# EE2331 Data Structures and Algorithms

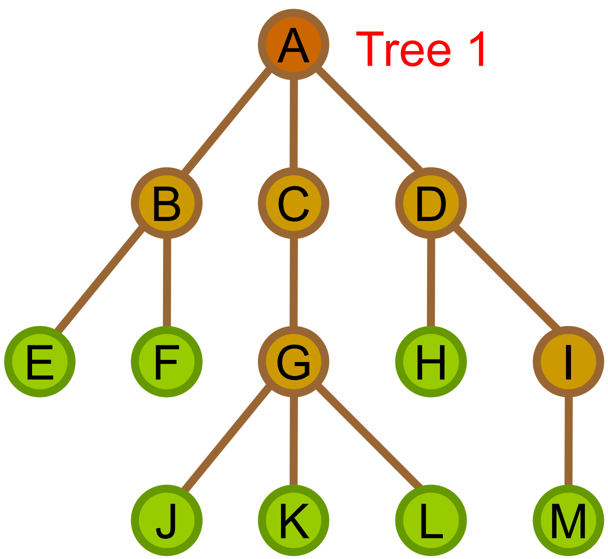
Trees: binary tree, operations

# Binary Tree

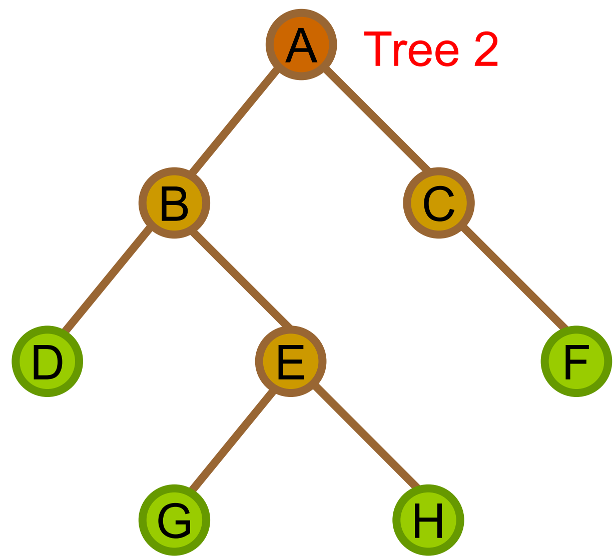
# Binary Trees

- Definition: each node has at most 2 children (left and right)
  - i.e. the node in binary tree should have either no children (leaf node), 1 child or 2 children
- Feature:
  - A special kind of tree
  - Simple design
  - Fixed max. degree of each node
    - Easier to represent with fixed data structure
  - Easy

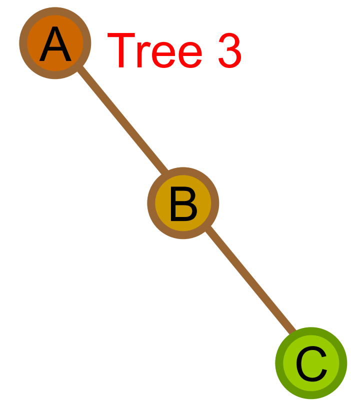
# Are They Binary Tree? Why?



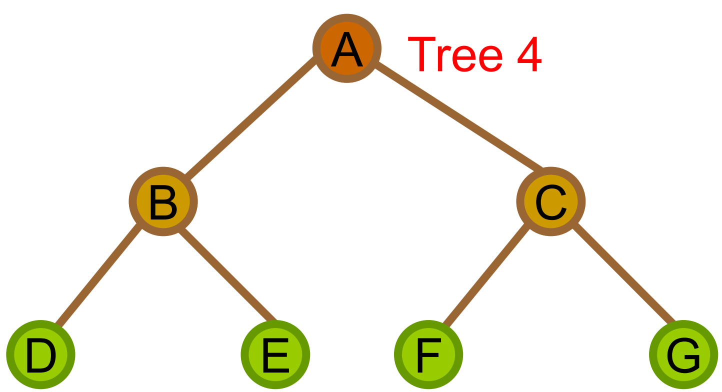

Tree 1



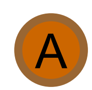

Tree 2




Tree 3




Tree 4

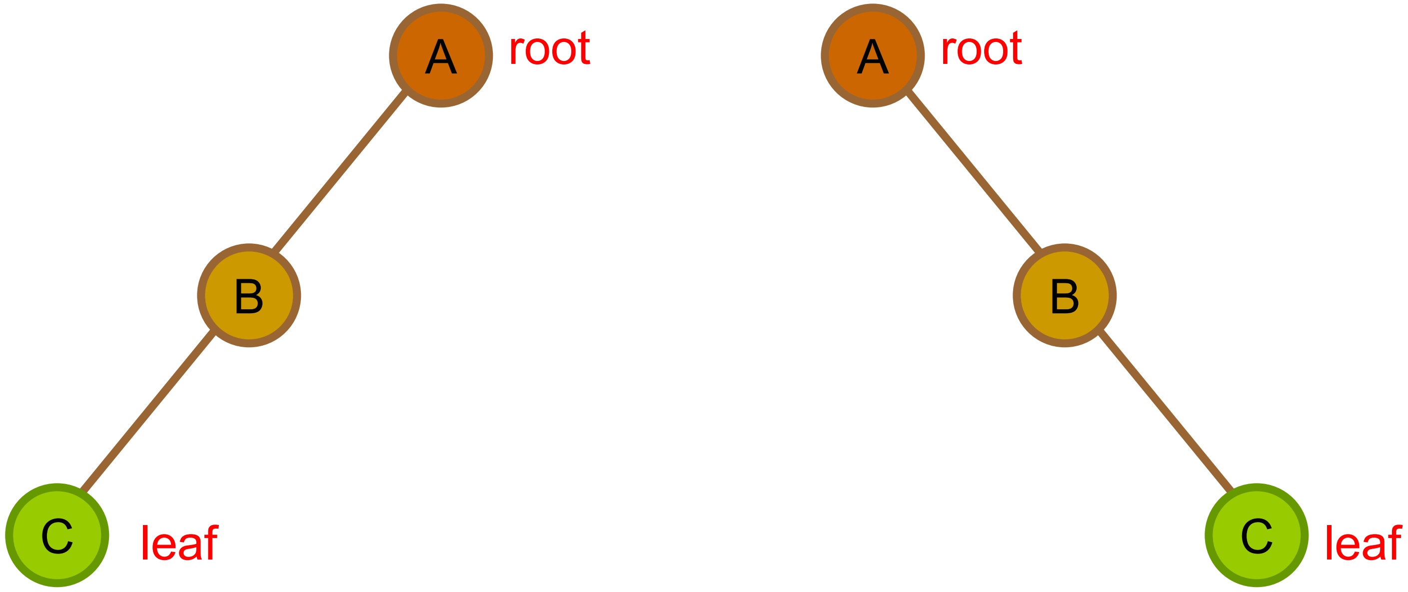



Tree 5 (root only)

Tree 6 (empty tree)

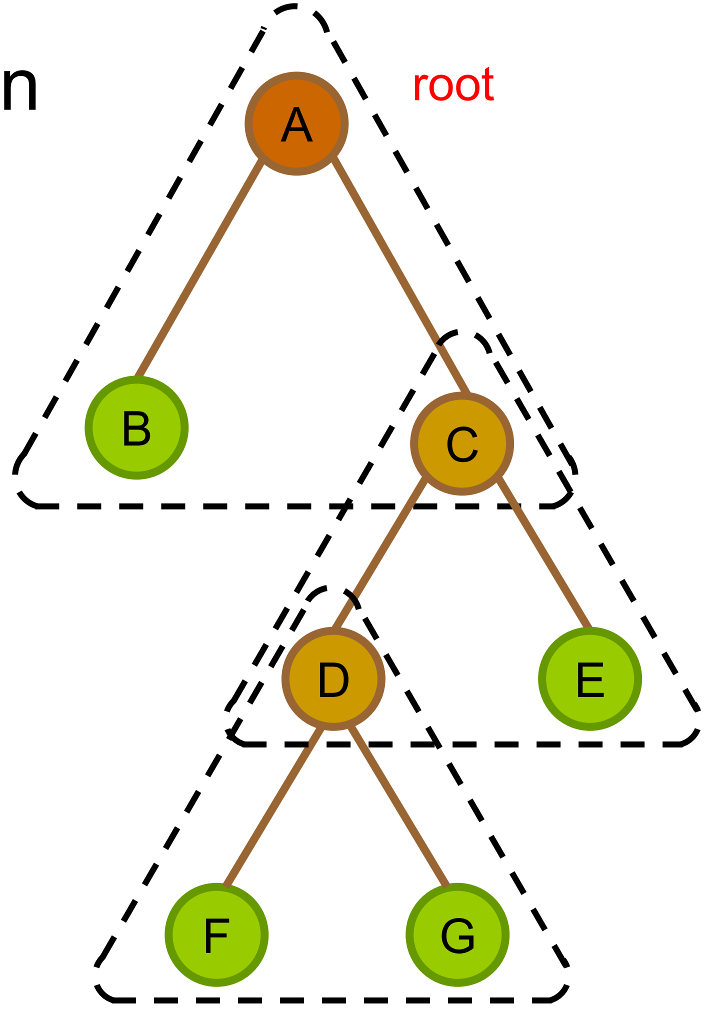
# Types of Binary Tree

- **Skewed tree**
- All nodes are either on the left hand side or right hand side



# Types of Binary Tree

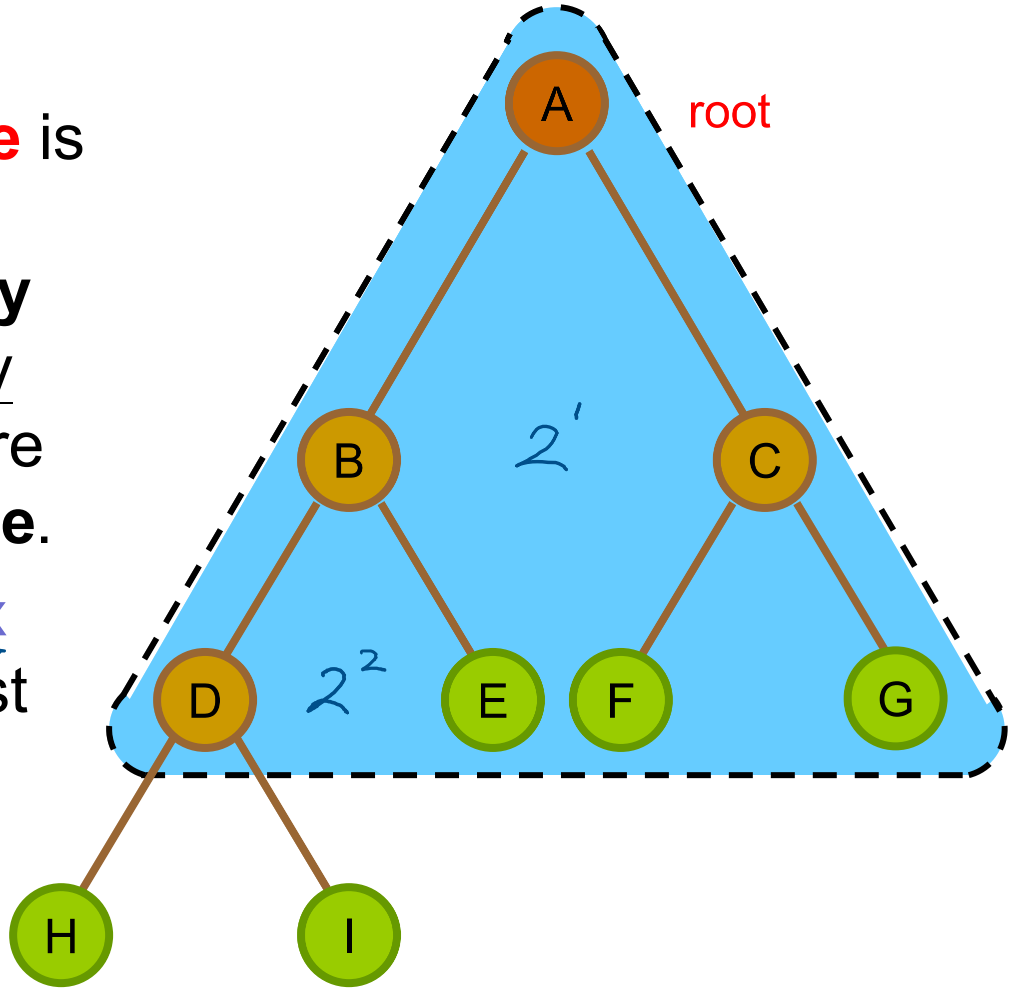
- **Full binary tree** is a tree in which every node in the tree has either 0 or 2 children.



# Types of Binary Tree

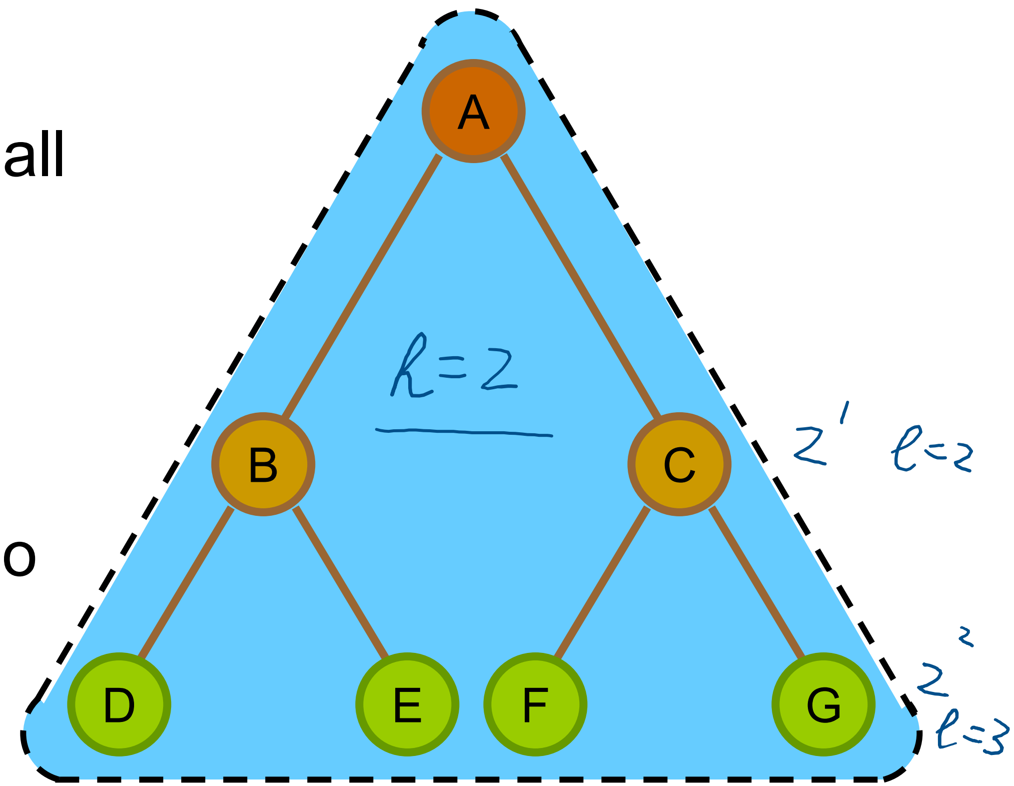
- **Complete binary tree** is a tree in which every level, **except possibly the last**, is completely filled, and all nodes are **as far left as possible**.
- It can have between x and y nodes at the last level *m*.

What is the range of the number of nodes at the last level m?



# Types of Binary Tree

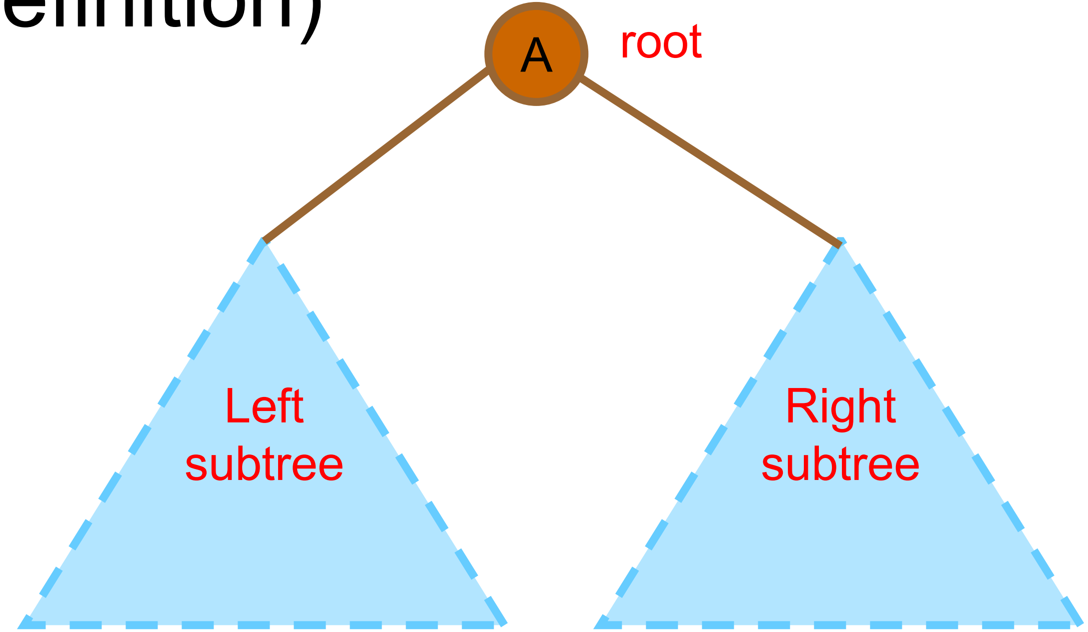
- **Perfect binary tree**
  is a binary tree in which all interior nodes have two children, *and* all leaves have the same *depth* or same *level*.
- Perfect binary tree is also full binary tree and complete binary tree.




Given a perfect binary tree T of height h, how many nodes does T have?

# Formation of Binary Trees

- It contains 3 parts, namely
  - root node, left subtree, right subtree
- For each subtree, it has 3 parts again (recursive definition)



# Properties of Binary Trees

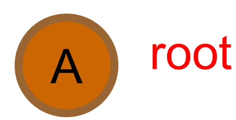
- Maximum no. of nodes on level *m* is $2^{m-1}$
- Maximum no. of nodes is $2^{h+1} - 1$, where *h* is the height of the tree

- Can you convince yourself by proving the above?

# Counting Binary Trees

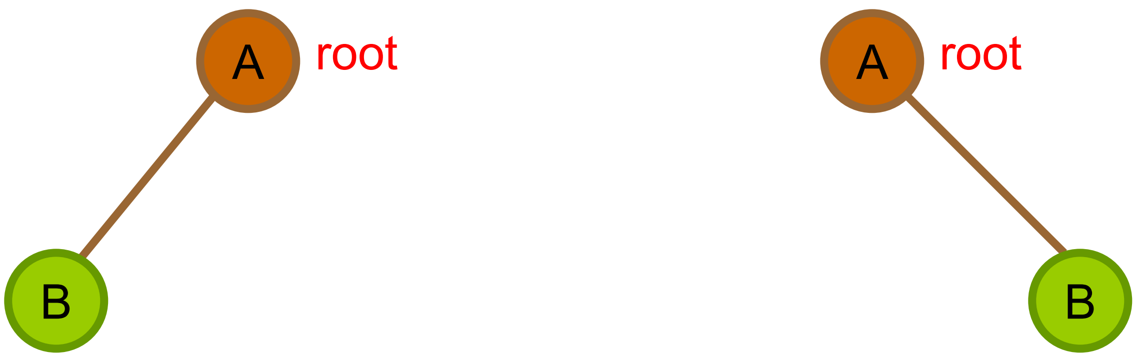
- How many different combination of a tree can have if it has *n* nodes?

- For $n$ = 1, only one combination



# Counting Binary Trees

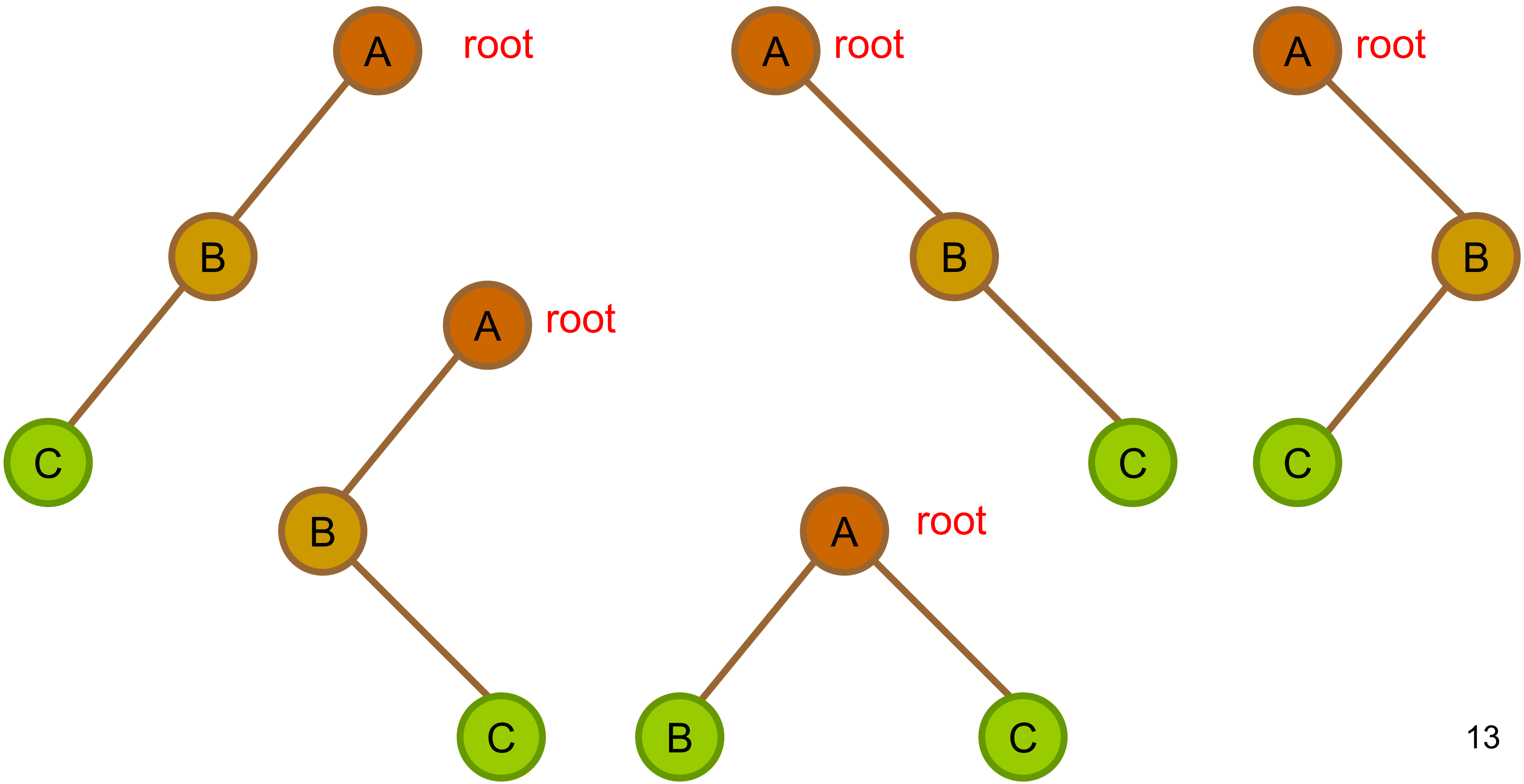
- For $n$ = 2, two combinations




Note: they are different!

