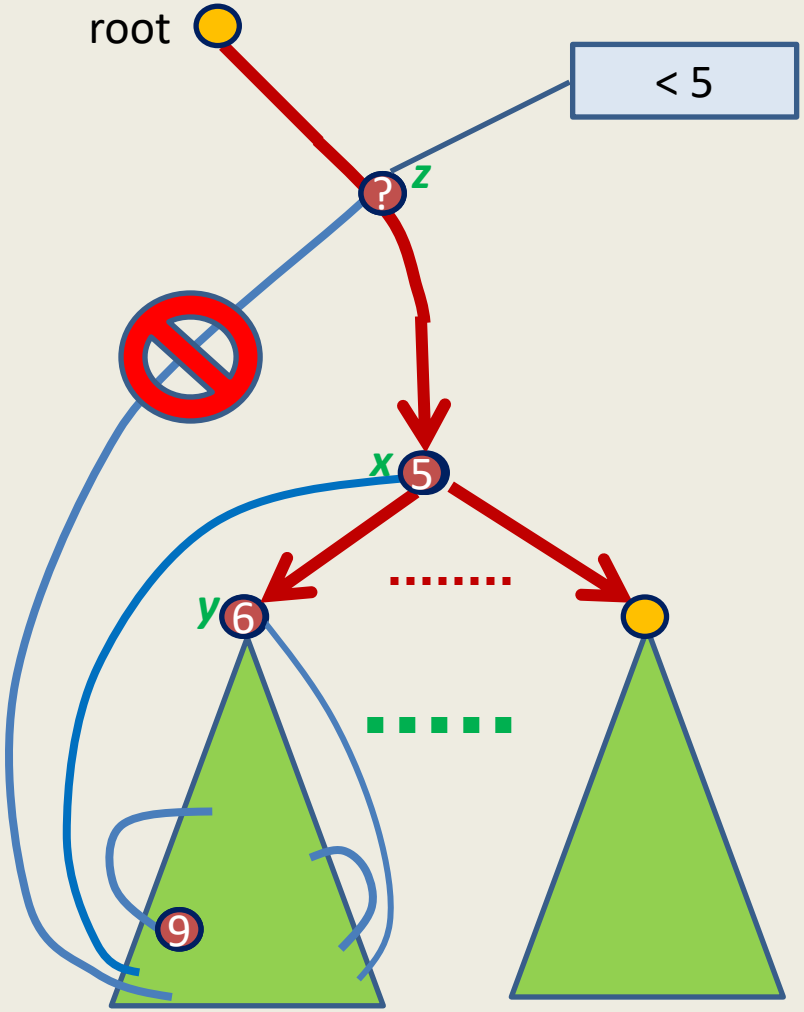
Some observations



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

What about an internal node ?

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.

