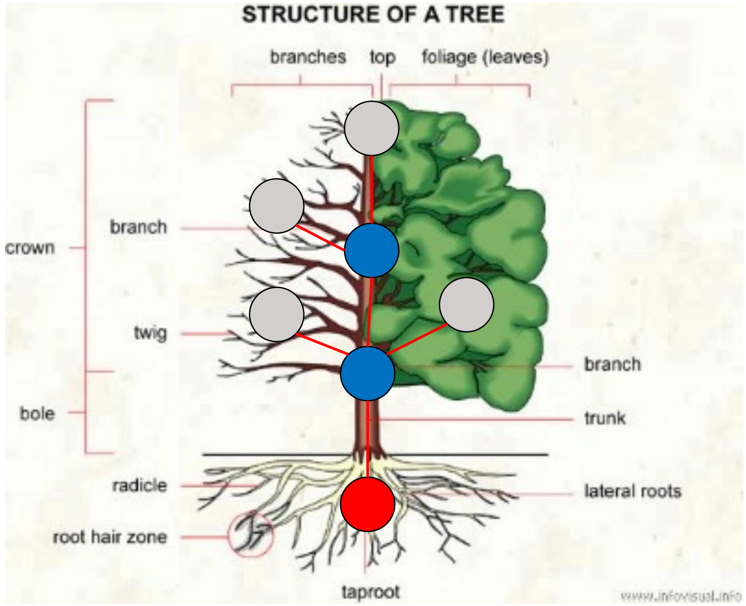
# [CSED233-01] Data Structure

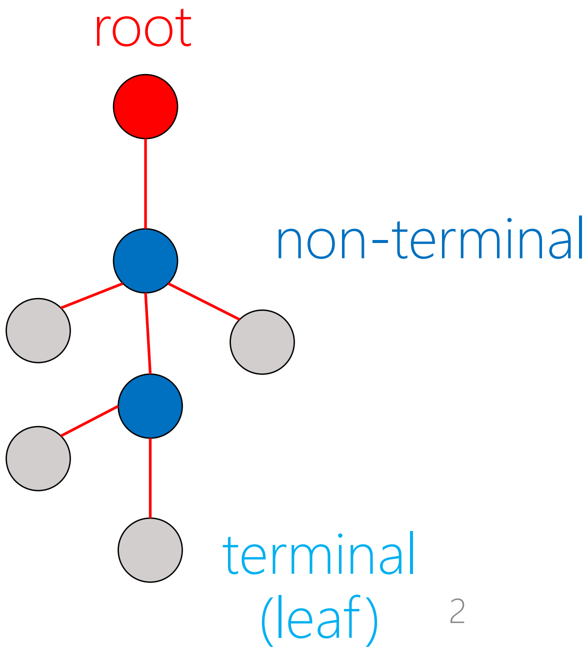
# Tree (1)