# Counting Binary Trees

- For $n$ = 3, five combinations



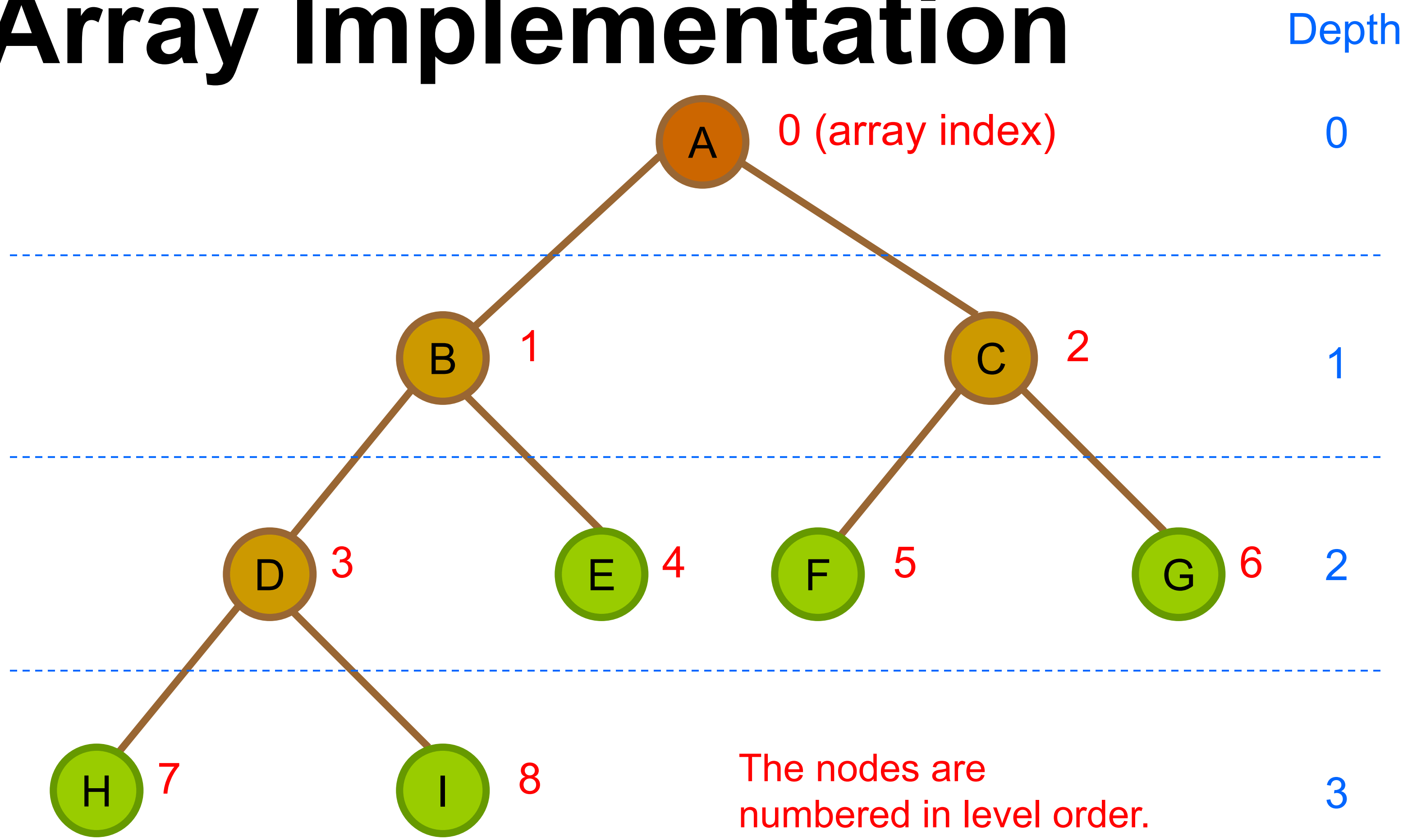
# Counting Binary Trees

- For *n* = 4, 14 combinations
  - Try yourself

- For *n* = *k*, $\frac{1}{k+1} \times \frac{(2k)!}{k!k!}$ combinations

# Array Implementation for binary tree (BT)

# Array Implementation

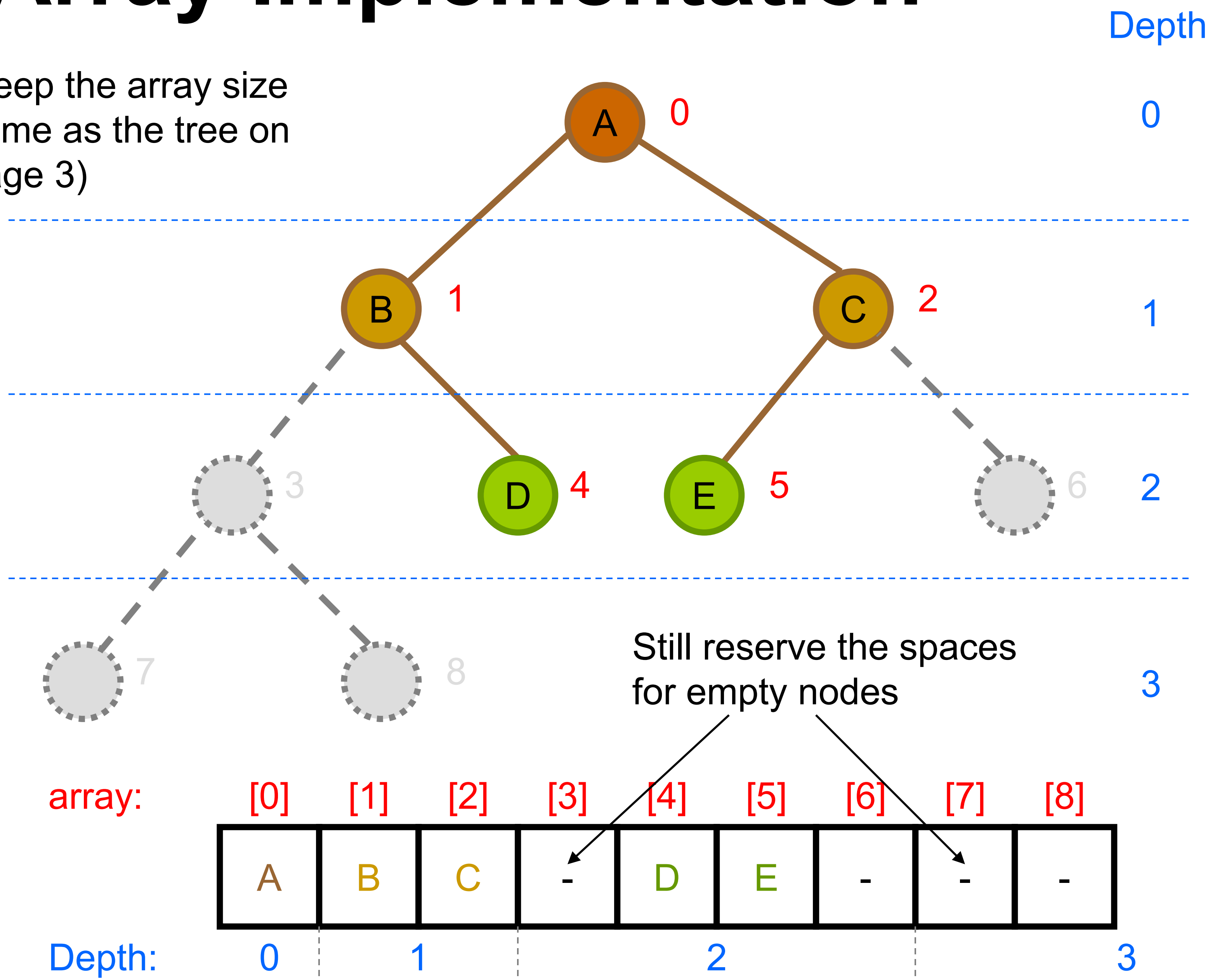



The nodes are numbered in level order.

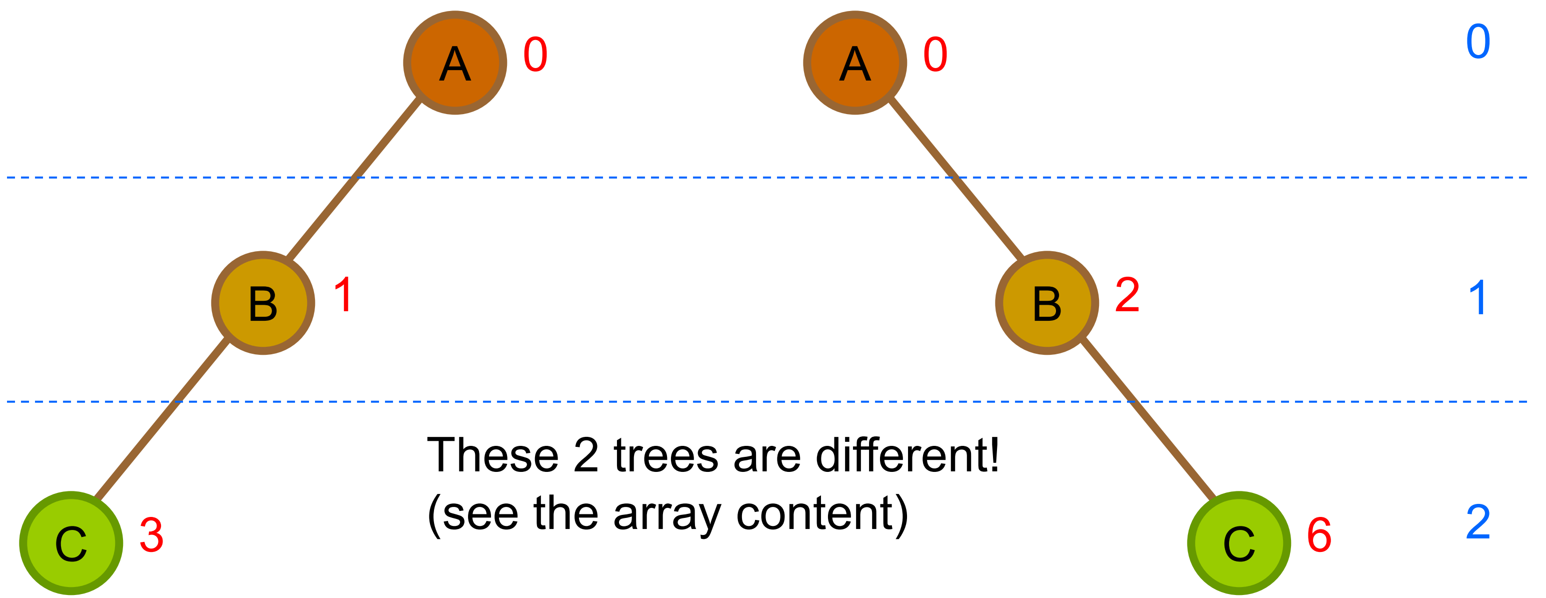
| array: | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I |
| Depth: | 0 | 1 | | 2 | | | | 3 | |

# Array Implementation

(keep the array size same as the tree on page 3)



# Array Implementation




array:

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| A | B | - | C | - | - | - | - | - |

array:

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| A | - | B | - | - | - | C | - | - |

A lot of unused space for non-complete binary tree

# Indicating Unused Nodes

Method 1:

array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | B | - | C | - | - | - | - | - |

Assign a special or invalid value
(e.g. -1, ‘\0’)

Method 2:

array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | B | ? | C | ? | ? | ? | ? | ? |

additional array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Create another boolean array to indicate the unused node

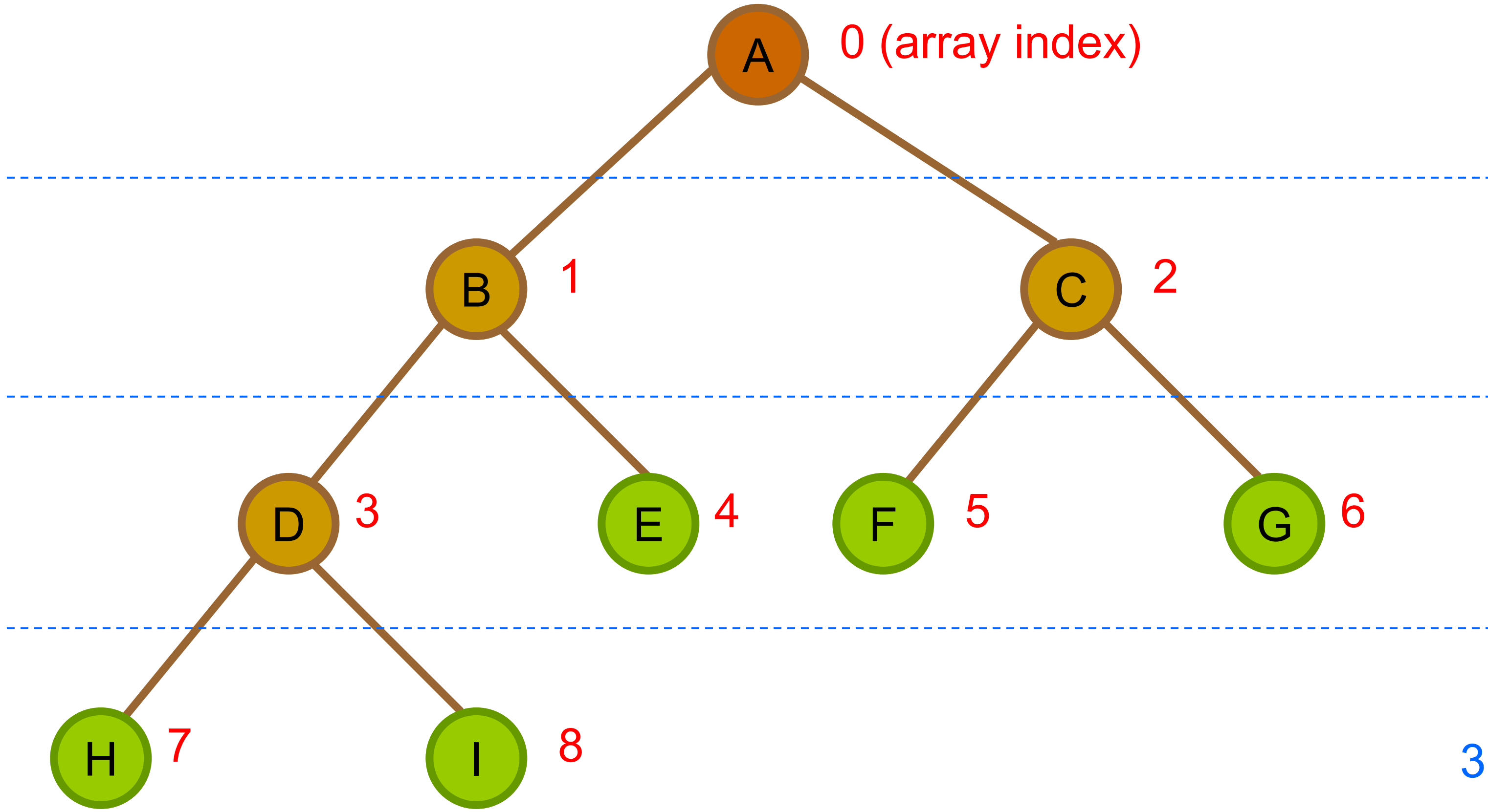
# Memory Efficiency

- For **complete binary tree**, array implementation is a very good approach
  - Simple
  - Utilize the memory very well
- But for other binary trees
  - Much memory has been wasted

# In-class exercise

- For a binary tree of height h, what is the smallest tree, plot it (sketch the key topology)
- For a binary tree of height h, what is the largest tree, plot it (sketch the key topology)

# Depth and node number



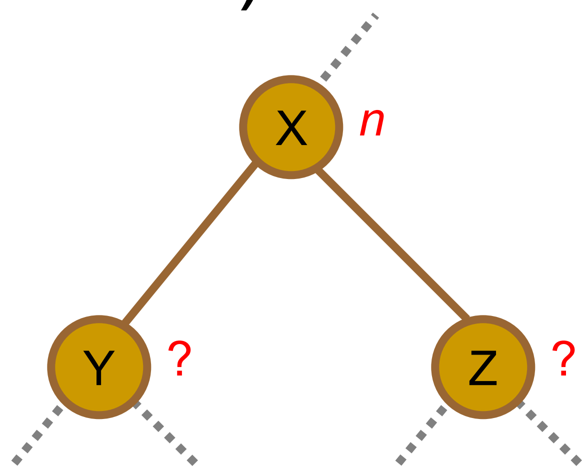

| Depth. | # nodes |
|---|---|
| 0 | $2^0$ |
| 1 | $2^1$ |
| 2 | $2^2$ |
| 3 | $<= 2^3$ |

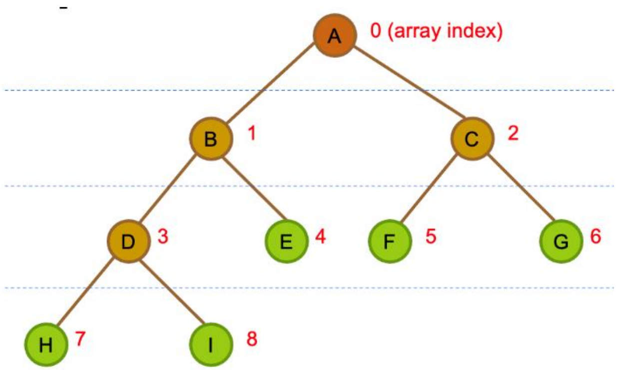
The total number of nodes for depth 0 to depth 2: 1+2+4

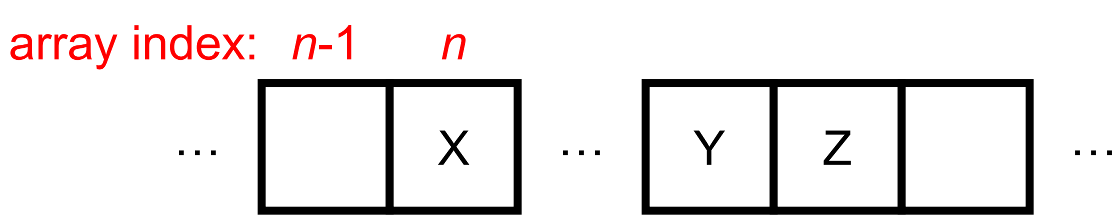
Generally, $2^0+2^1+2^2 + \ldots + 2^d = 2^{d+1} - 1$

# Determine the Index of Children

- If the array index of node *x* is *n*, what are the array indexes of the children of node *x* (i.e. node *y* and *z*)?



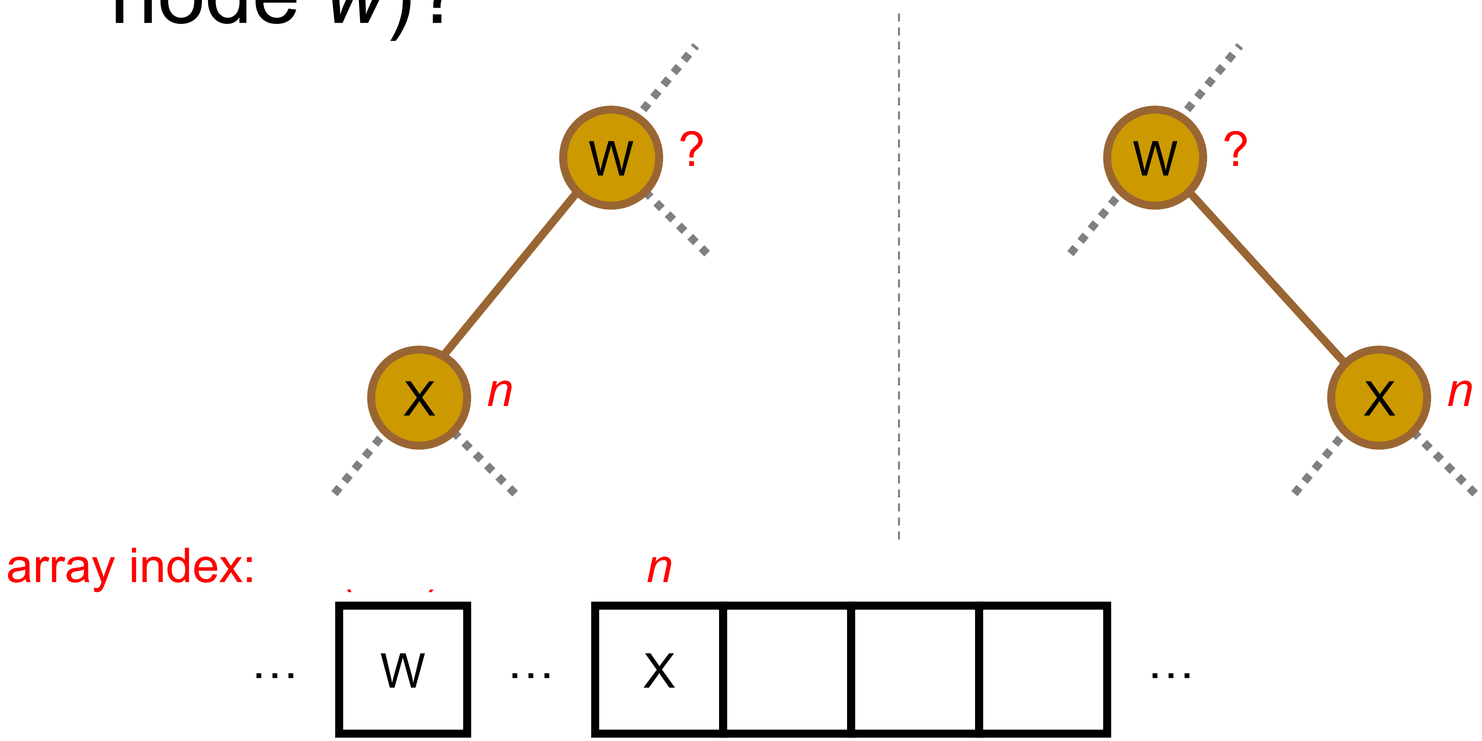



# How to prove it?

- Hint: let a node be the *i*th node at depth d, now figure out its index and also its left child's index.