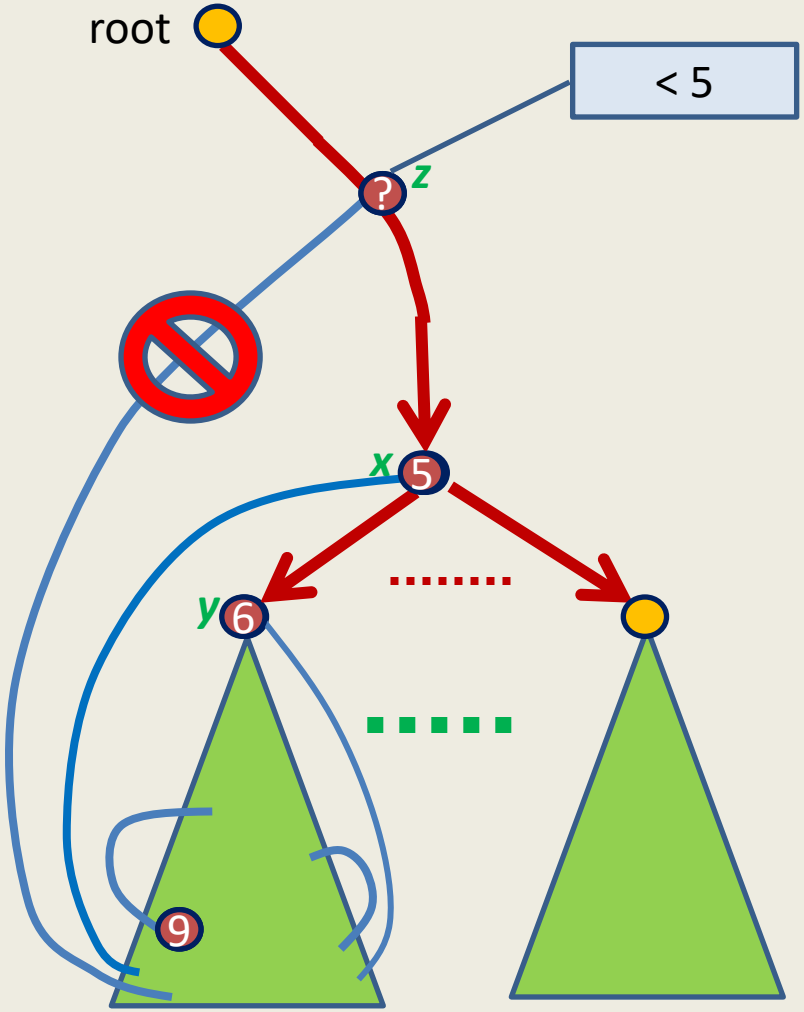
no back edge from **subtree**(**y**) to **ancestor** of ***x***.

➔ 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.

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

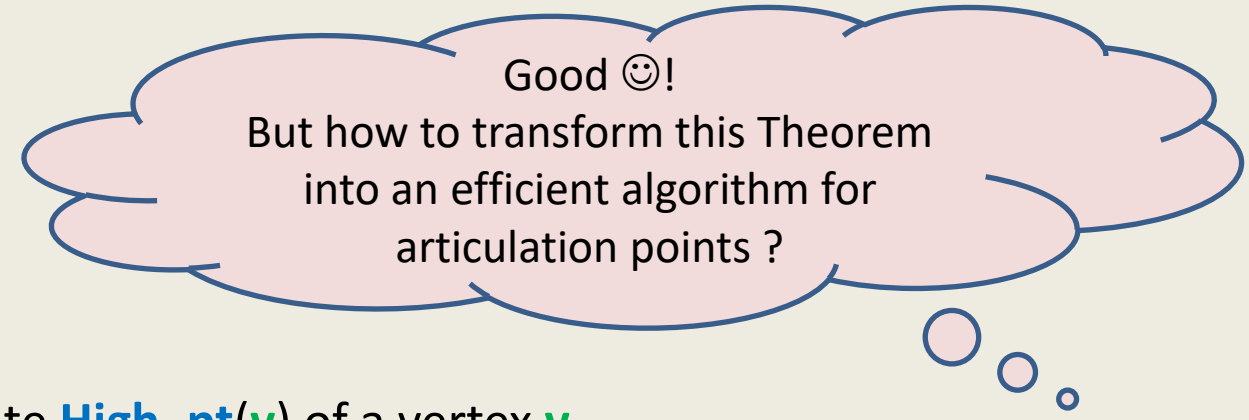
High_pt(**y**) $\geq$ **DFN**(***x***).

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***).