Jaesik Park

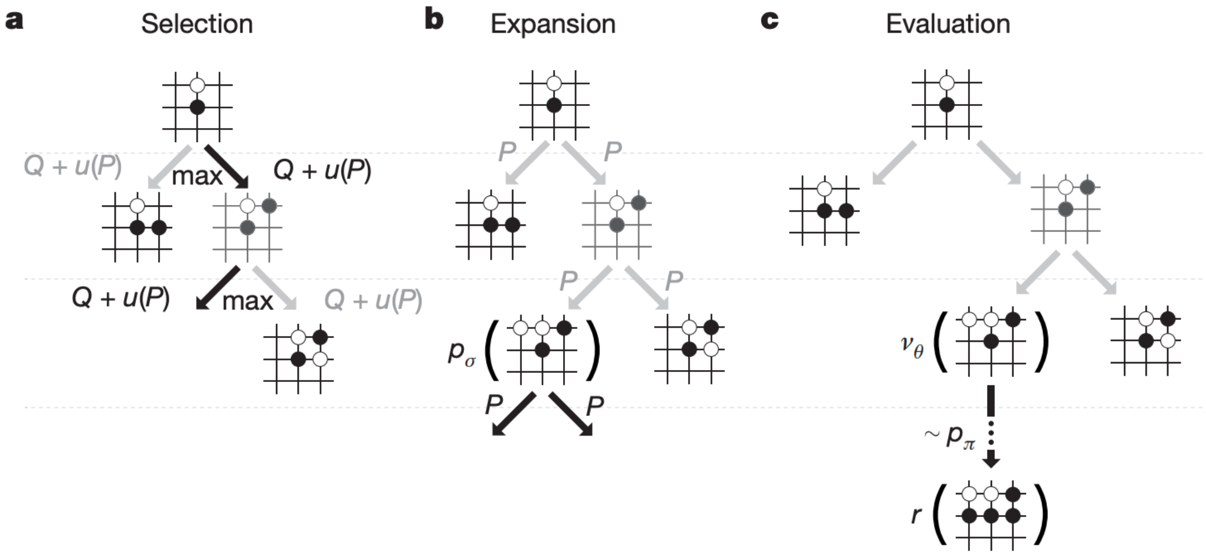
# Tree

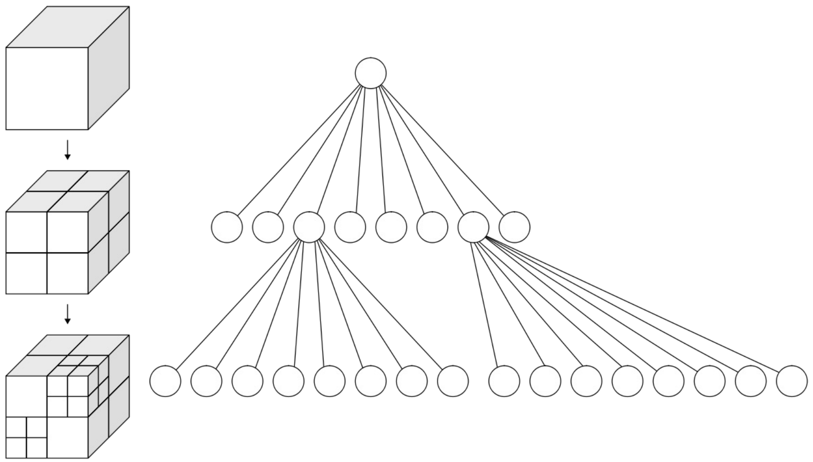
- A collection of nodes and edges
  - with one node distinguished as a "*root*"
  - along with "*parenthood*" relation
  - One edge connects two nodes ☺
  - Tree is a type of graph, which we will learn later
- Each node in the tree can be connected to many children, but must be connected to exactly one parent, except for the *root* node
- No cycles or "loops"



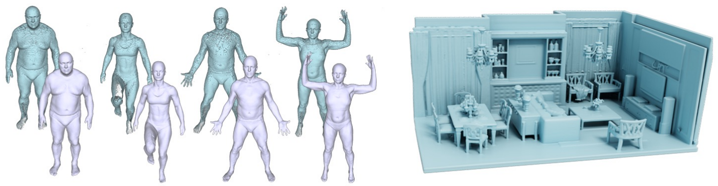
# Trees in Computer Science

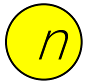
Dual Octree Graph Networks for Learning Adaptive Volumetric Shape Representations

PENG-SHUAI WANG, Microsoft Research Asia, China
YANG LIU, Microsoft Research Asia, China
XIN TONG, Microsoft Research Asia, China