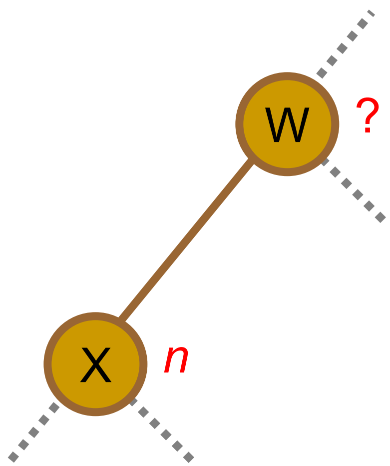
# Determine the Index of Parent

- If the array index of node *x* is *n*, what is the array index of the parent of node *x* (i.e. node *w*)?



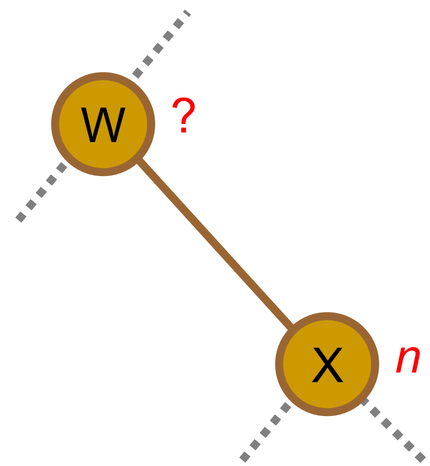
# Left or Right Child?

- If the array index of node *x* is *n*, how to determine if node *x* is the left child or right child?




left child if *n* is odd

if (n % 2 == 1) { /* left */ }




right child if *n* is even

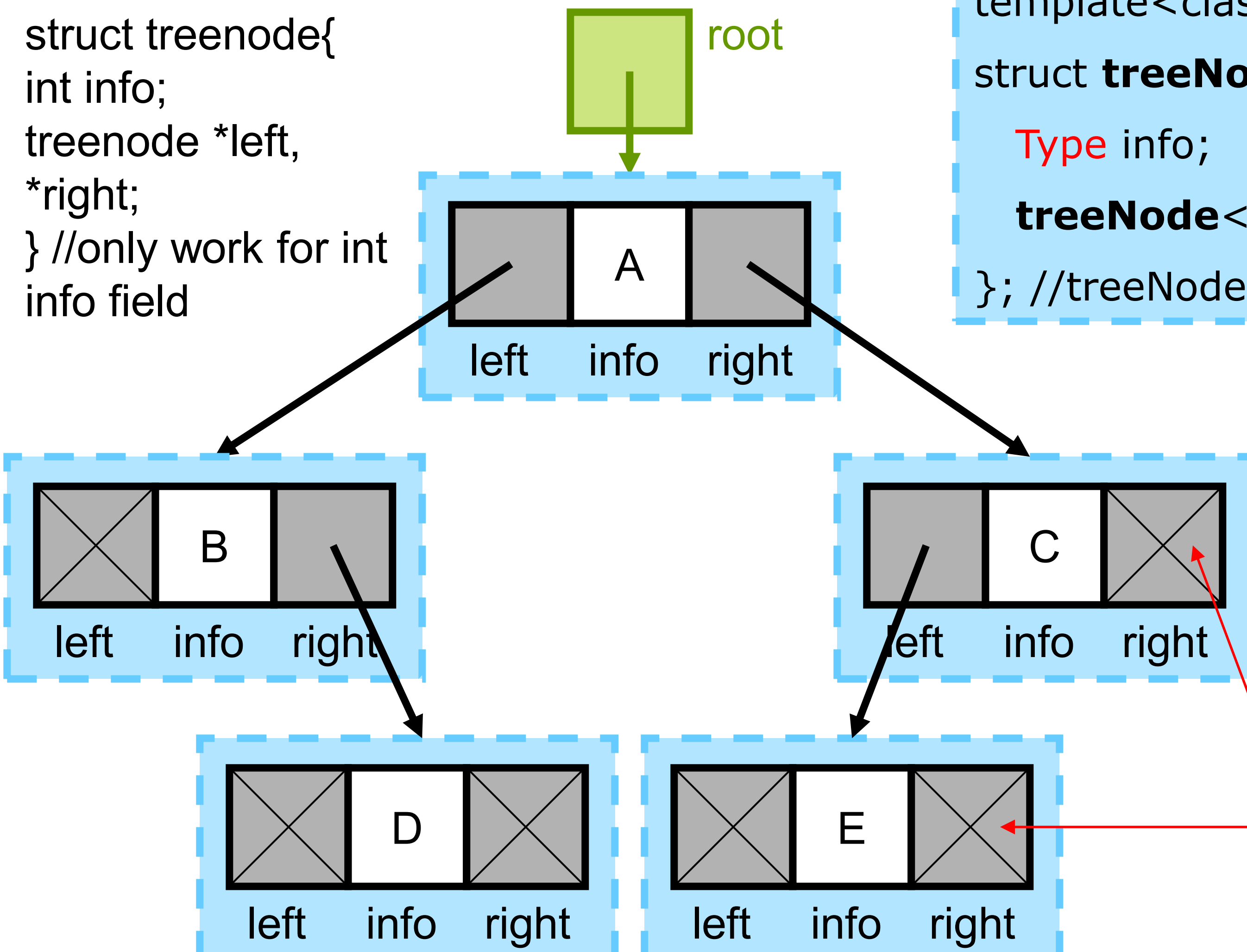
if (n % 2 == 0) { /* right */ }

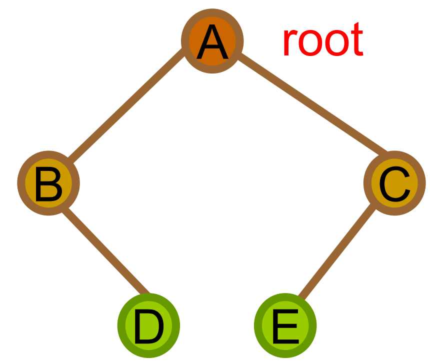
# Linked List Implementation

# Linked List Implementation

struct treenode{
int info;
treenode *left,
*right;
} //only work for int
info field

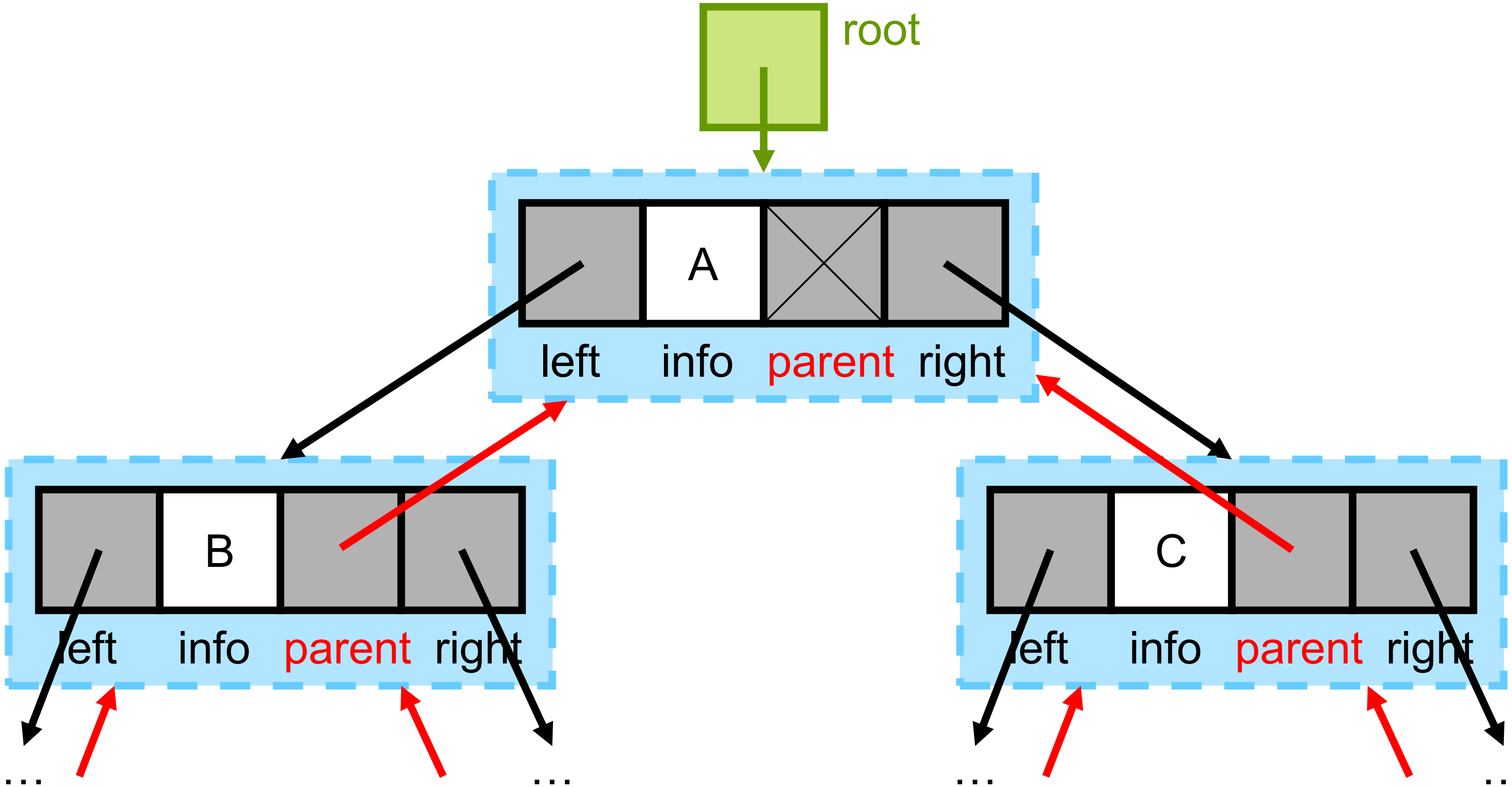
```
template<class Type>
struct treeNode {
    Type info;
    treeNode<Type> *left, *right;
}; //treeNode<int> *root;
```




NULL indicating no subtrees

# Possible Variations



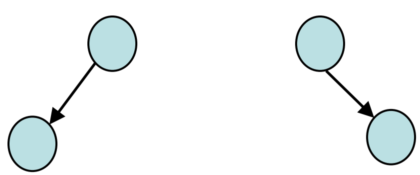

Each node has 3 references:

left, right and parent

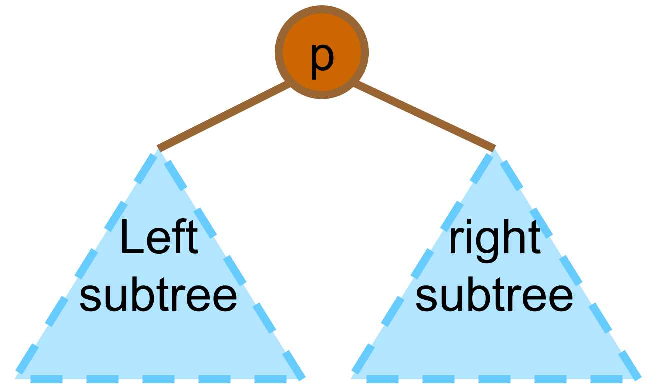
# Common Operations

# Compute the Height

```
template<class Type>
int height(treeNode<Type> *tree)
{
   if (tree == NULL)
      return -1;          // some definitions of empty tree’s height = 0

   if ((tree->left == NULL) && (tree->right == NULL))
      return 0;

   int HL = height(tree->left);
   int HR = height(tree->right);

   if (HL > HR)
      return 1+HL;
   else
      return 1+HR;
}
```



Height of root =
max(1 + $h_{\text{left}}$ , 1 + $h_{\text{right}}$)




Height of left subtree = $h_{\text{left}}$

Height of right subtree = $h_{\text{right}}$

# Count No. of Nodes / Leaves

```
template<class Type>
int nodeCount(treeNode<Type> *tree) {
  if (tree == NULL)
    return 0;
  return 1 + nodeCount(tree->left) + nodeCount(tree->right));
}
```

```
template<class Type>
int leavesCount(treeNode<Type> *tree) {
  if (tree == NULL)
    return 0;
  else if ((tree->left == NULL) && (tree->right == NULL))
    return 1;
  else
    return leavesCount(tree->left) + leavesCount(tree->right);
}
```

# Copy Binary Tree

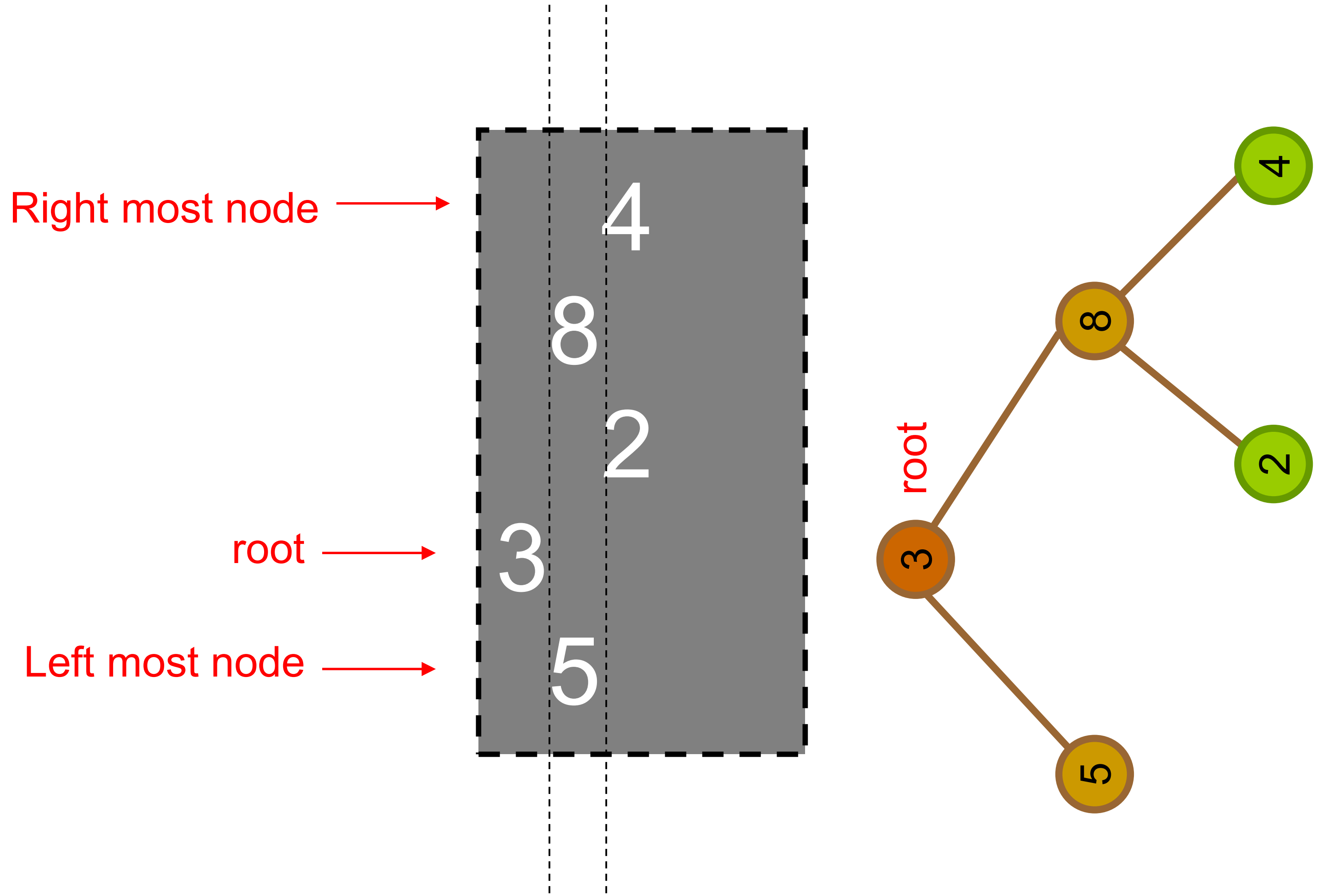
```
template<class Type>
treeNode<Type>* copyTree_2(treeNode<Type> *other)
{
   if (other == NULL)
      return NULL;

   treeNode<Type> *p = new treeNode<Type>;
   p->info = other->info;
   p->left = copyTree_2(other->left);
   p->right = copyTree_2(other->right);

   return p;
}
```

# Compare Two Binary Tree

- The two binary trees are identical iff
  - Their root nodes are equal;
  - their left subtrees are equal and;
  - their right subtrees are equal.

```
template<class Type>
bool equal(treeNode<Type> *tree1, treeNode<Type> *tree2) {

  if ((tree1 == NULL) && (tree2 == NULL))
    return true;

  if ((tree1 != NULL) && (tree2 != NULL)) {
     if ((tree1->info == tree2->info) &&
          equal(tree1->left, tree2->left) &&
          equal(tree1->right, tree2->right))
             return true;
  }

  return false;

}
```

# Printing a Binary Tree



# Printing a Binary Tree

- Print the right subtree first

Try to modify the code to print out the left subtree first.

```
#include <iomanip>          //setw(), set width


template<class Type>
void printTree(treeNode<Type> *p, int indent) {
    if (p != NULL) {
        //print right subtree, root, and then left subtree
        printTree(p->right, indent+3);
        cout << setw(indent) << p->info << endl;
        printTree(p->left, indent+3);
    }
}
```

# Four Basic Traversal Orders

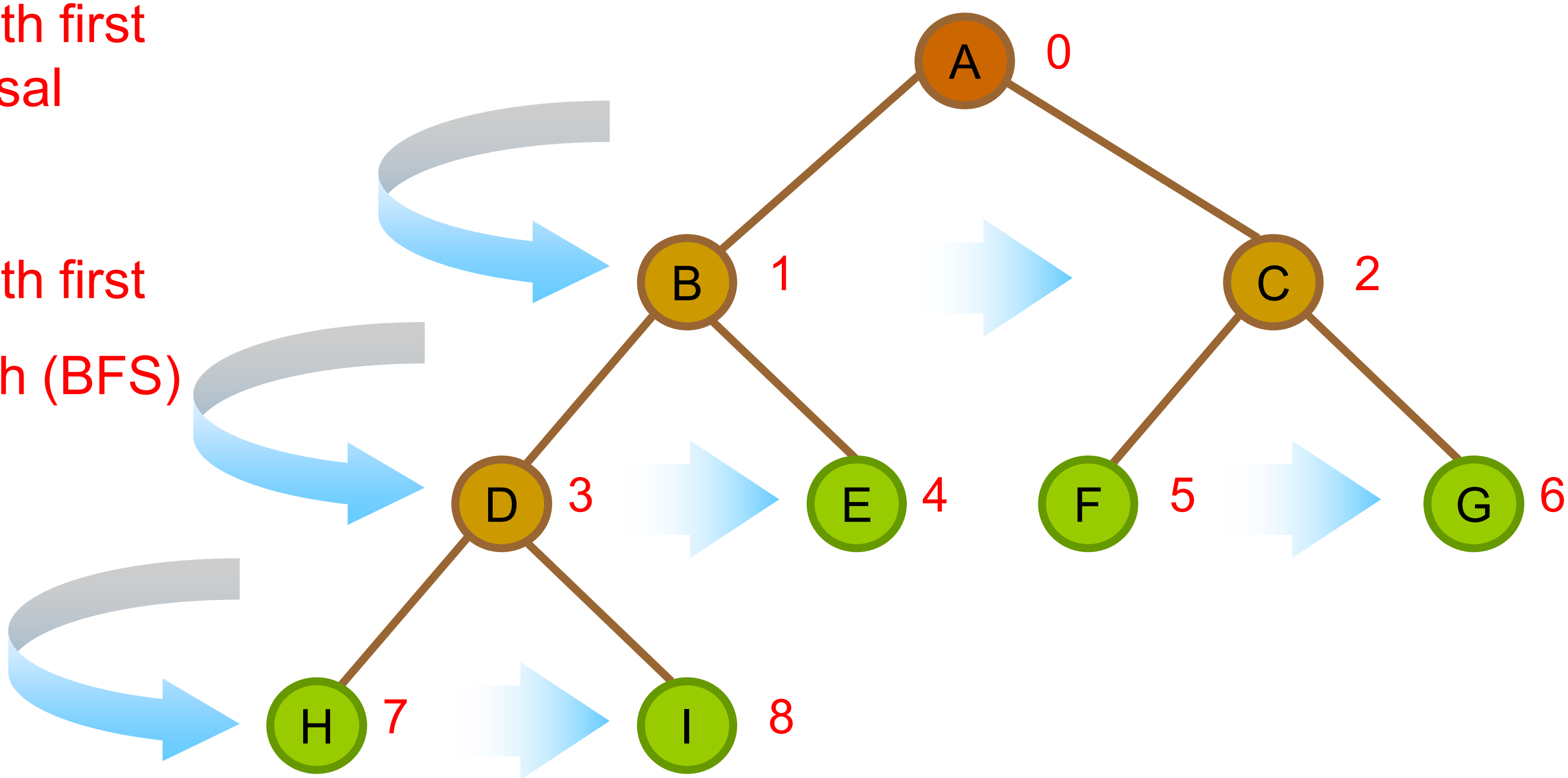
- Describe the way to visit every nodes of the entire tree
- **Level order**
  - visit the nodes from left to right, level by level starting from the root

Breadth first traversal

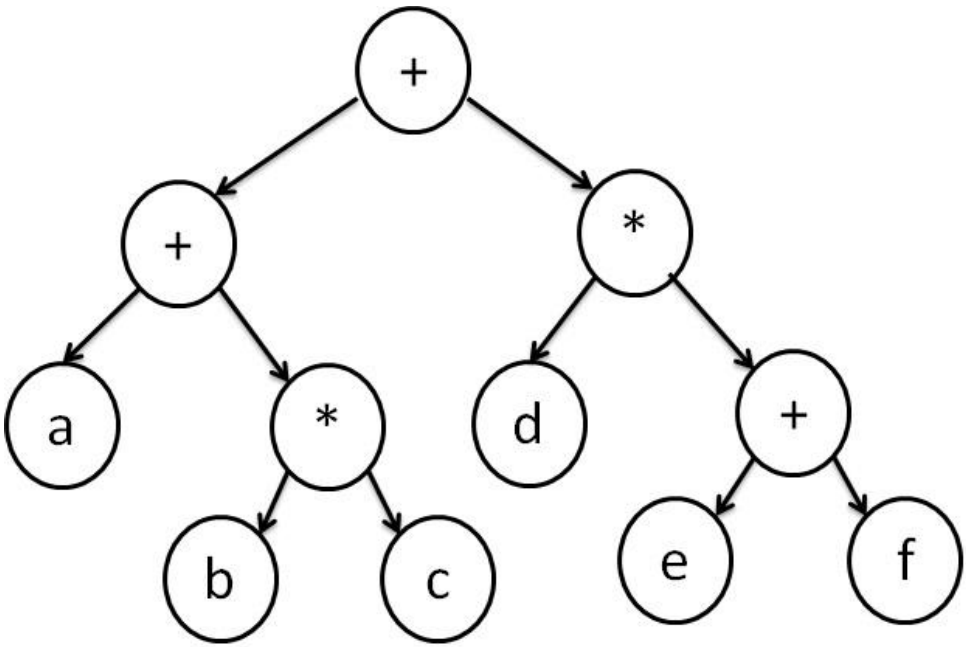
Or

Breadth first

Search (BFS)



# Other traversal orders




expression tree

What is the expression/formular represented by this tree?

