Data Structures and Algorithms I SCS1201 - CS

Dr. Dinuni Fernando

Senior Lecturer

Based on Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.

Lecture 9

UCSC

Learning outcomes

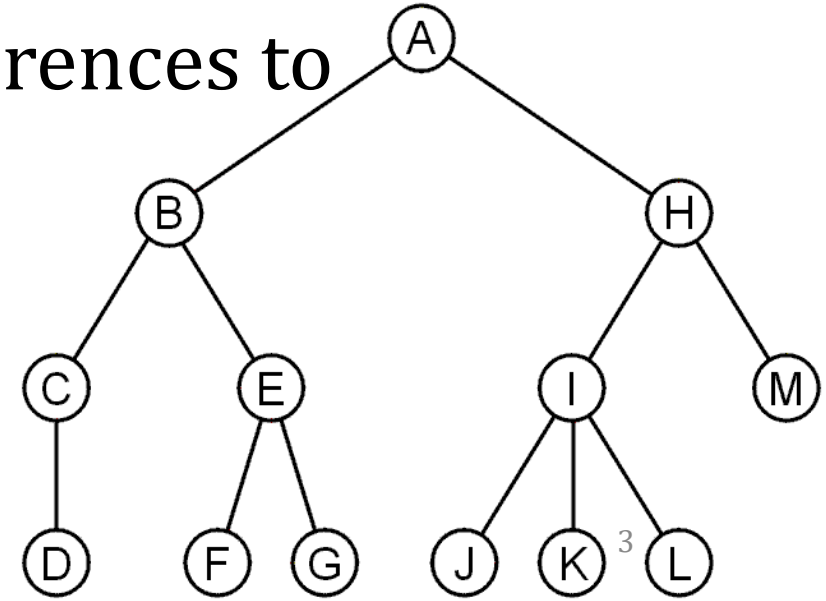
In this topic, we will cover:

- Definition of a tree data structure and its components
- Concepts of:
  - Root, internal, and leaf nodes
  - Parents, children, and siblings
  - Paths, path length, height, and depth
  - Ancestors and descendants
  - Ordered and unordered trees
  - Subtrees

Trees

A rooted tree data structure stores information in *nodes*

- Similar to linked lists:
  - There is a first node, or *root*
  - Each node has variable number of references to successors



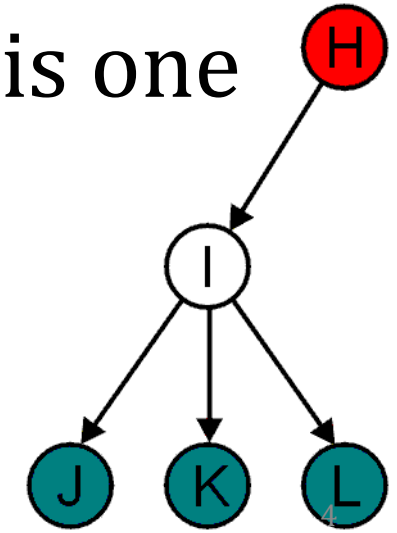
Terminology

All nodes will have zero or more child nodes or *children*

- I has three children: J, K and L

For all nodes other than the root node, there is one parent node

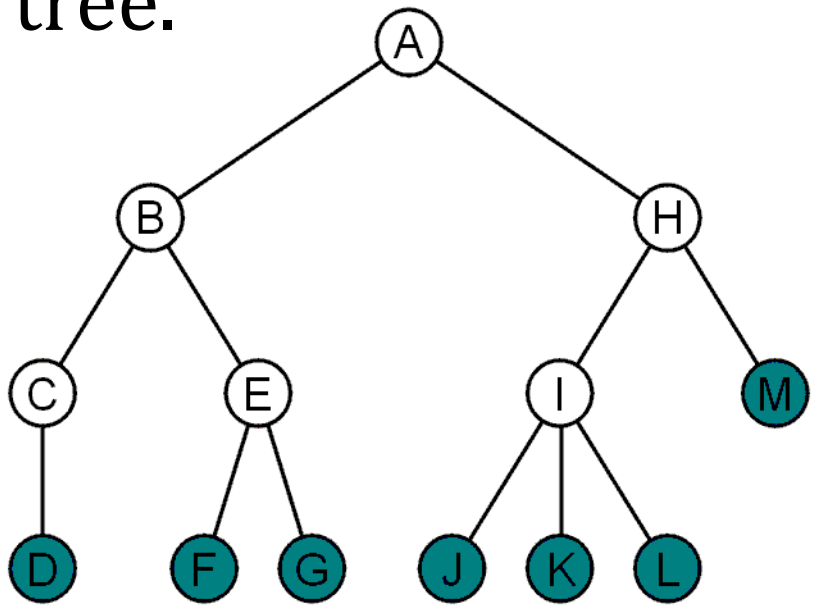
- H is the parent I



Terminology

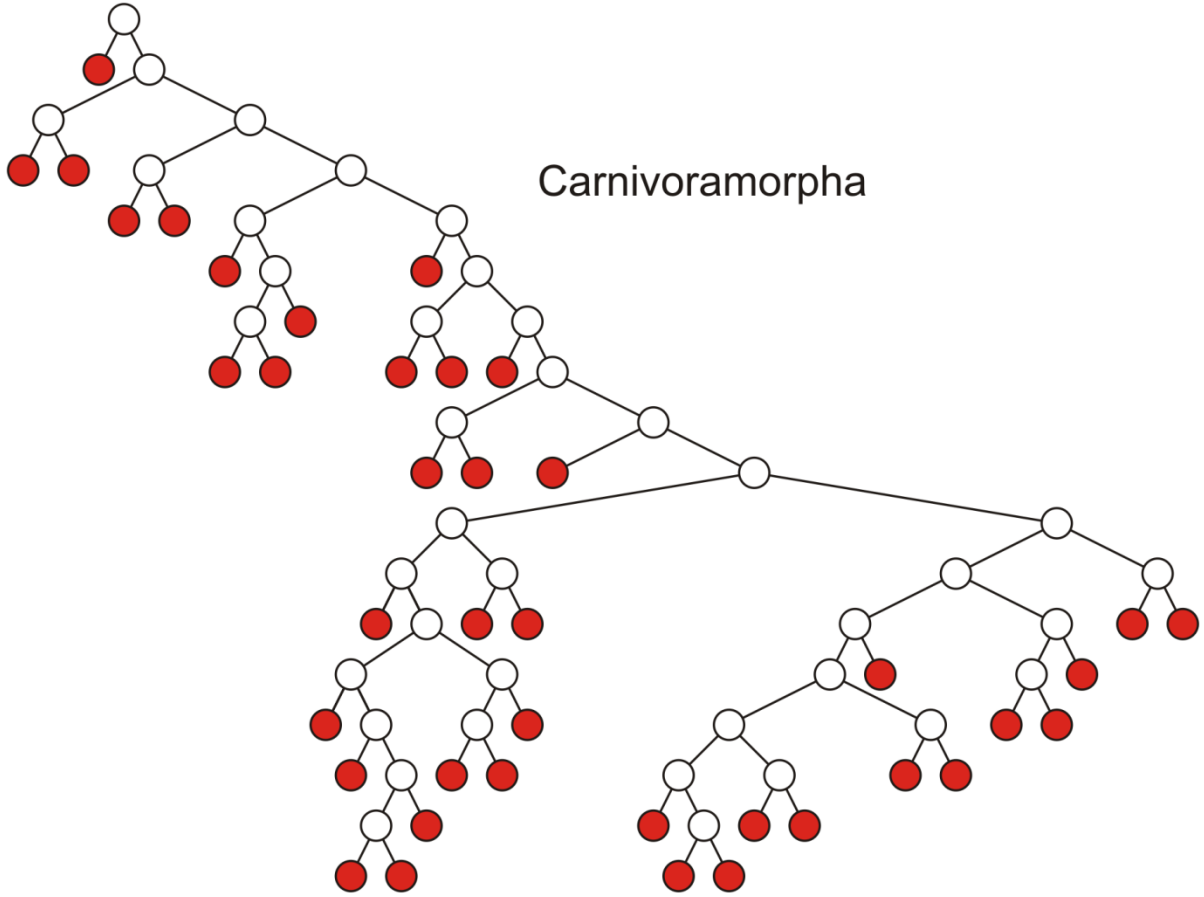
Nodes with degree zero are also called *leaf nodes.*

All other nodes are said to be *internal nodes*, that is, they are internal to the tree.



Terminology

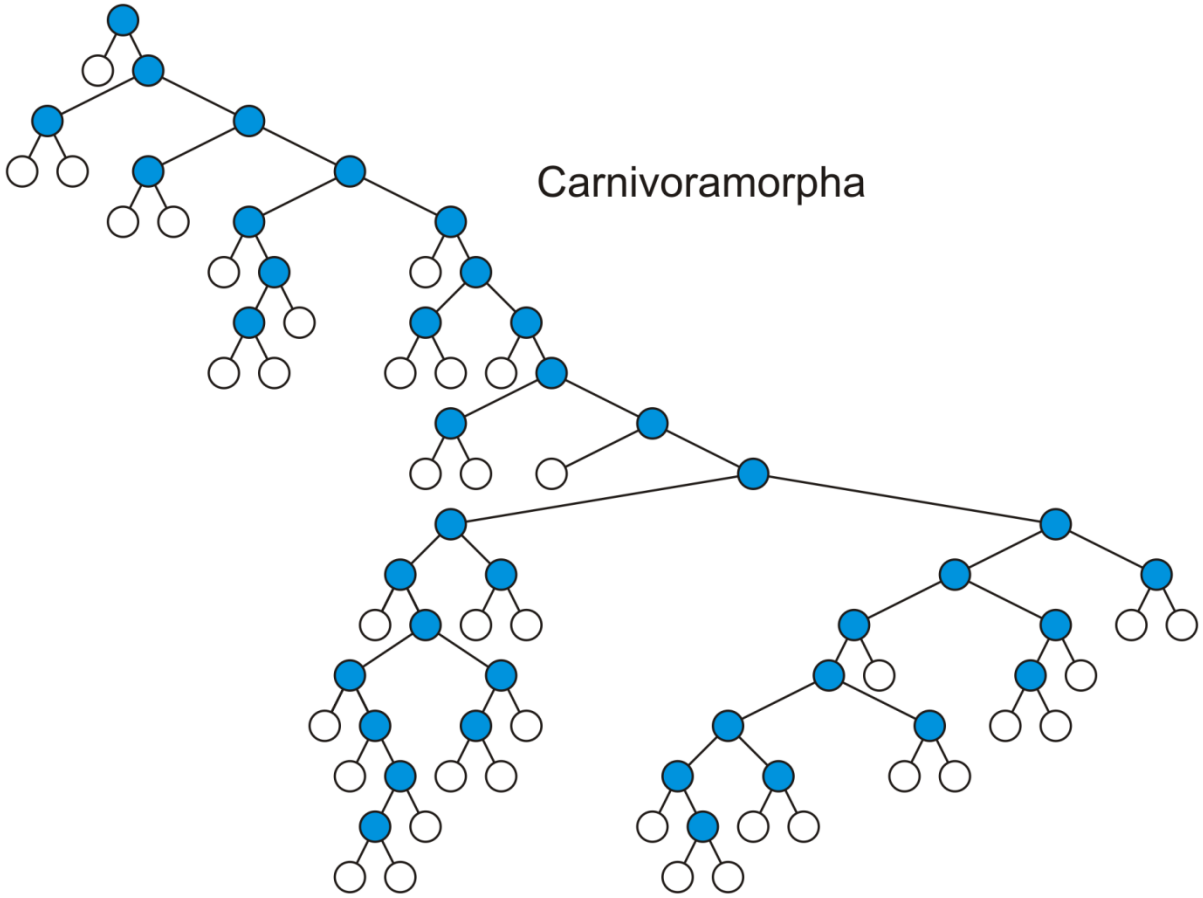
Leaf nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

Internal nodes:

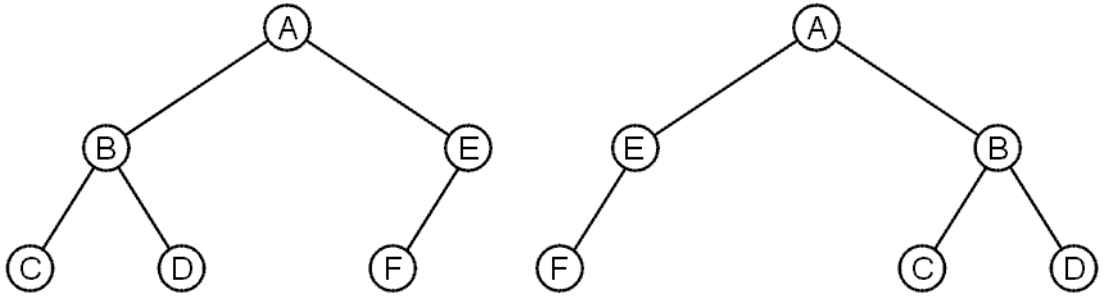


Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

These trees are equal if the order of the children is ignored

- *unordered trees*

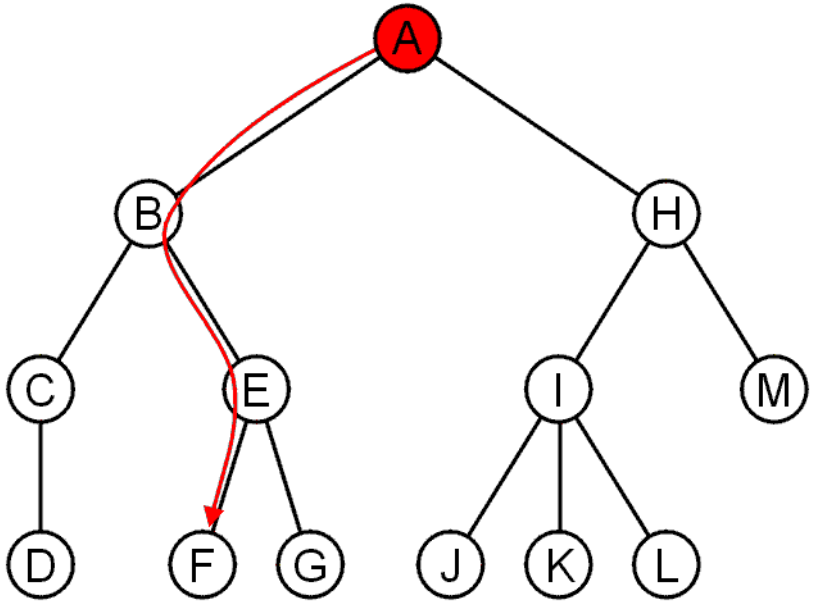


They are different if order is relevant (*ordered trees*)

- We will usually examine ordered trees (linear orders)
- In a hierarchical ordering, order is not relevant

Terminology

The shape of a rooted tree gives a natural flow from the *root node*, or just *root*

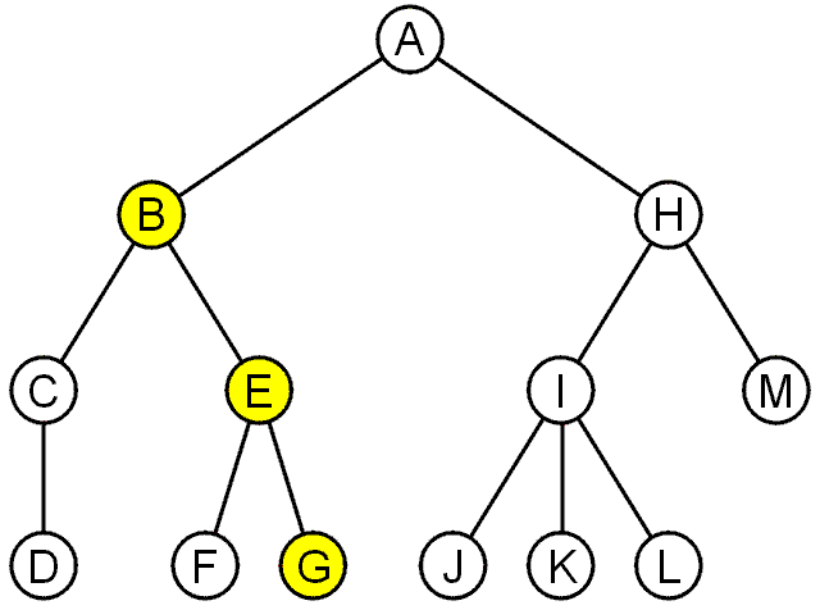


Terminology

A path is a sequence of nodes $(a_0, a_1, ..., a_n)$ where a_{k+1} is a child of a_k is