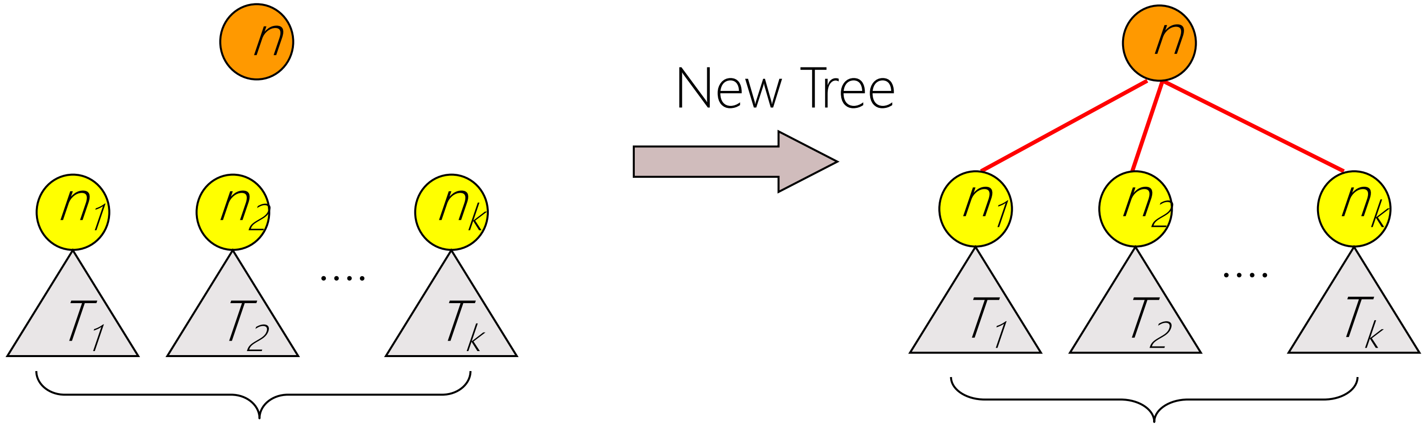
# Tree: Recursive Definition

- Starts with a single node




Single node

- We can build a new tree by making other trees as subtrees of the single node




Trees with roots $n_1, n_2, \dots, n_k$

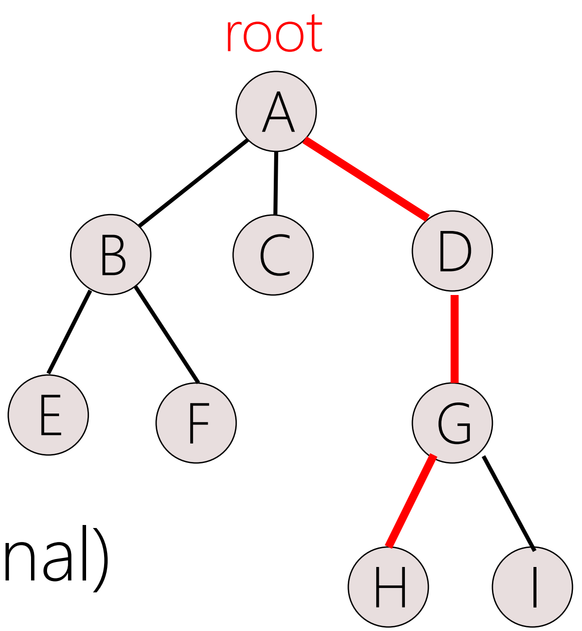
Subtrees

# Parenthood Relations

- Parent/child: A is the parent of C, C is the child of A
- Ancestor/descendant: A is the ancestor of G, G is the descendant of A
- Siblings: B,C,D are siblings

- Path $<n_1, n_2, \ldots, n_k>$
  - $n_i$ is the parent of $n_{i+1}$ $(1 \leq i < k)$
  - length = $k$ -1
    - Number of edges connecting the path

- Terminal (leaf, external) ↔ Non-terminal (internal)
  - Whether the node has any child nodes
  - E,F,C,H,I ➔ leaf
  - B,A,D,G ➔ Non-terminal