(a+b*c)+(d*(e+f))

+ (+a(*(bc)))(*(d)(+ef))

# Four Basic Traversal Orders

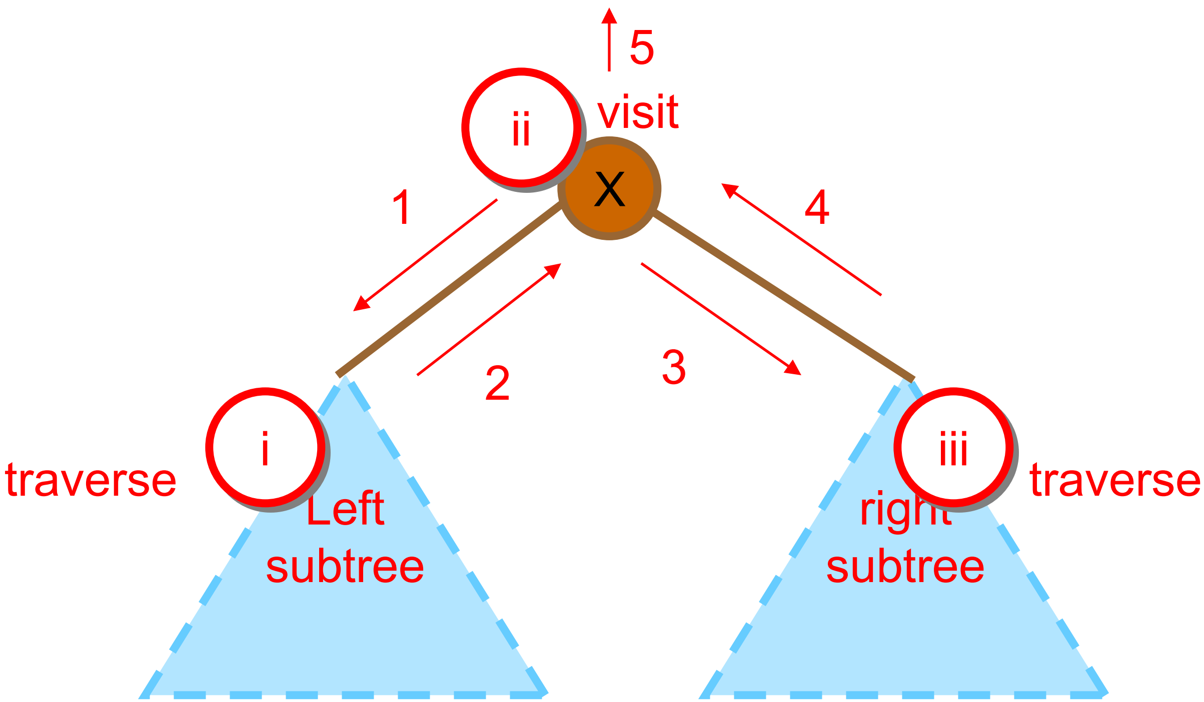
- **Preorder**
  - visit the root (V)
  - visit the left subtree in preorder (L)
  - visit the right subtree in preorder (R)
- **Inorder**
  - visit the left subtree in inorder (L)
  - visit the root (V)
  - visit the right subtree in inorder (R)
- **Postorder**
  - visit the left subtree in postorder (L)
  - visit the right subtree in postorder (R)
  - visit the root (V)

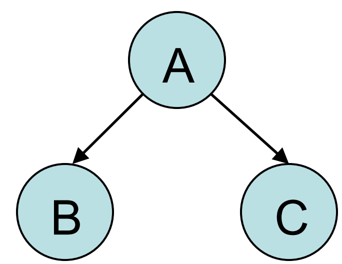
Depth first traversal

(Depth first search (DFS))

# Example: LVR

- Step i) go to left subtree (**recursion**)
- Step ii) visit node x
- Step iii) go to right subtree (**recursion**)

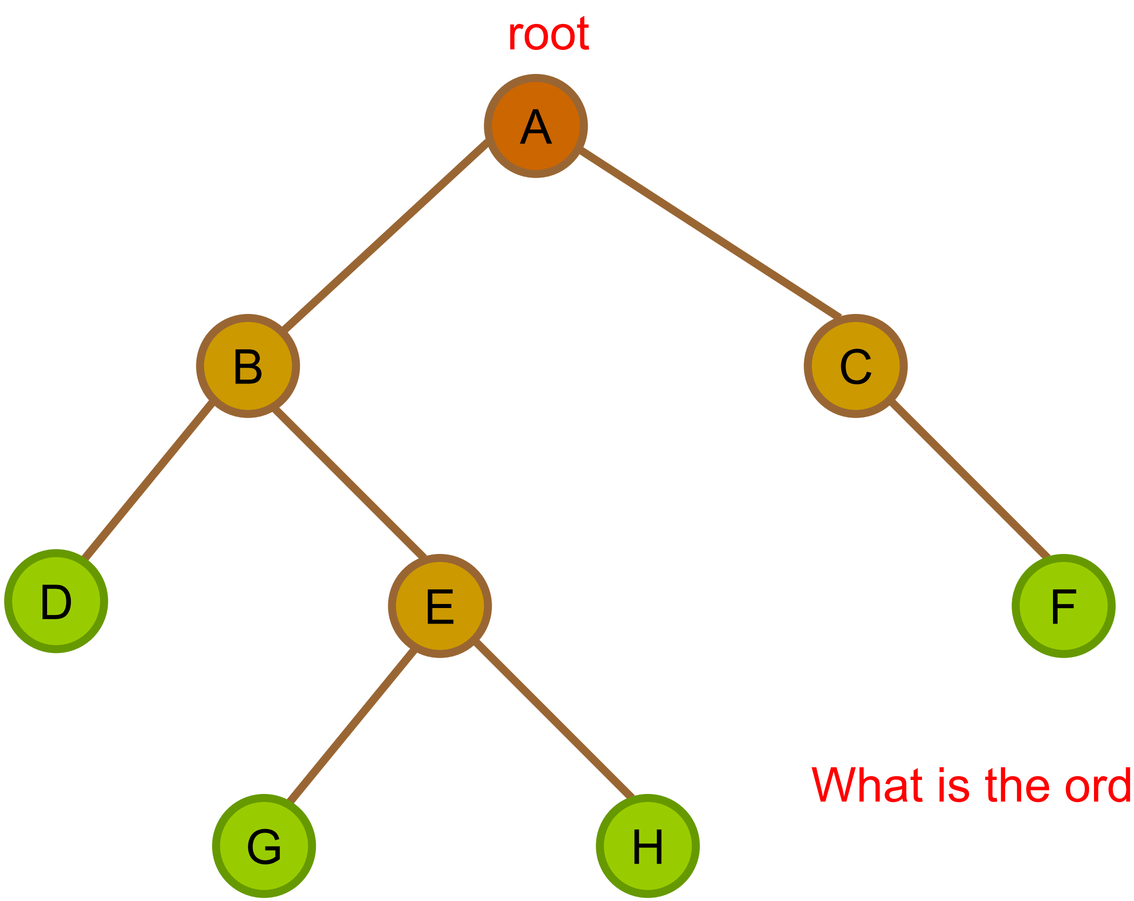



Trivial case:
BAC

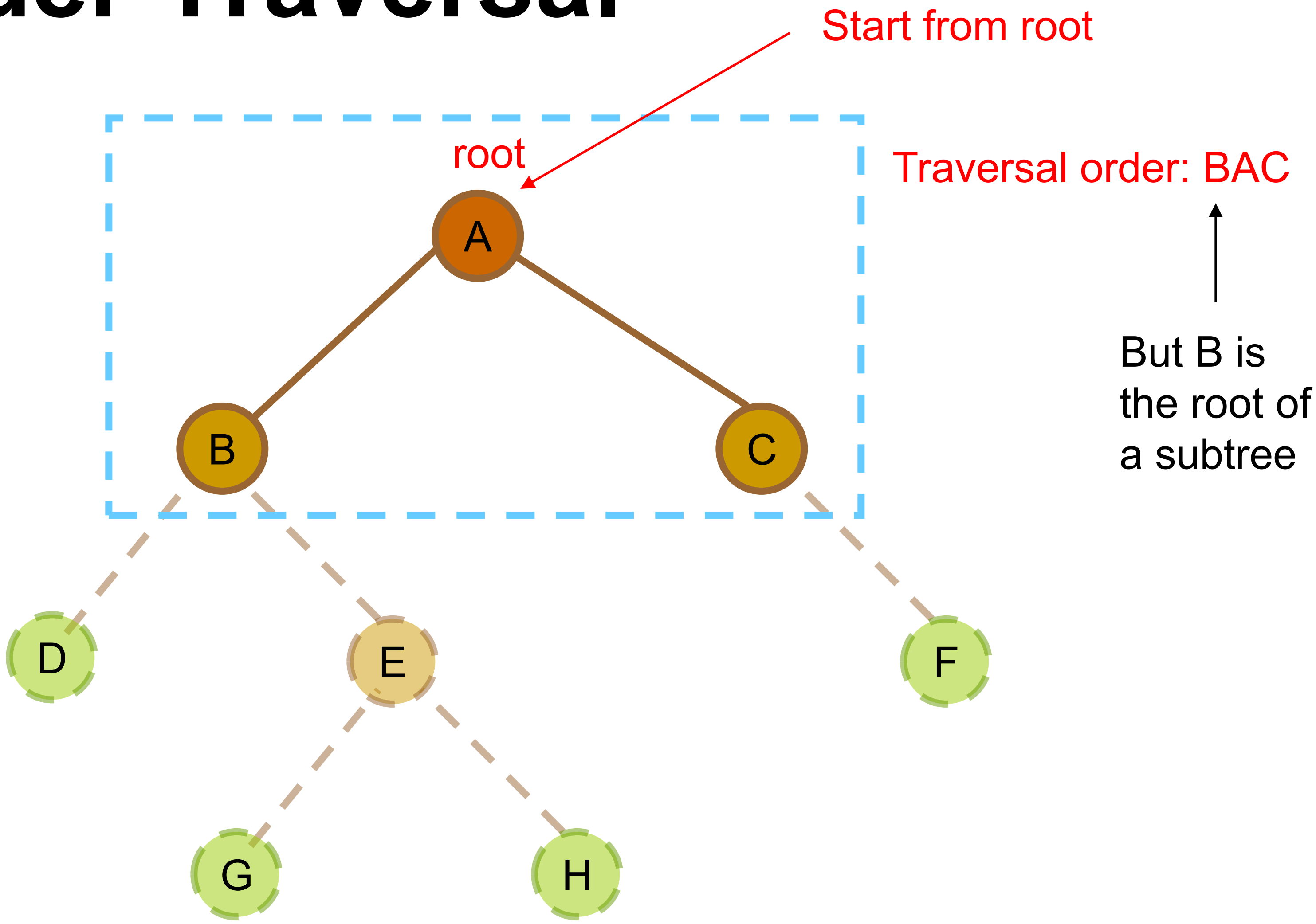
# Inorder Traversal

- LVR

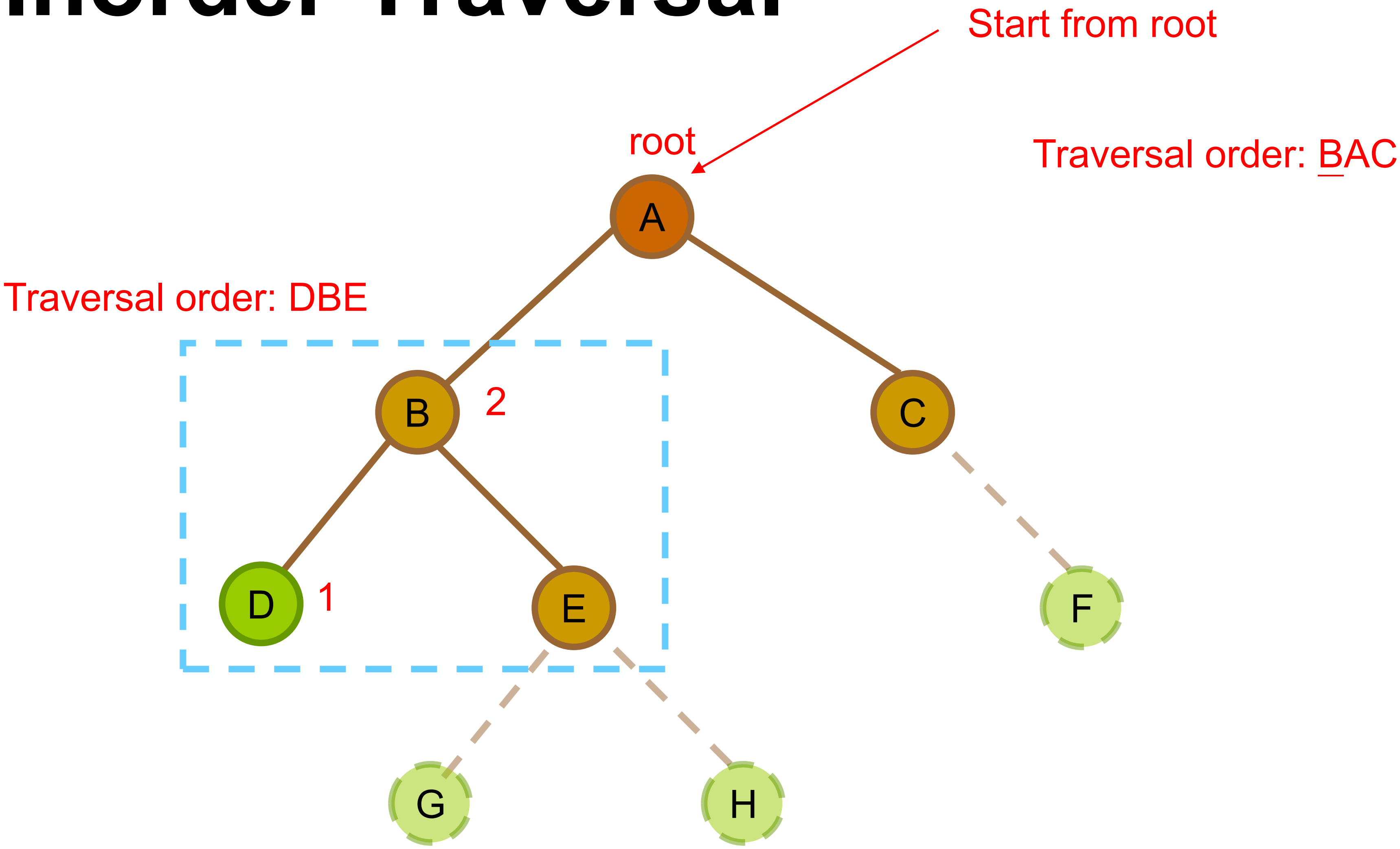



What is the order of traversal?