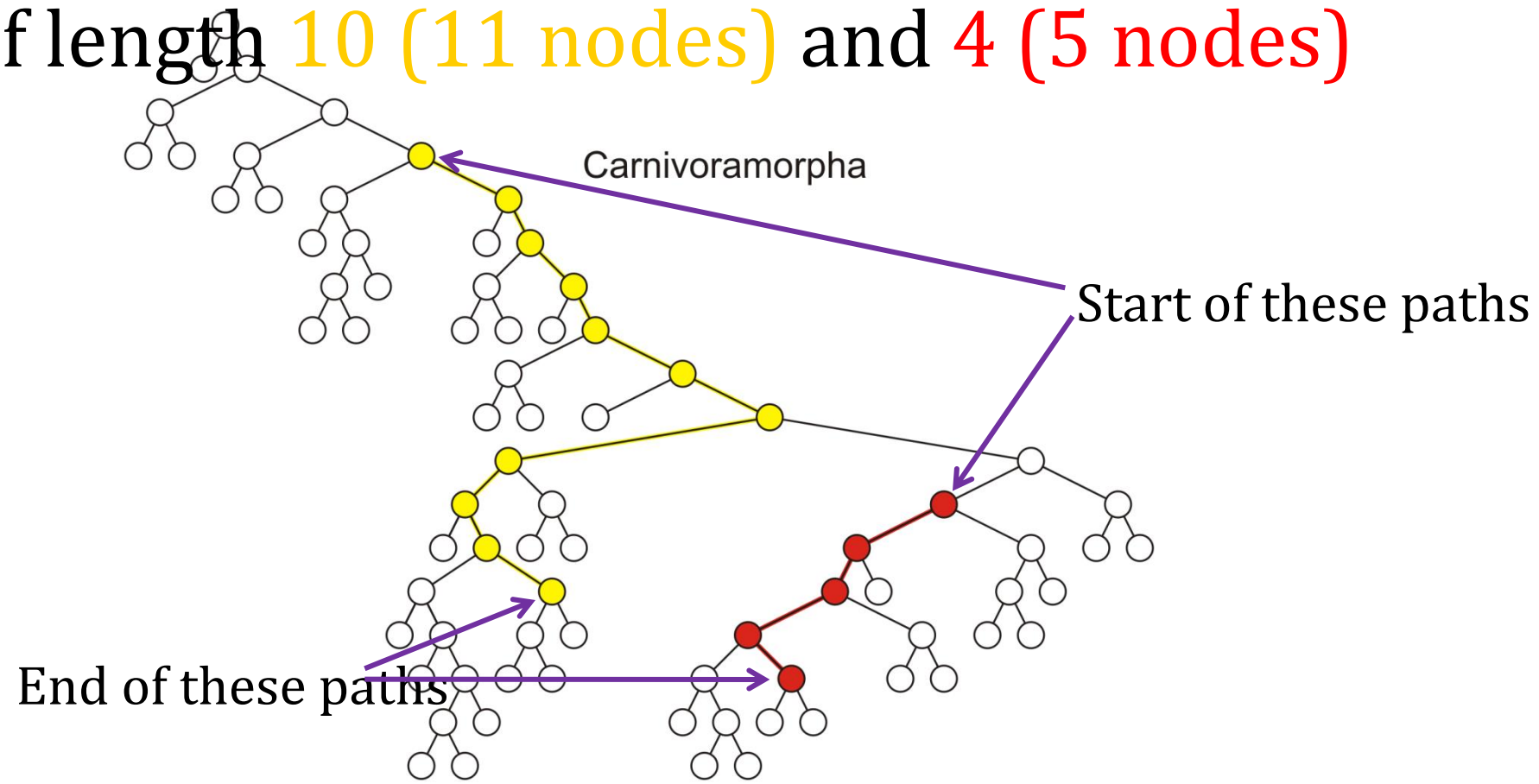
The length of this path is n

E.g., the path (B, E, G) has length 2



Terminology

Paths of length 10 (11 nodes) and 4 (5 nodes)



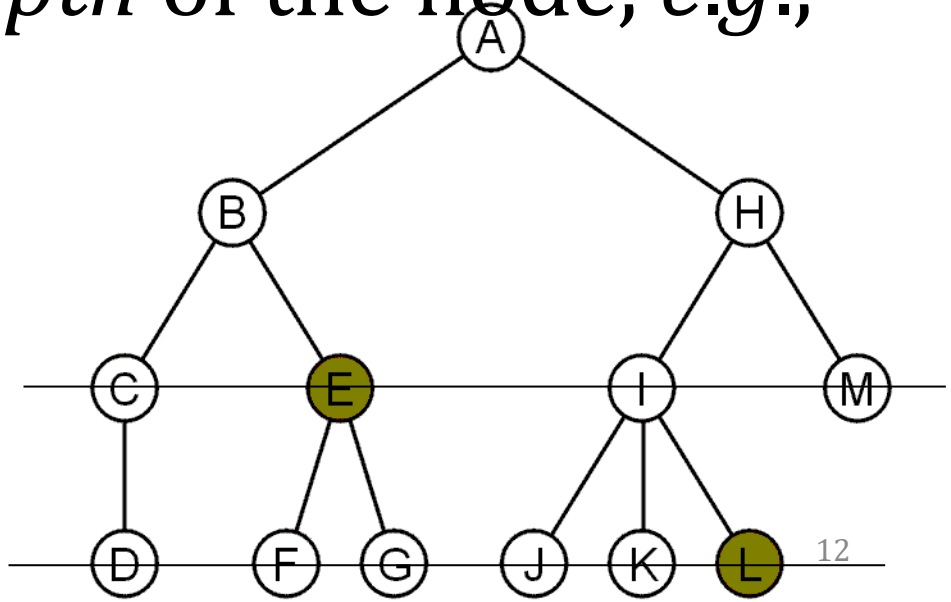
Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology

For each node in a tree, there exists a unique path from the root node to that node

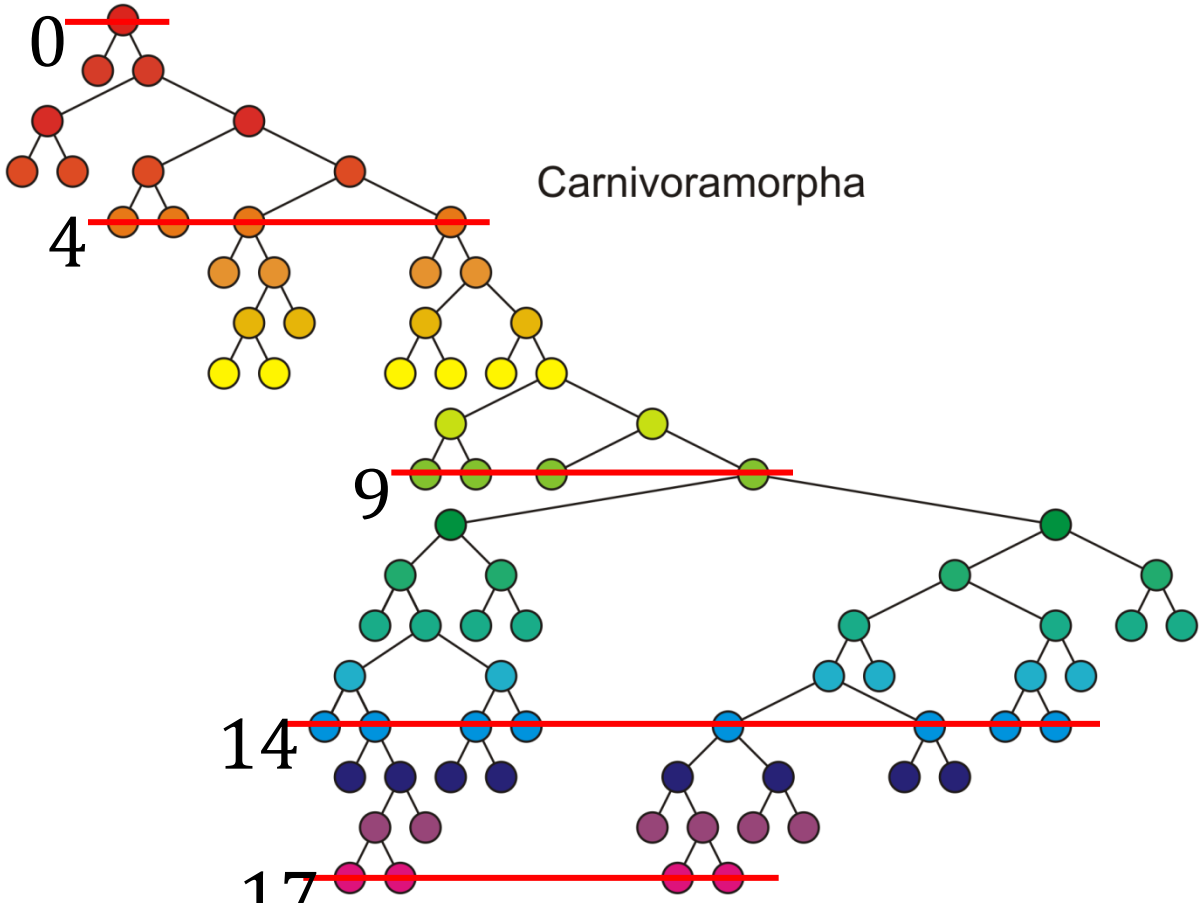
The length of this path is the *depth* of the node, *e.g.*,

- E has depth 2
- L has depth 3



Terminology

Nodes of depth up to 17



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

The *height* of a tree is defined as the maximum depth of any node within the tree

The height of a tree with one node is 0

- Just the root node

For convenience, we define the height of the empty tree to be –1

Terminology

If a path exists from node *a* to node *b*:

- *a* is an *ancestor* of *b*
- *b* is a *descendent* of *a*

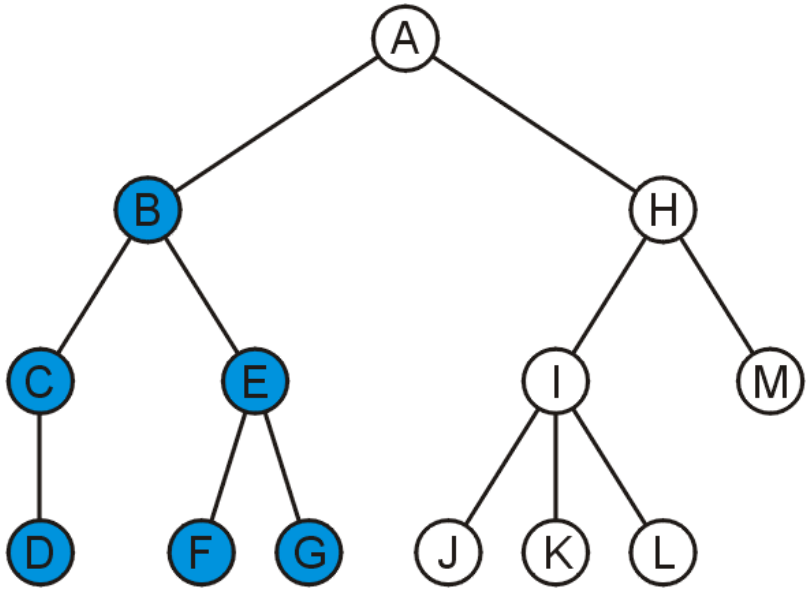
Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective *strict* to exclude equality: *a* is a *strict descendent* of *b* if *a* is a descendant of *b* but $a \neq b$

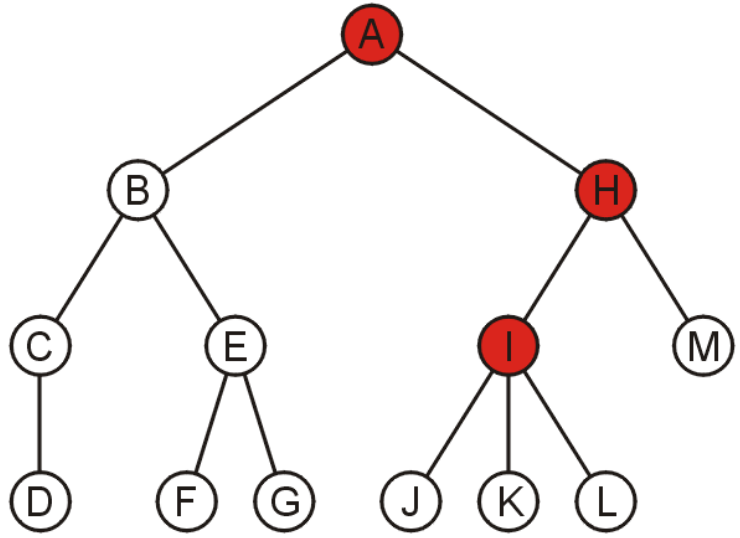
The root node is an ancestor of all nodes

Terminology

The descendants of node B are B, C, D, E, F, and G:



The ancestors of node I and H, are A:



General Trees

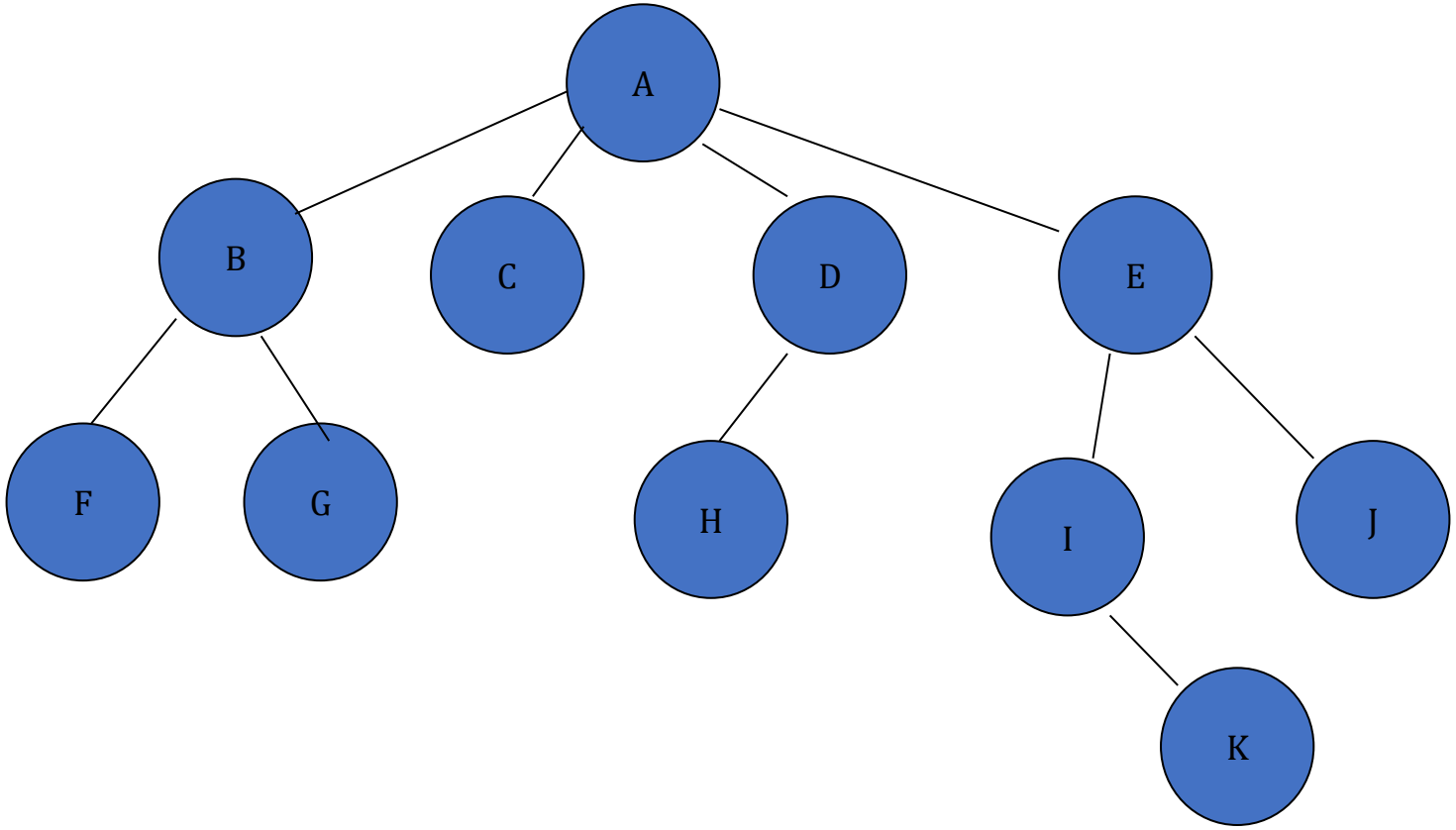
Trees can be defined in two ways :

- Recursive
- Non- recursive

One natural way to define a tree is recursively.