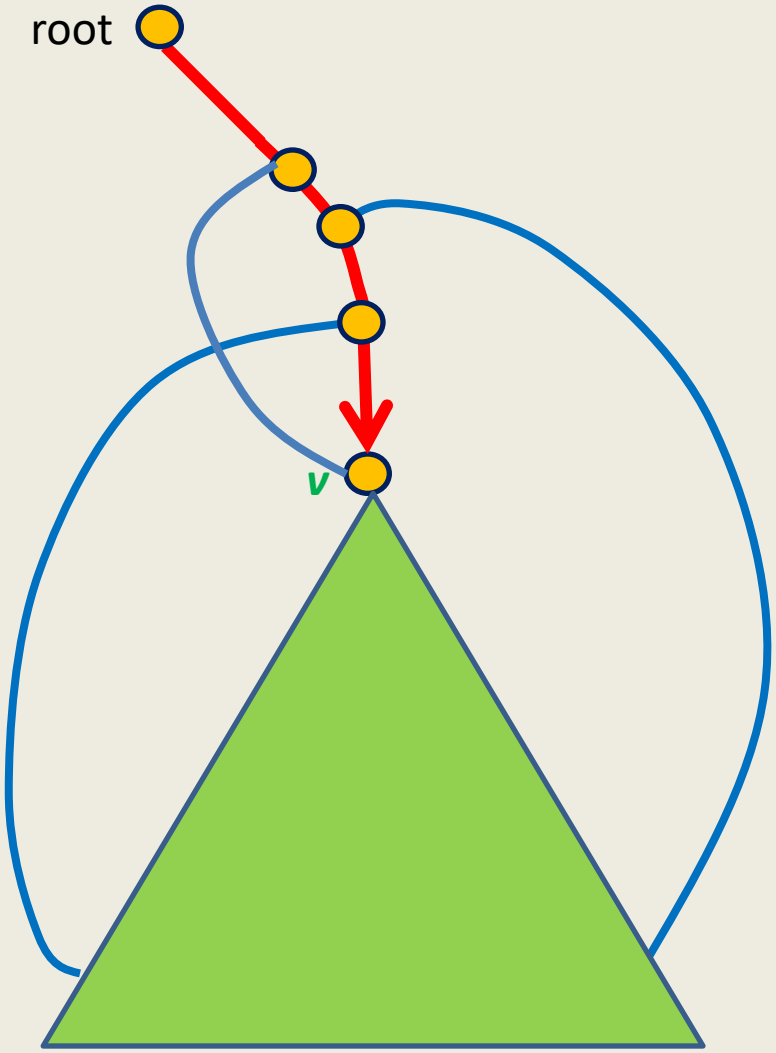
In order to compute **High_pt**(**v**) of a vertex **v**,

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 ☹

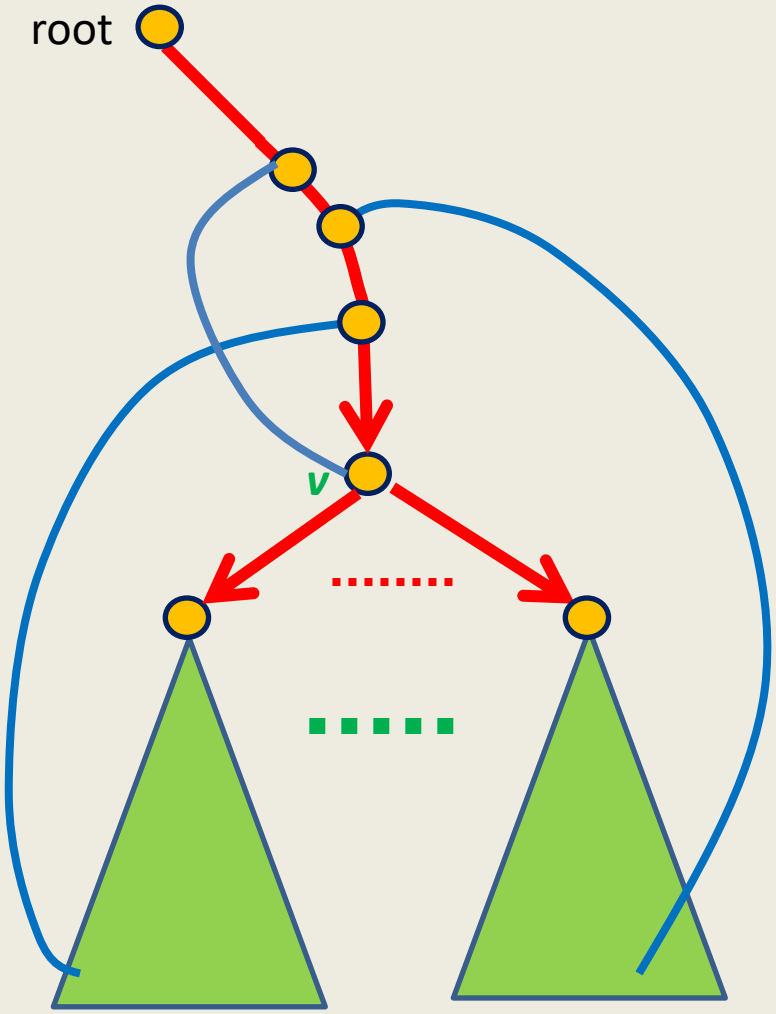
How to compute High_pt(v) efficiently ?