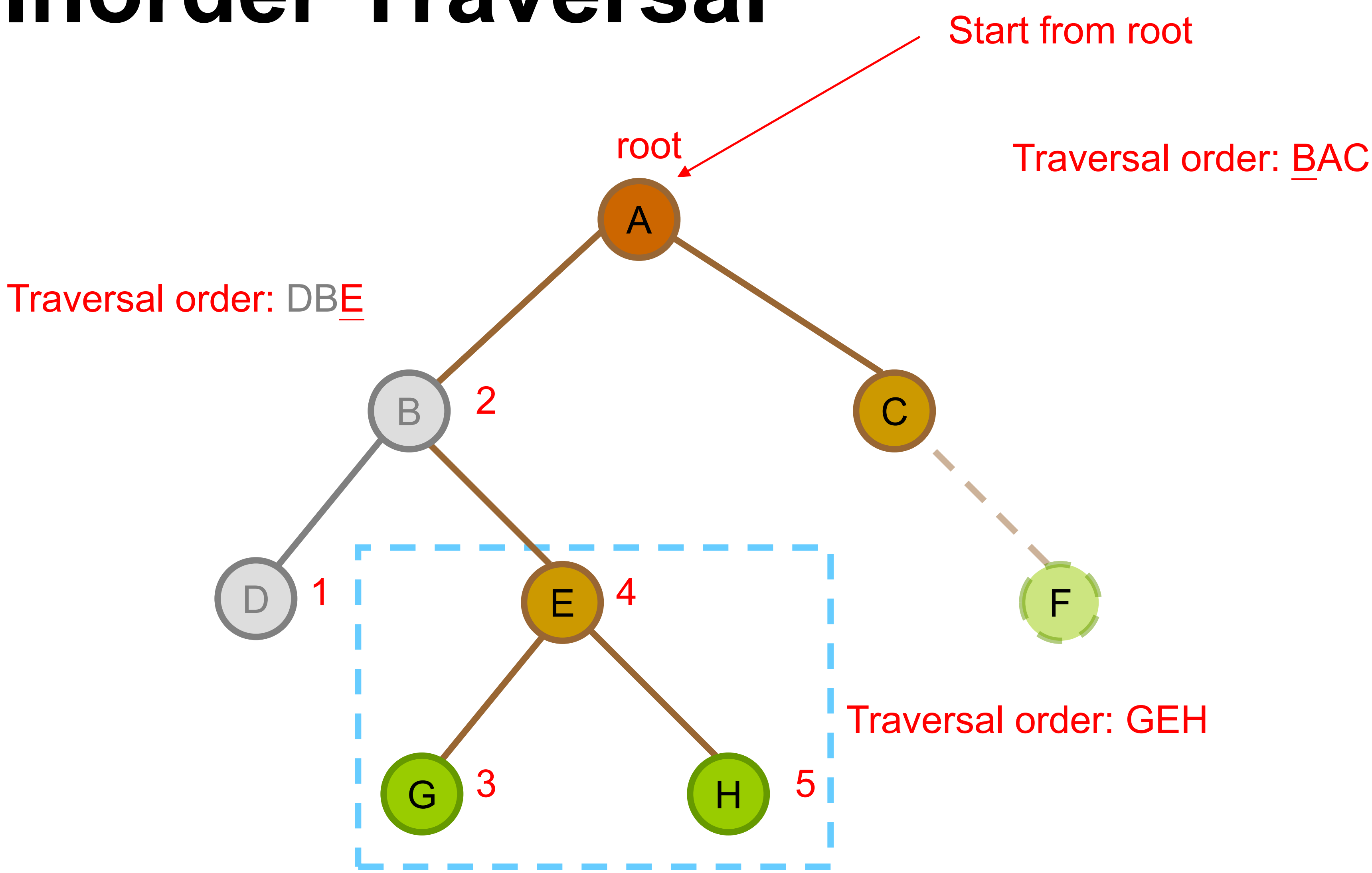
# Inorder Traversal



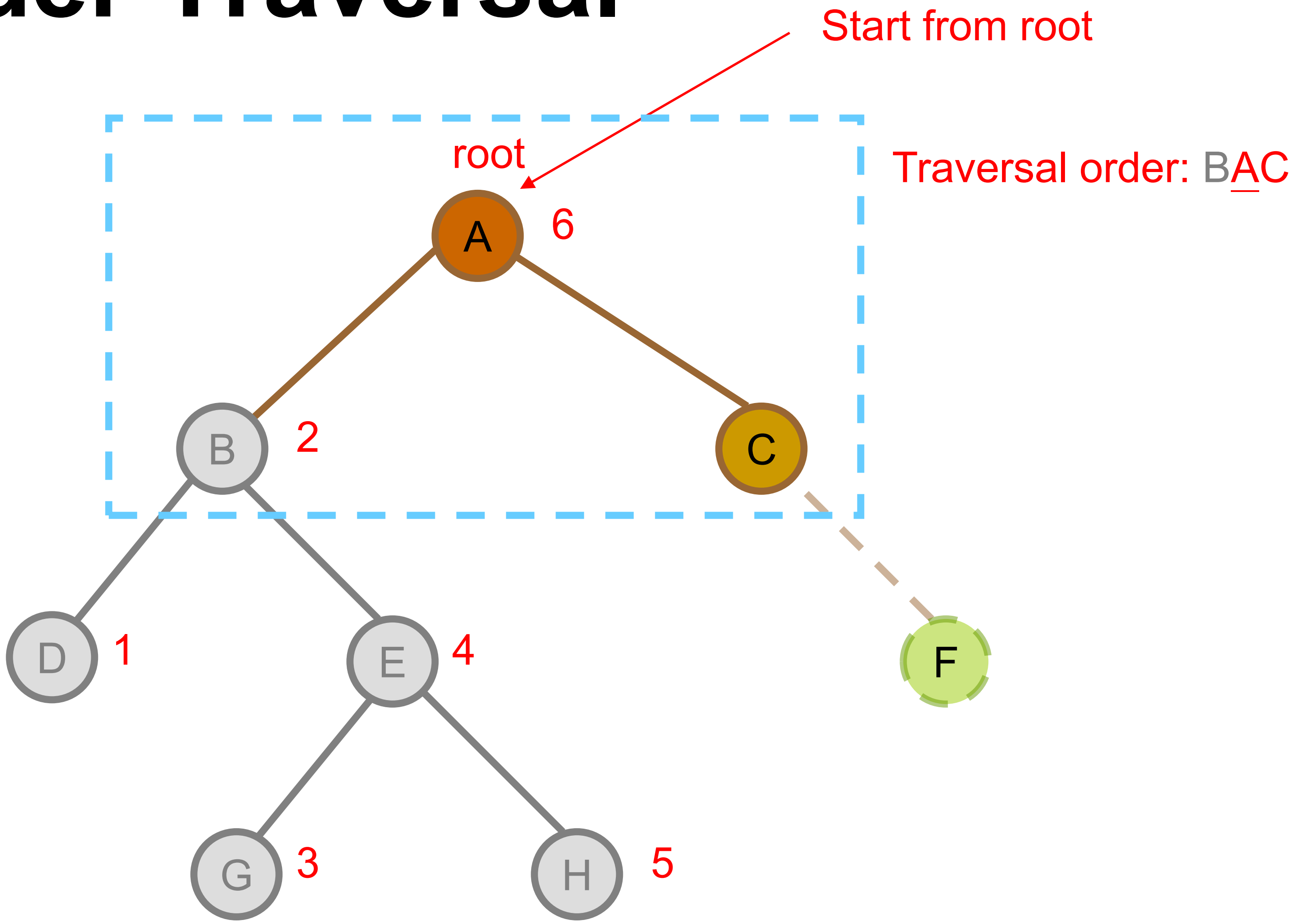
# Inorder Traversal



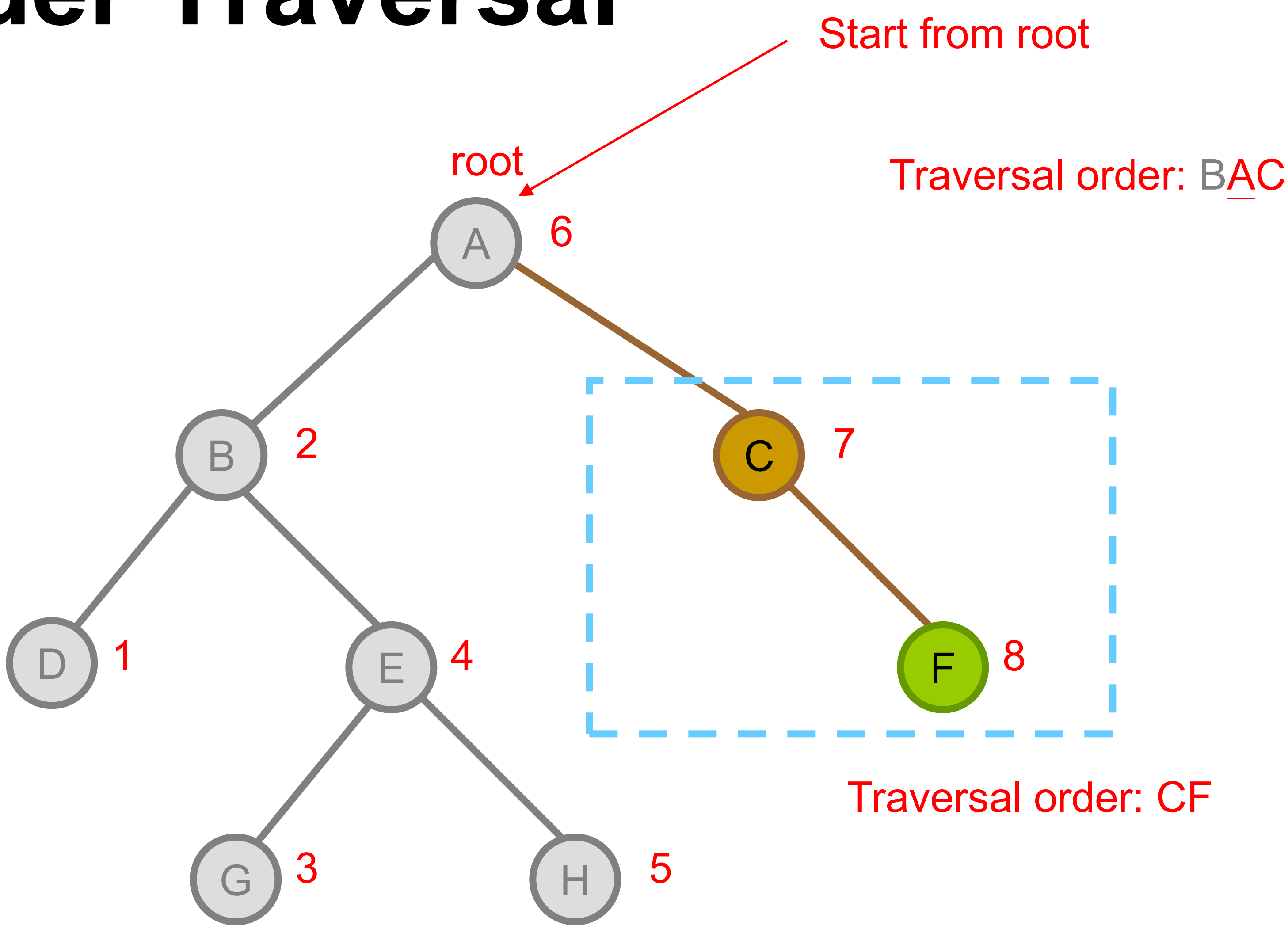
# Inorder Traversal



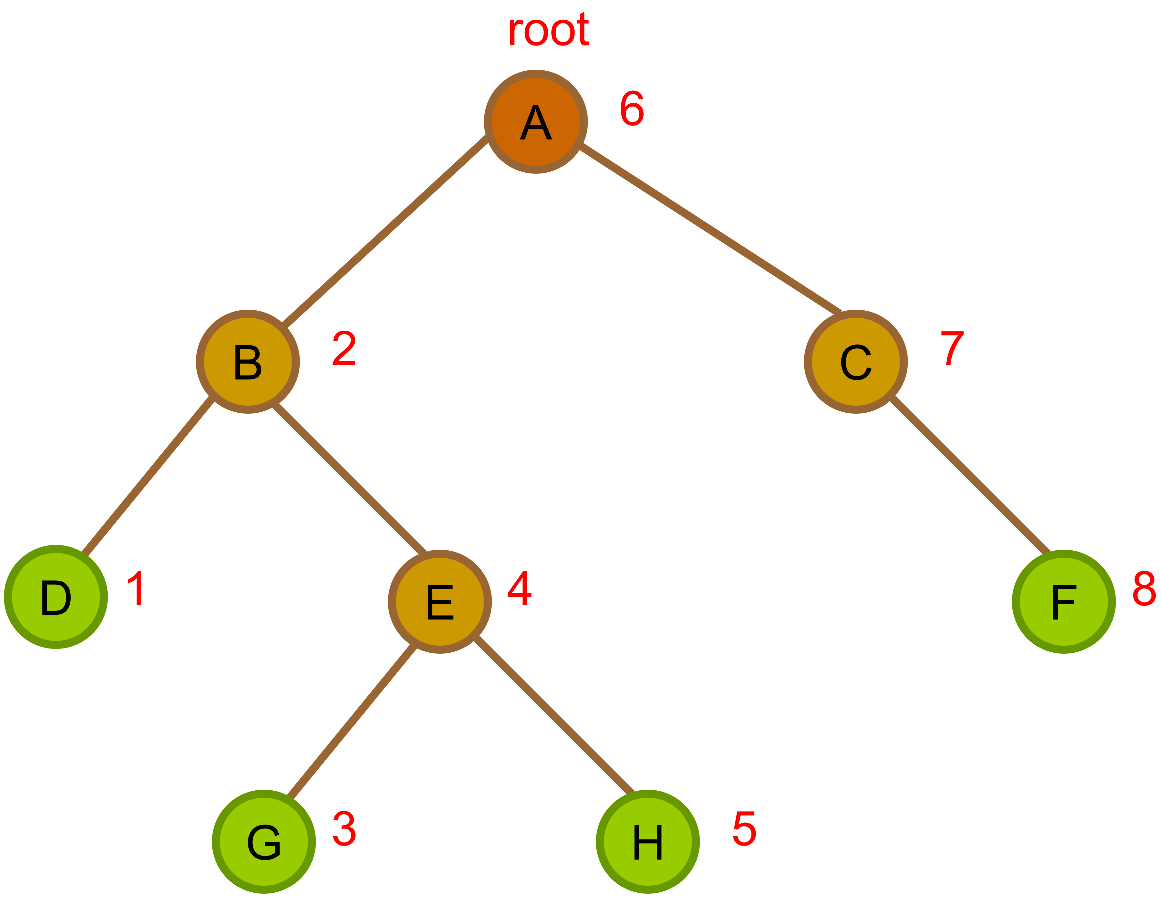
# Inorder Traversal



# Inorder Traversal



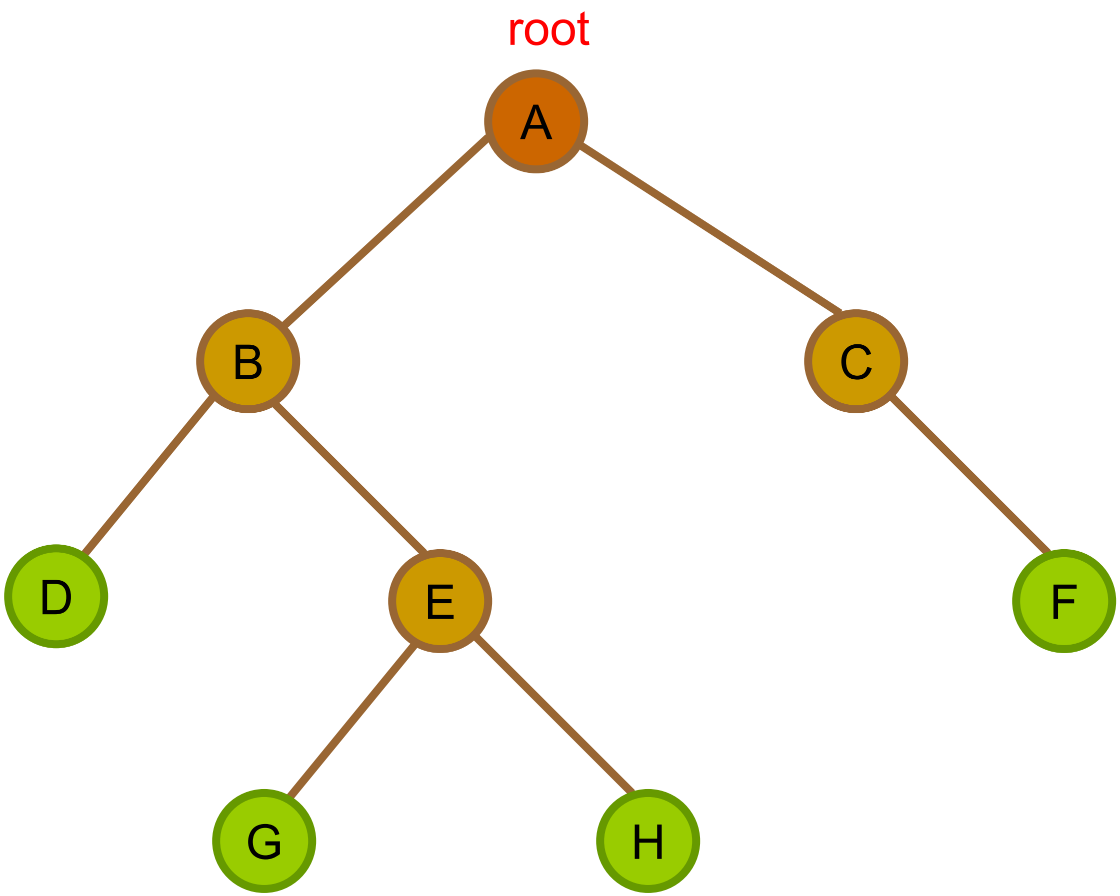
# Inorder Traversal




The final sequence: DBGEHACF

# Postorder Traversal

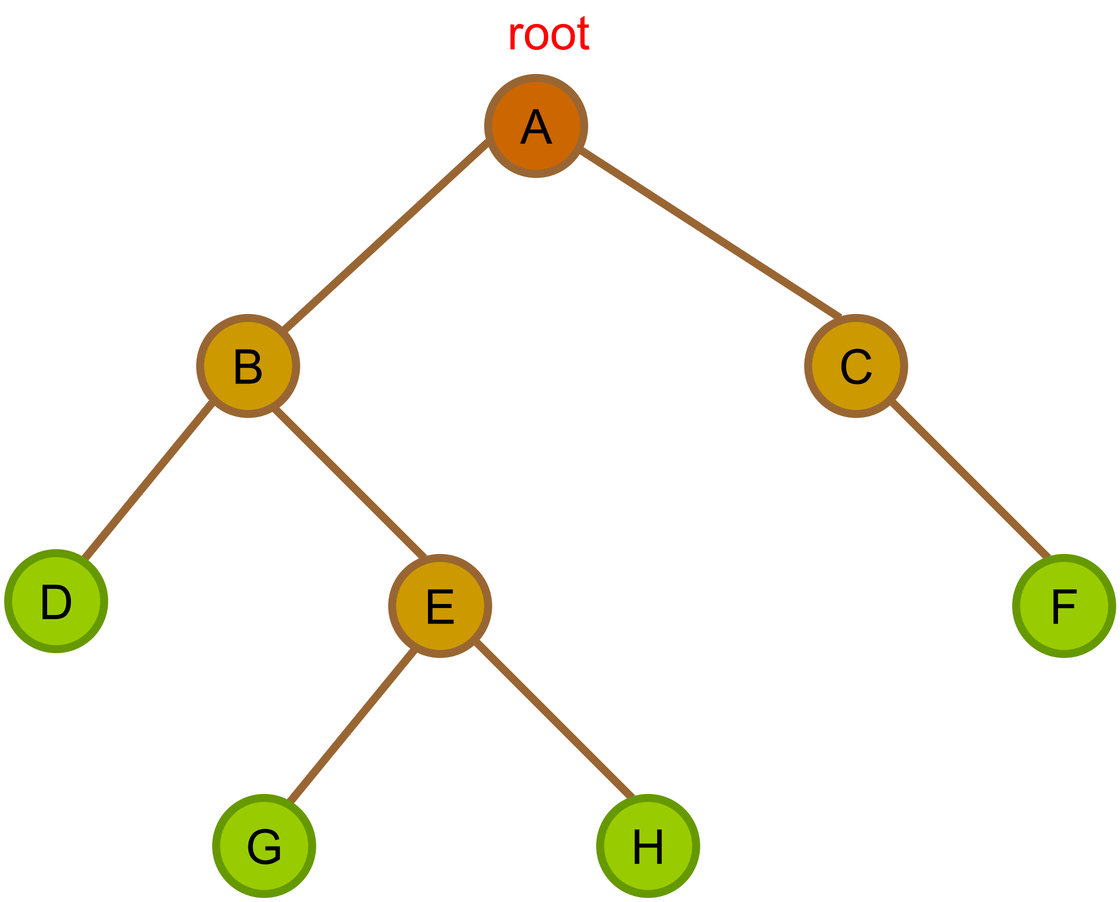
- LRV




The final sequence: ____________________

# Preorder Traversal

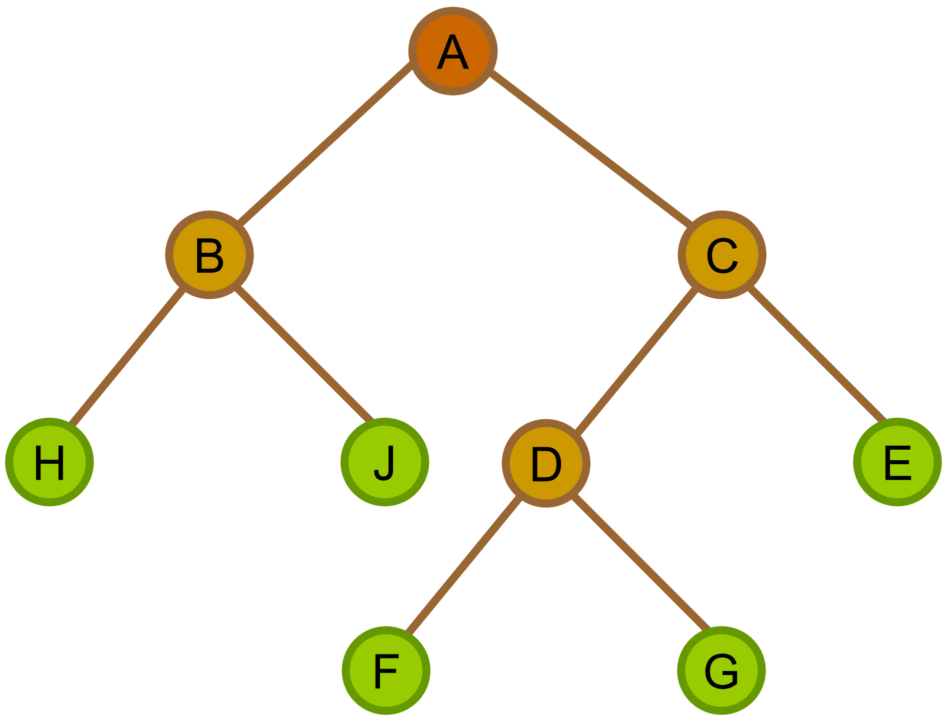
- VLR




The final sequence: ________________

# In-class exercise

- Show the in-order, post-order, and pre-order traversal of the following tree.



# Preorder Traversal

```
template<class Type>
void preorder(treeNode<Type> *p)
{
   if (p != NULL)
   {
      cout << p->info << “ ”;         //visit the node
      preorder(p->left);              // visit left subtree
      preorder(p->right);             // visit right subtree
   }
}
```

Go to right subtree (i.e.
p->right) by recursion

# Inorder & Postorder Traversal

```
template<class Type>
void inorder(treeNode<Type> *p) {
   if (p != NULL) {
      inorder(p->left);
      cout << p->info << " ";        //visit the node
      inorder(p->right);
   }
}

template<class Type>
void postorder(treeNode<Type> *p) {

}
```

# Reconstruction of Binary Tree

# Question to ponder:

- Can you draw the binary tree if the **postorder** and **inorder** traversal of the tree are HJBFGDECA and HBJAFDGCE respectively?

# Reconstruction of Binary Tree

- The structure of a binary tree can be obtained if either preorder or postorder plus inorder traversal sequences are given
- Preorder + postorder
  - Fail to reconstruct the binary tree
- Only **inorder + preorder**, or **inorder + postorder** can provide sufficient information to reconstruct a binary tree

# The Reconstruction Algorithm

- Step 1) Determine the root node, left and right subtrees
  - From **postorder**, the last node is the root
    - e.g. node A
  - Then from **inorder**, the nodes on the left hand side of node A belongs to the left subtree of node A, nodes on the right hand side belongs to its right subtree
- Step 2) Consider the traversal sequence of the subtrees, and determine its root, left and right subtrees recursively

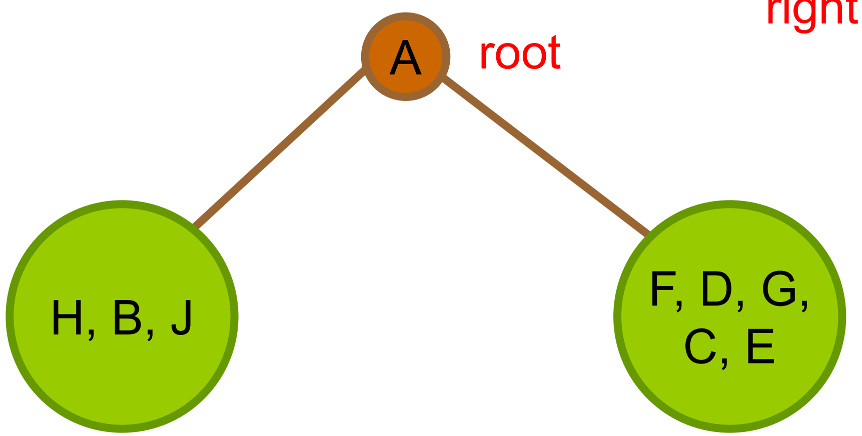
# First Determine the Root

root node

- Postorder: H J B F G D E C A
- Inorder: $\underline{\text{H B J}}$ A $\underline{\text{F D G C E}}$