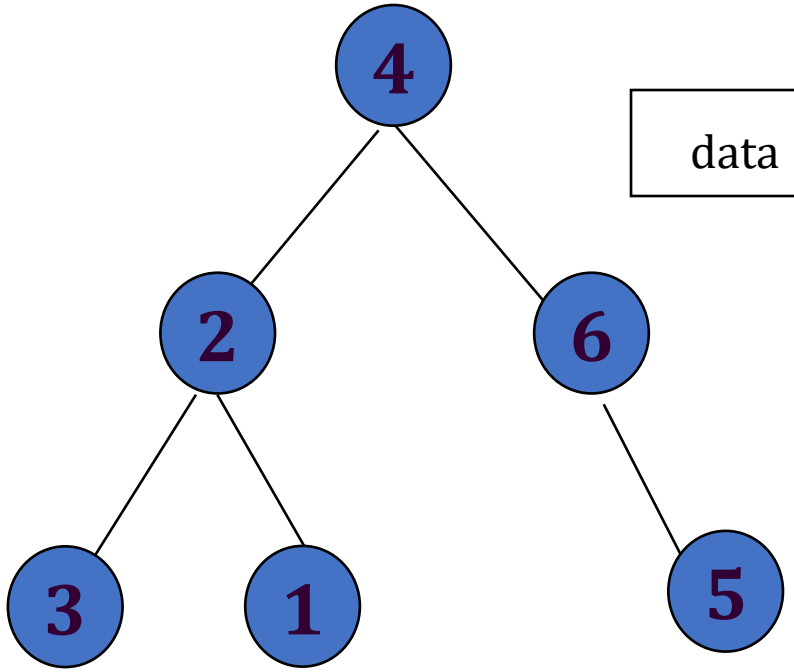
General trees-Definition

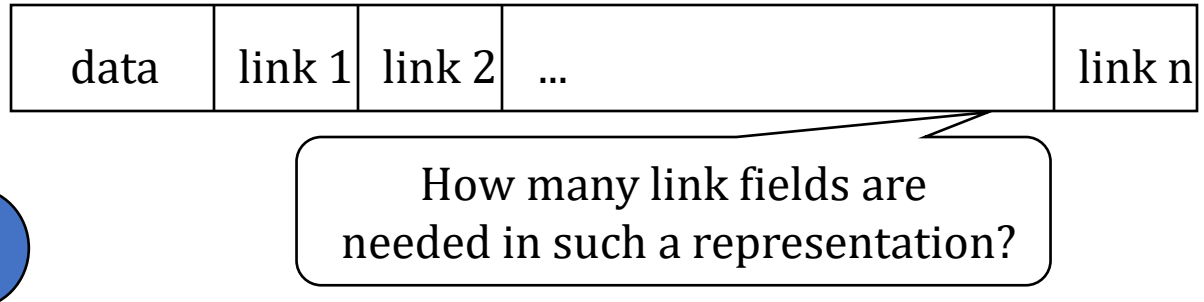
- A tree consists of set of nodes and set of edges that connected pair of nodes.



Representation of Trees

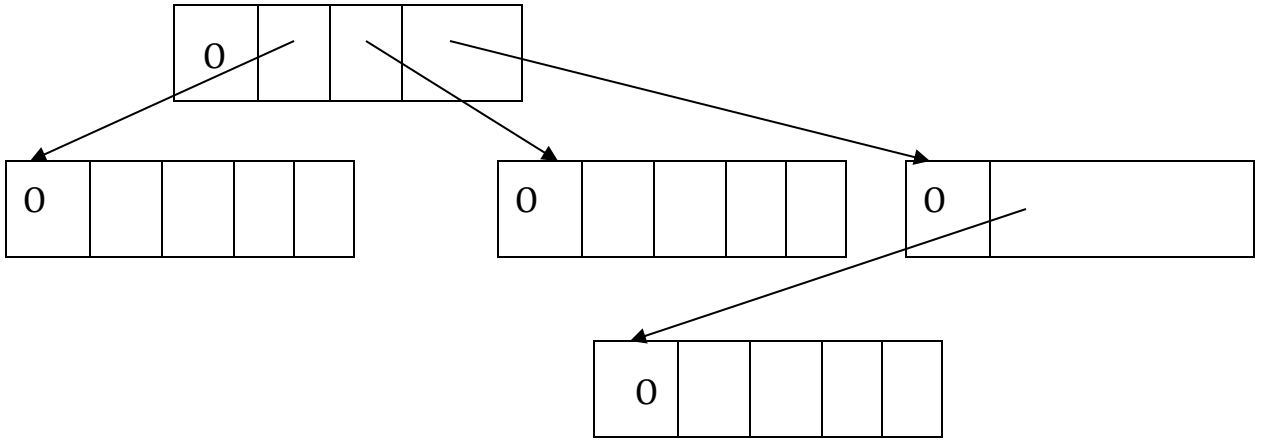
- List Representation
  - 4(2 (3)(1))(6(5))
  - The root comes first, followed by a list of sub-trees





A Tree Node

- Every tree node:
  - object – useful information
  - children – pointers to its child nodes



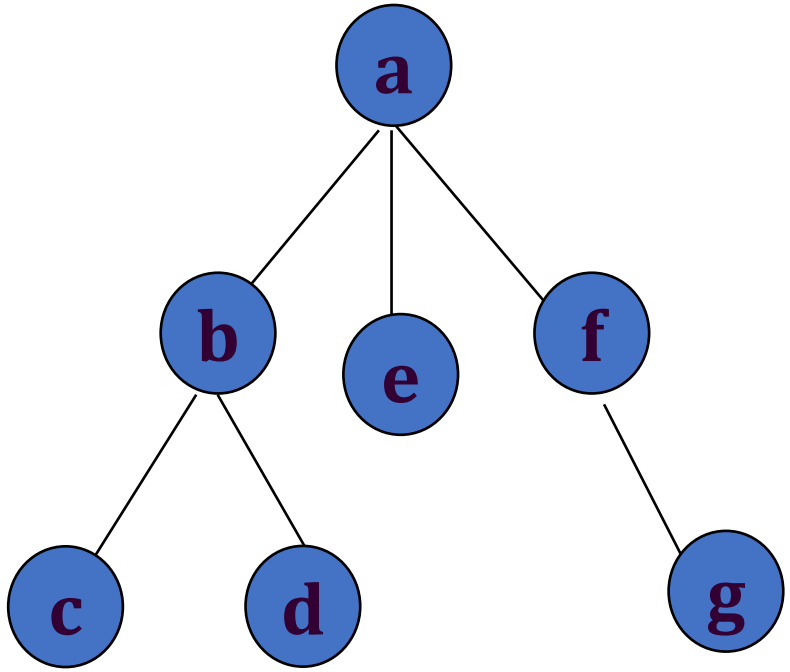
Tree - Implementation

- One way to implement a tree would be to have in each node a reference to each child of the node in addition to its data.
- However, the number of children per node can vary so greatly and is not known in advanced.
- It might be infeasible to make the children direct links in the data structure.
- These would be too much wasted space.

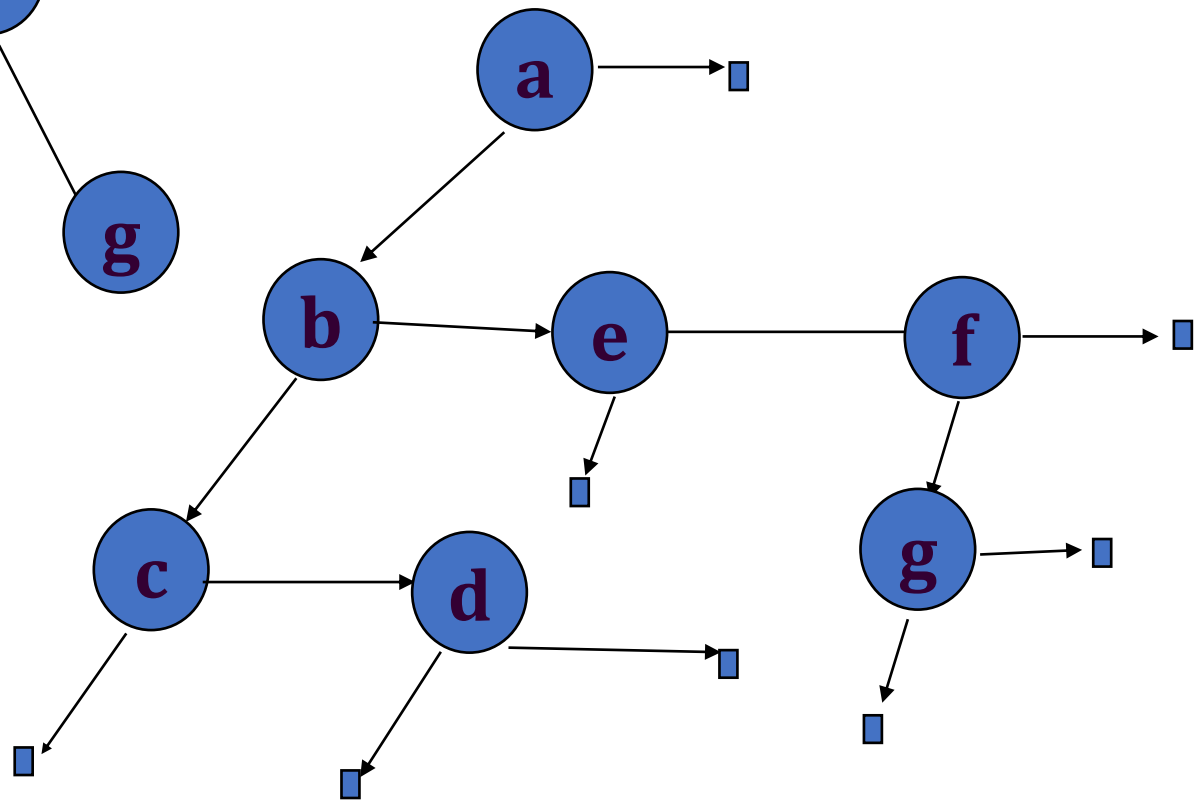
The solution is simple :

- Keep the children of each node in a linked list of tree nodes. Thus, each node keeps two references : one to its leftmost child and other one for its right sibling.

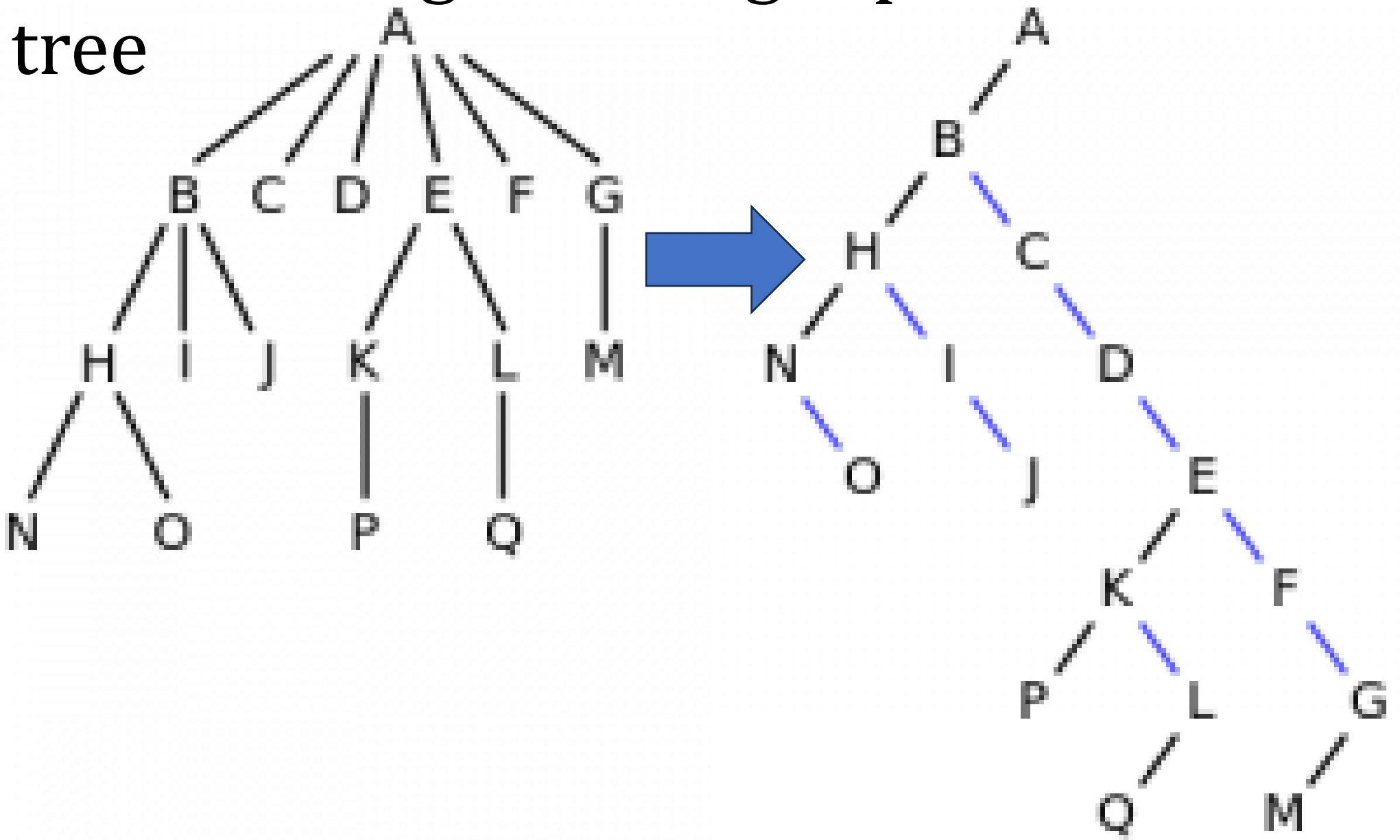
Example



data	
left child	right sibling



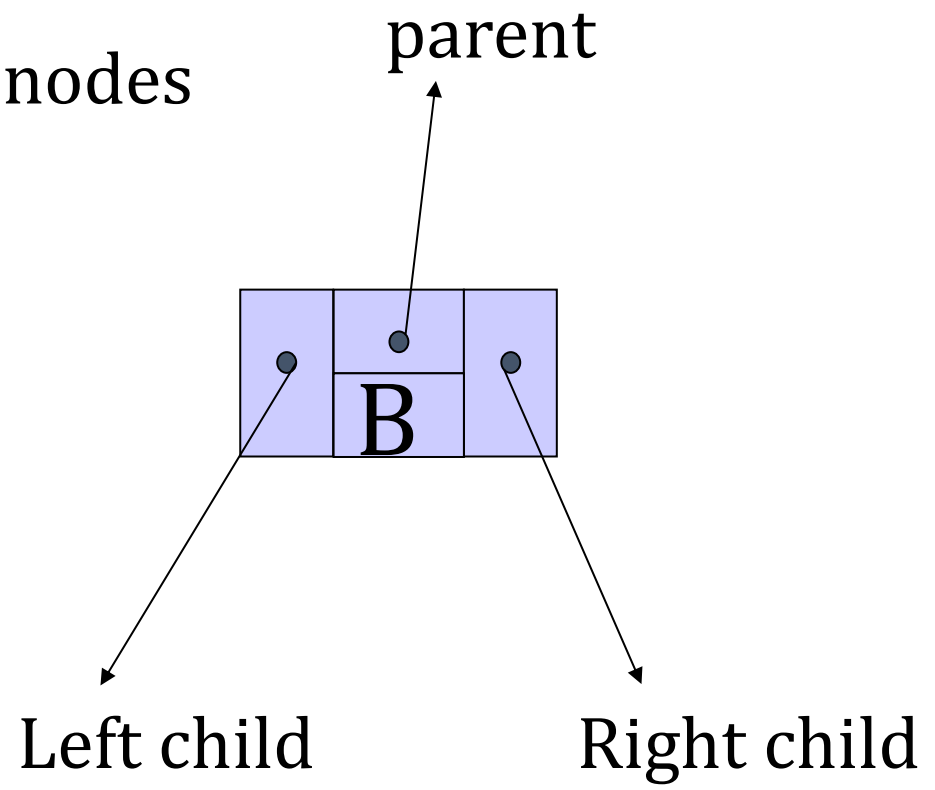
Left Child - Right Sibling representation of a tree