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***) =

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

The novel algorithm

Output : an array **AP[]** s.t.

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*

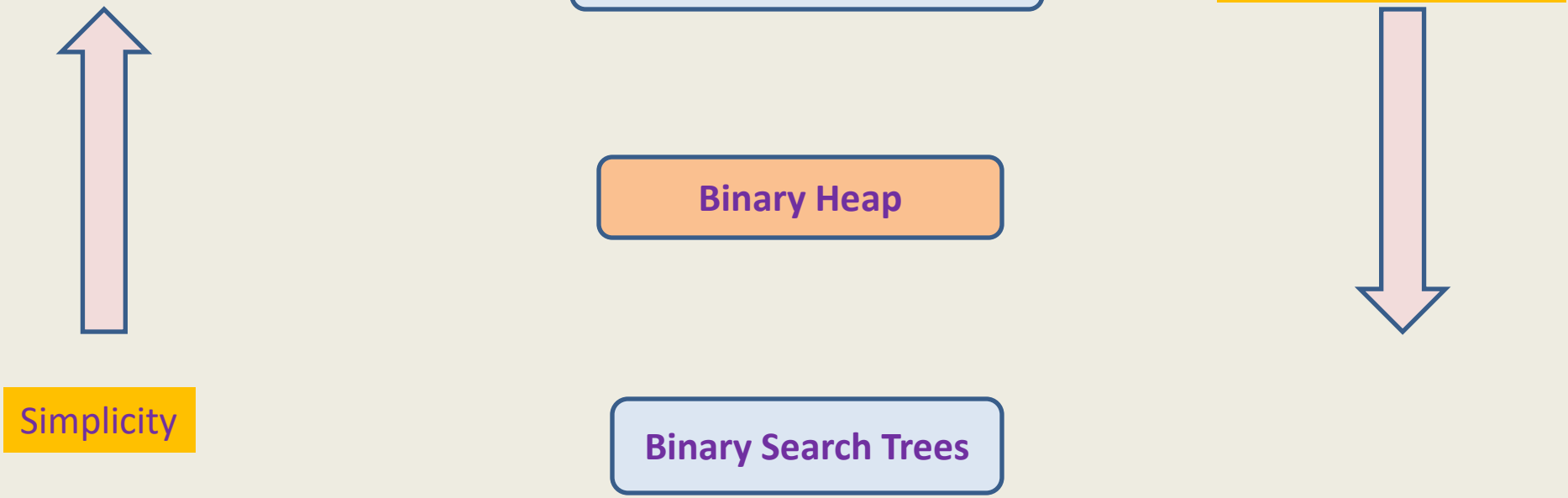
```
DFS(v)
{  Visited(v) ← true; DFN[v] ← dfn ++;  High_pt[v]←∞ ;
   For each neighbor w of v
   {       if (Visited(w)  = false)
           {    Parent(w)← v;   DFS(w);
                High_pt(v) ← min(                    ,                    );
                If  High_pt(w) ≥ DFN[v]
           }
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

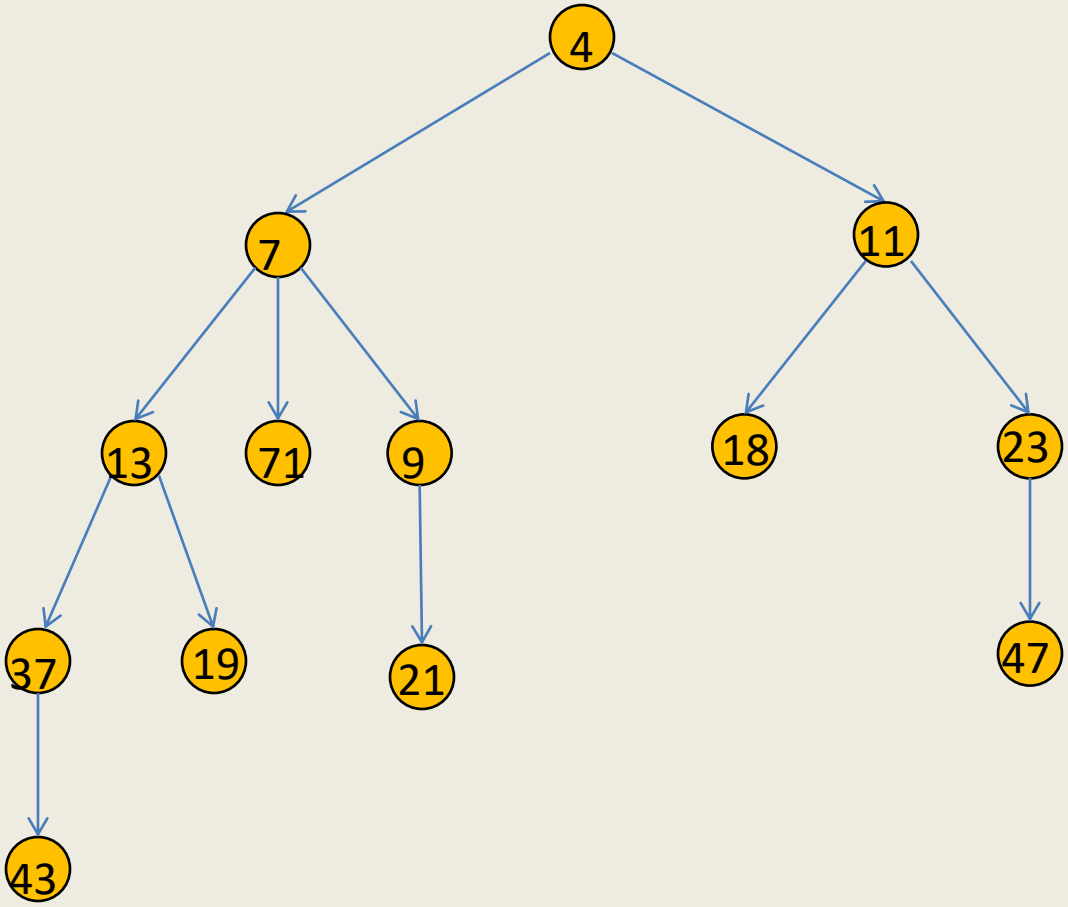