left subtree

right subtree



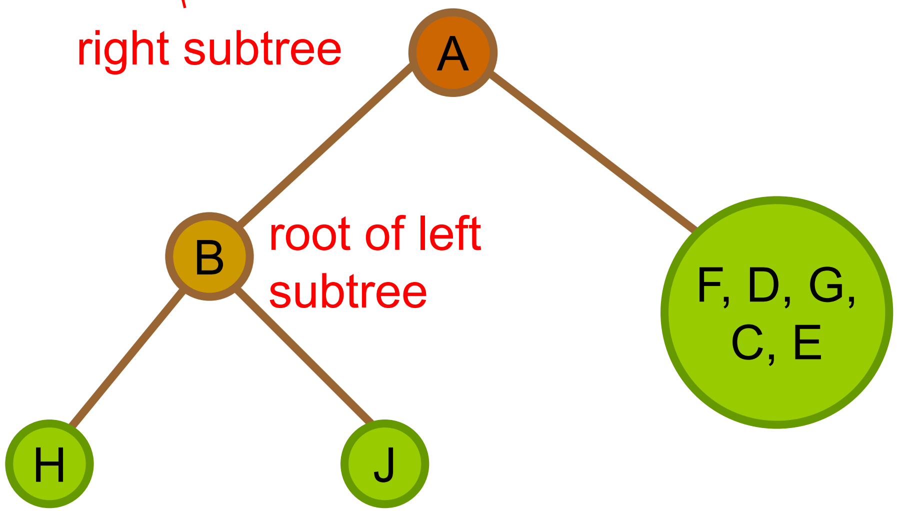
# Consider Left Subtree of Root

root node of the subtree

- Postorder: H J B F G D E C ~~A~~
- Inorder: H B J ~~A~~ F D G C E

left subtree

right subtree



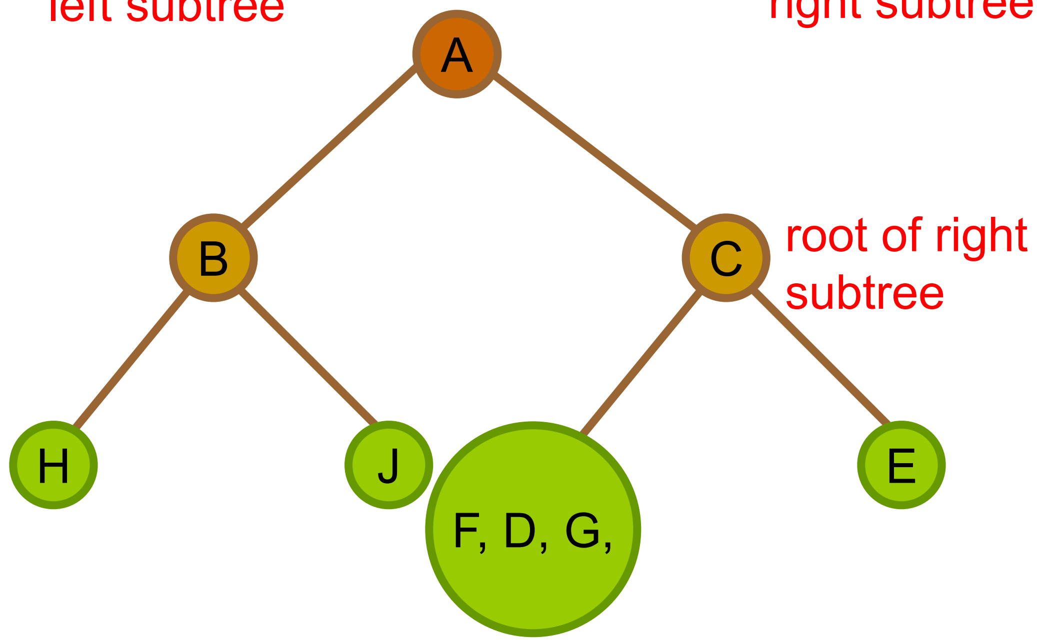
# Consider Right Subtree of Root

root node of the subtree

- Postorder: ~~H J B~~ F G D E C ~~A~~
- Inorder: ~~H B J A~~ F D G C E

left subtree

right subtree



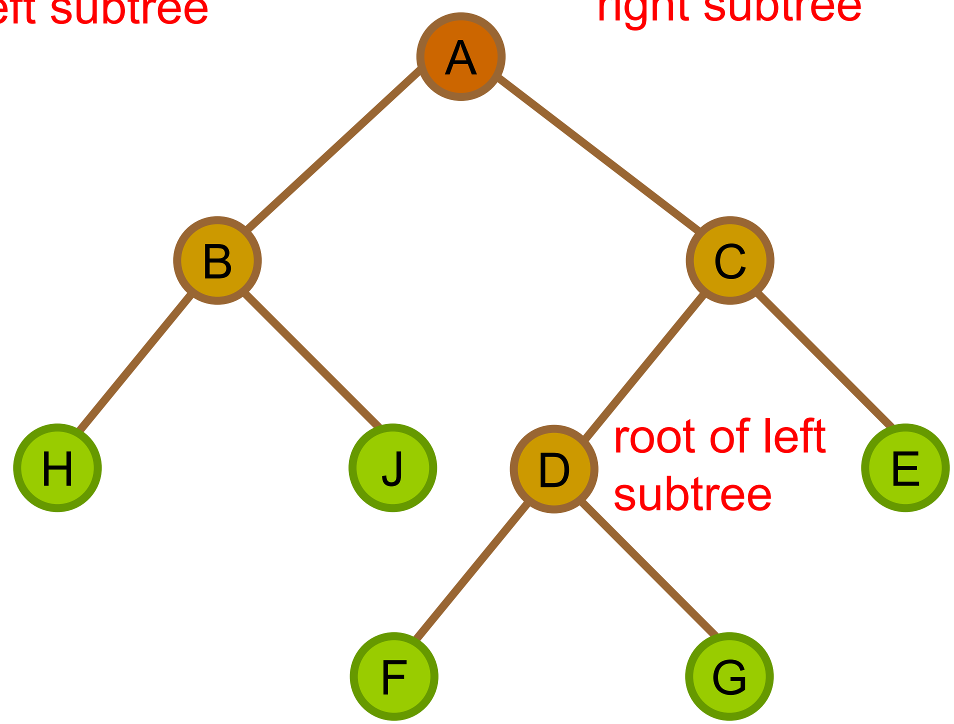
# Consider Left Subtree of C

root node of the subtree

- Postorder: ~~H~~ ~~J~~ ~~B~~ F G D ~~E~~ ~~C~~ ~~A~~
- Inorder: ~~H~~ ~~B~~ ~~J~~ ~~A~~ F D G ~~C~~ ~~E~~

left subtree

right subtree

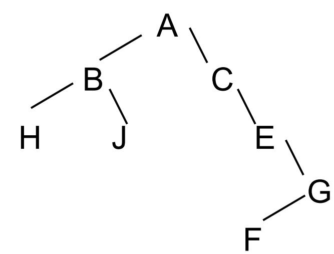


# In-class exercise

- Plot the tree given its in-order traversal and post-order traversal as:

in-order: HBJACEFG

post-order: HJBFGECA




Reasoning process: given the post-order, root is A. Left subtree's post-order is HJB, in-order is HBJ. Thus, B is the root, H and J are its left/right child, respectively.
Right subtree's in-order: CEFG, its post-order is FGEC. Thus, the root is C. C's left subtree is empty (CEFG). C's right subtree's in and post are: EFG and FGE. So E is the root and its left subtree is empty. Its right subtree's in and post are: FG and FG. G is the root and F is the left child.

# Summary

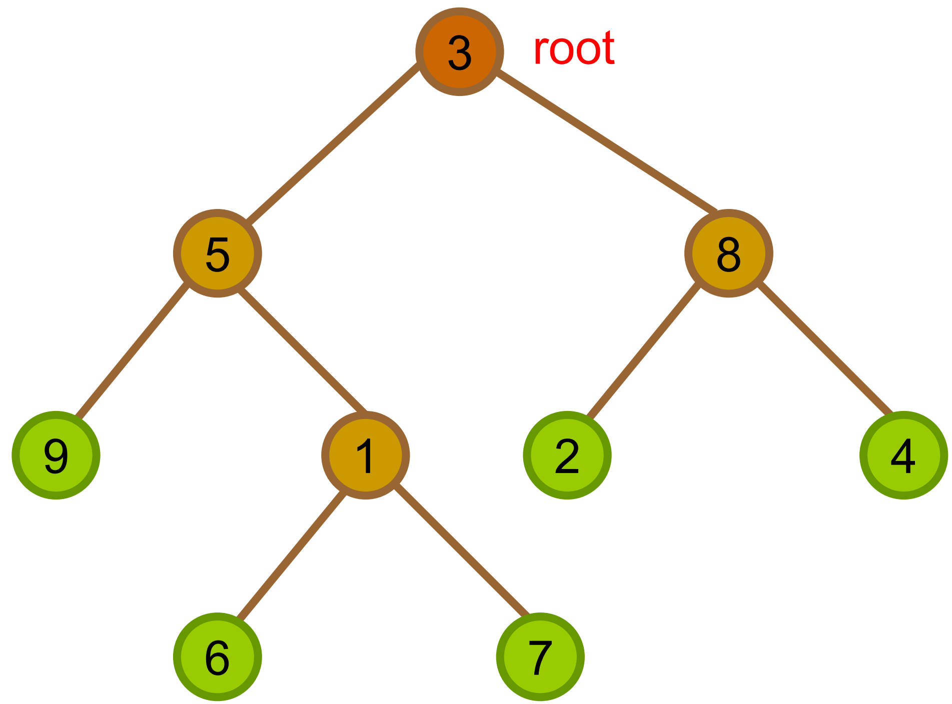
- For postorder, the **last node** is the root
- For preorder, the **first node** is the root
- For inorder, the nodes on the **left hand side** of last node of postorder (or first node of preorder) belongs to the left subtree, nodes on the **right hand side** belongs to its right subtree
- Apply this principle recursively in left/right subtrees

# Binary Search Tree (BST)

and its operations

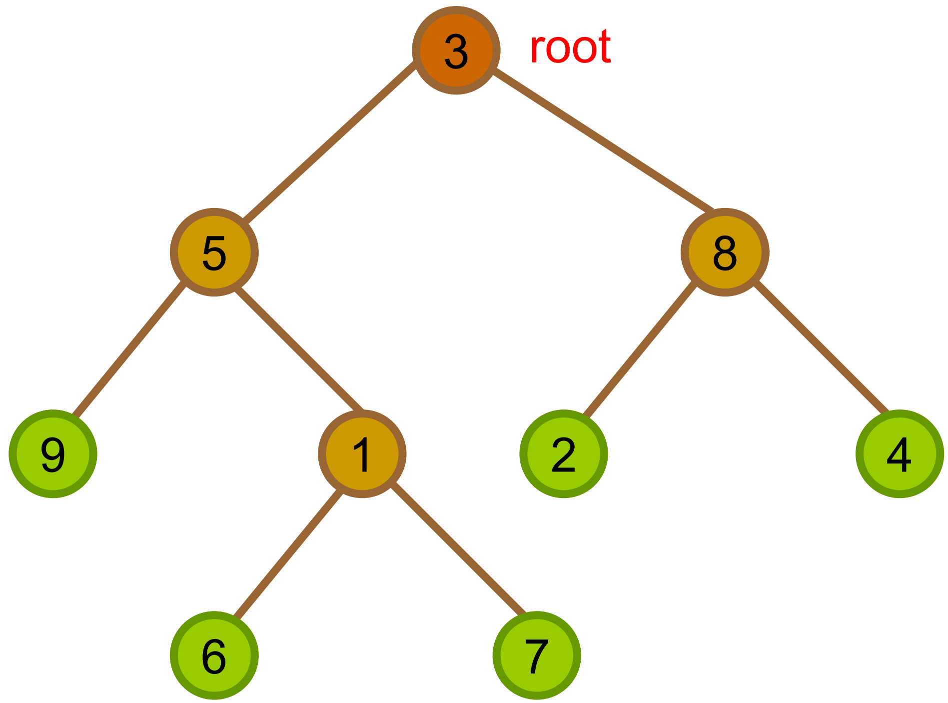
# How to Search a Tree?

- Suppose we have a binary tree like this
- Each node contains an integer data
- How do you find the node that contain value = *k*?
- Can you determine the max./min. node value?




Recall that “search” is an important operation for database. E.g. the tree represents cityu students. Each node represents one student.

# How to Search a Tree?




array:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 8 | 9 | 1 | 2 | 4 | - | - | 6 | 7 |

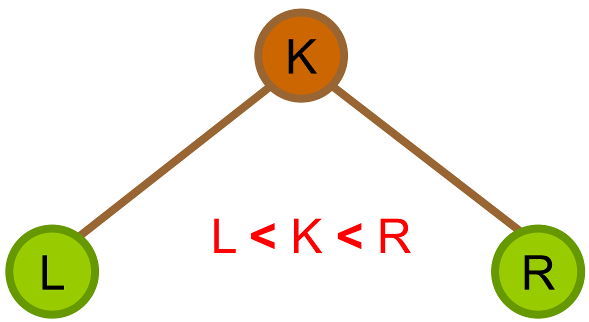
for-loop

In order to find the max. node, min. node or a node equal to particular value, you have to visit the entire tree once

How about linked list implemented tree?

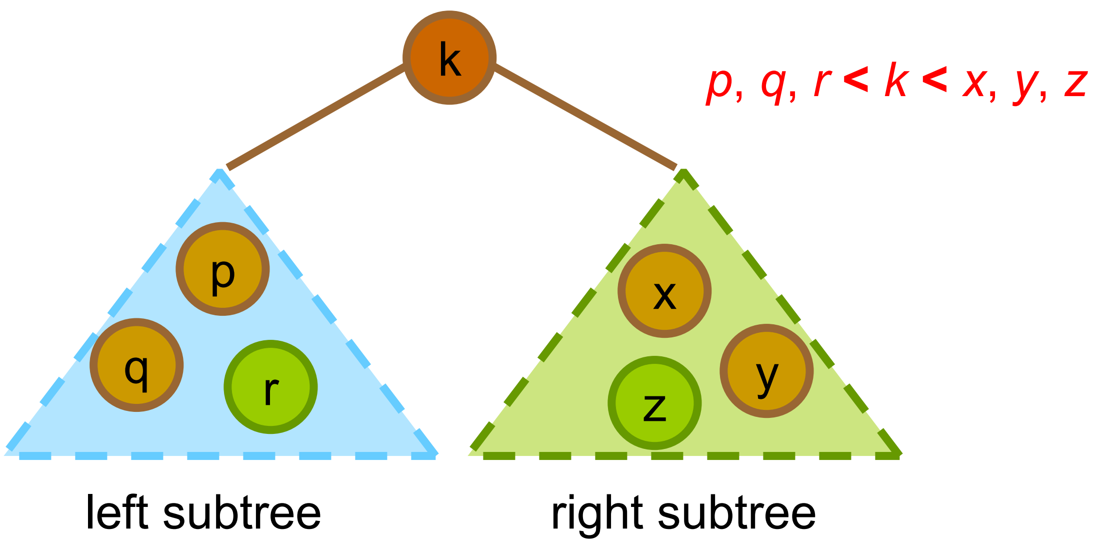
# Pre-sort Tree

- How about if the tree is pre-sorted in some sense
- The value stored at a node is greater than the value stored at its left child, but less than the value stored at its right child
- This arrangement of nodes allow us to make decision of a searching going along its left or right path.

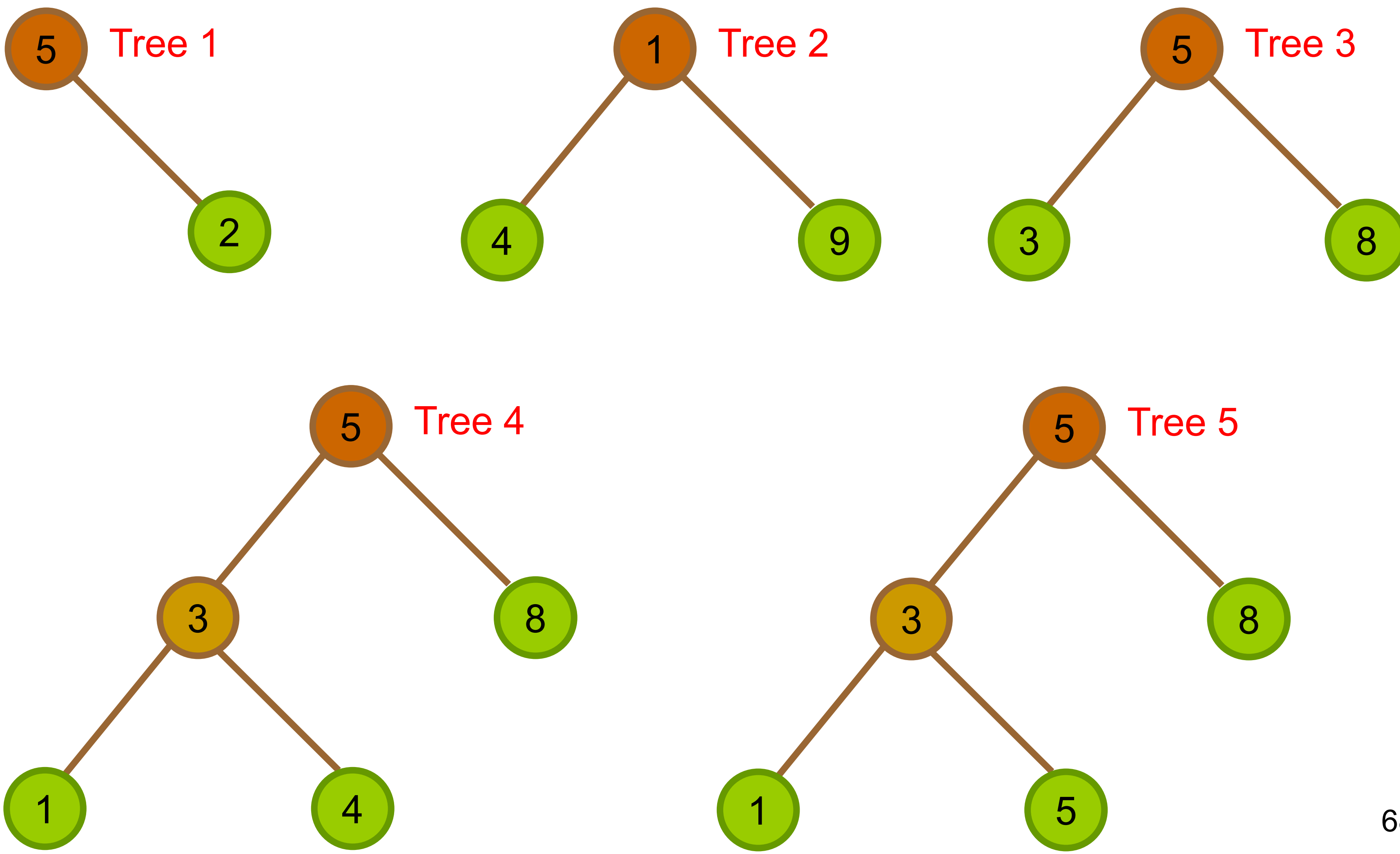


# Binary Search Tree (BST)

- A binary search tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:
- Every element has a key field and no two elements in the BST have the same key, i.e. all keys are distinct. (Example, student ID is a key field in the student record.)
- The keys (if any) in the **left subtree** are smaller than the key in the root.
- The keys (if any) in the **right subtree** are larger than the key in the root.
- The left and right subtrees are also BST (recursively applied).

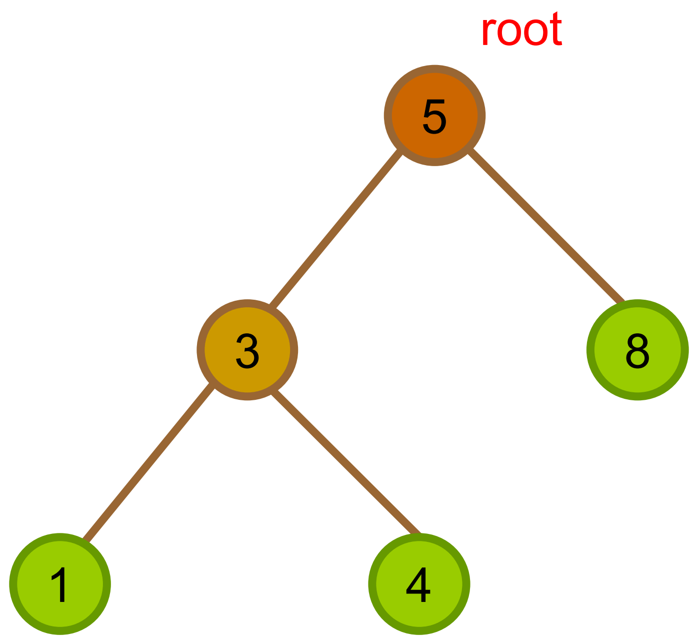


# Exercise: Are They BST?



# Find a Node in BST

- How to find a node with value = *k*?



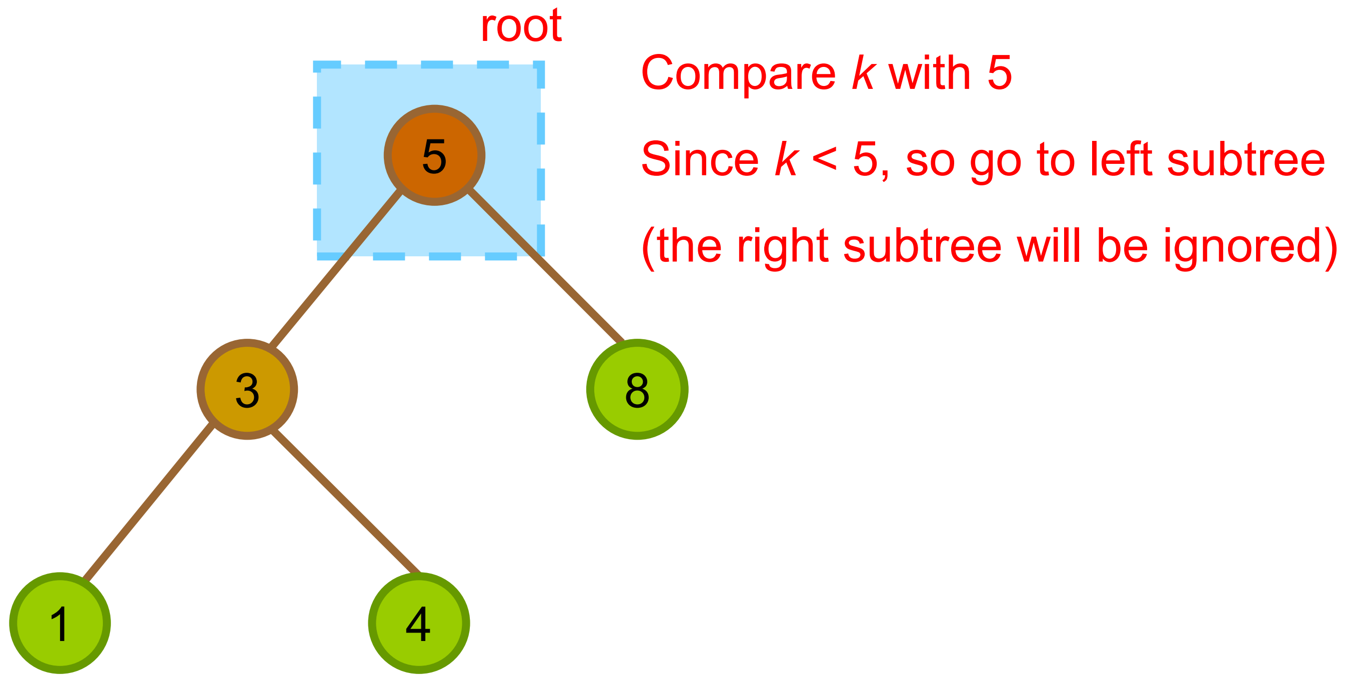
# Find a node in BST

- Compare *k* with the value of root
- If value of root == *k*, the answer is root!
- If value of root > *k*, go to the left subtree
- If value of root < *k*, go to the right subtree

- Continue to compare recursively until it meets a leaf node

# Find a Node in BST

- e.g. $k = 1$



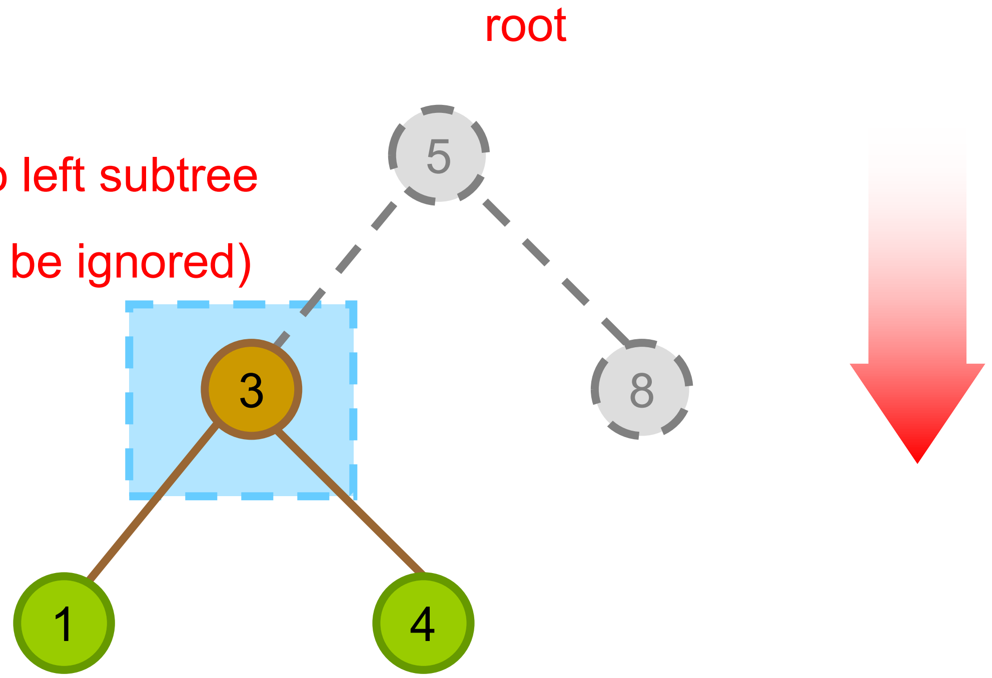
# Find a node in BST

- e.g. $k$ = 1

Compare $k$ with 3

Since $k$ < 3, so go to left subtree

(its right subtree will be ignored)



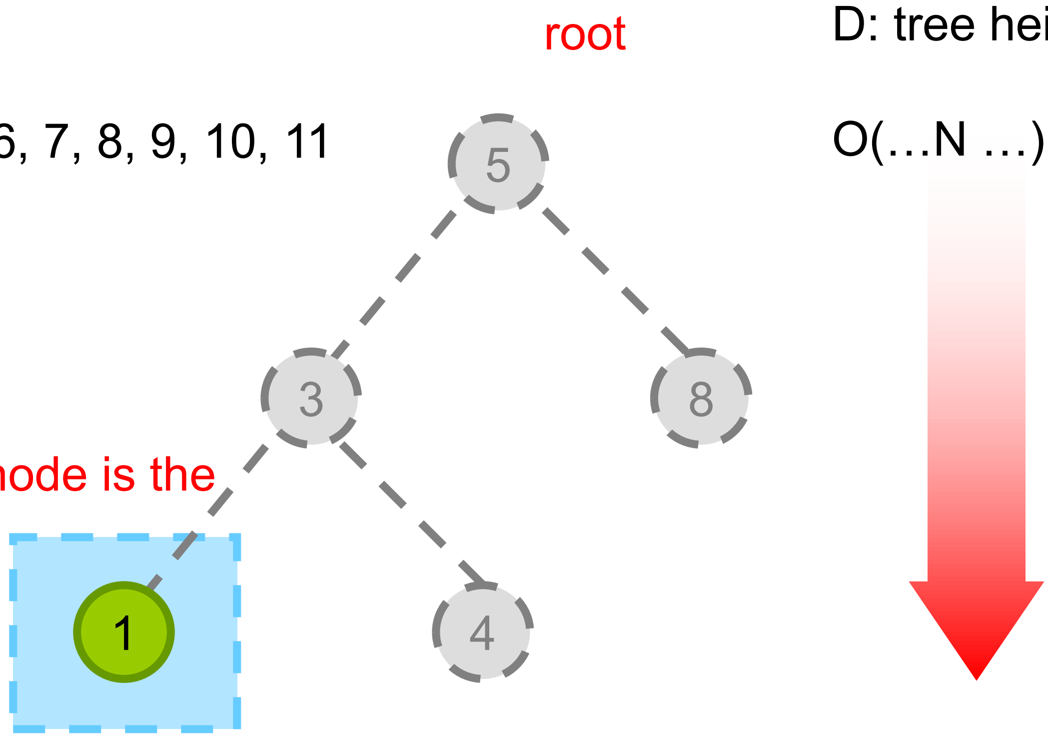
# Find a node in BST

- e.g. $k = 1$

N: number of nodes
D: tree height/depth

O(…N …)

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11




Compare $k$ with 1

Since $k$ == 1, this node is the solution!

How about if we want to find a node with value equal to 2?

# Time Complexity

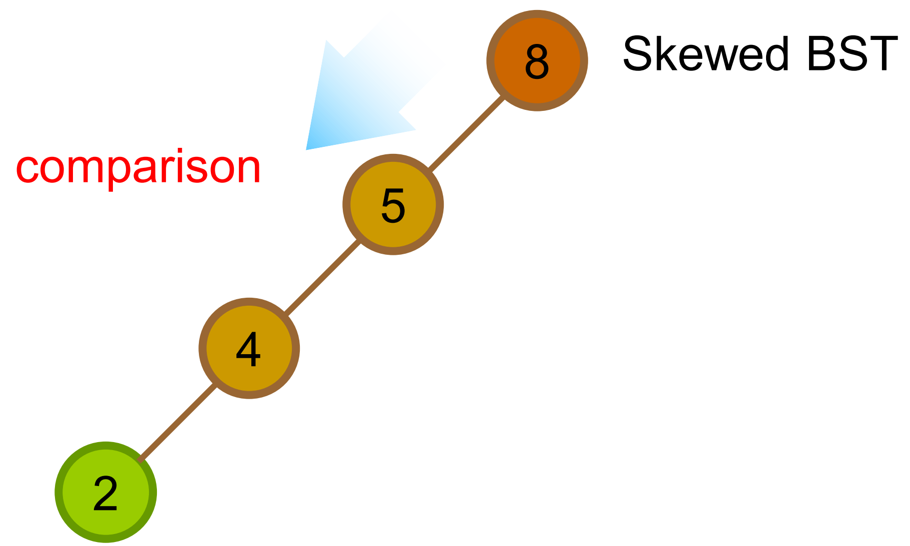
- What's the time complexity of the **find** function?
  - Time complexity is proportional to the no. of comparison
  - The max. no. of comparison = no. of levels of the tree

# Complete BST

- If it is a complete BST
  - After each comparison, either left subtree or right subtree will be **ignored**
  - About half nodes do not require to consider after <u>each comparison</u>
  - The depth of the tree is *floor($log_2 n$)*
- **Average case**: O($\log_2 n$), where $n$ is the total no. of nodes

# Skewed BST

- If it is a skewed BST
  - The depth of the tree is *n-1*
- **Worst case**: O(*n*)




Conclusion: it is very important to maintain a complete BST

# Non-Recursive Search BST

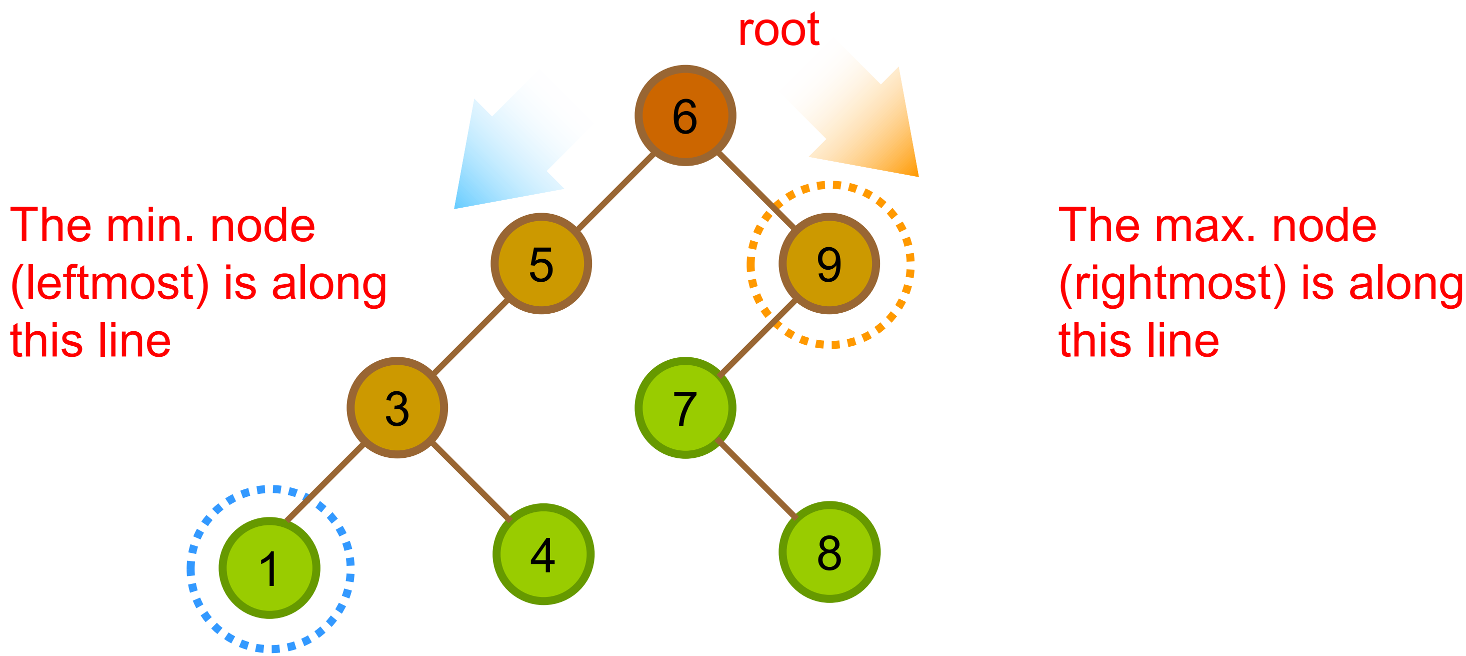
```
// implemented as a member function of BST class
template<class Type>
treeNode<Type>* search(const Type& x) {

  treeNode<Type> *p = root;      // point to root node of the tree
  while (p != NULL && x != p->info) {    // compare key field
     if (x < p->info)
       p = p->left;
     else
       p = p->right;
  }
  return p;
}
```

# Recursive Search BST

```
template<class Type>
treeNode<Type>* search(treeNode<Type> *p, const Type& x) {
  if (p == NULL)
    return NULL;

  if (x == p->info)
    return p;

  if (x < p->info)
    return search(p->left, x);
  else
    return search(p->right, x);
}
```

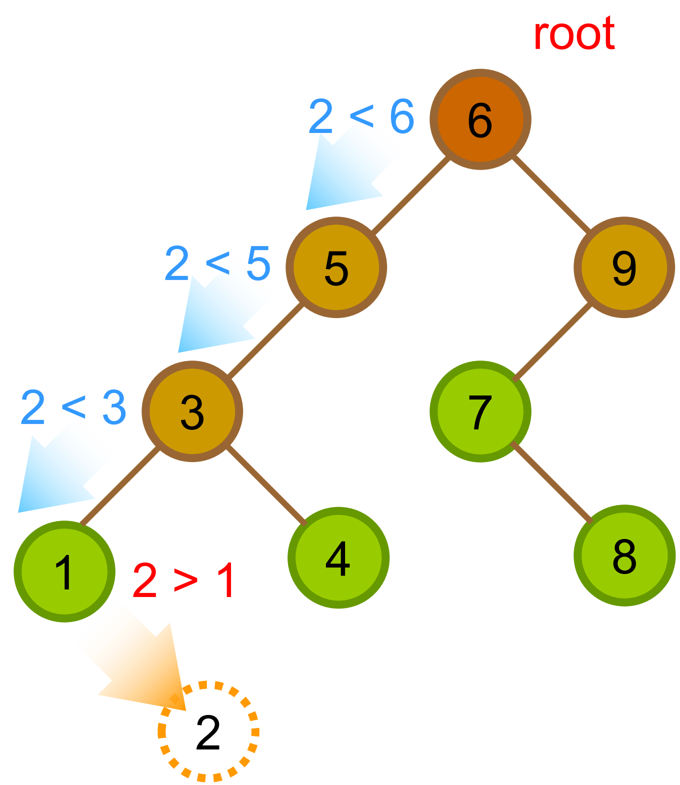
# In-Class Exercise: Min & Max Node of BST

- Exercise: write the code to find the min and max node (using recursion/iteration)



# Insert a Node In BST

- How to insert a node in BST?
  - e.g. insert(2)
- Two major steps:
  - Verify if the new element is not in the BST
  - Determine the point of insertion

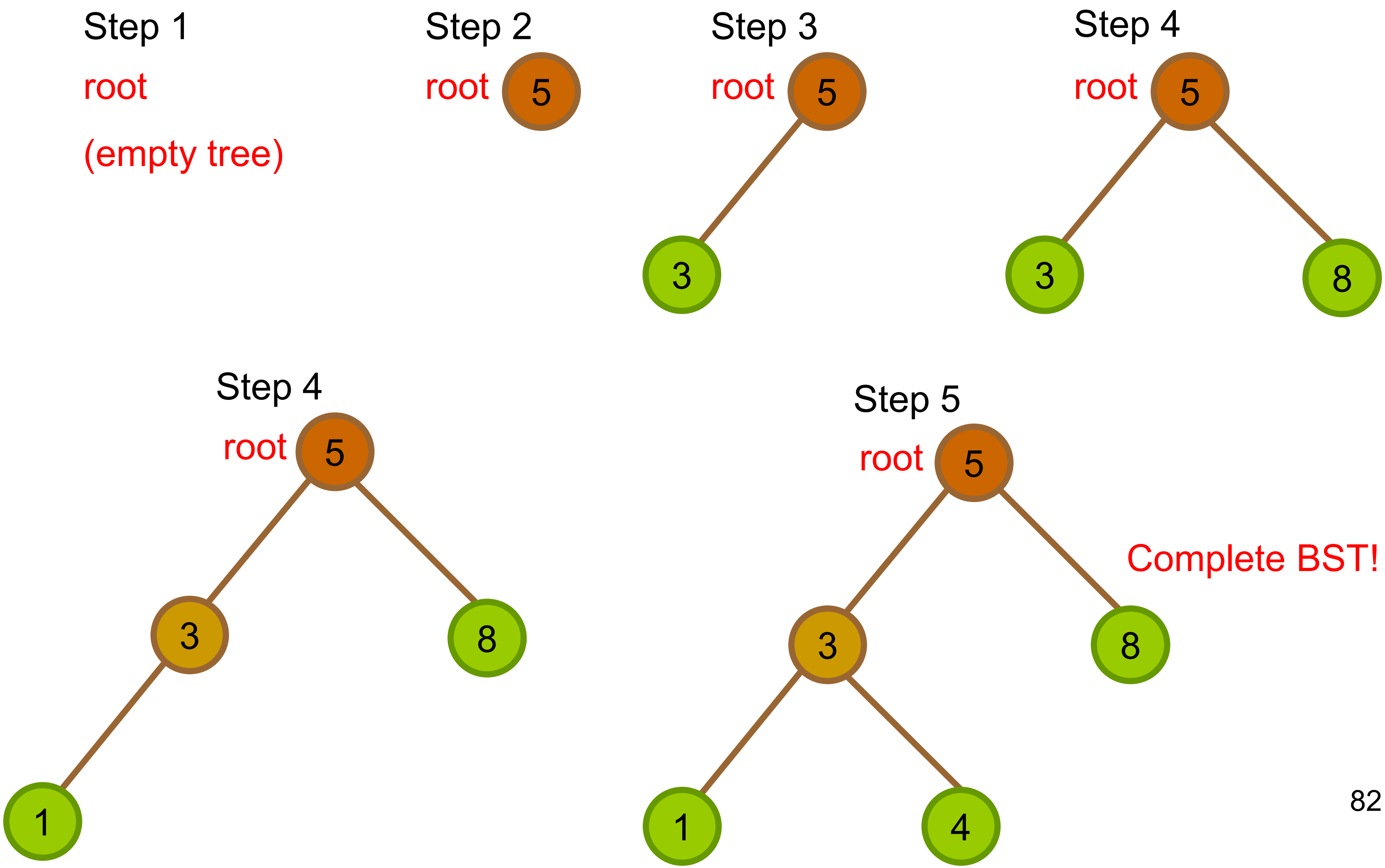



https://www.cs.usfca.edu/~galles/visualization/BST.html //visualization

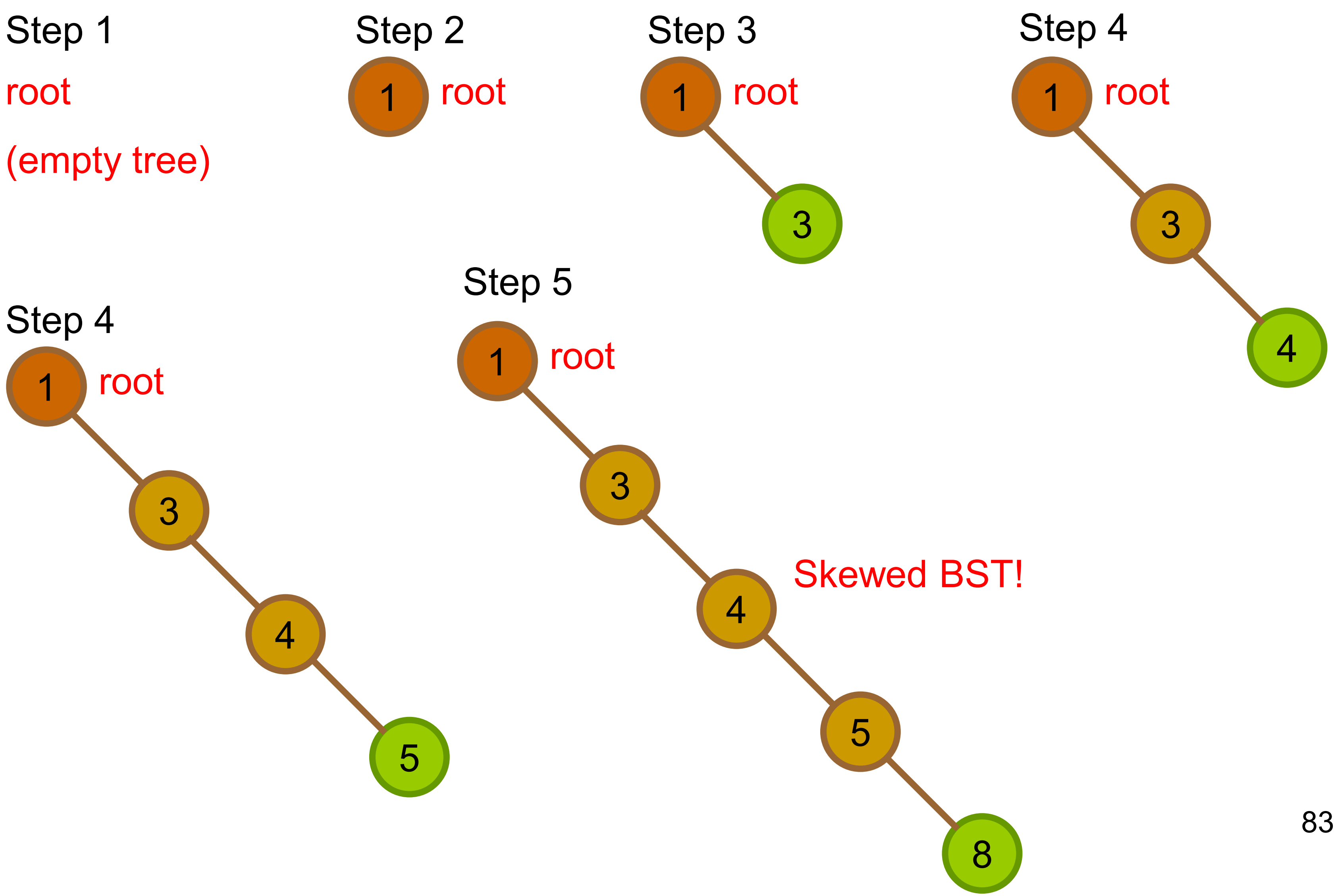
# Order of Inserting Elements

- Does the order of inserting elements into a BST matter?
  - Yes, certain orders could produce very unbalanced trees
  - e.g. compare the resultant tree if inserting the elements in these order:
    - 1) 5, 3, 8, 1, 4 and
    - 2) 1, 3, 4, 5, 8
- Unbalanced trees are not desirable because search time increases

# Insert Order: 5, 3, 8, 1, 4



# Insert order: 1, 3, 4, 5, 8



# Insert Node to BST

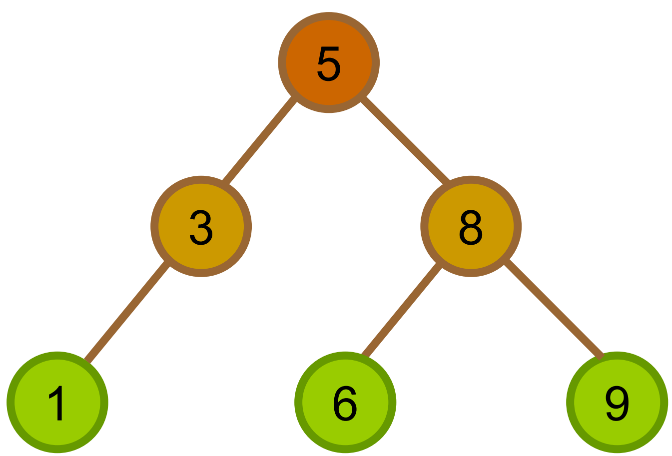
- The insertion function returns the pointer to the newly inserted node or the node with the given key value.

```
template<class Type>
treeNode<Type>* insert(const Type& x) {
  treeNode<Type> *p, *q;

  q = NULL;  // parent of p
  p = root;  // point to root
  while (p != NULL) {
    //element already exists
    if (x == p->info)
         return p;
    q = p;
    if (x < p->info)
         p = p->left;
    else
         p = p->right;
  }
```

```
  treeNode<Type> *v = new treeNode<Type>;
  v->info = x;
  v->left = v->right = NULL;

  if (q == NULL)   // empty tree
     root = v;
  else if (x < q->info)
     q->left = v;
  else
     q->right = v;

  return v;
}
```