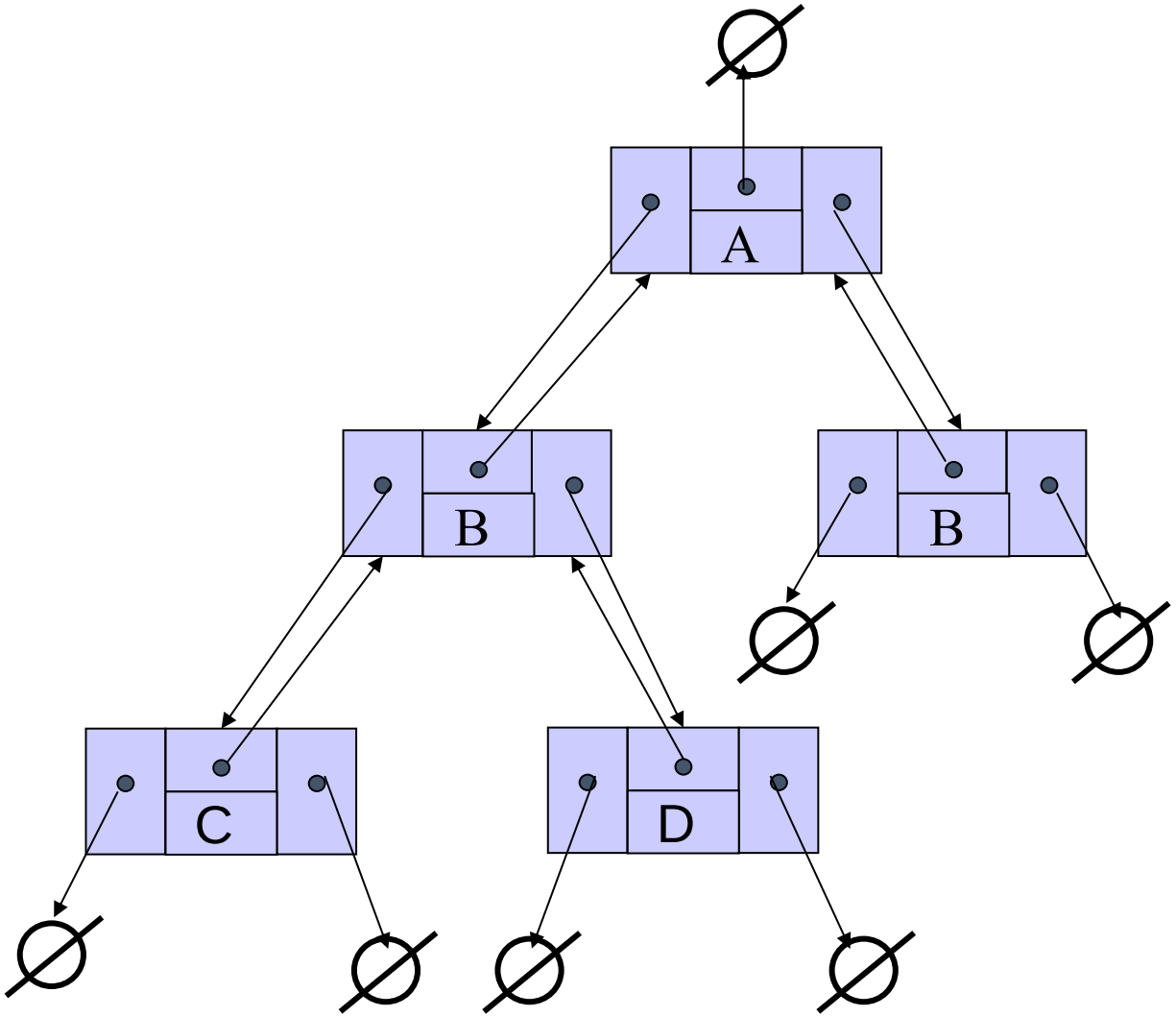
A linked structure for Binary Trees

We represent each node of a binary tree by an object which stores

- Element
- References to its parent and children nodes

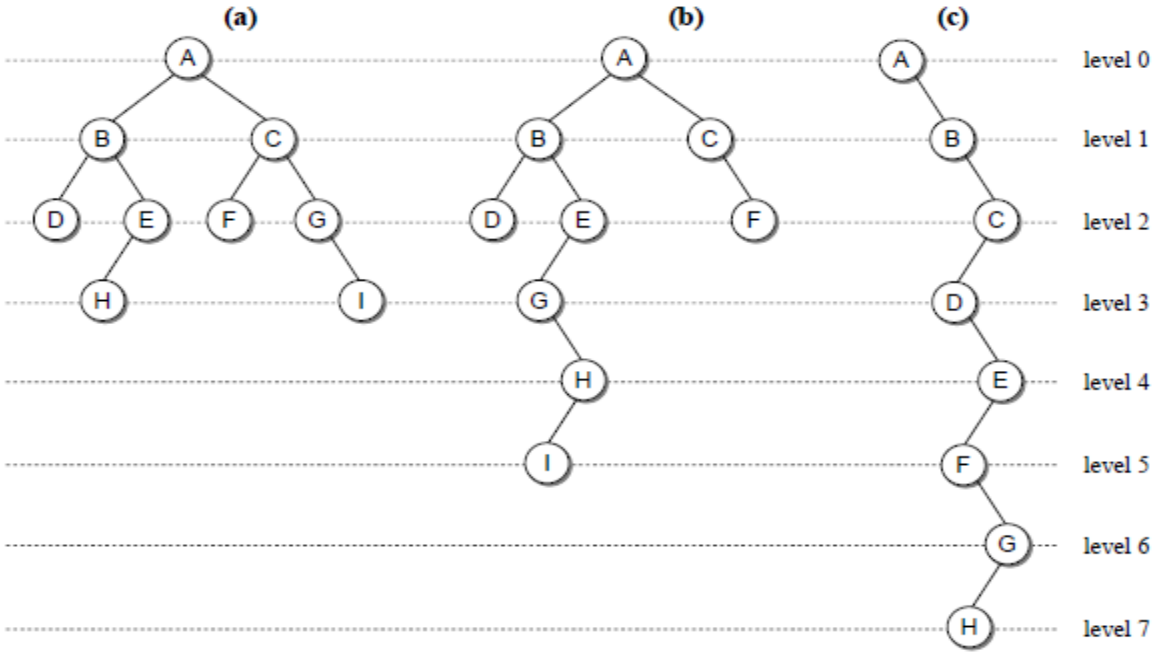


A linked structure for Binary Trees



Properties of Binary Tress

- Binary trees come in many deferent shapes and sizes. The shapes vary depending on the number of nodes and how the nodes are linked.



Three different arrangements of nine nodes in a binary tree.

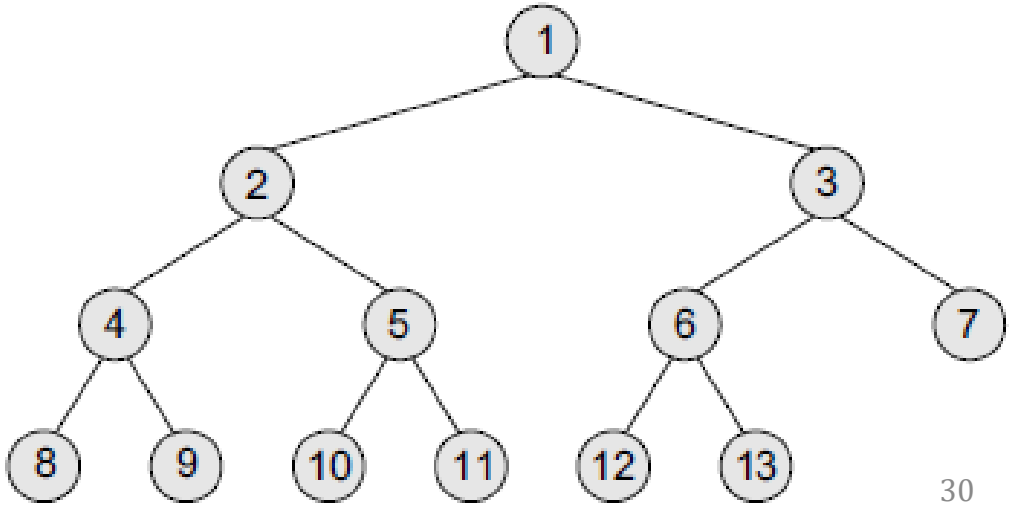
Complete Binary Trees

A *complete binary tree* is a binary tree that satisfies following 3 properties.

- In a complete binary tree, every level, except possibly the last, is completely filled.
- All nodes appear as far left as possible.
- Each non leaf node has exactly two child nodes

Example

- In a complete binary tree Tn, there are exactly n nodes and level r of T can have at most 2^r nodes
- Note that in the diagram, level 0 has $2^0 = 1$ node, level 1 has $2^1 =$ 2nodes, level 2 has $2^2 = 4$ nodes,
- Level 3 has 6 nodes which is less than the maximum of $2^3 = 8$ nodes.



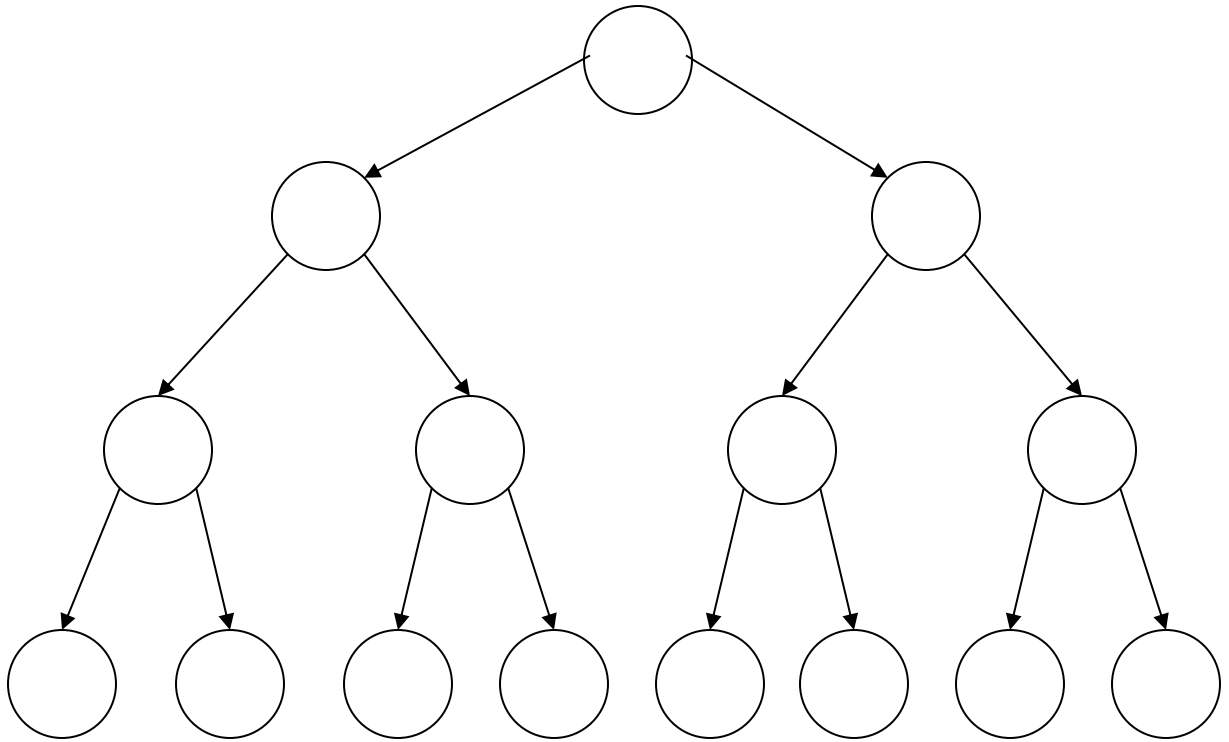
Example (Cont'd)

- In the above tree has exactly 13 nodes. They have been purposely labelled from 1 to 13,
- So that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node.
- The formula can be given as—if K is a parent node, then its left child can be calculated as 2 × K and its right child can be calculated as 2 × K + 1.
- For example, the children of the node 4 are 8 (2 × 4) and 9 (2 × 4 + 1).
- Similarly, the parent of the node K can be calculated as | K/2 |.
- Given the node 4, its parent can be calculated as | 4/2 | = 2.
- The height of a tree T_n having exactly n nodes is given as:
- H_n = | log2 (n + 1) |
- if a tree (T) has 10,00,000 nodes, then its height is 21.

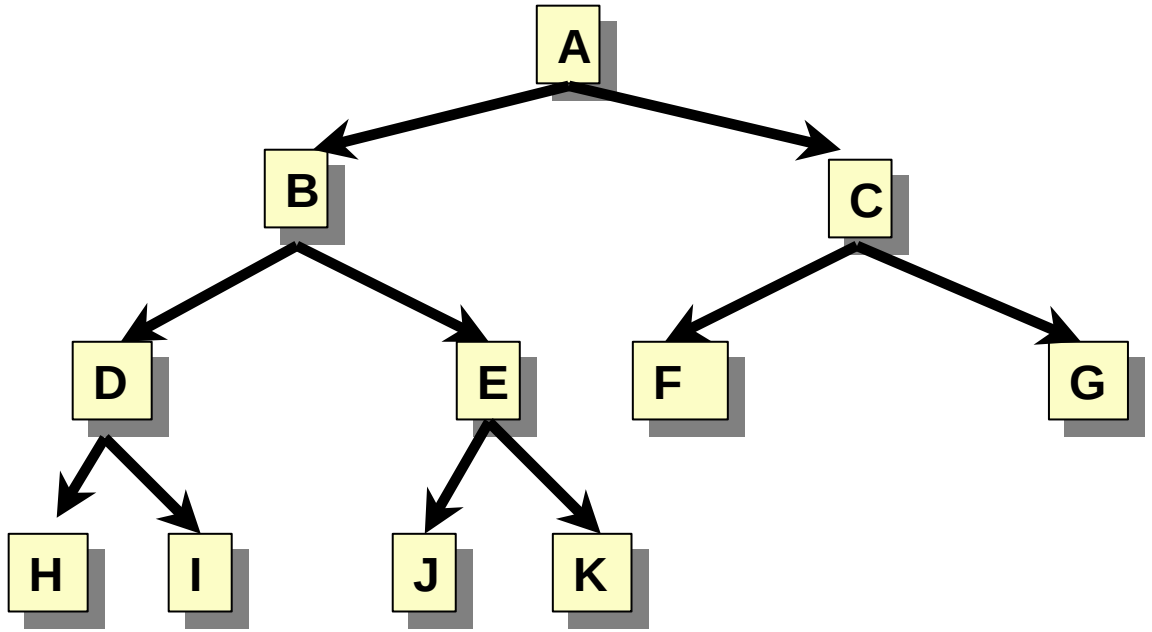
Properties of Binary Trees

- A binary tree is a **full** binary tree if and only if:
  - Each non leaf node has exactly two child nodes
  - All leaf nodes have identical path length
- It is called **full** since all possible node slots are occupied

A Full Binary Tree - Example



Complete Binary Trees - Example



Properties of binary trees.

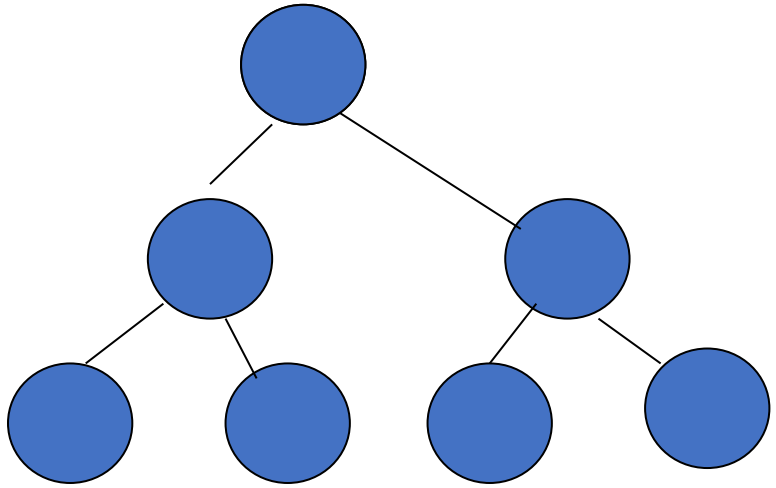
- If a binary tree contains m nodes at level L, then it contains at most 2m nodes at level L+1.
- A binary tree can contain at most 2^L nodes at L

At level 0 B-tree can contain at most $1= 2^0$ nodes

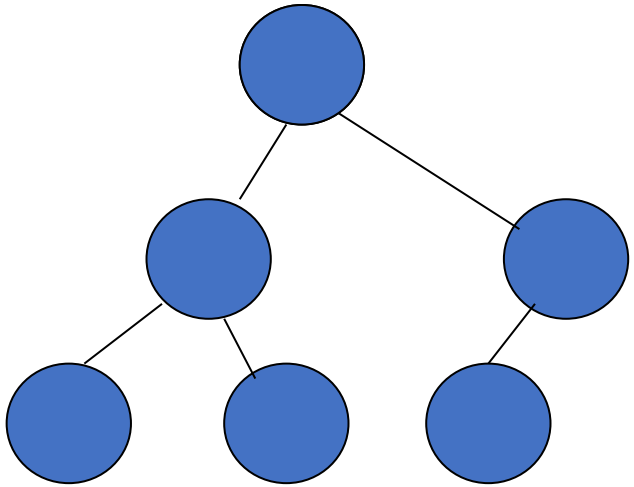
At level 1 B-tree can contain at most $2= 2^1$ nodes At level 2 B-tree can contain at most $4= 2^2$ nodes At level L B-tree can contain at most → 2^L nodes

Full B-tree

- A full B-tree of depth d is the B-tree that contains exactly 2^L nodes at each level between 0 and d (or 2^d nodes at d)



full B-tree



Not a full B-tree

The total number of nodes (Tn) in a full binary tree of depth d is 2^{d+1}-1

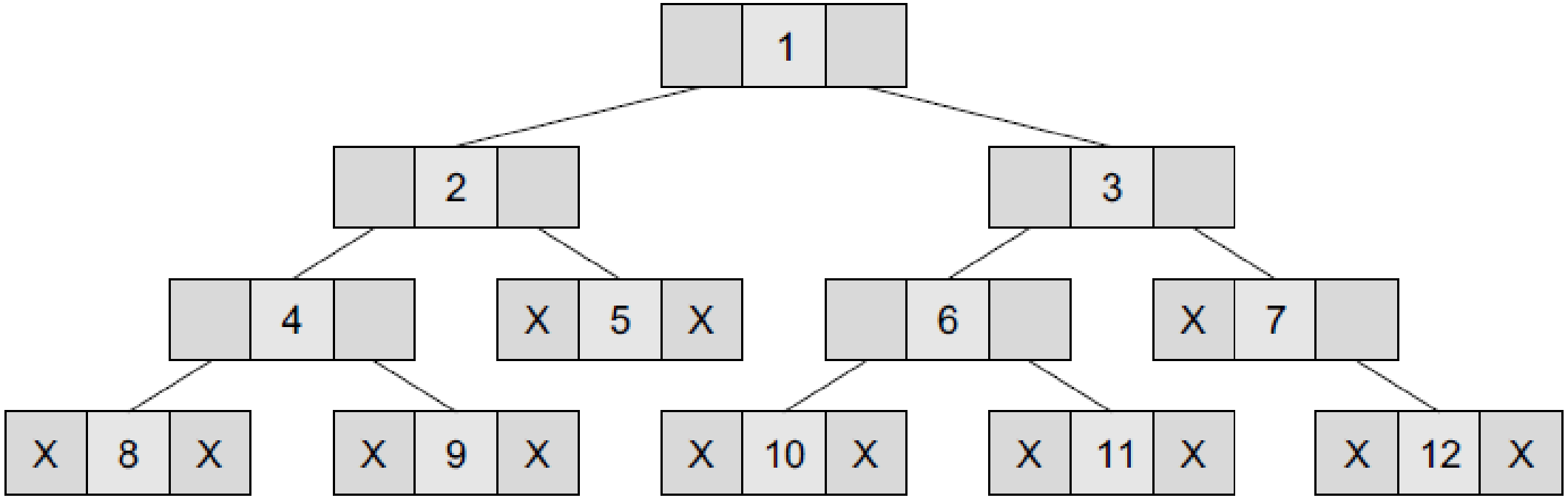
- $Tn=2^0+2^1+2^2+\ldots\ldots 2^d\ldots..(1)$
- $2Tn=2^1+2^2+\ldots\ldots\ 2^{d+1}\ldots.(2)$
- (2)-(1)→
- $Tn=2^{d+1}-1$

Representation of Binary Trees in the Memory

- In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.
- ***Linked representation of binary trees***
- In the linked representation of a binary tree, every node will have three parts:
  - the data element,
  - a pointer to the left node, and
  - a pointer to the right node.

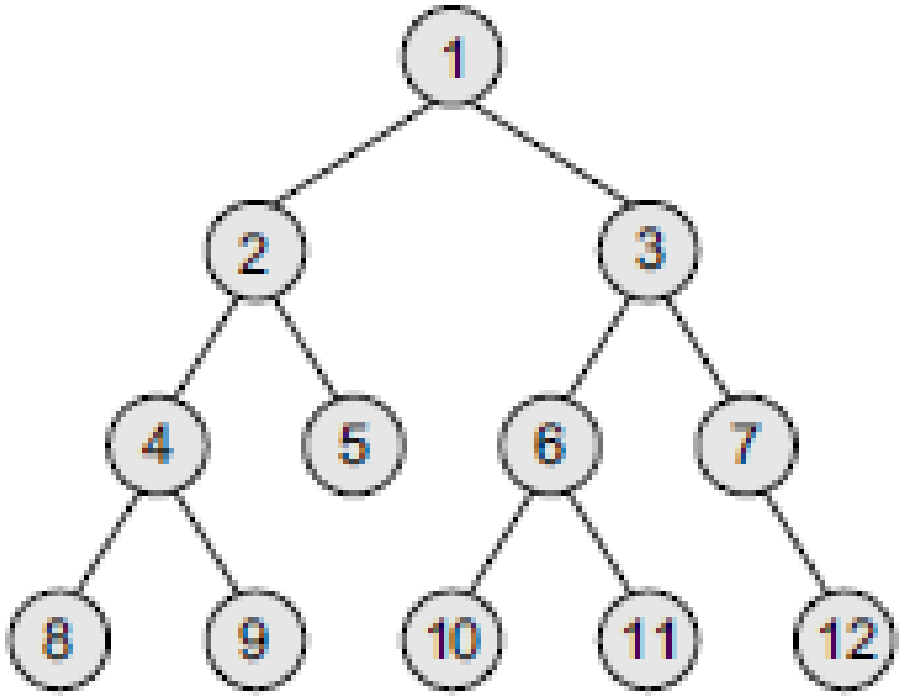
```
struct node {
   struct node *left;
   int data;
   struct node *right;
};
```

Representation of Binary Trees in the Memory (Cont'd)



The above tree is represented in the main memory using a linked list as follows:

Left node index	Current node value	Right node index

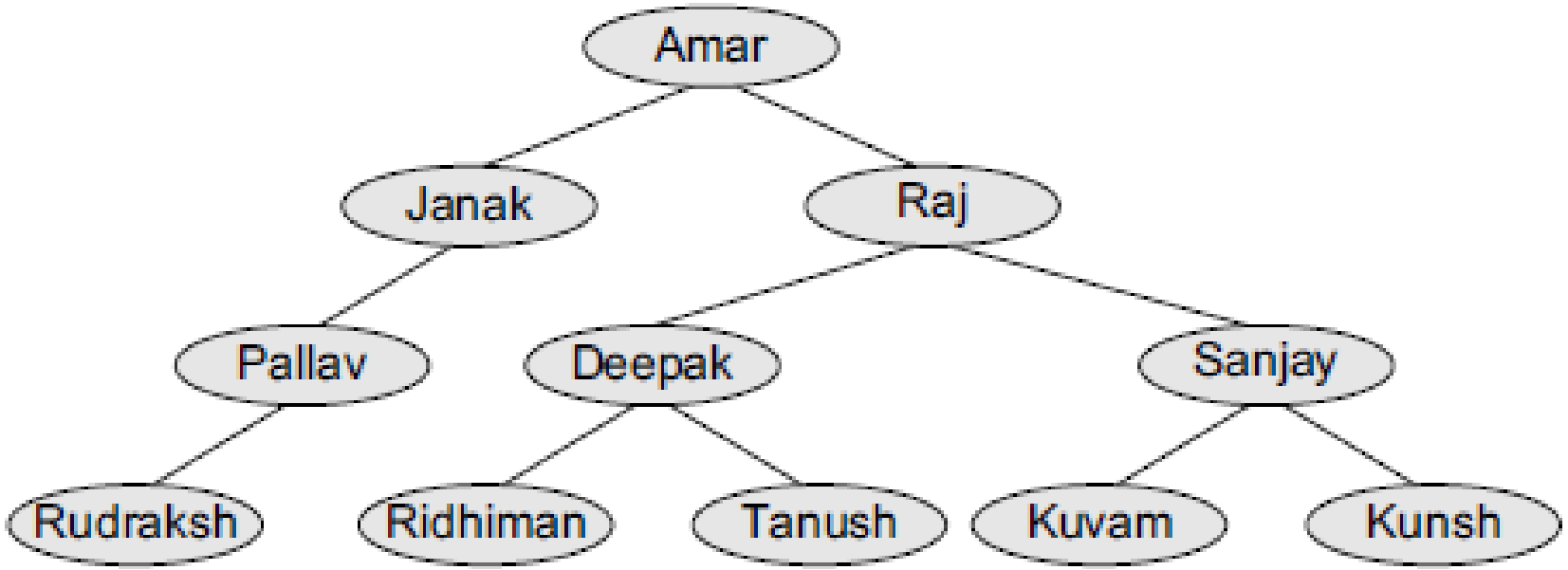


ROOT = 3

AVAIL = 15

	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16

Given the memory representation of a tree that stores the names of family members, construct the corresponding tree from the given data.

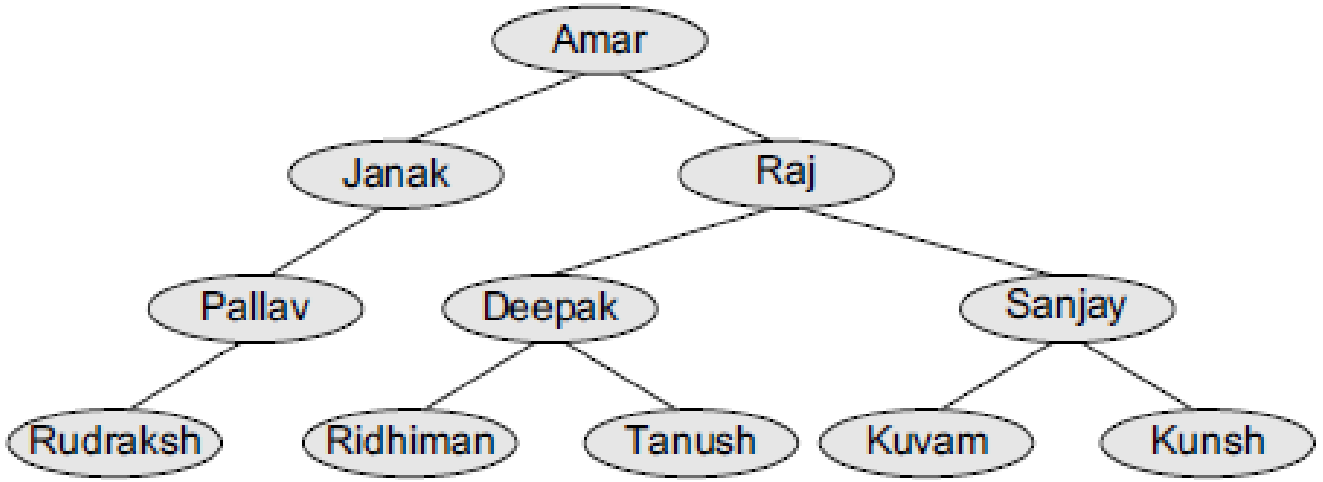


Solution :Given the memory representation of a tree that stores the names of family members, construct the corresponding tree from the given data.

ROOT 3

AVAIL 7

	LEFT	NAMES	RIGHT
1	12	Pallav	–1
2			
3	9	Amar	13
4			
5			
6	19	Deepak	17
7			
8			
9	1	Janak	–1
10			
11	–1	Kuvam	–1
12	–1	Rudraksh	–1
13	6	Raj	20
14			
15	–1	Kunsh	–1
16			
17	–1	Tanush	–1
18			
19	–1	Ridhiman	–1
20	11	Sanjay	15



Sequential representation of binary trees

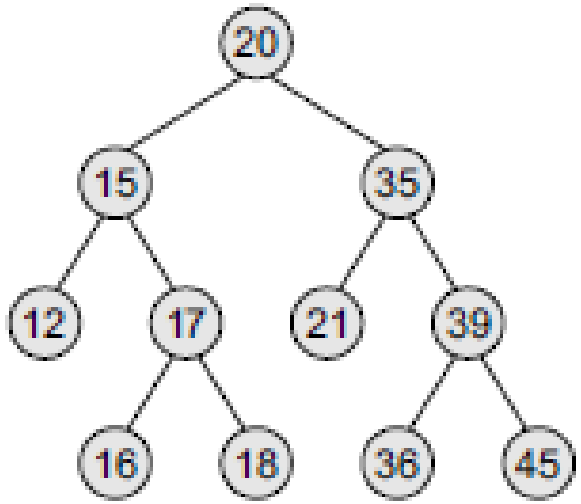
- Sequential representation of trees is done using single or one-dimensional arrays.
- Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space.

How can you represent binary trees in sequential manner ?

Sequential representation of binary trees (Cont'd)

- A one-dimensional array, called TREE, is used to store the elements of tree.
- The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
- The maximum size of the array TREE is given as $(2^{h+1}–1)$, where h is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

The following figure shows a binary tree and its corresponding sequential representation. The tree has 11 nodes, and its height is 3.



1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

Tree Traversals

- A means of visiting all the objects in a tree data structure
- We will look at
  - Breadth-first traversals
  - Depth-first traversals
- Applications
- General guidelines

Background

All the objects stored in an array or linked list can be accessed sequentially.