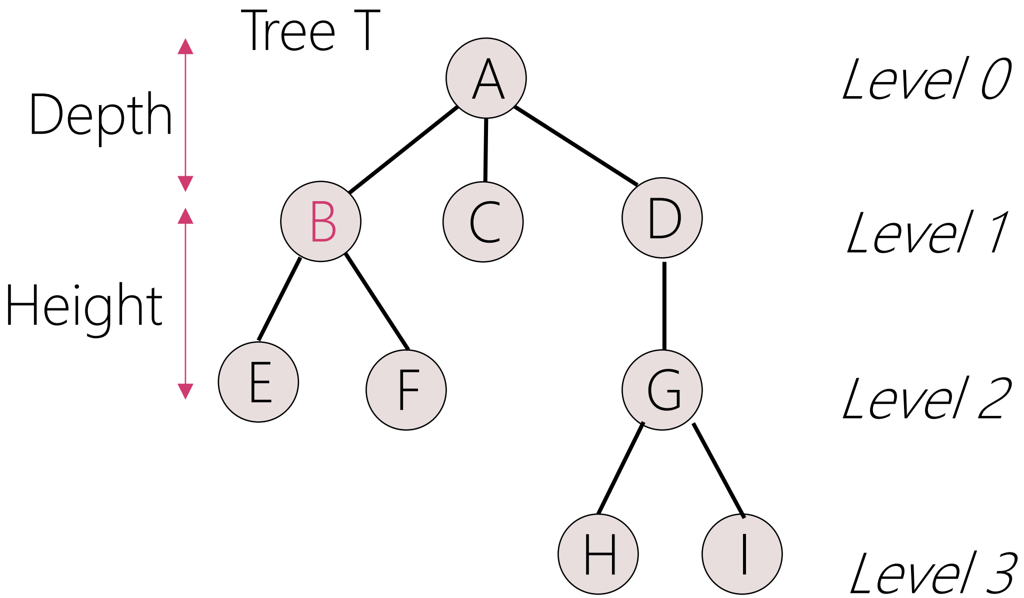



*Path from A to H*

*(length = 3)*

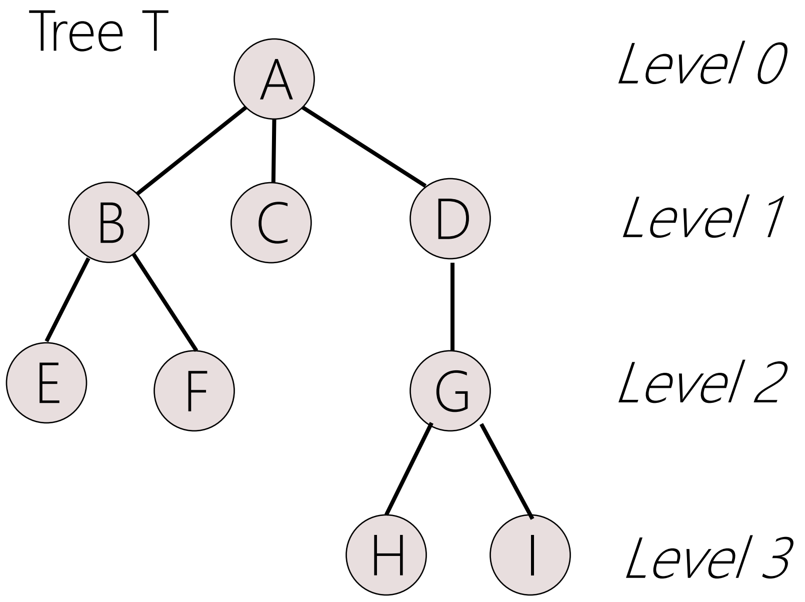
# Depth, Height, Level

- Depth/Height
  - Depth of node *M*
    - the length of the path from the root to *M*
    - the # of ancestors of *M* (excluding *M* itself)
  - Height of node *M*
    - the length of a longest path from *M* to a leaf
  - Height of *tree*
    - the height of the root
- Level
  - Root is at level 0
  - Its children are at level 1
  - Their children are level 2, and so on



# Depth, Height, Level

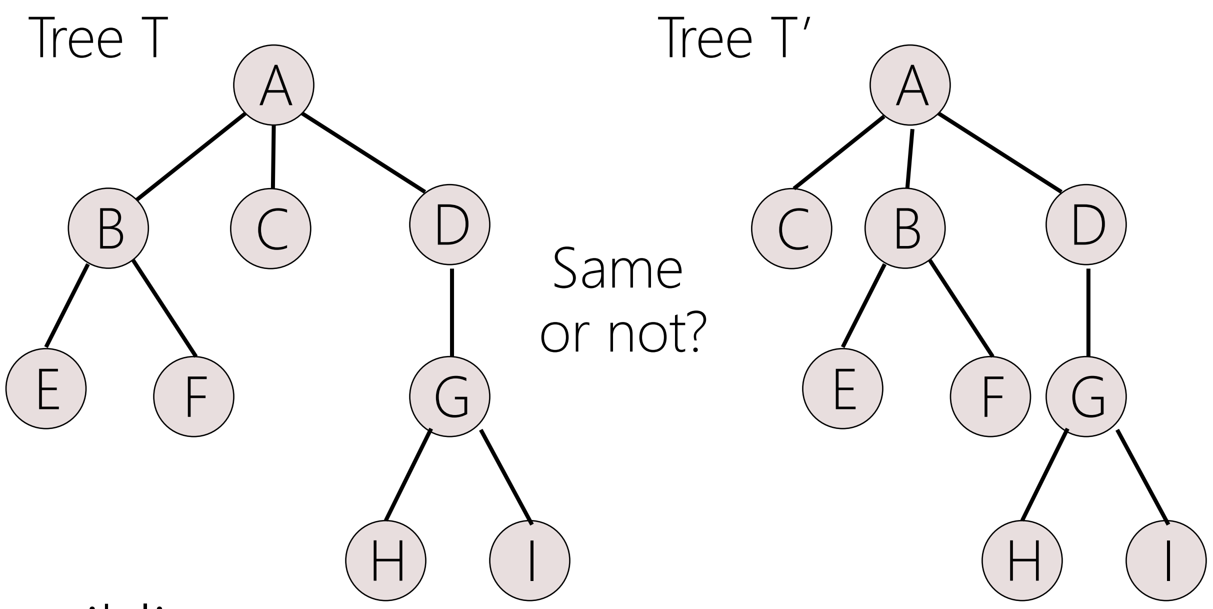
- Examples
  - *Depth* (node *G*) = 2
  - *Height* (node *G*) = 1
  - *Height* (node *F*) = 0
  - *Height* (tree *T*) = 3




- *Different definition* of height & depth
- Level number could start at 1 (rather than 0)
  - Again, no rule here