Data Structures



Heap

Definition: a tree data structure where :

value stored in a node $<$ value stored in each of its children.



Operations on a heap

Query Operations

- **Find-min**: report the smallest key stored in the heap.

Update Operations

- **CreateHeap**(**H**) : Create an empty heap **H**.
- **Insert**(**x**,**H**) : Insert a new key with value **x** into the heap **H**.
- **Extract-min**(**H**) : delete the smallest key from **H**.
- **Decrease-key**(***p***, Δ, **H**) : decrease the value of the key ***p*** by amount Δ.
- **Merge**(H_1,H_2) : Merge two heaps H_1 **and** H_2.

Why heaps when we can use a binary search tree ?

Compared to binary search trees, a heap is usually

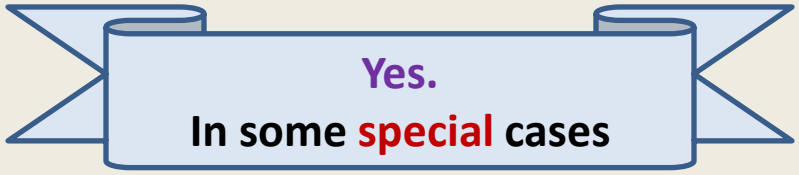
-- much **simpler** and

-- more **efficient**

Existing heap data structures

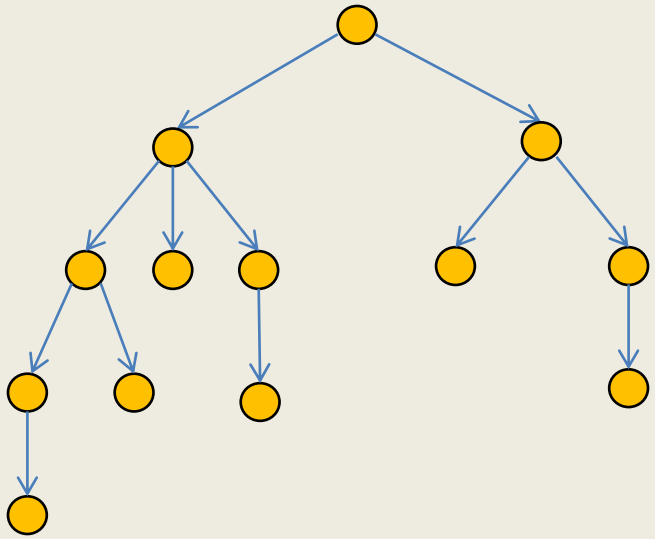
- **Binary heap**
- **Binomial heap**
- **Fibonacci heap**
- **Soft heap**

Can we implement
a binary tree using an array ?



fundamental question

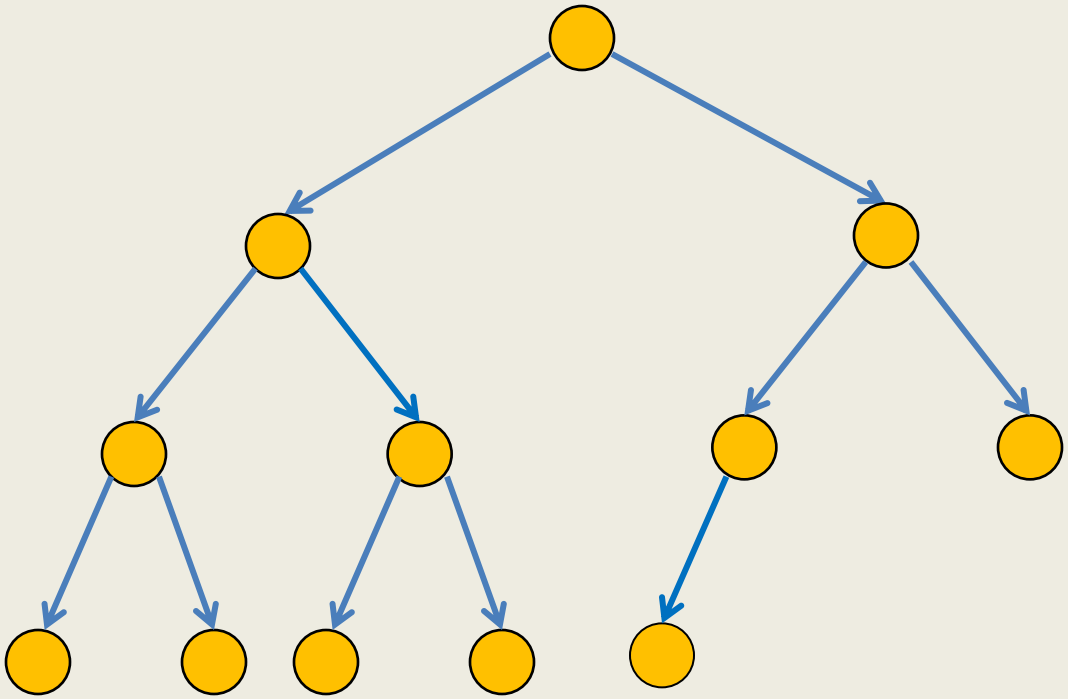
Question: What does the implementation of a tree data structure require ?



Answer: a mechanism to

- access **parent** of a node
- access **children** of a node.

A complete binary tree



A complete binary of 12 nodes.

A complete binary tree