Question: how can we iterate through all the objects in a tree in a predictable and efficient manner

- Requirements: $Q(n)$ run time and $o(n)$ memory

Types of Traversals

- The breadth-first traversal visits all nodes at depth *k* before proceeding onto depth *k* + 1 (aka Level order traversal)
  - Easy to implement using a queue

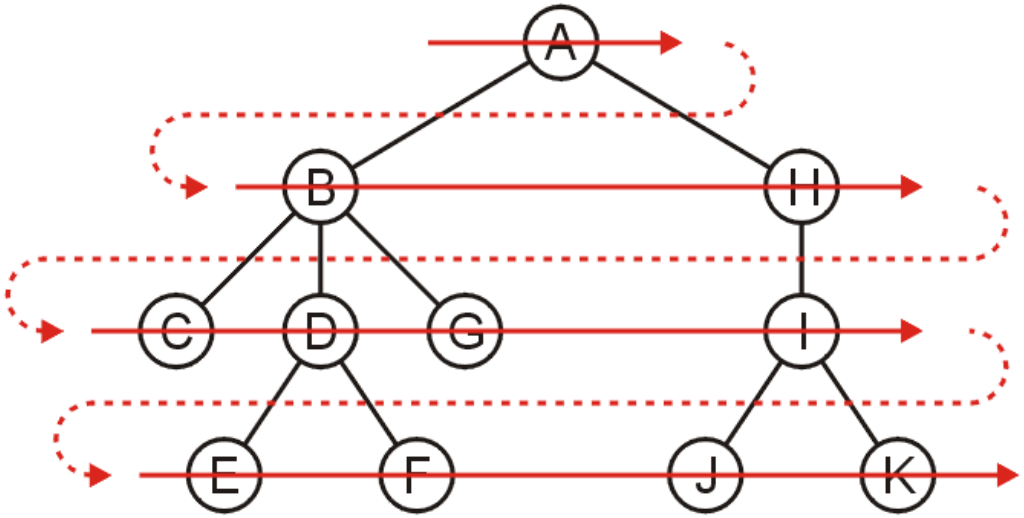
- Another approach is to visit always go as deep as possible before visiting other siblings: *depth-first traversals*
  - *In-order traversal*
  - *Pre-order traversal*
  - *Post-order traversal*

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

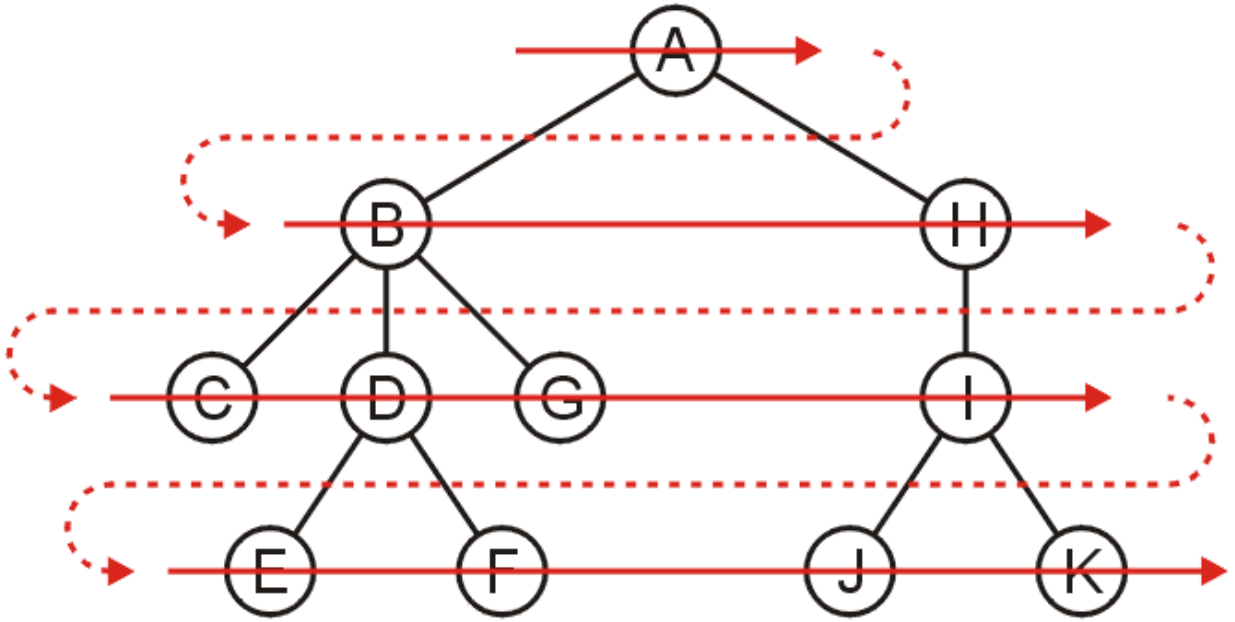
Idea : expand a frontier one step at a time.

- Can be implemented using a queue (FIFO)
- Run time is Q(n)
- Memory is potentially expens
  depth
- Order: A B H C D G I E F J K



Breadth-First Traversal

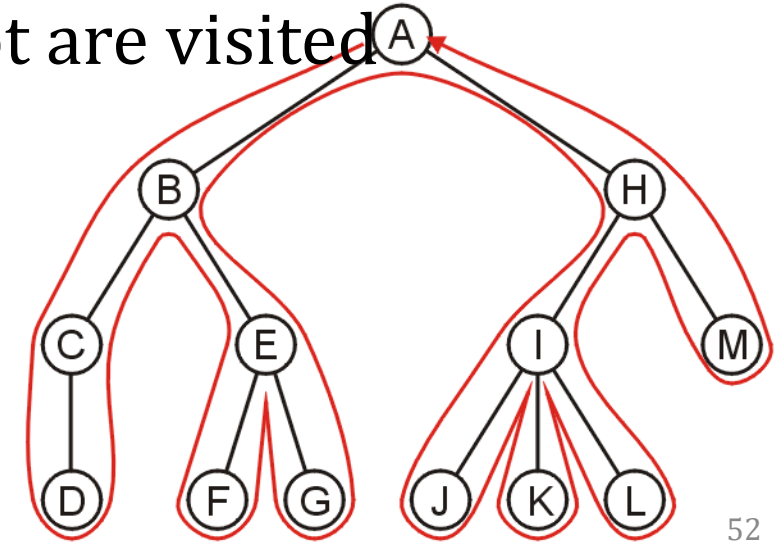
- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node



Backtracking Algorithm

- At any node, we proceed to the first child that has not yet been visited
- Or, if we have visited all the children (of which a leaf node is a special case), we backtrack to the parent and repeat this decision making process.

- We end once all the children of the root are visited.



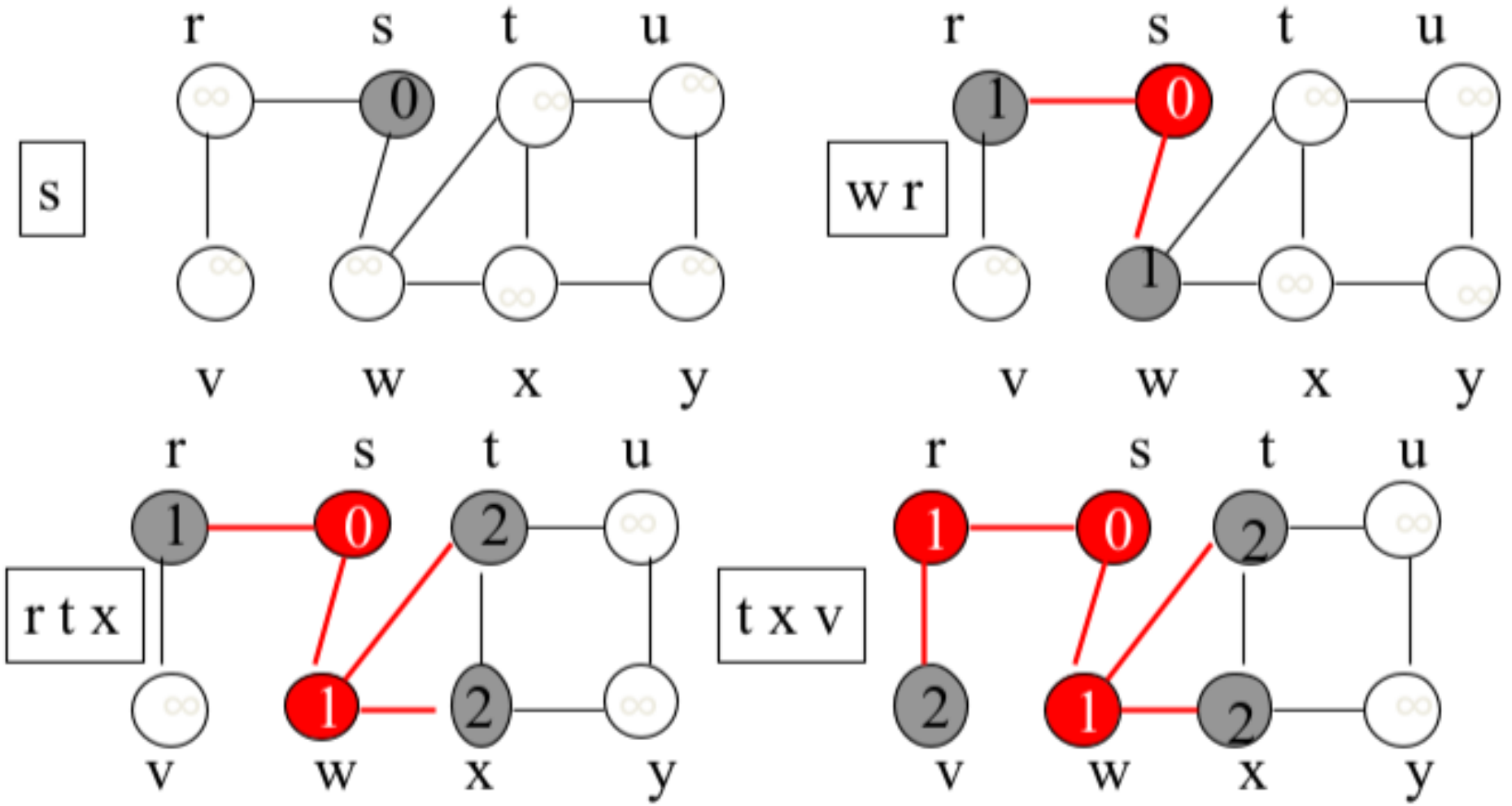
DFS

```
DFS (G:graph; var color:carray;  parent:parray);

    for each vertex u do
            color[u]=white;  parent[u]=nil;
 end for
 for each vertex u do
     if color[u] == white then
          DFS-Visit(u);
     end if
 end for
end DFS
```

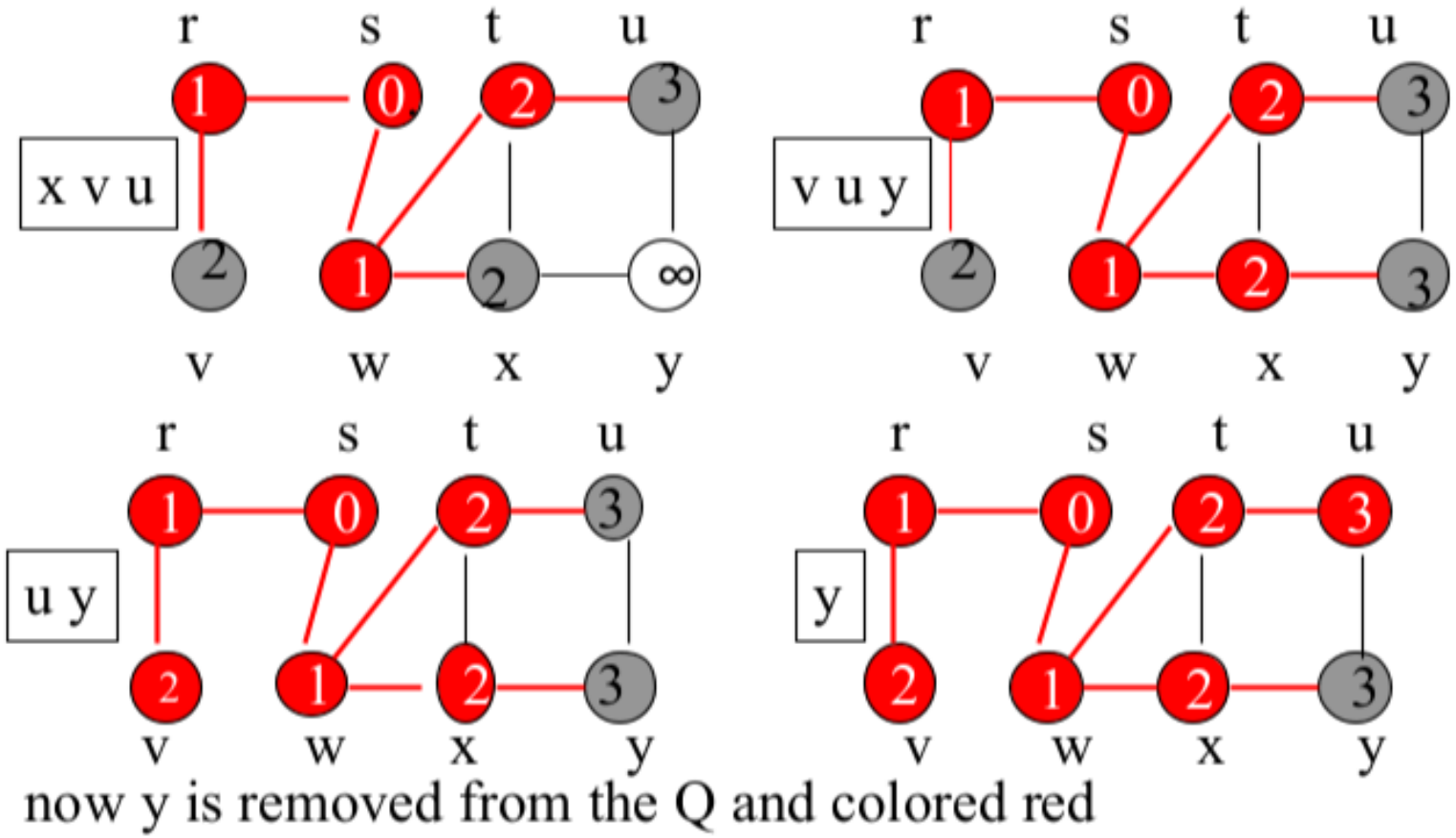
DFS-Visit(u)

```
DFS-Visit(u)
{
 color[u]=gray;

 for each v in adj[u] do
          if color[v] = white {
              parent[v] = u;
              DFS-Visit(v);
          }

 color[u] = red;
}
```

BFS – Example



BFS – Example



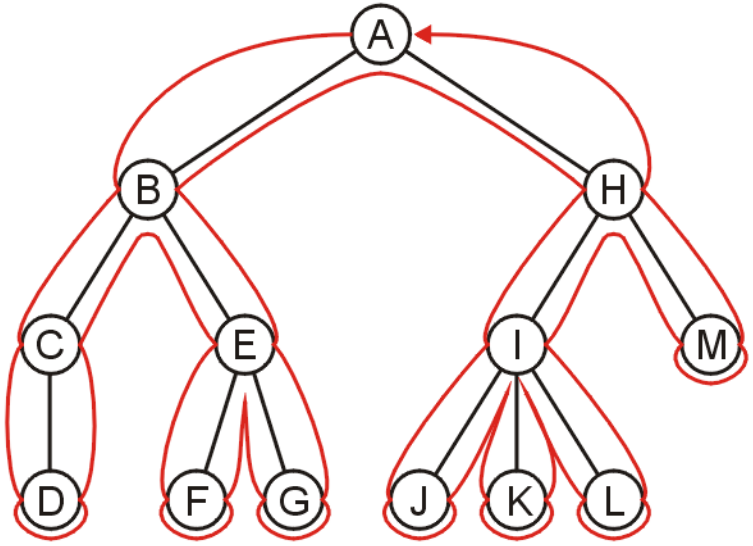
now y is removed from the Q and colored red

Depth-first Traversal

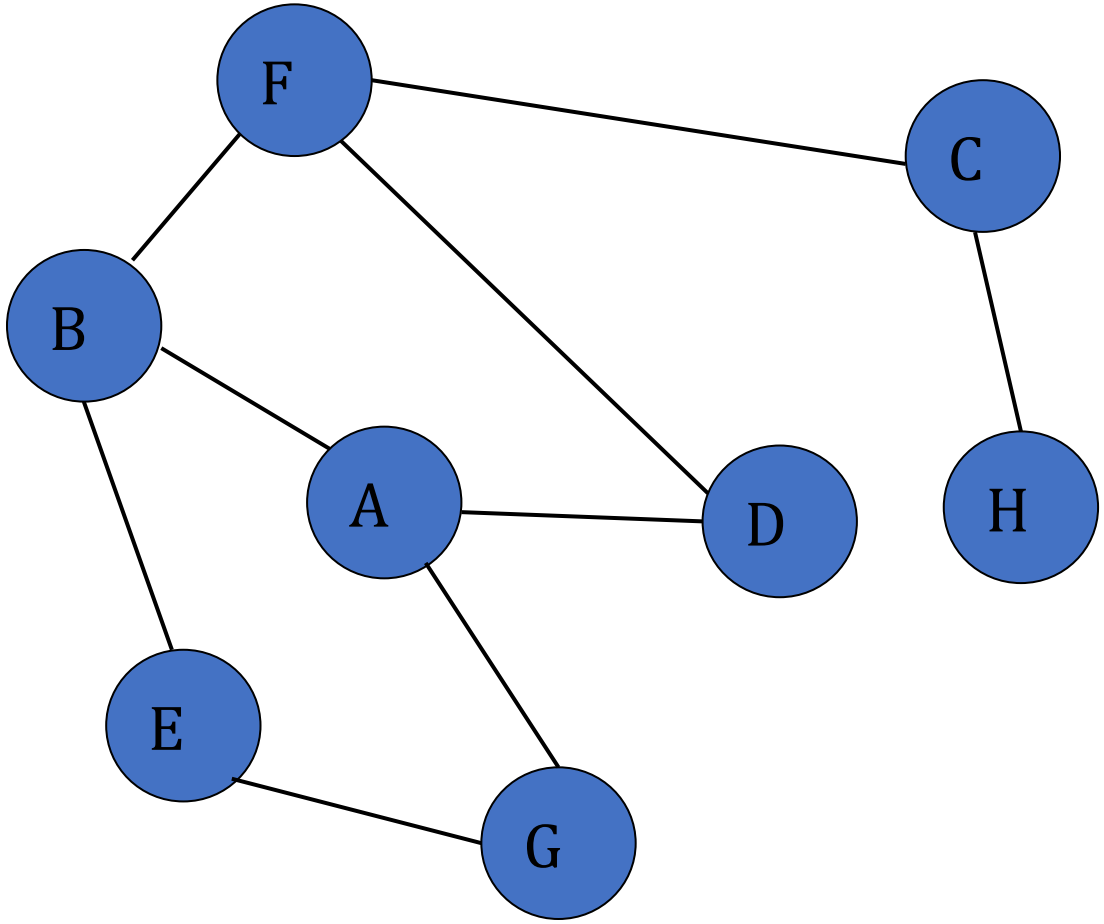
Idea : Explore every node and edge of the graph. We go deeper whenever is possible.

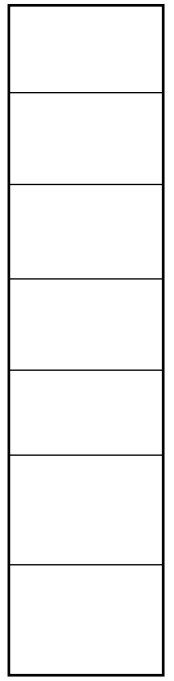
We note that each node could be visited twice in such a scheme

- The first time the node is approached (before any children)
- The last time it is approached (after all children)



Depth-first Traversal

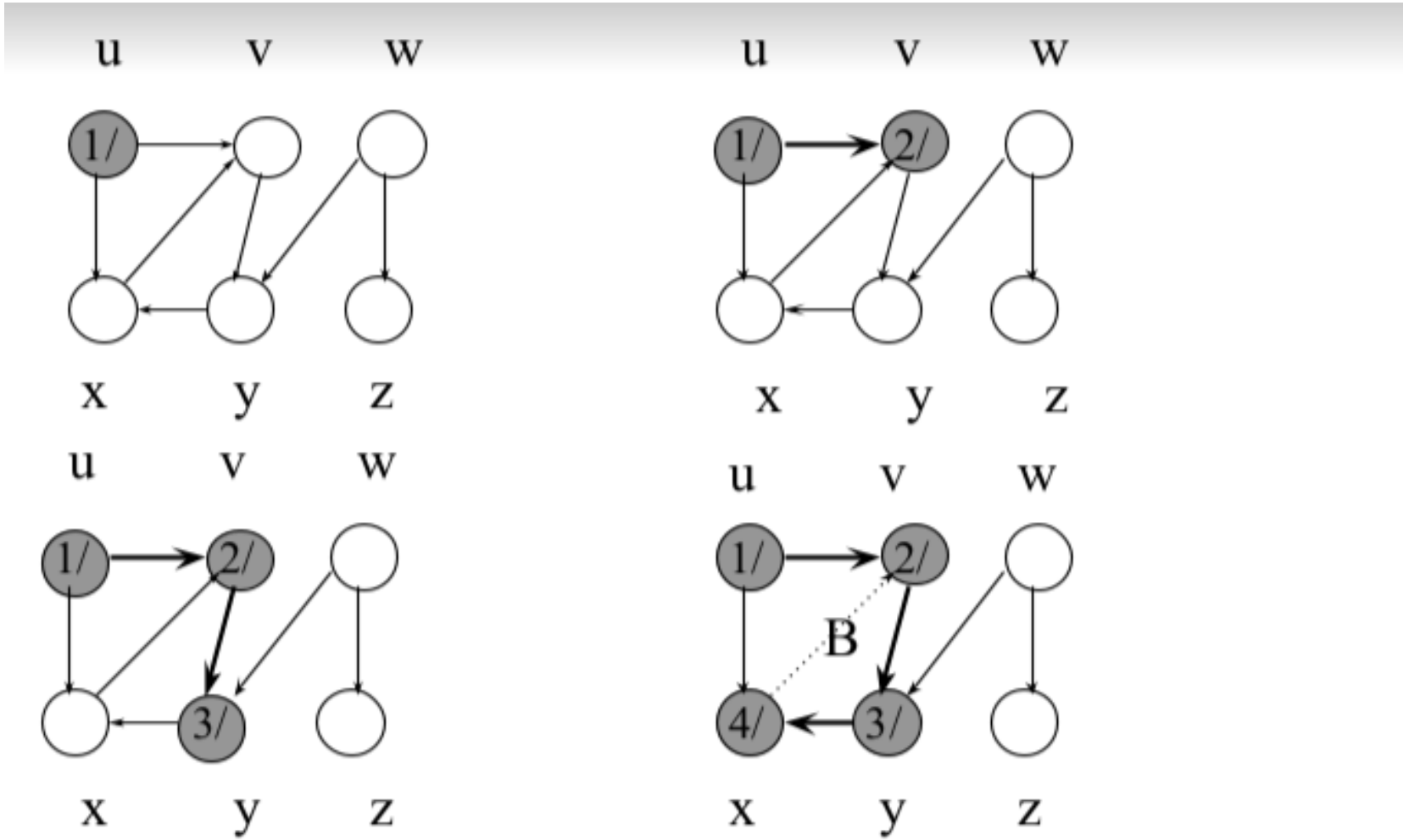




Stack

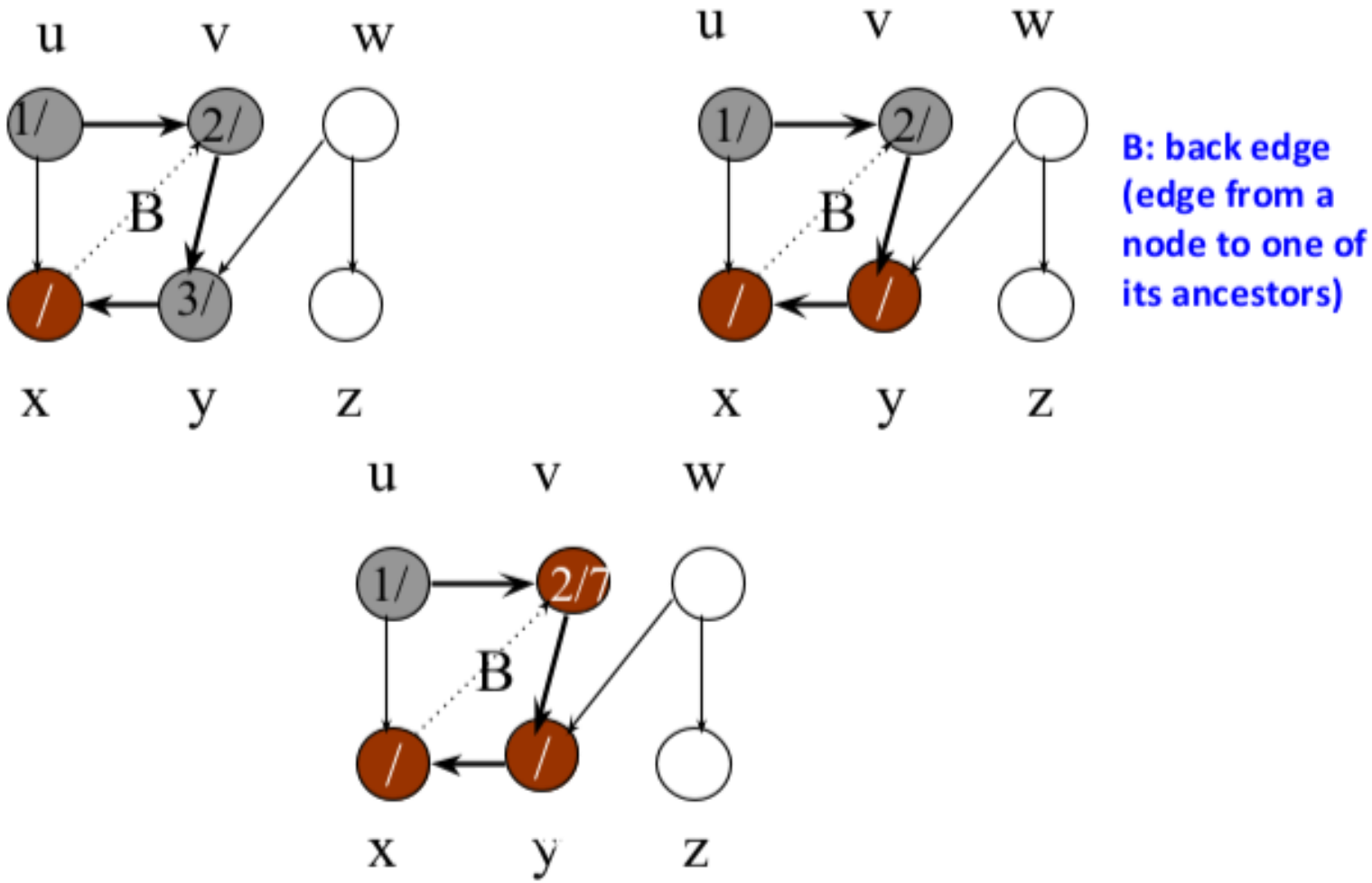
Output : ABEGFCHD

Depth-first Traversal



B: Back edge (edge from a node to one of its ancestors)

Depth-first Traversal

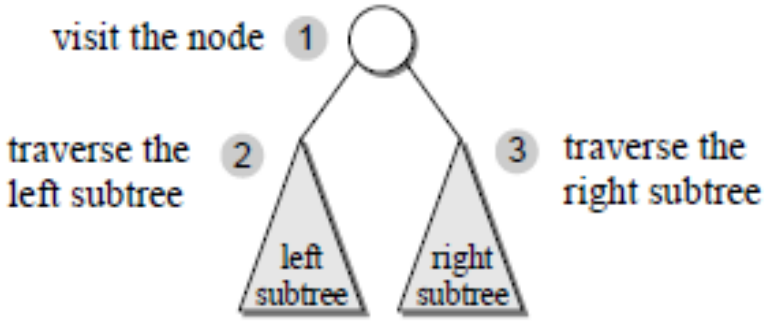


Traversals of Binary Trees

- A traversal of a tree is a systematic way of accessing or “visiting” all the nodes in the tree.
- There are three basic traversal schemes:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

Pre-Order Traversal

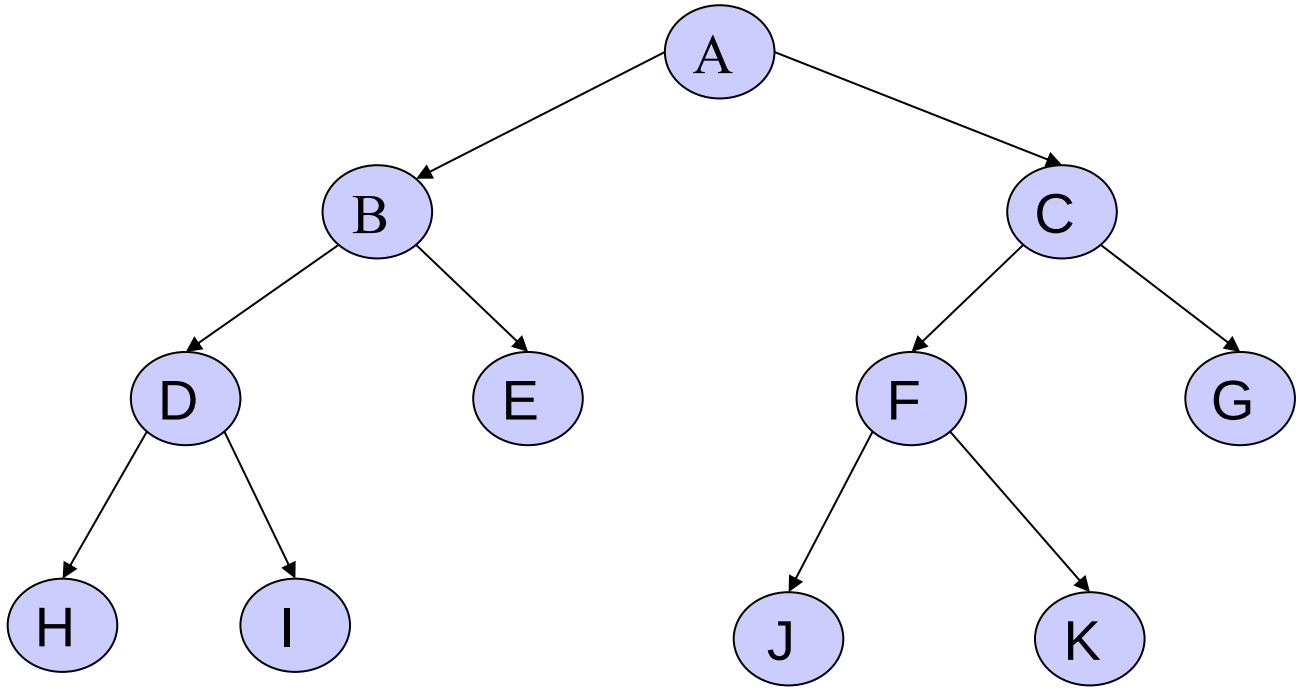
- A pre-order traversal has three steps for a nonempty tree:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.



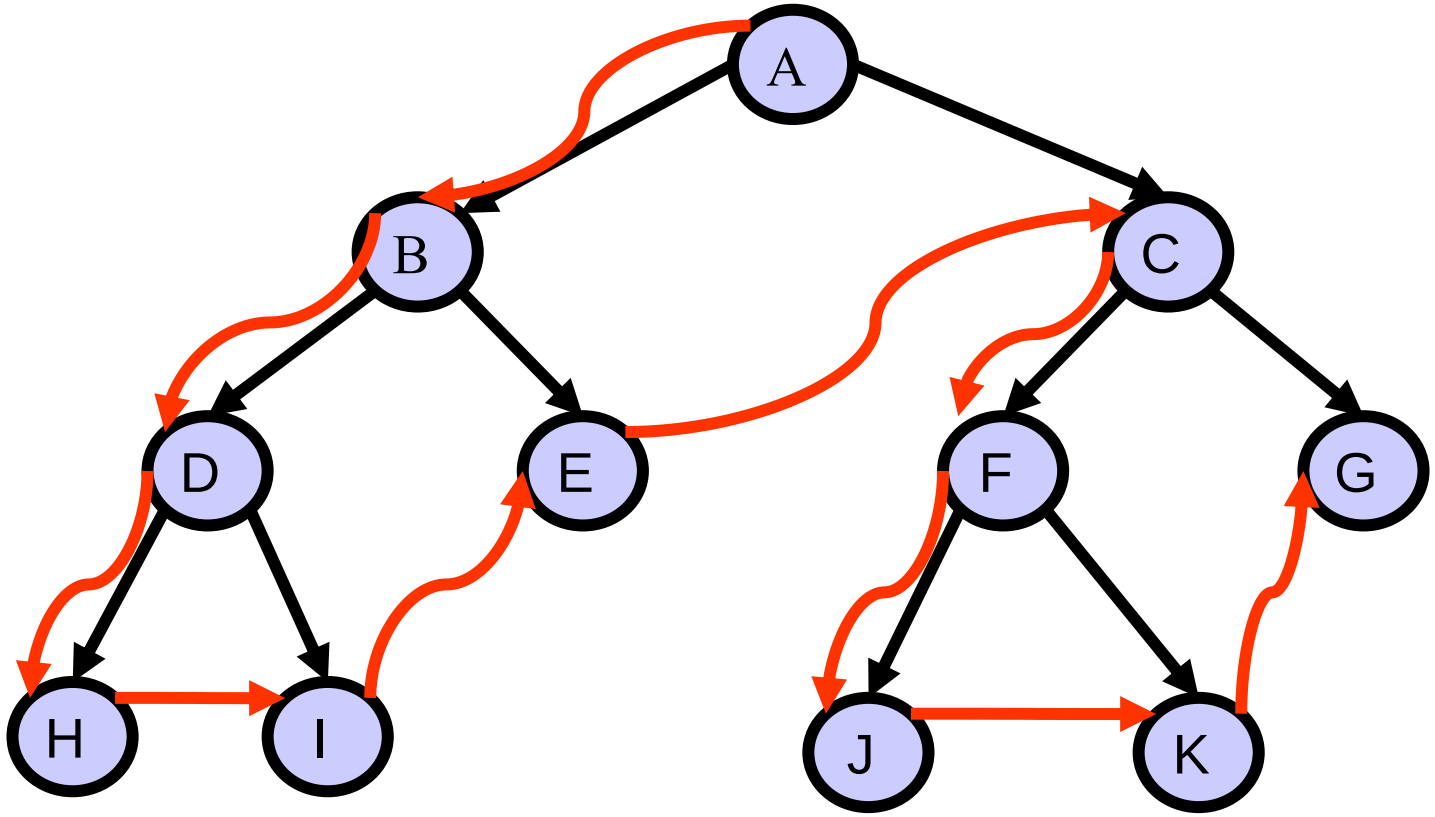
Pre-Order Traversal

- To traverse a non-empty binary tree in pre-order (also known as depth first order), we perform the following operations.
- Visit the root (or print the root)
- Traverse the left in pre-order (Recursive)
- Traverse the right tree in pre-order (Recursive)

Pre-Order Traversal



Pre-Order Traversal



Algorithm for pre-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

In-order Traversal

- An in-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

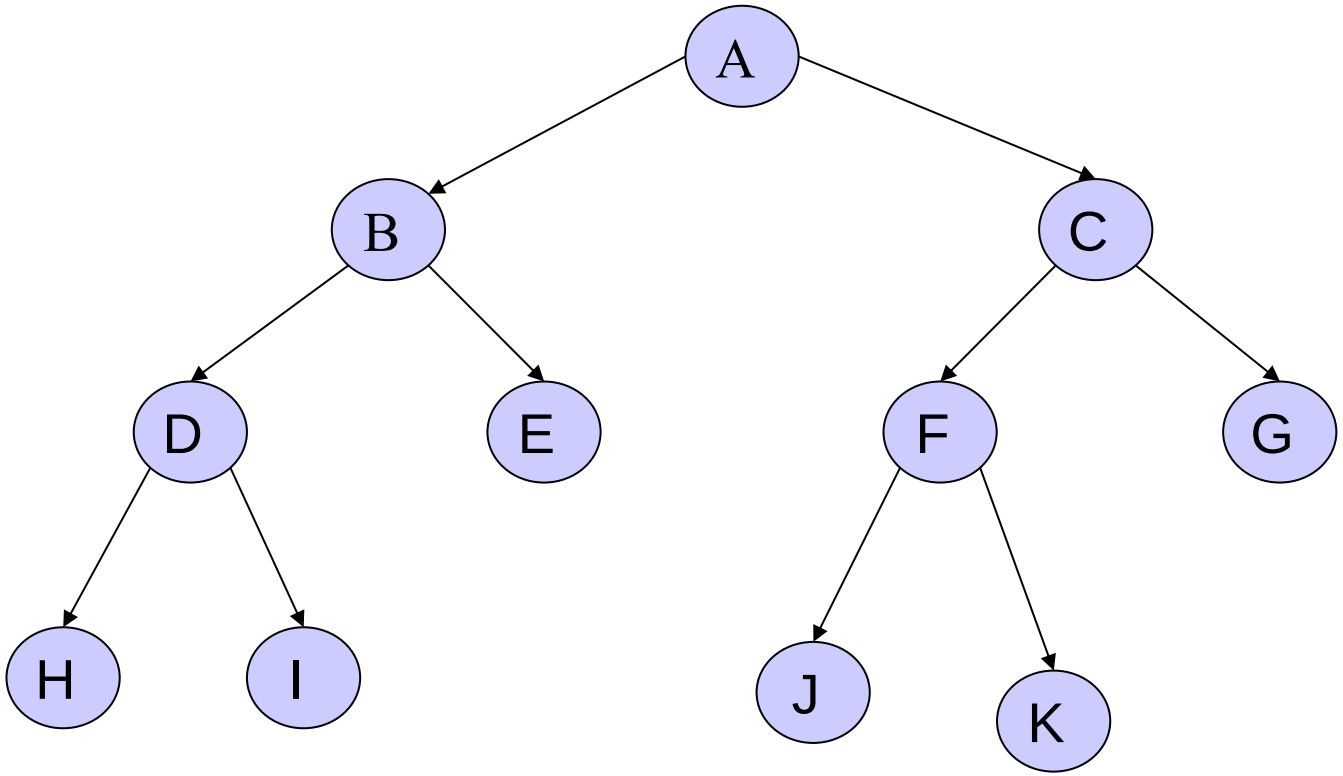
In-order traversal

The in-order listing of the nodes of T is the nodes of T_1 in in-order, followed by n,followed by the nodes $T_1,T_2,....T_n$, each group of nodes in in-order.

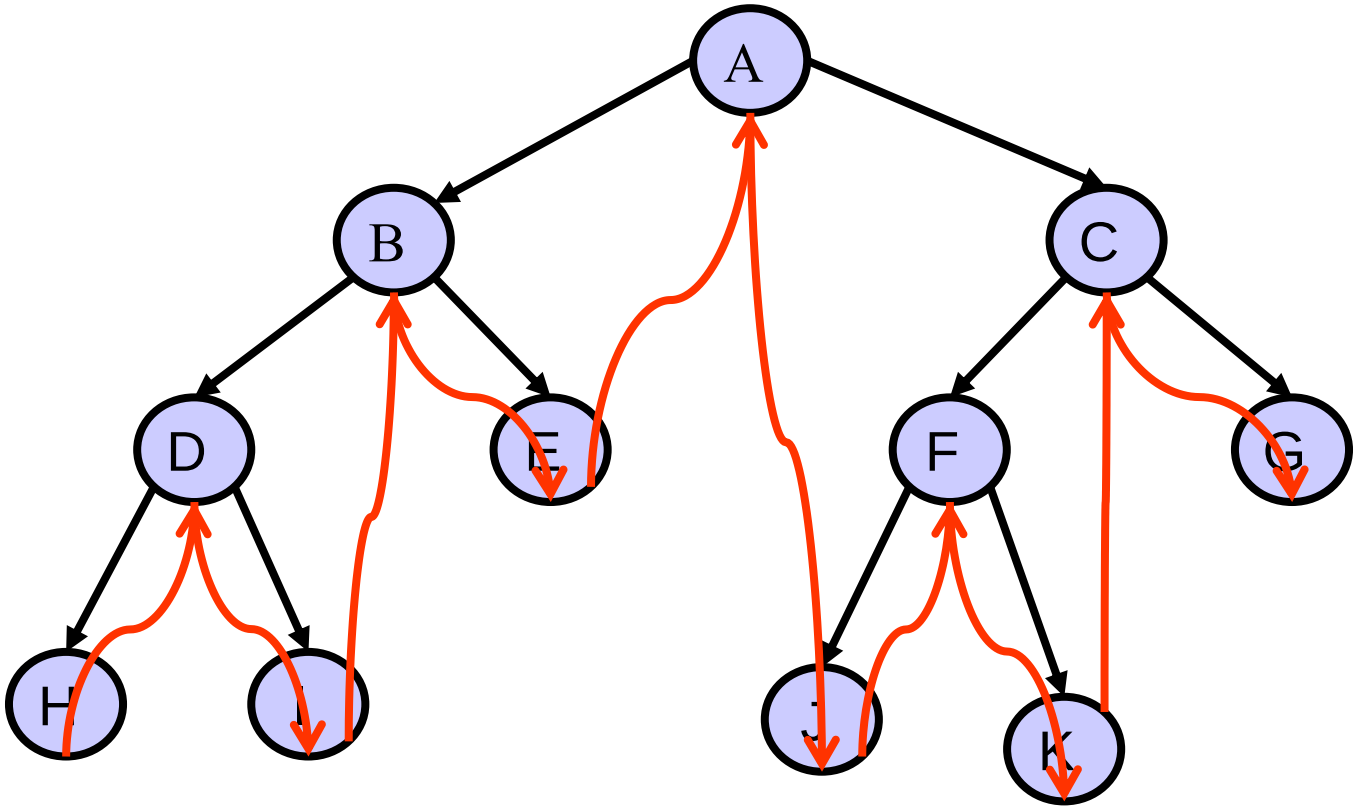
Algorithm :

- Traverse the left-subtree in in-order
- Visit the root
- Traverse the right-subtree in in-order.

In-order traversal



In-order traversal



Algorithm for in-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Post-order Traversal

- A post-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.

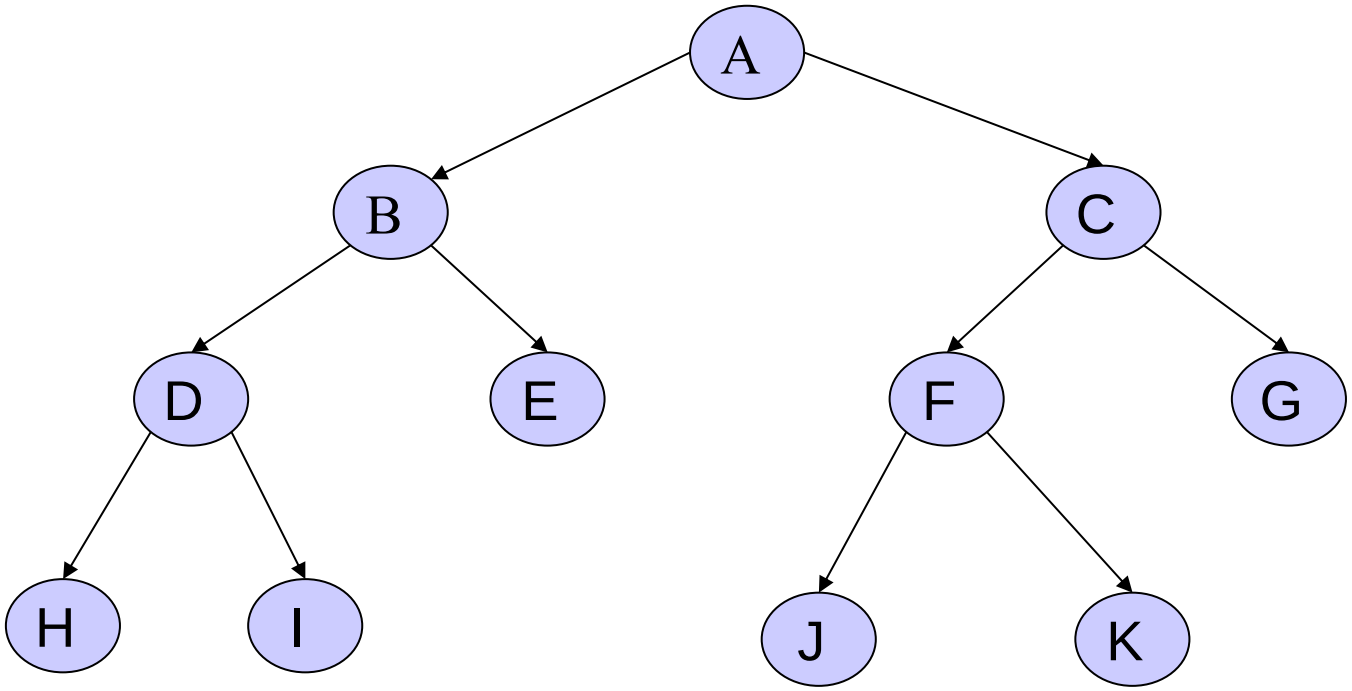
Post-order Traversal

The post-order listing of the nodes of T is the nodes of T_1 in post-order, then the nodes of T_2 in post order and so-on up to T_k, all followed by n

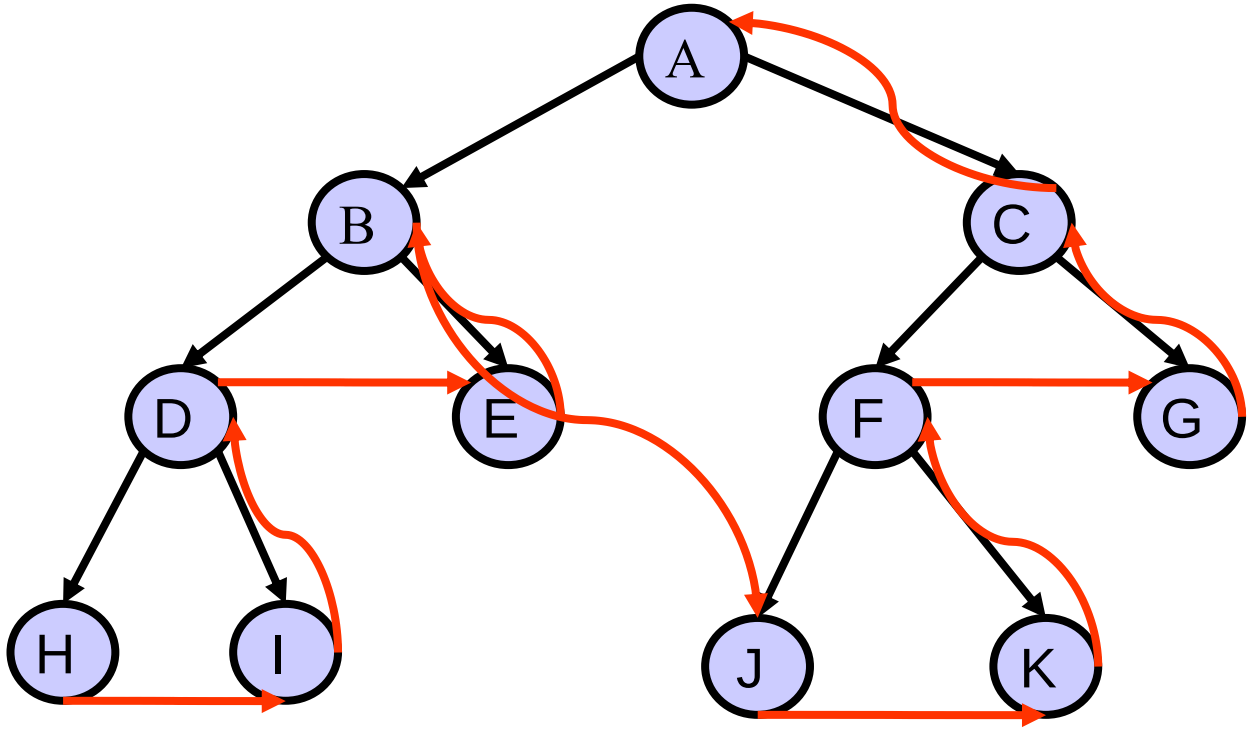
Algorithm

- Travers the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- Visit the root

Post-order Traversal



Post-order Traversal



Algorithm for post-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

Thank you