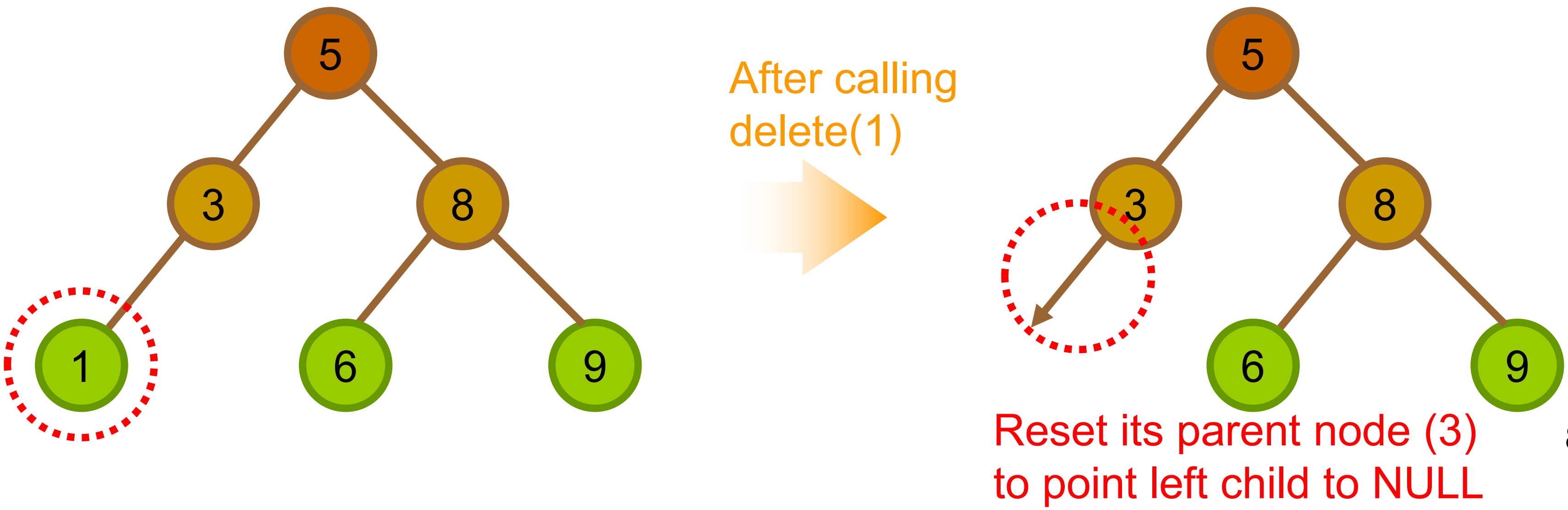
# Delete a Node in BST

- The property of BST must be **preserved** after deletion
- We have to consider 3 different cases
  - The node to be deleted is:
  - 1) A leaf node (e.g. node 1)
  - 2) A node has only one child (e.g. node 3)
  - 3) A node has two children (e.g. node 5)



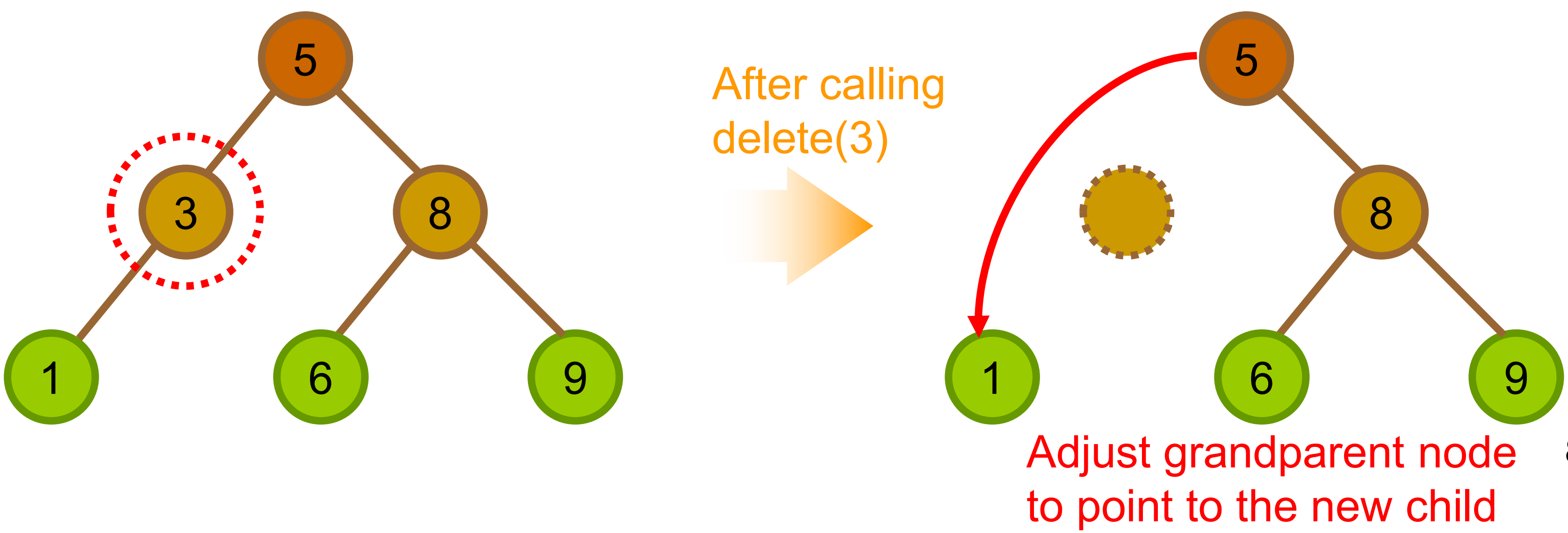
# Delete Node: Case 1

- **Degree 0 Node (leaf node)**
  - Just delete it
  - Then reset the reference of its parent node



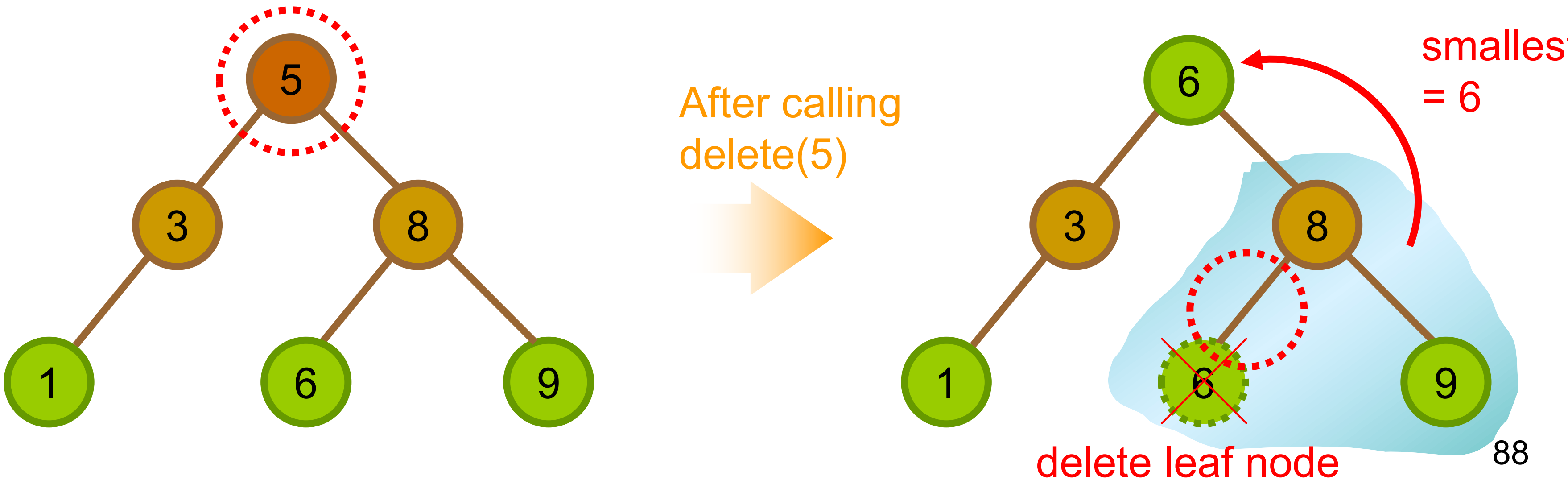
# Delete Node: Case 2

- **Degree 1 Node (with 1 child)**
  - Before deletion, adjust the pointer of parent to point to the grandson
  - Then simply delete it



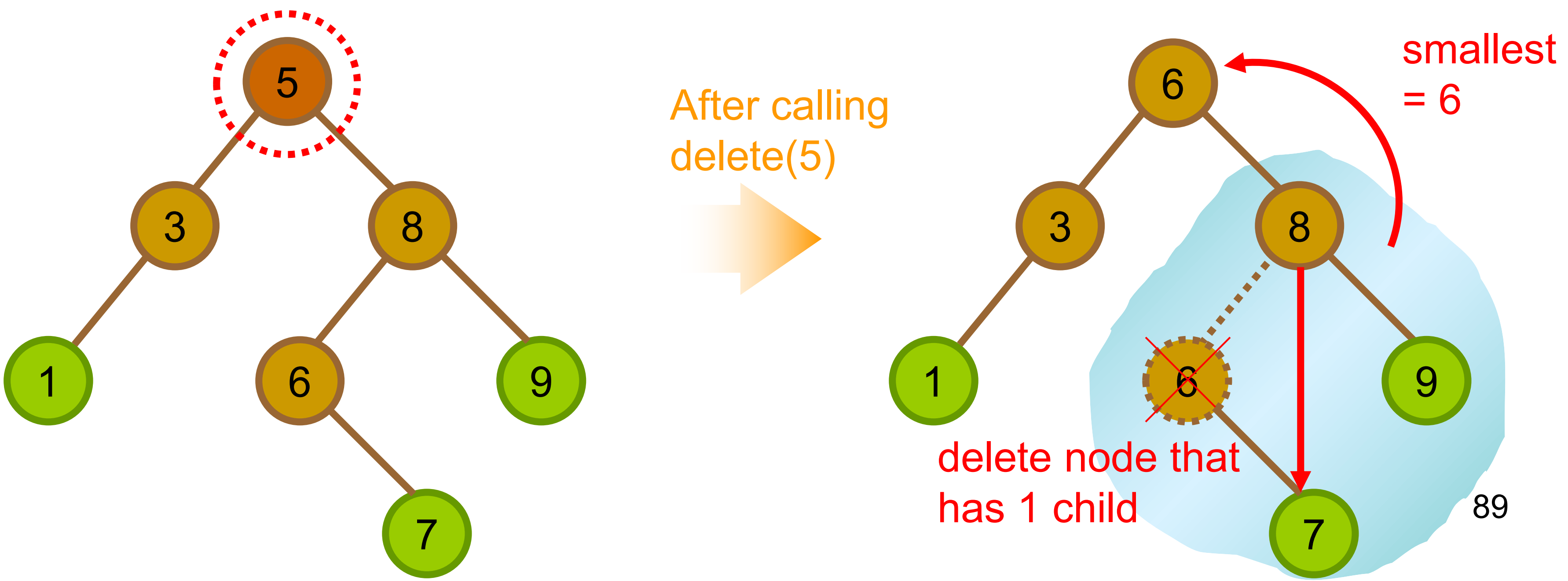
# Delete Node: Case 3

- **Degree 2 Node (with 2 children)**
  - Replace the deleted node with its inorder predecessor (biggest node in left subtree) or inorder successor (smallest node in right subtree)



# Delete Node: Case 3

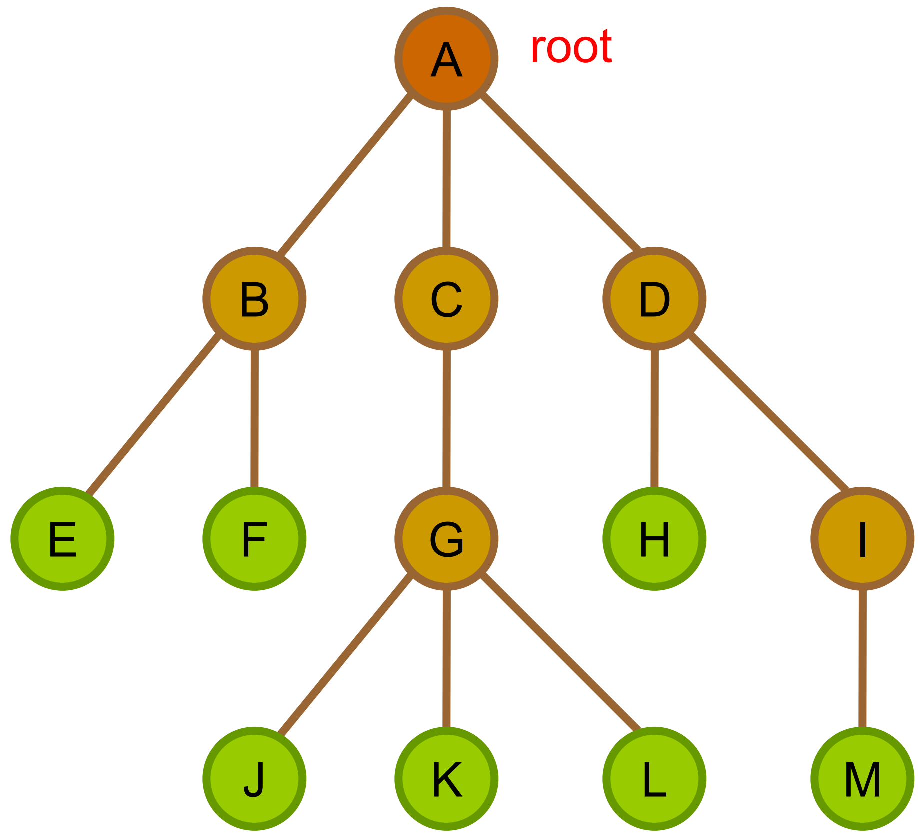
- If the inorder successor or predecessor has a child, delete it in turn with the same steps in case 2.



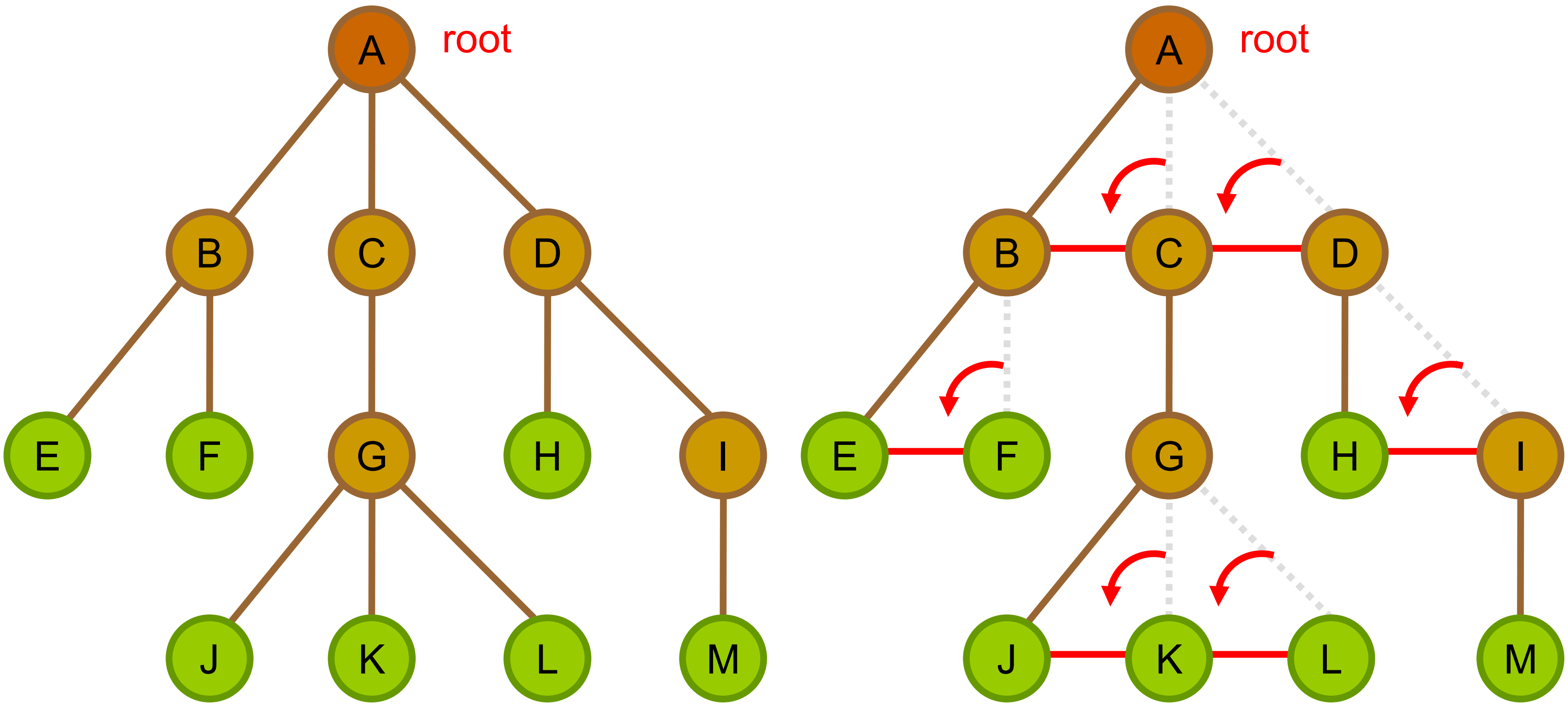
# General Tree to Binary Tree Conversion

# General tree

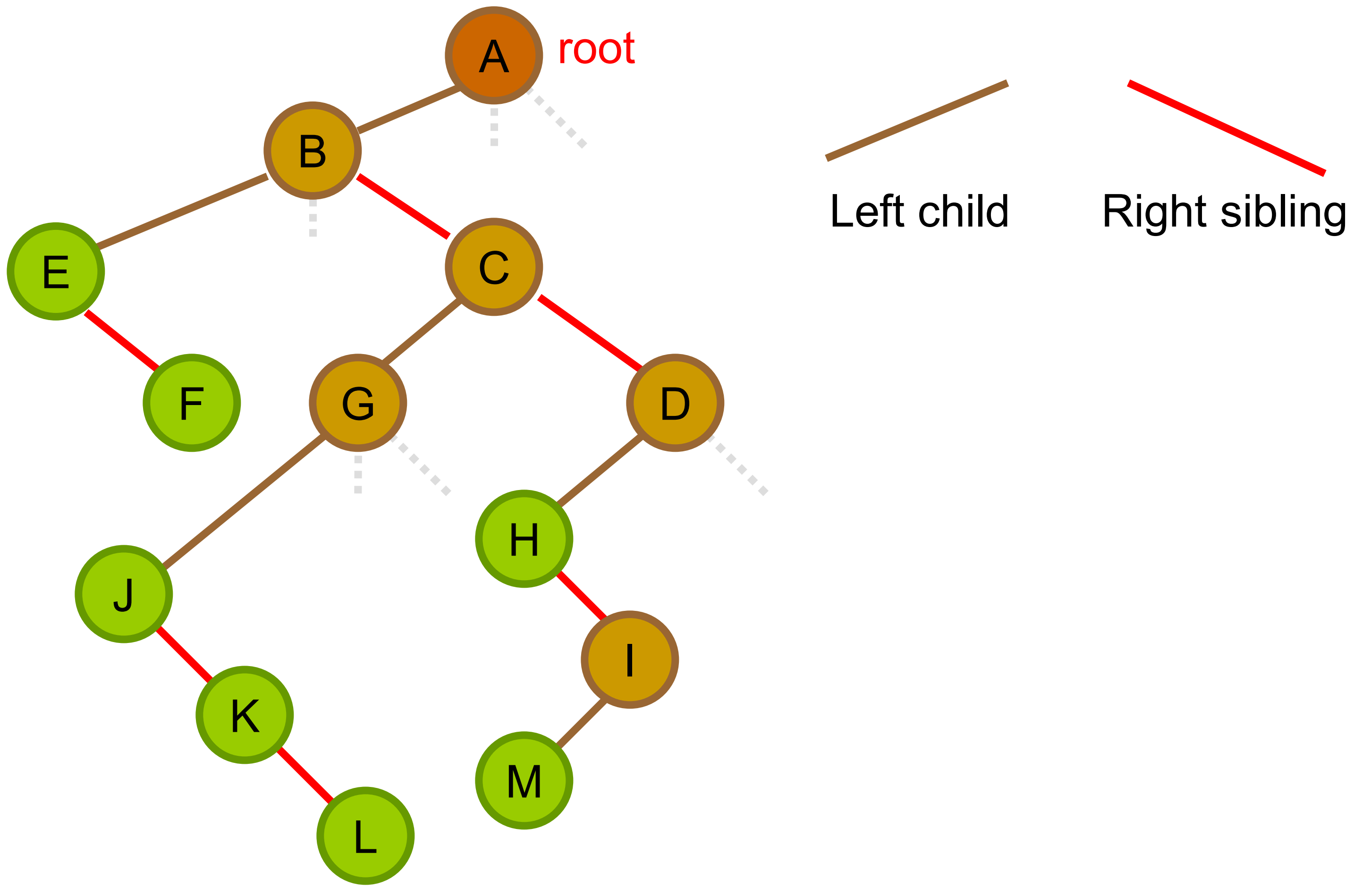
- We go back to the very beginning problem
- How to represent a general tree using binary tree?
  - Left Child Right Sibling Representation



# Left Child Right Sibling



# Left Child Right Sibling



# Count the No. of Leaf Nodes

```
template<class Type>
int countLeaf(treeNode<Type> *p) {
   // p is a general tree represented as a binary tree
   int count;

   if (p == NULL)            // tree is empty
      return 0;

   if (p->left == NULL)      // root has no subtree
      return 1;

   // root has 1 or more subtree.
   // no. of leaf nodes = sum of leaf nodes in the subtrees of the root
   count = 0;
   p = p->left;
   while (p != NULL) {       //for each subtree
      count += countLeaf(p);
      p = p->right;          //move on to the next subtree
   }
   return count;
}
```

# Determine the Height

```
template<class Type>
int height(treeNode<Type> *p)  {
   // p is a general tree represented as a binary tree
   int h, t;

   if (p == NULL)
      return -1;   // leaf node's height is 0

   h = -1;
   p = p->left;
   while (p != NULL) {
      t = height(p);
      if (t > h)
         h = t;
      p = p->right;
   }

   // h = max height of all subtrees
   return h+1;
}
```

# Applications of trees

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Scanning, parsing , generation of code and evaluation of arithmetic expressions in Compiler design.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie : Used to implement dictionaries with prefix lookup.
8. Suffix Tree : For quick pattern searching in a fixed text.
9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks
10. As a workflow for compositing digital images for visual effects.
11. Decision trees.
12. Organization chart of a large organization.
13. In XML parser.
14. Machine learning algorithm.
15. For indexing in database.
16. IN server like DNS (Domain Name Server)
17. In Computer Graphics.
18. To evaluate an expression.
19. In chess game to store defense moves of player.