# Degree, Ordered, Forest

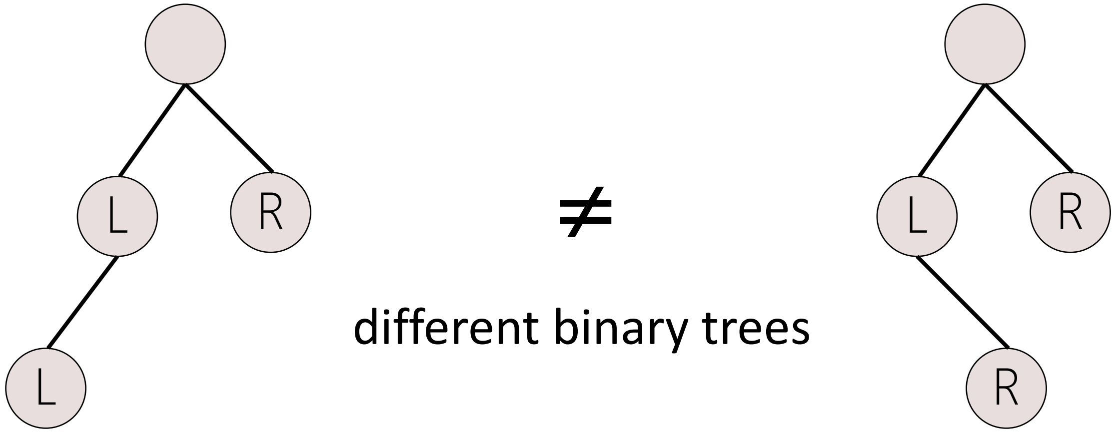
- Degree of node
  - # of children
  - Degree (leaf) = 0
- Degree of tree
  - maximum of its node degrees
- Ordered tree
  - If there exists a linear ordering between siblings
- Forest
  - A collection/set of trees ☺




# Binary Trees

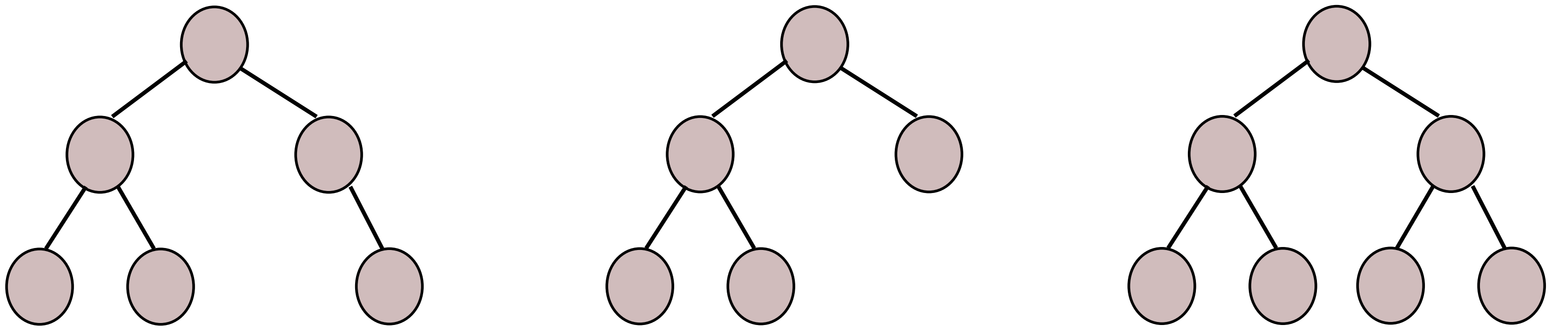
- Binary tree
  - Every node has at most two children
  - Each child is designated as a left child or a right child

- Examples:



# Proper, Full Binary Trees

- Proper
  - if each node has either zero or two children
- Full
  - If it has a maximum # of nodes at each level
  - A full binary tree of height $h$ has $(2^{h+1} - 1)$ nodes



*proper (not full)*

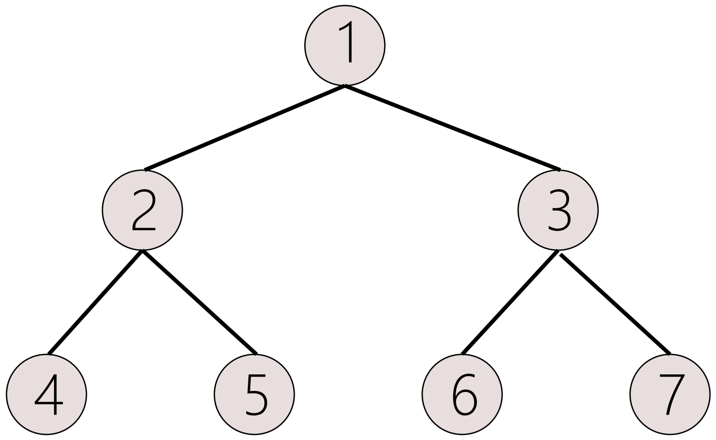
*full*

*Height = 2*

- Caution: *different definitions* in some texts
  - In our text, proper binary trees are also known as full binary trees

# Node Numbering

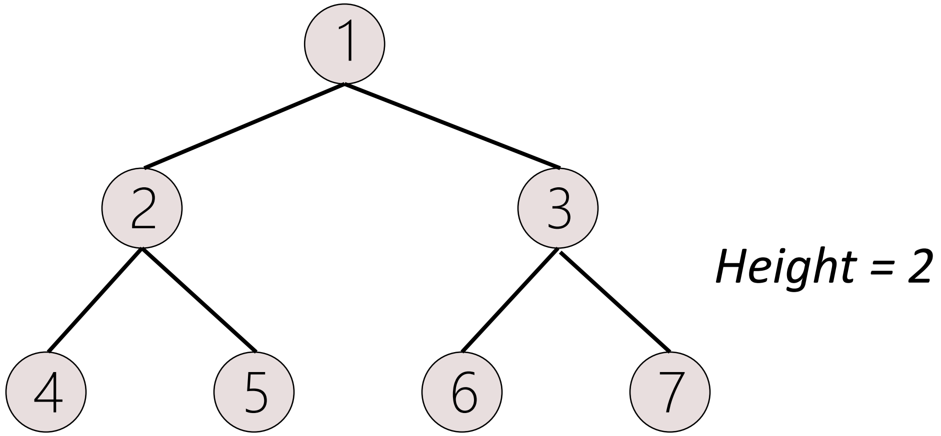
- How to assign numbers to nodes (in a full binary tree)
  - Number the nodes 1 through $2^{h+1} - 1$
  - Number by levels from top to bottom
  - Within a level, number from left to right




*Height = 2*

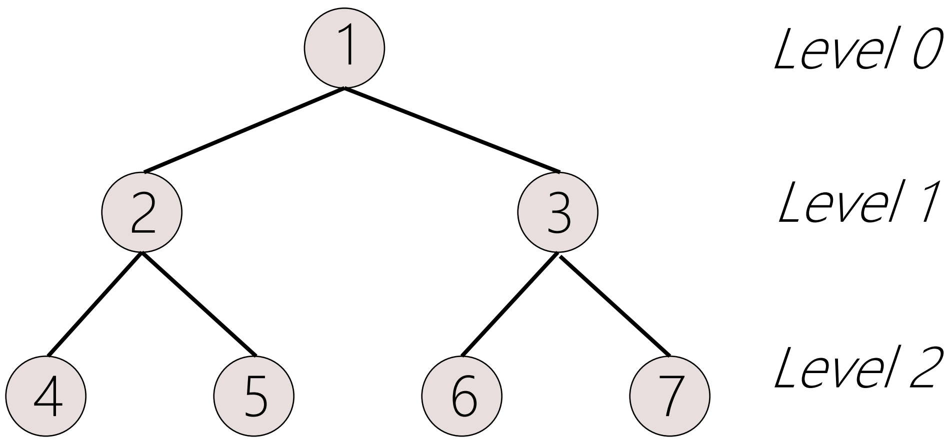
# Node Numbering

- Properties of node numbers
  - Node 1 is the root
  - *Parent* (node $k$)
    - node $\lfloor k/2 \rfloor$ (if $k \neq 1$)
  - *Left_child* (node $k$)
    - node $2k$ (if $2k < n$)
  - *Right_child* (node $k$)
    - node $2k + 1$ (if $2i + 1 < n$)
  - *Left_sibling* (node $k$)
    - node $k - 1$ (if $k$ is odd)
  - *Right_sibling* (node $k$)
    - node $k + 1$ (if $k$ is even and $k+1 < n$)

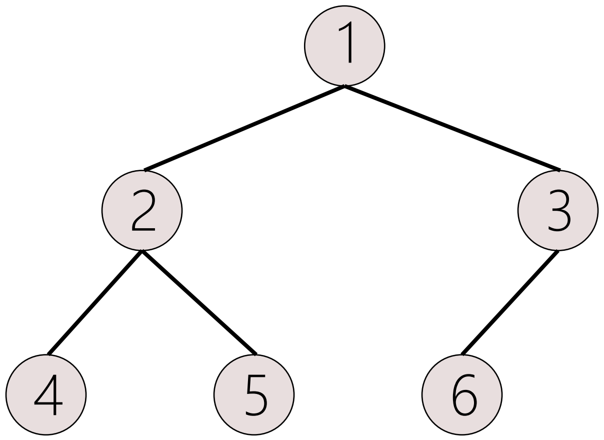


# Complete Binary Tree

- Relaxed definition of a full binary tree
- A binary tree of height *h* is complete, if
  - All levels (possibly except *h*) are completely full
  - Level *h* (leaf level) is filled from left to right




**Height = 2**




*full binary tree with 7 nodes*

*Complete binary tree with 6 nodes*

# Binary Tree Implementations

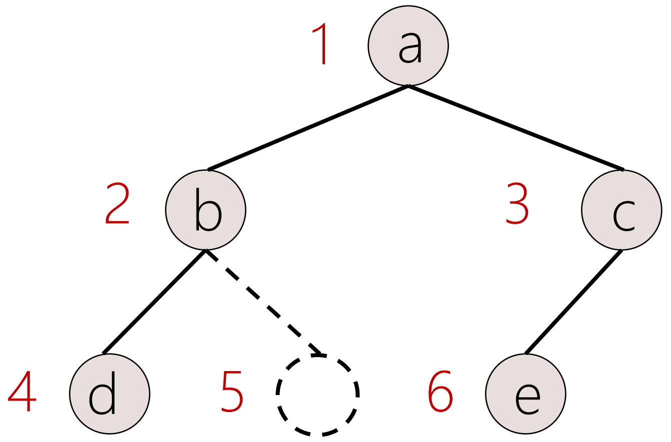
- Array implementation
  - Elements for a complete binary tree
  - Space waste when many nodes are missing
  - char binary_tree[123];

- Linked implementation
  - The most popular way
    - Each node has two pointer fields
  - struct elem{

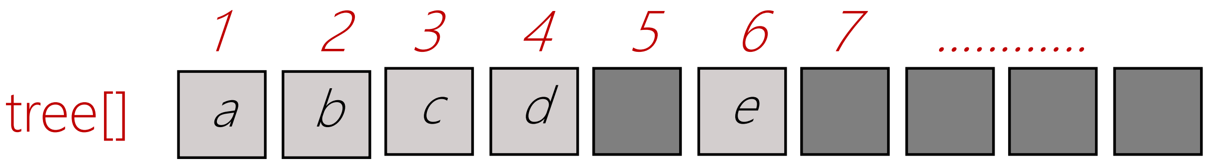
```
    char val;
    elem* left;
    elem* right;
}
```

# Array Implementation: Binary

- Using the numbering scheme for a full binary tree
- The node numbered *k* is stored in an array tree[*k*]

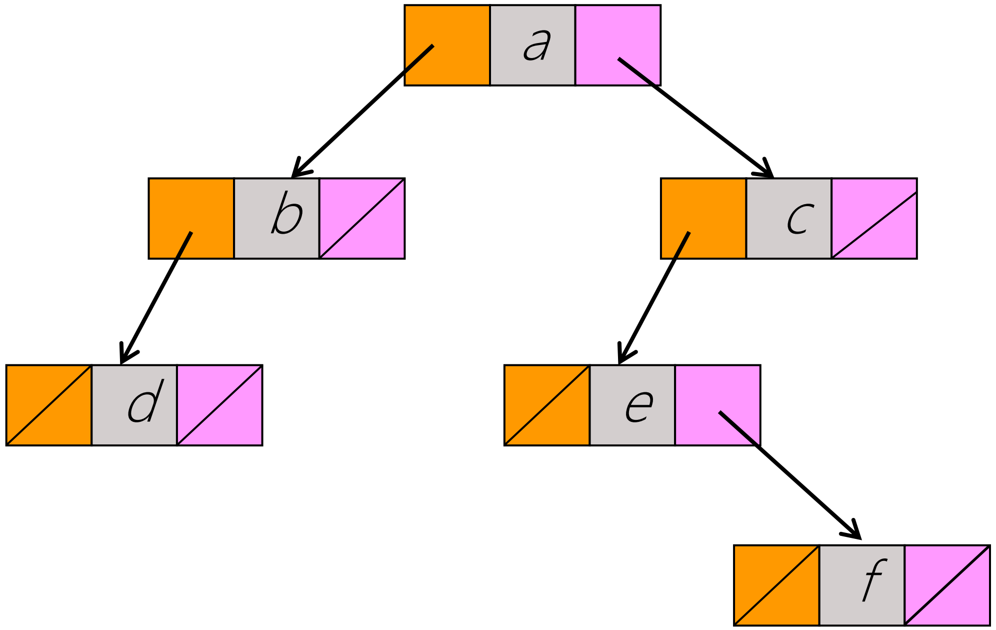



*Complete binary tree with 6 nodes and 1 missing one*



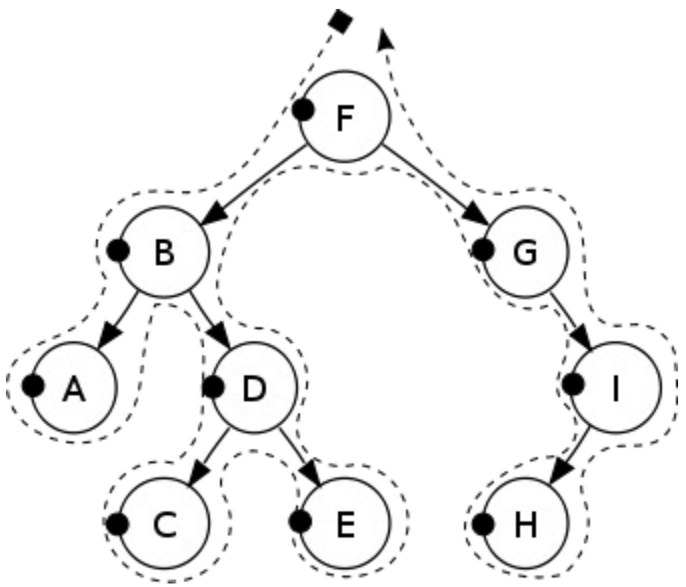
# Linked Implementation: Binary

- Each binary tree node has
  - Value field
  - Two pointer fields (for left & right children)



# Tree Traversal

- *Passing* through the tree & *visit* each of its nodes
  - *Visiting* each tree node *exactly once* in a systematic way
    - During the visit, actions are taken
      - Update, check, evaluate, ….
  - Linearization of tree




- Traversal: F B A D C E G I H (visiting *once*)
  - This is called depth-first search

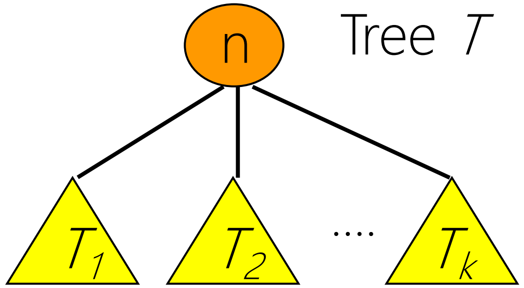
- Recursive function call can implement the tree traversal

```
Func traverse (node){

  if the node have a left child
    call traverse for the left child;

  if the node have a right child
    call traverse for the right child;


}
```

Call stack!

# Tree Traversal

- Types of traversals




Preorder →

Inorder →

Postorder →

```
Func traverse (node){

  if the node have a left child
    call traverse for the left child;

  if the node have a right child
    call traverse for the right child;

}
```

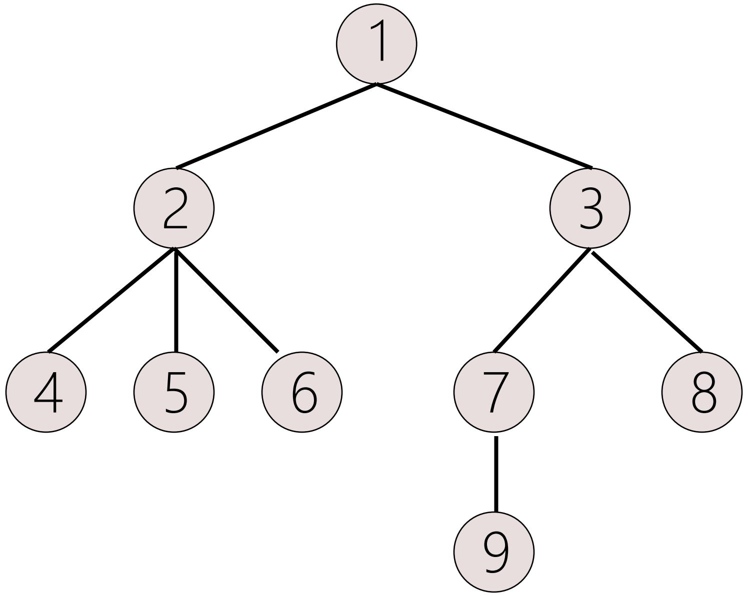
- $Preorder(T) = <n,\ Preorder(T_1), \dots, Preorder(T_k)>$
- $Postorder(T) = <Postorder(T_1), \dots, Postorder(T_k),\ n>$
- $Inorder(T) = <Inorder(T_1),\ n,\ Inorder(T_2), \dots, Inorder(T_k)>$
  - No natural definition of *Inorder*

# Tree Traversals: Examples

- *Preorder*( *T*)
  = 1,2,4,5,6,3,7,9,8

- *Postorder*( *T*)
  = 4,5,6,2,9,7,8,3,1

- *Inorder*( *T*)
  =




  - $Preorder(T) = <n,\ Preorder(T_1), \ldots, Preorder(T_k)>$
  - $Postorder(T) = <Postorder(T_1), \ldots, Postorder(T_k),\ n>$
  - $Inorder(T) = <Inorder(T_1),\ n,\ Inorder(T_2), \ldots, Inorder(T_k)>$
    - No natural definition of *Inorder*

# References

- Further reading list and references
  - https://en.wikipedia.org/wiki/Tree_(data_structure)
  - https://www.geeksforgeeks.org/binary-tree-data-structure/
  - Silver et al., Mastering the game of Go with deep neural networks and tree search
  - Wang et al., Dual Octree Graph Networks for Learning Adaptive Volumetric Shape Representations
  - Takikawa et al., Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee