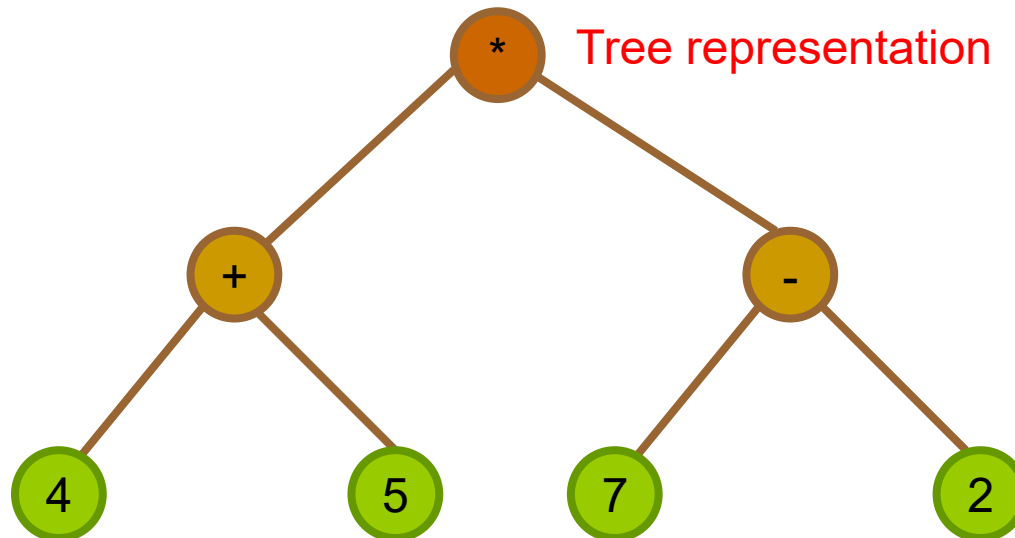


EE2331 Data Structures and Algorithms

Trees

Remember?

- How does a computer evaluate mathematical expressions?
 - e.g. $(4 + 5) * (7 - 2)$
 - Use **postfix expressions** $(4\ 5\ +\ 7\ 2\ -\ *)$
- May we transform it to tree representation?



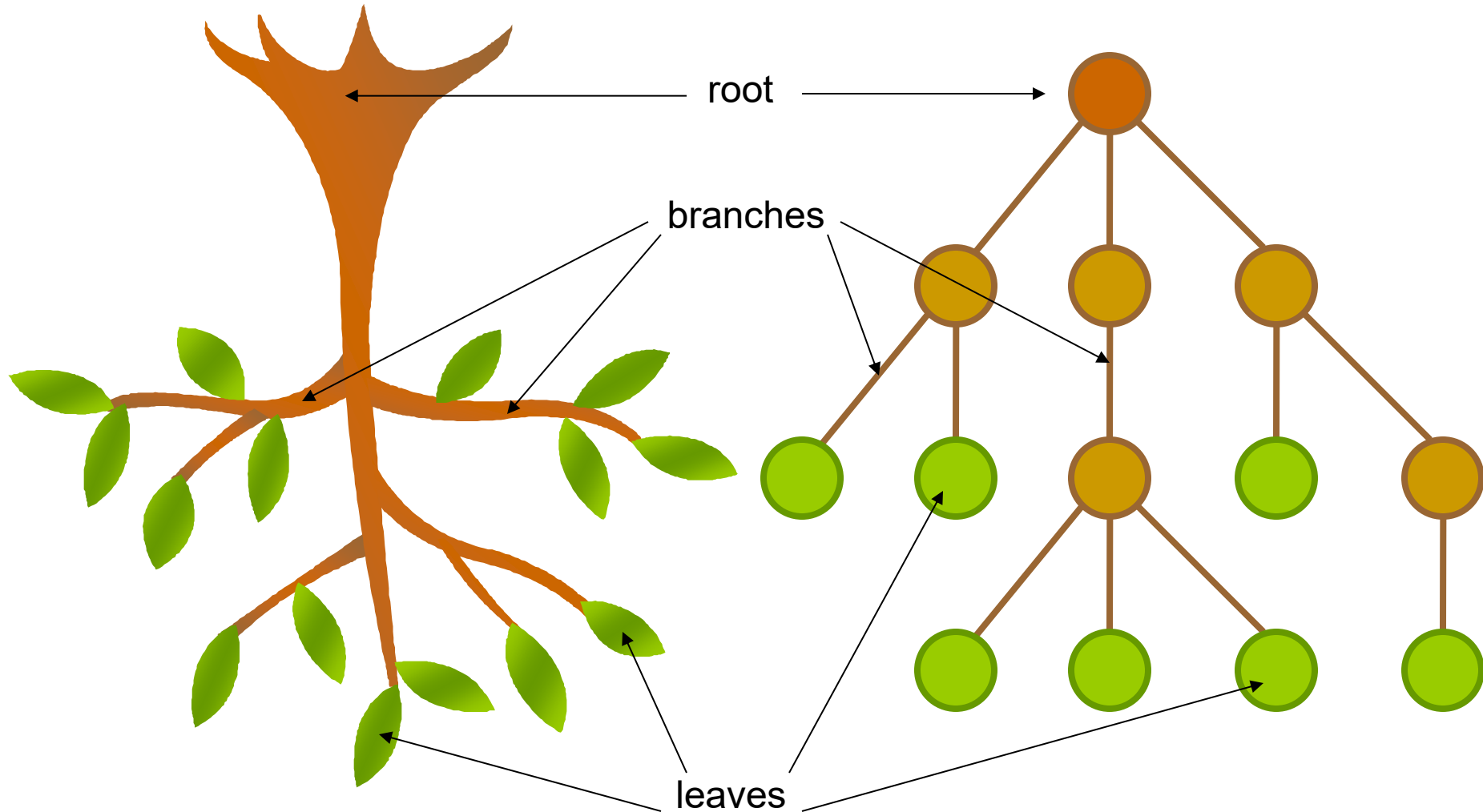
Stacks, Queues vs. Trees

- Data structures discussed so far are **linear**
 - One preceding/succeeding element
 - e.g. **linked lists, stacks, queues**
- **Tree** is a **non-linear** linked data structure
 - Multiple **succeeding elements**
 - **Tree structure is recursively defined**, so tree operations often involve **recursion** and **linked list**

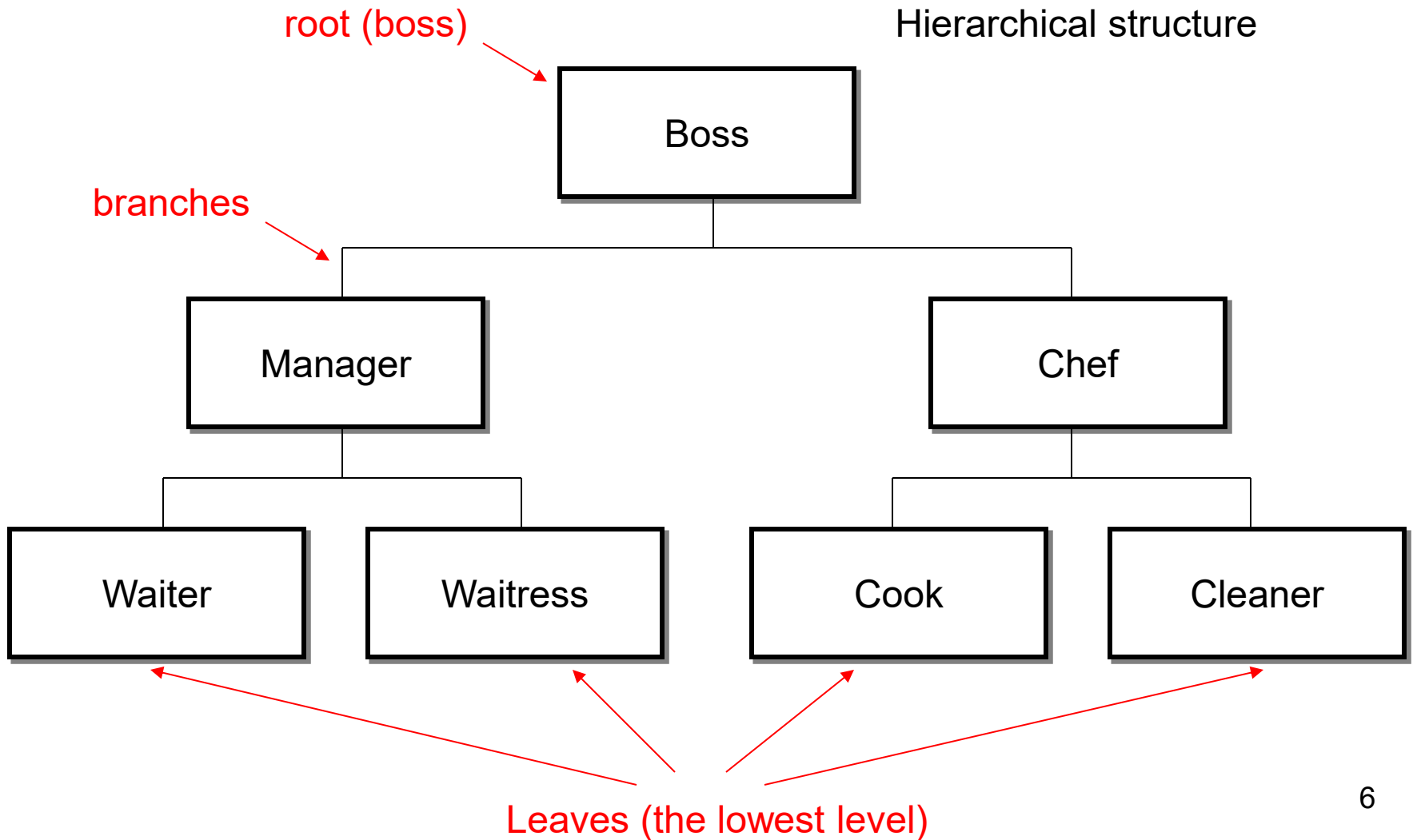
Outline

- Terminology
- Representation
- Binary Trees
- Implementations with Array and Linked List
- Common Operations of Binary Tree
- Trees Traversal
 - Preorder, inorder, postorder, level order
- Relationship between Trees, Stacks and Queues
- Reconstruction of Binary Trees
- Special Binary Trees
 - Binary Search Trees
 - Heap Trees
- Applications
- General Trees and Other tree structures

An Inverted Tree



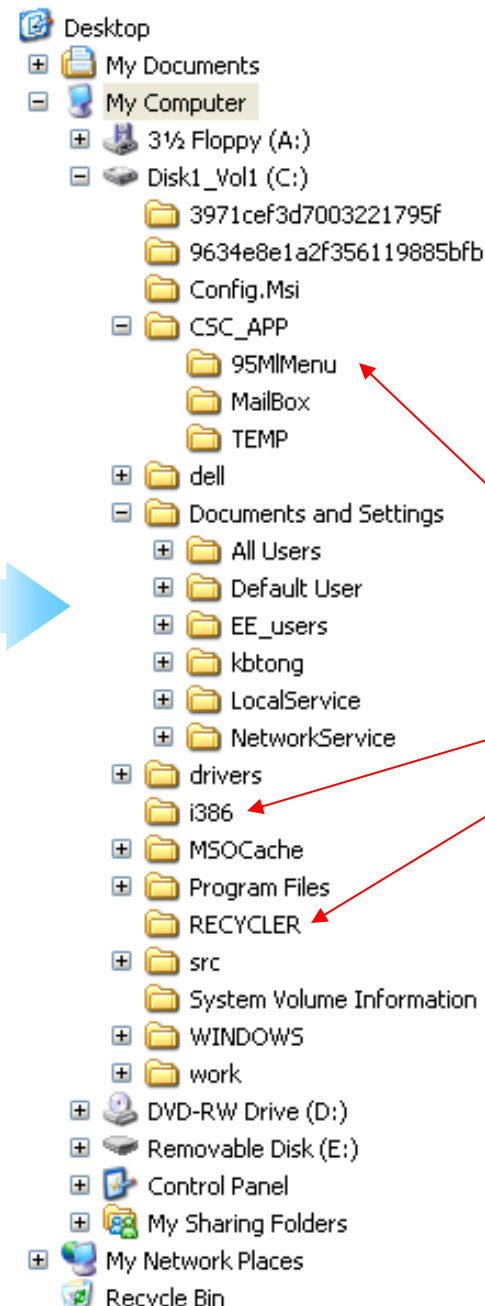
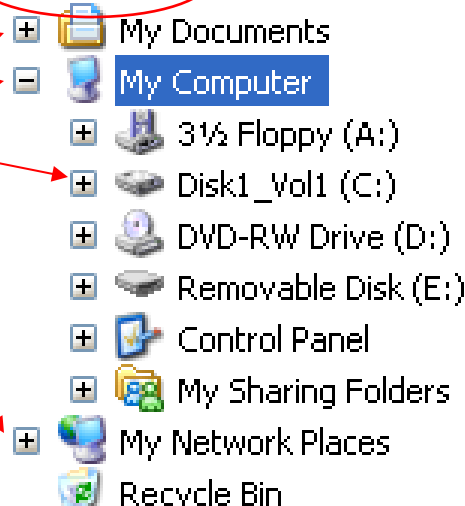
Tree Example: Restaurant



File System

root (desktop)

branches

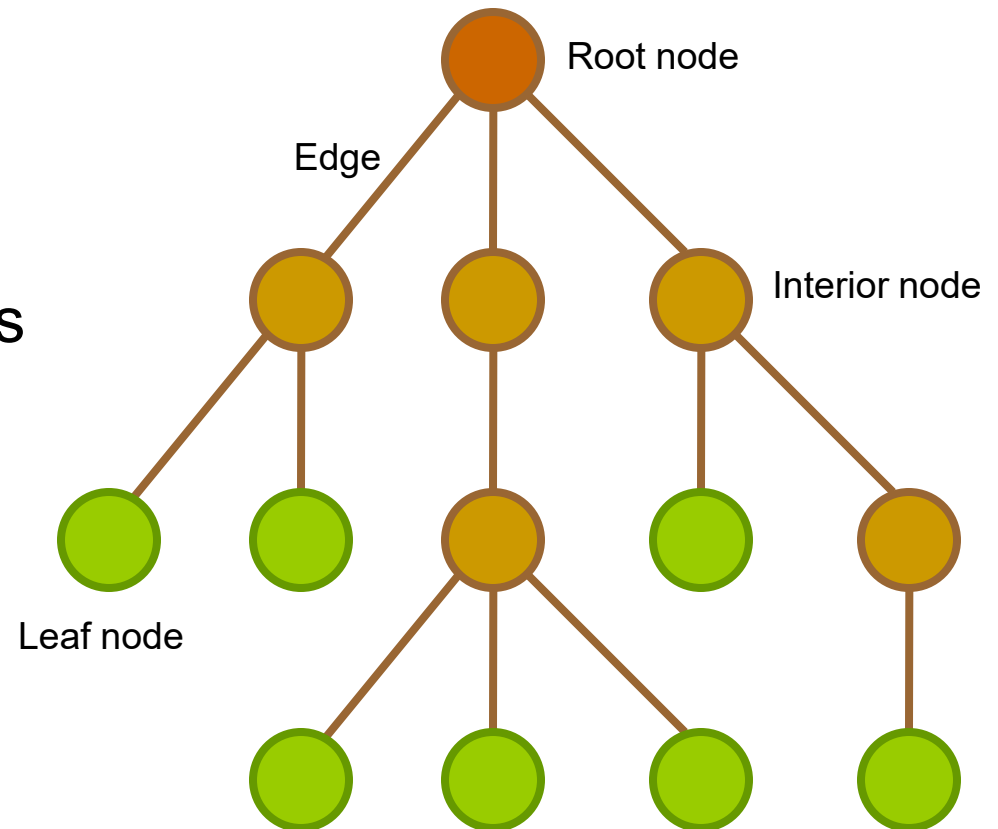


leaves

Composition of a Tree

■ Types of tree node

- Root node (the top node in a tree)
- Interior nodes (nodes with at least one child)
- Leaf nodes (nodes with no children)

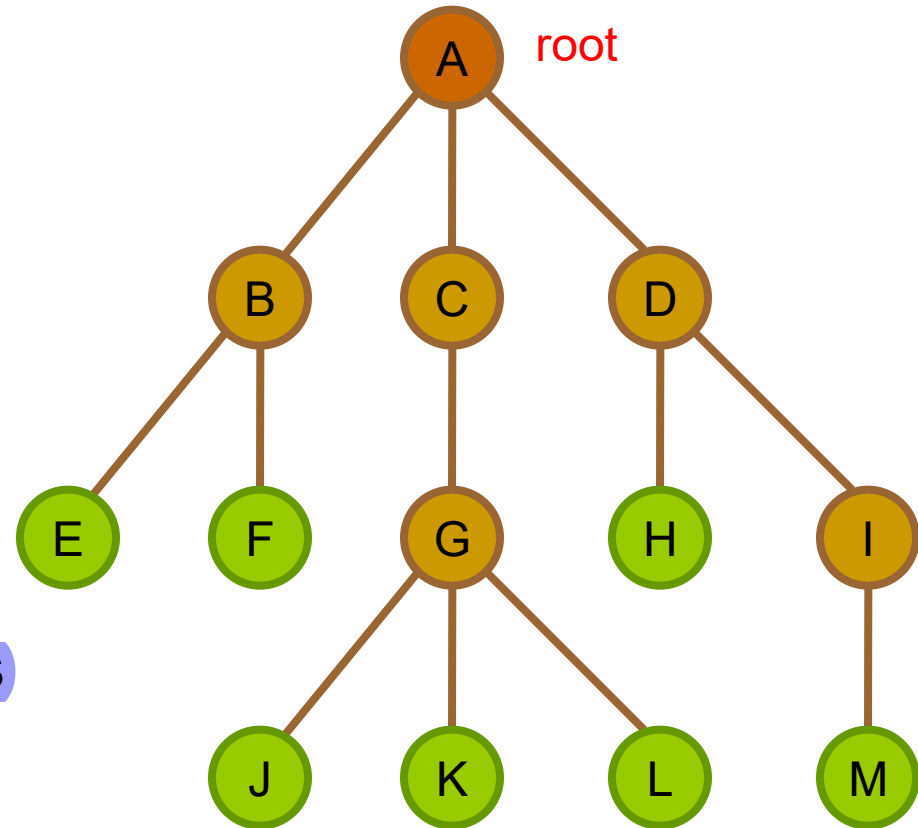


Property of Trees

- Nodes represent information (data)
 - Branches represent **links** between the **nodes**
 - If the total number of nodes (i.e. **root node**, **interior nodes** and **leaf nodes**) is **n** , how many branches in the tree?
 - Number of branches is **$n - 1$**
-

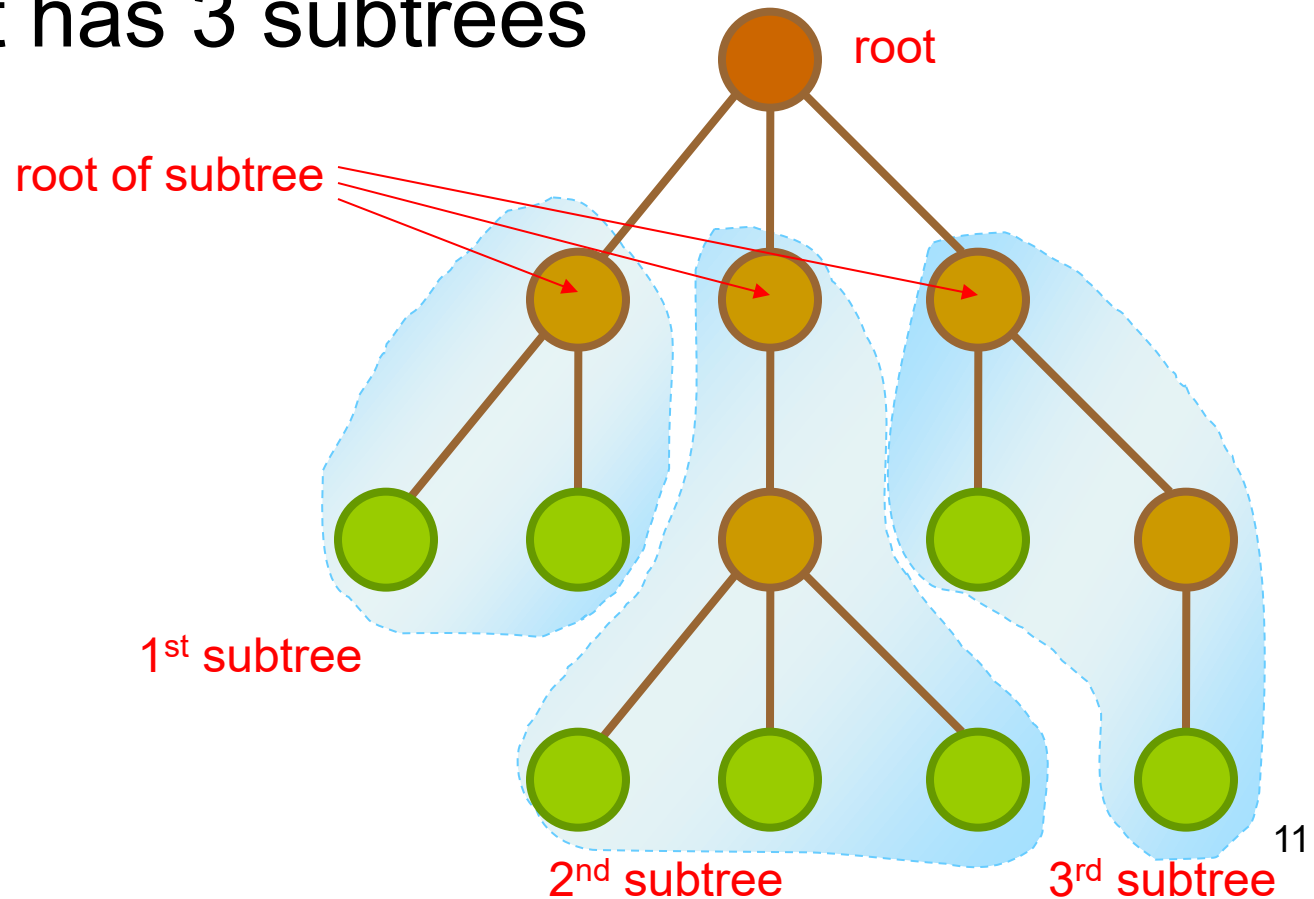
Parent, Children & Sibling

- This tree has 13 nodes
- Node A has 3 children
 - Nodes B, C and D
- Node A is the parent of
 - B, C and D
- Node G is the parent of
 - Nodes J, K and L
- Node G is the child of C
- J, K and L are sibling nodes (share the same parent)



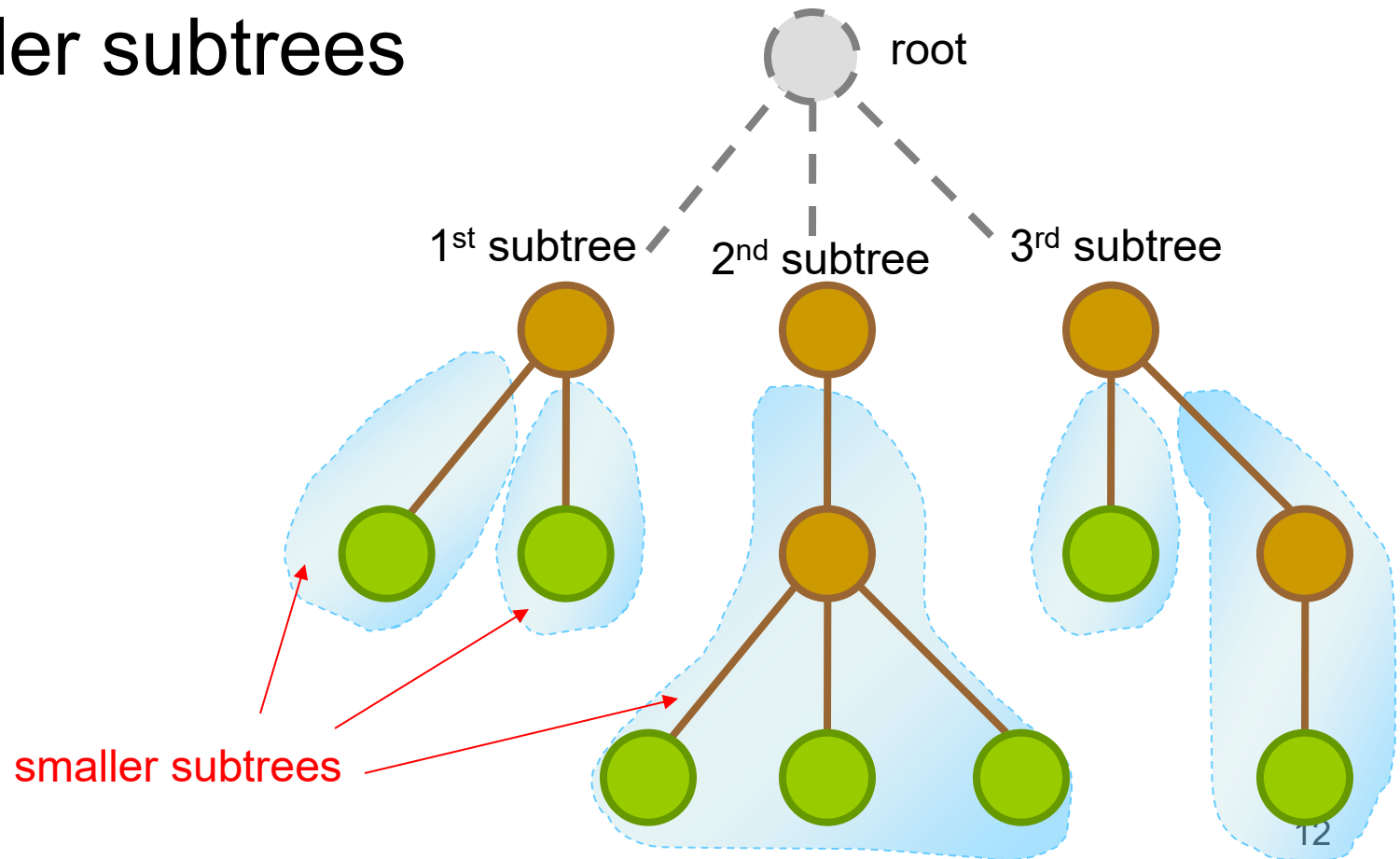
Subtrees

- A tree is composed of several subtrees
- e.g. root has 3 subtrees



Smaller subtrees

- A subtree can be further broken down into smaller subtrees

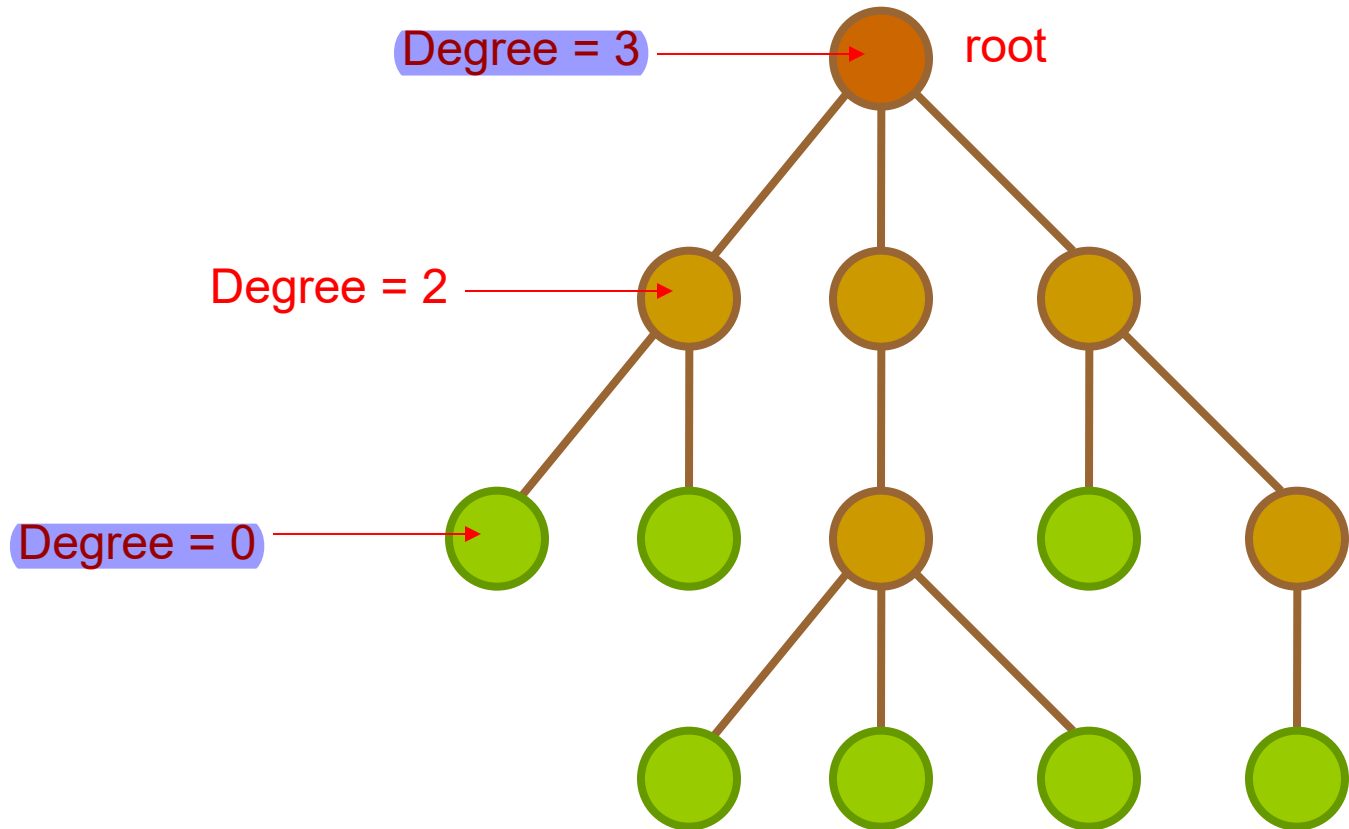


Ancestor and Descendant

- A simple path is a sequence of nodes n_1, n_2, \dots, n_k such that the nodes are all distinct and there is an edge between each pair of nodes $(n_1, n_2), (n_2, n_3), \dots, (n_{k-1}, n_k)$
- The nodes along the simple path from the root to node x are the ancestors of x
- The descendants of a node x are the nodes in the subtrees of x
- Length of a path = no. of branches on the path

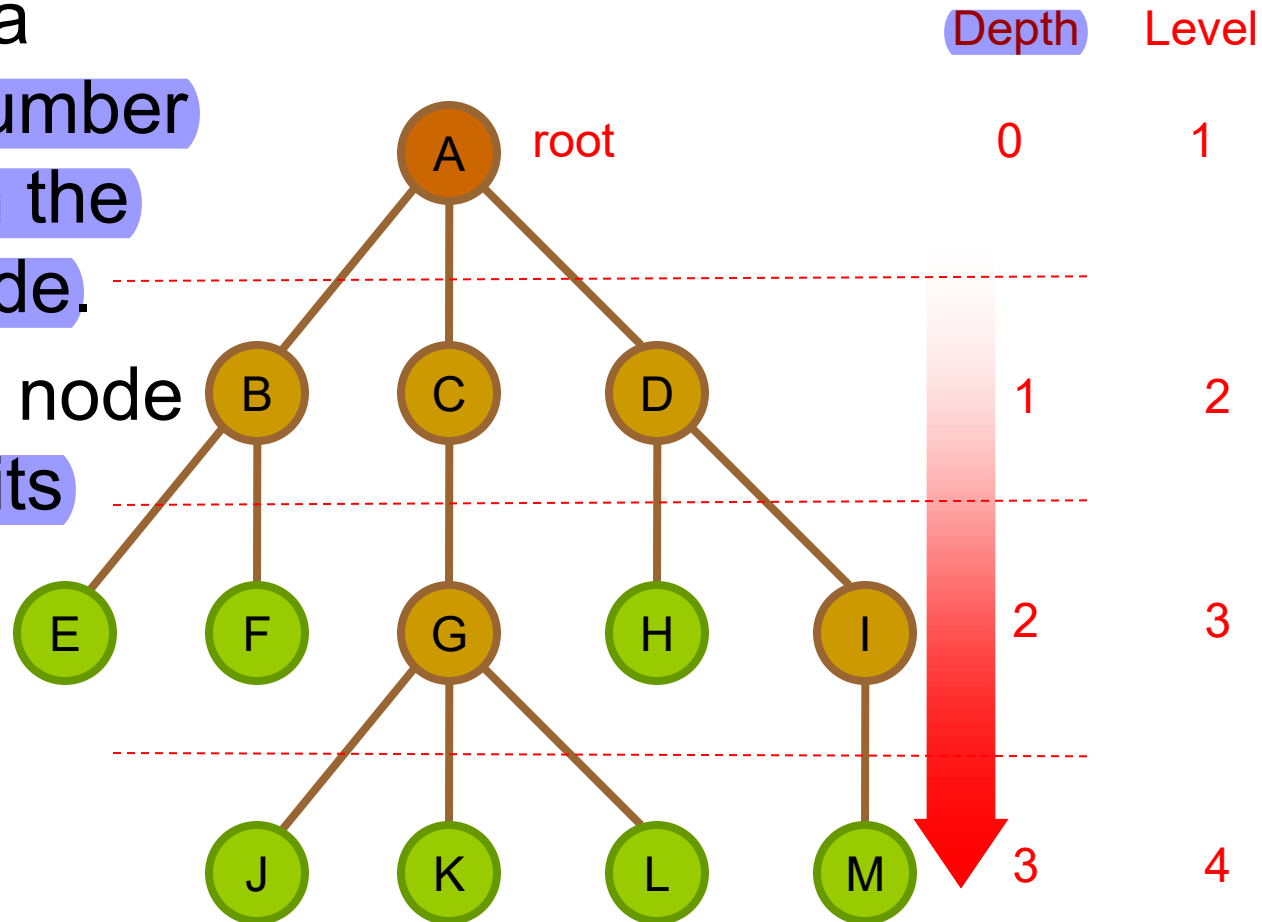
Degree

- The number of subtrees of a node



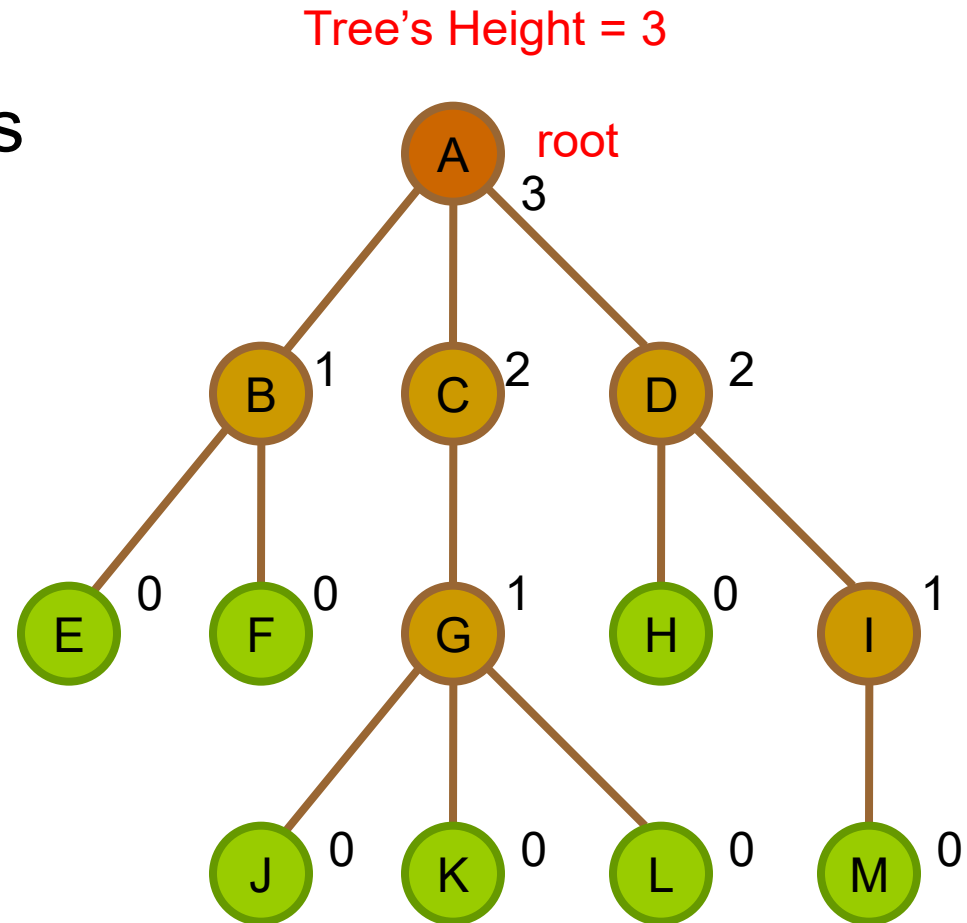
Depth and Level

- The depth of a node is the number of edges from the root to the node.
- The level of a node is defined by its depth plus 1.



Height

- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.



Class Exercise

- A node has no parent is called _____
- A node has no children is called _____
- A node has both parent and children is called _____

- If a tree has 5 branches, how many nodes does this tree contain?

- What is the degree of a leaf node? _____
- Can a node has more than one parent? _____
- Is there a unique path from root to every node? _____

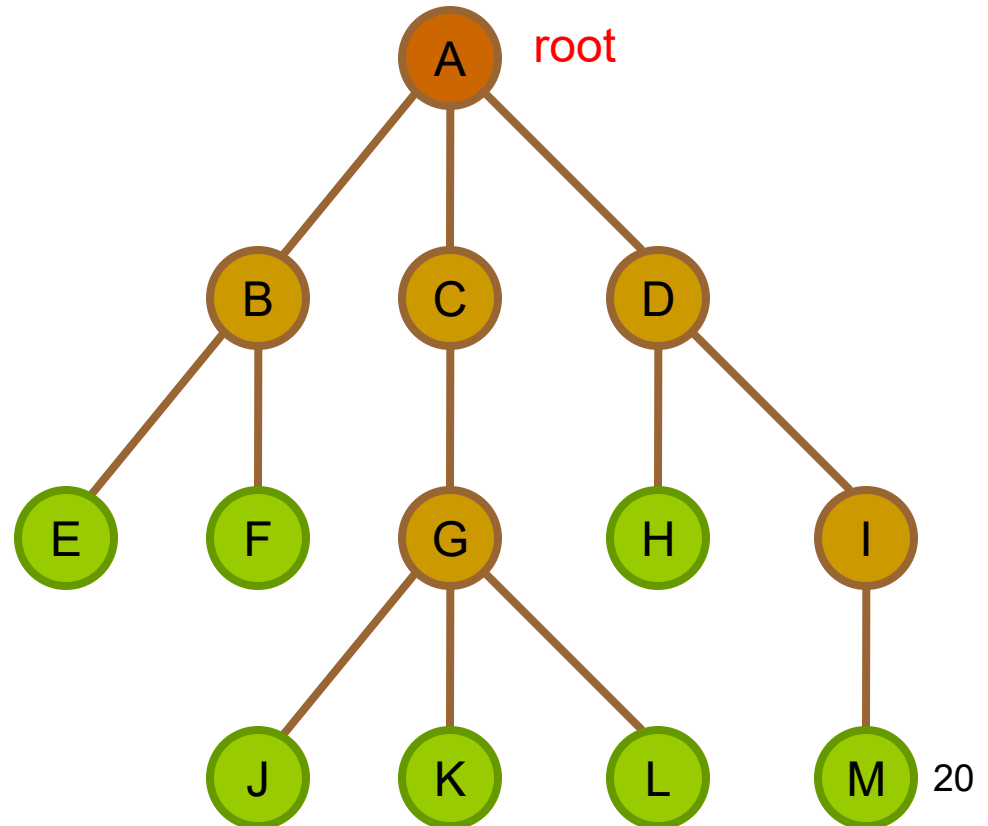
Tree Representation

Representation of Trees

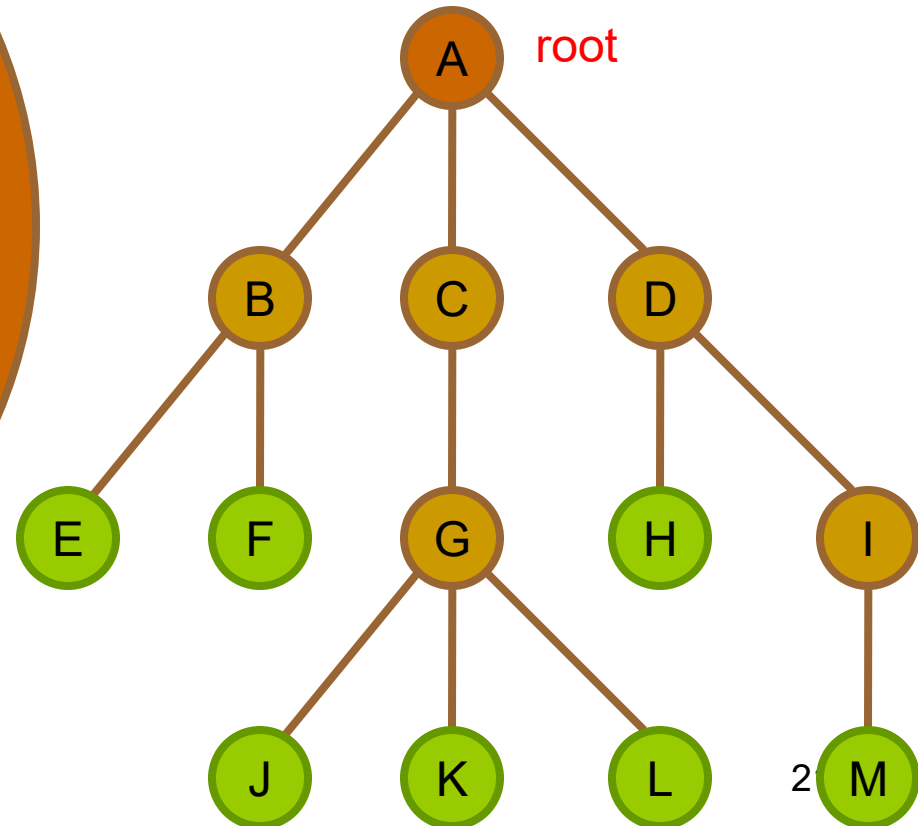
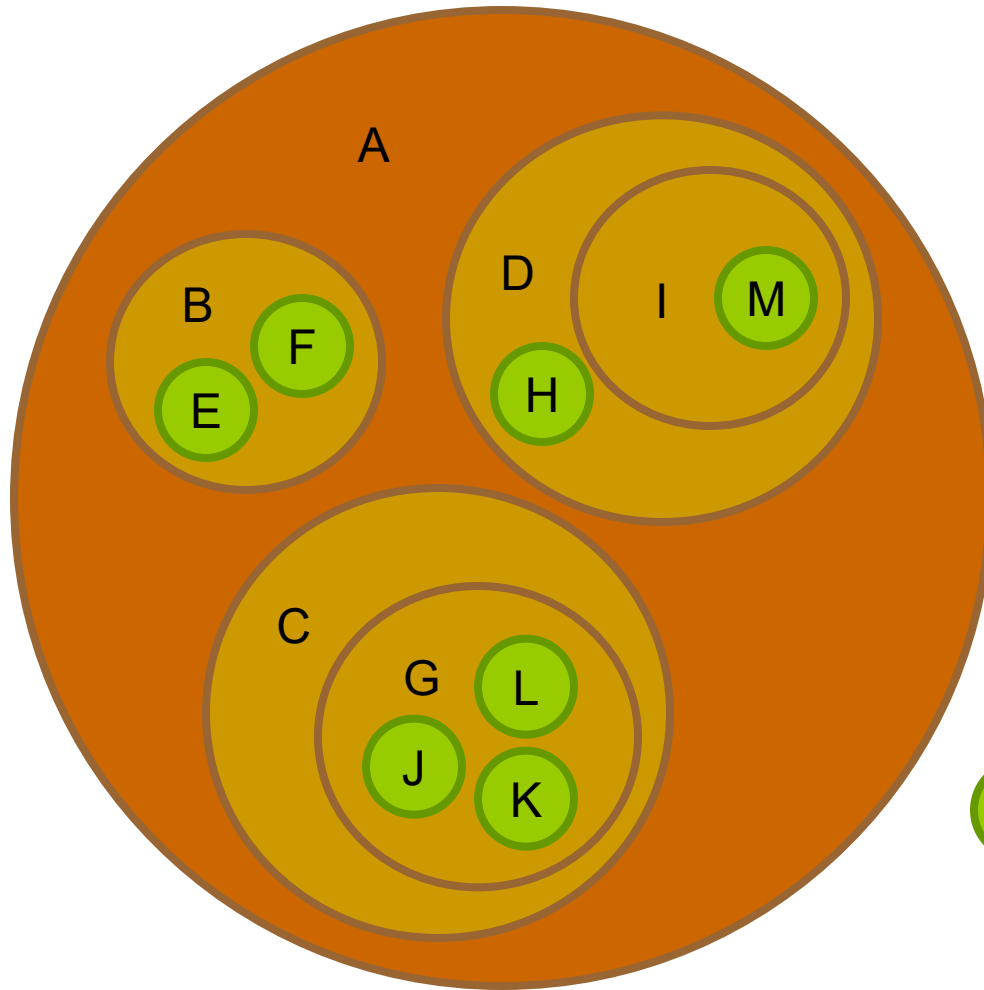
- List representation
- Set representation
- Indentation

List Representation

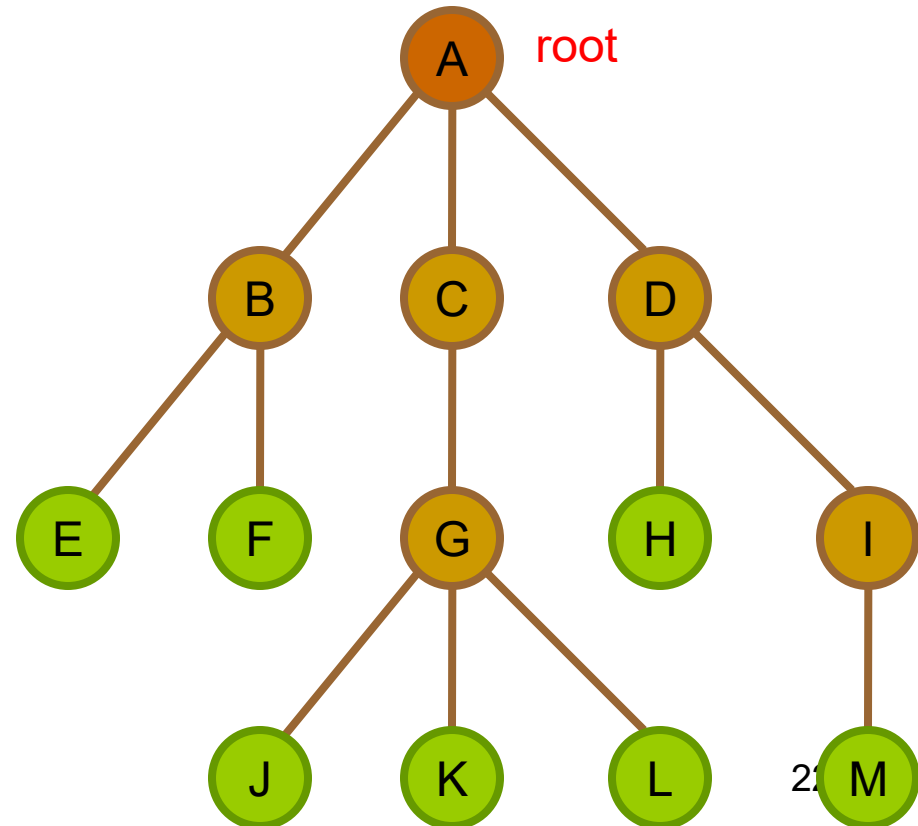
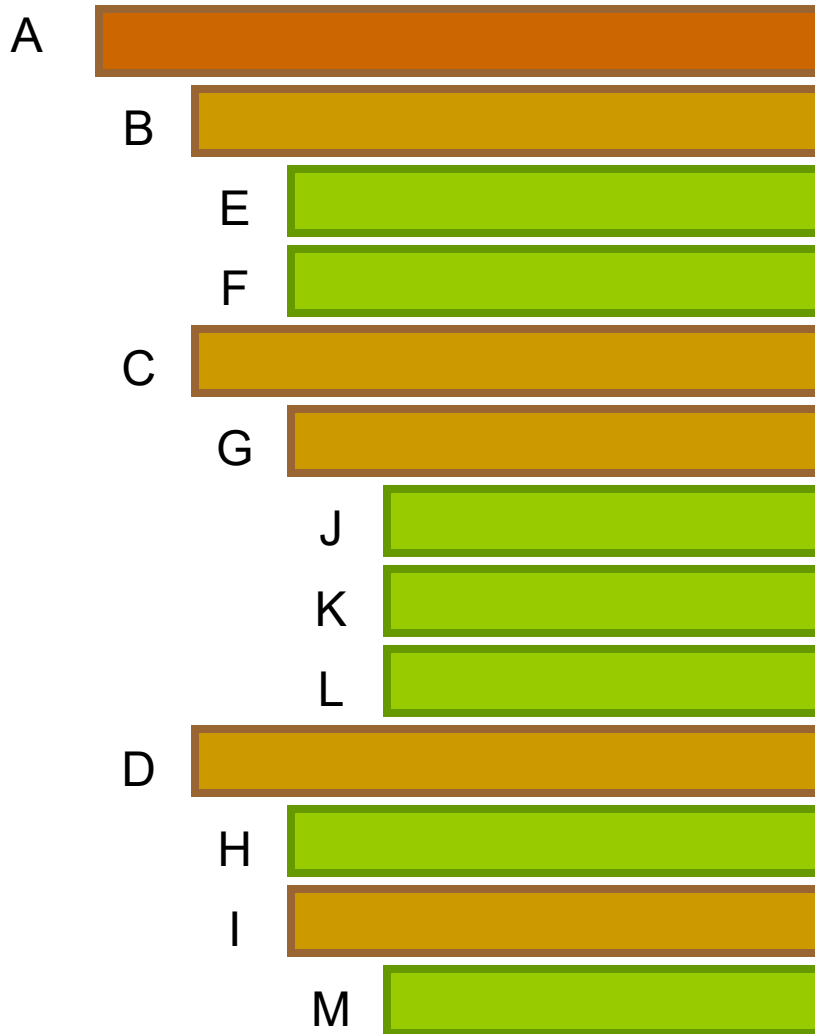
- The tree can be represented by this list
 - (A(B(E, F), C(G(J, K, L), D(H, I(M))))))



Set Representation

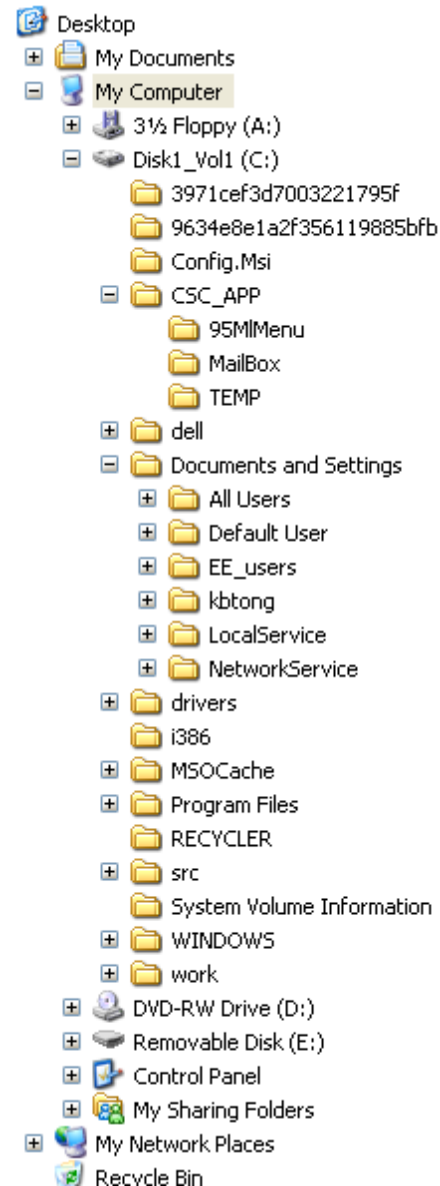


Indentation Representation



They Are Also Indentation

■ A
 ■ B
 ■ E
 ■ F
 ■ C
 ■ G
 ■ J
 ■ K
 ■ L
 ■ D
 ■ H
 ■ I
 ■ M

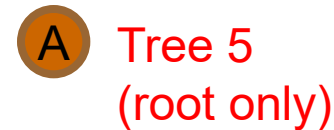
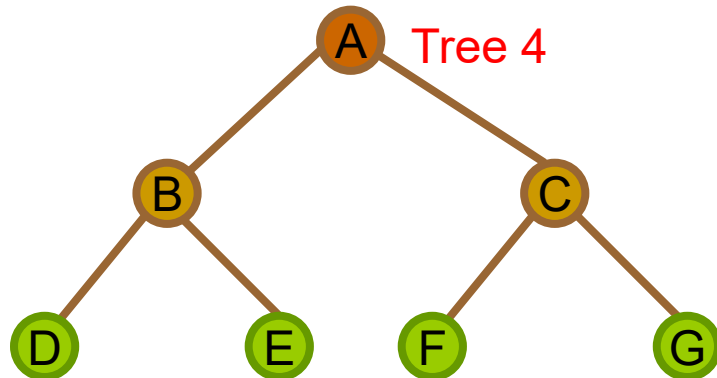
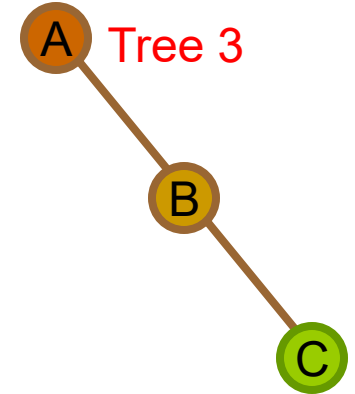
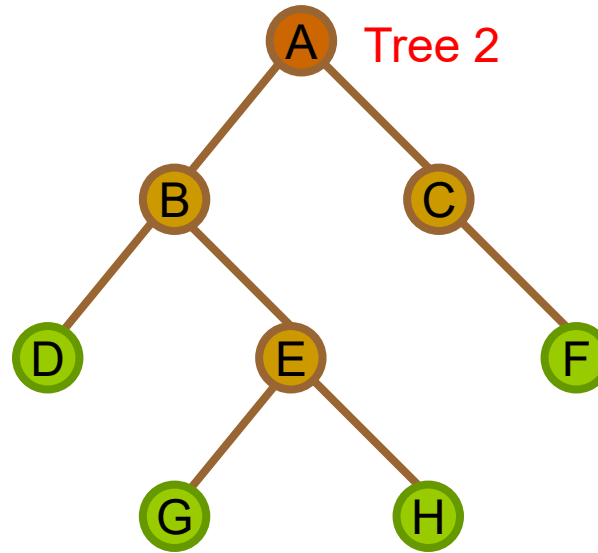
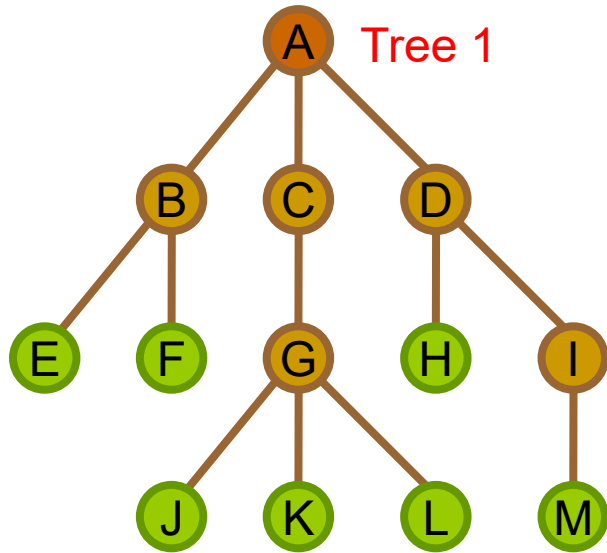


Binary Tree

Binary Trees

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

Are They Binary Tree? Why?

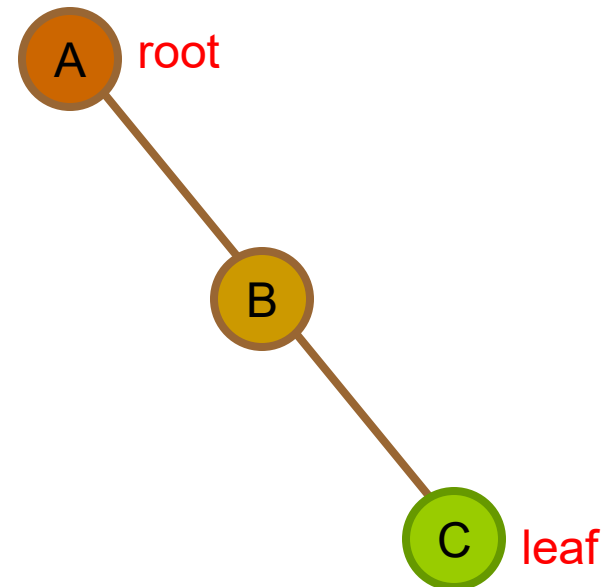
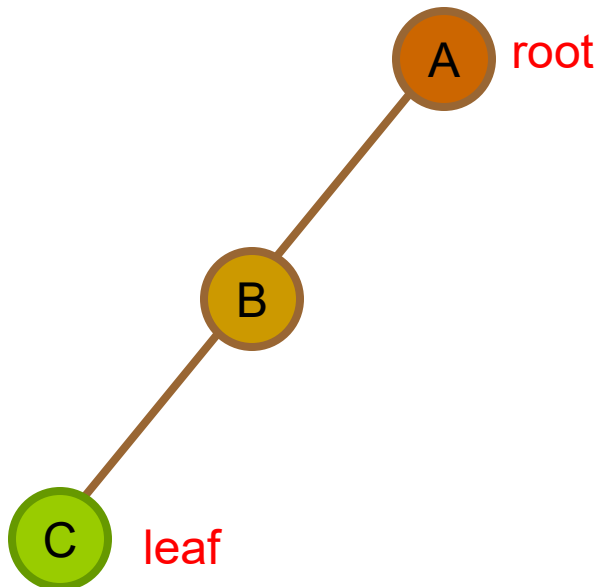


Tree 6 (empty tree)

Types of Binary Tree

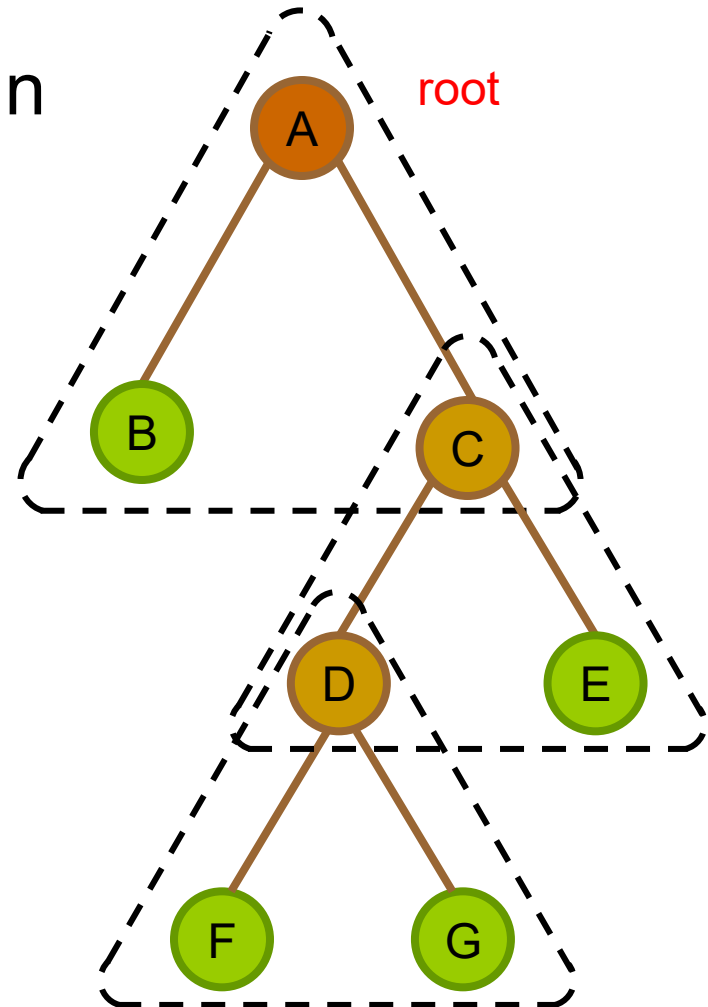
- **Skewed tree**

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



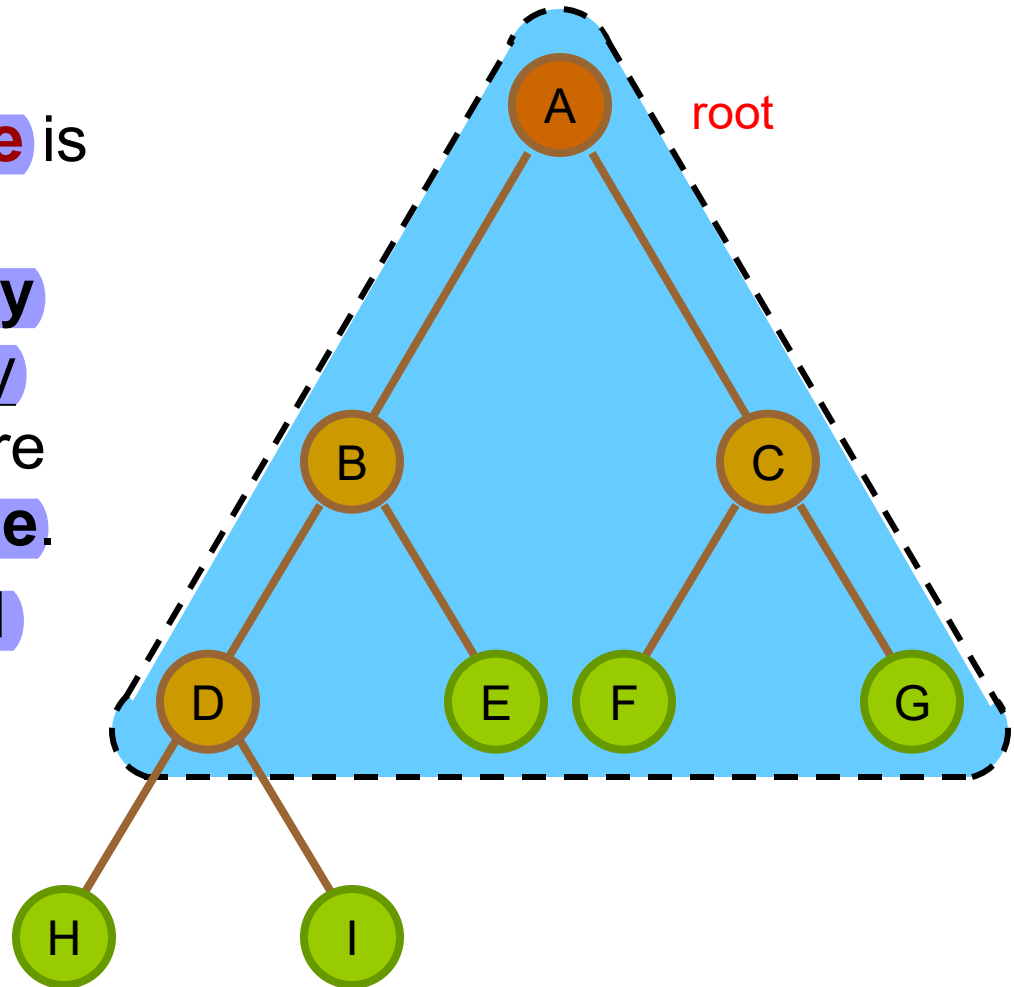
Types of Binary Tree

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



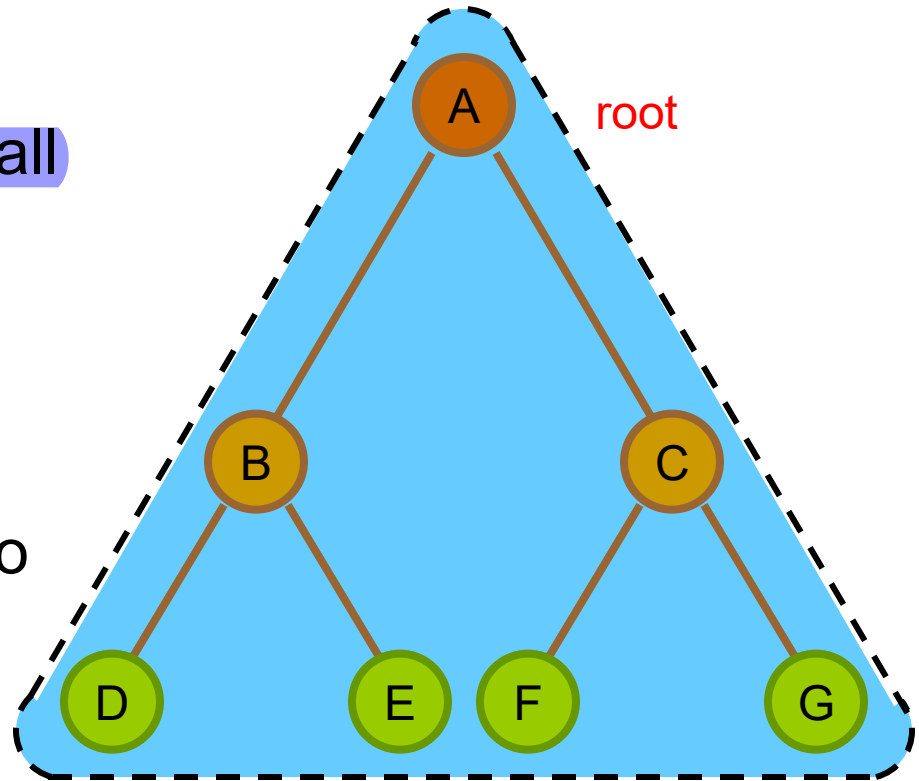
Types of Binary Tree

- **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 **1** and **2^{m-1}** nodes at the last **level m** .



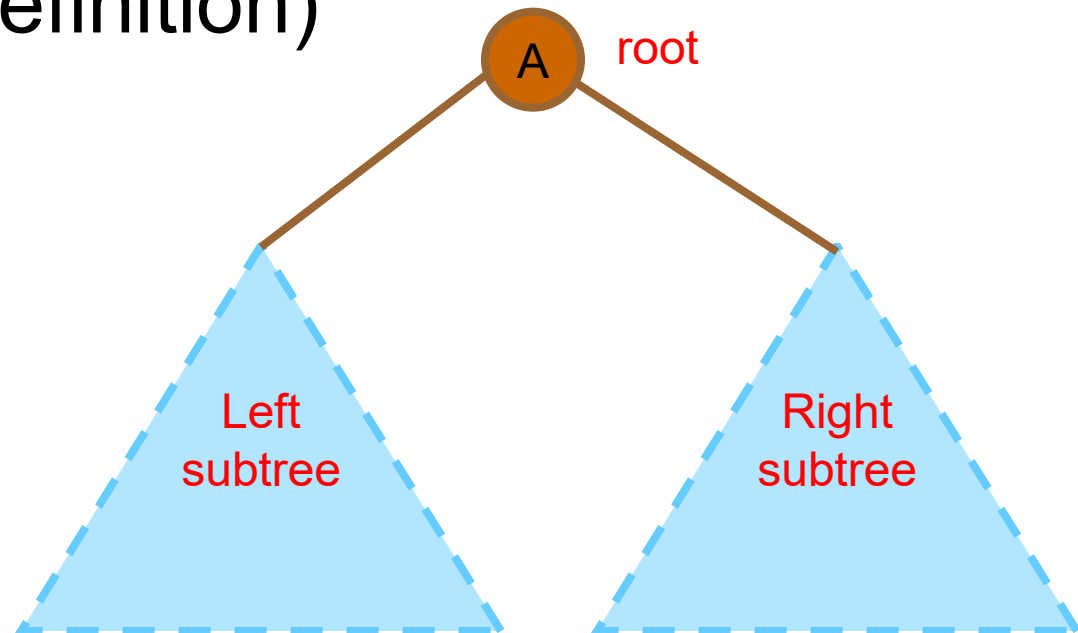
Types of Binary Tree

- **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.



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
- For a non-empty binary tree, if n_0 is the total no. of leaf nodes and n_2 is total no. of degree 2 nodes, then $n_0 = n_2 + 1$

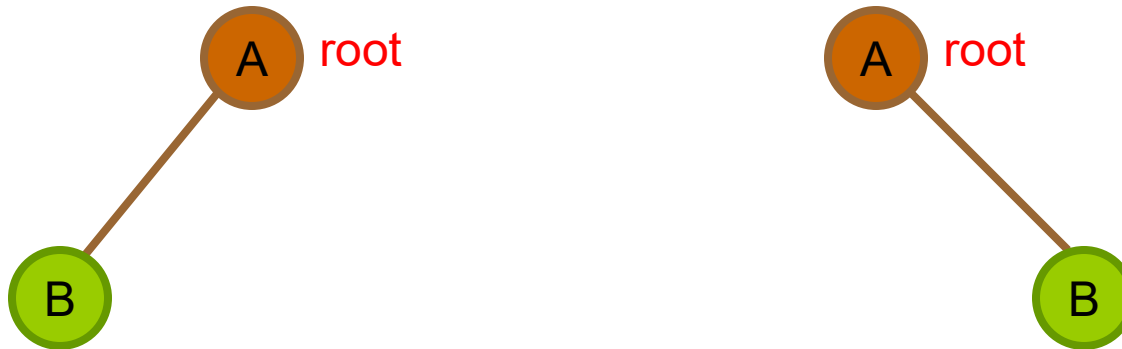
Counting Binary Trees

- How many different combination of a tree can have if it has n nodes?
- For $n = 1$, only one combination



Counting Binary Trees

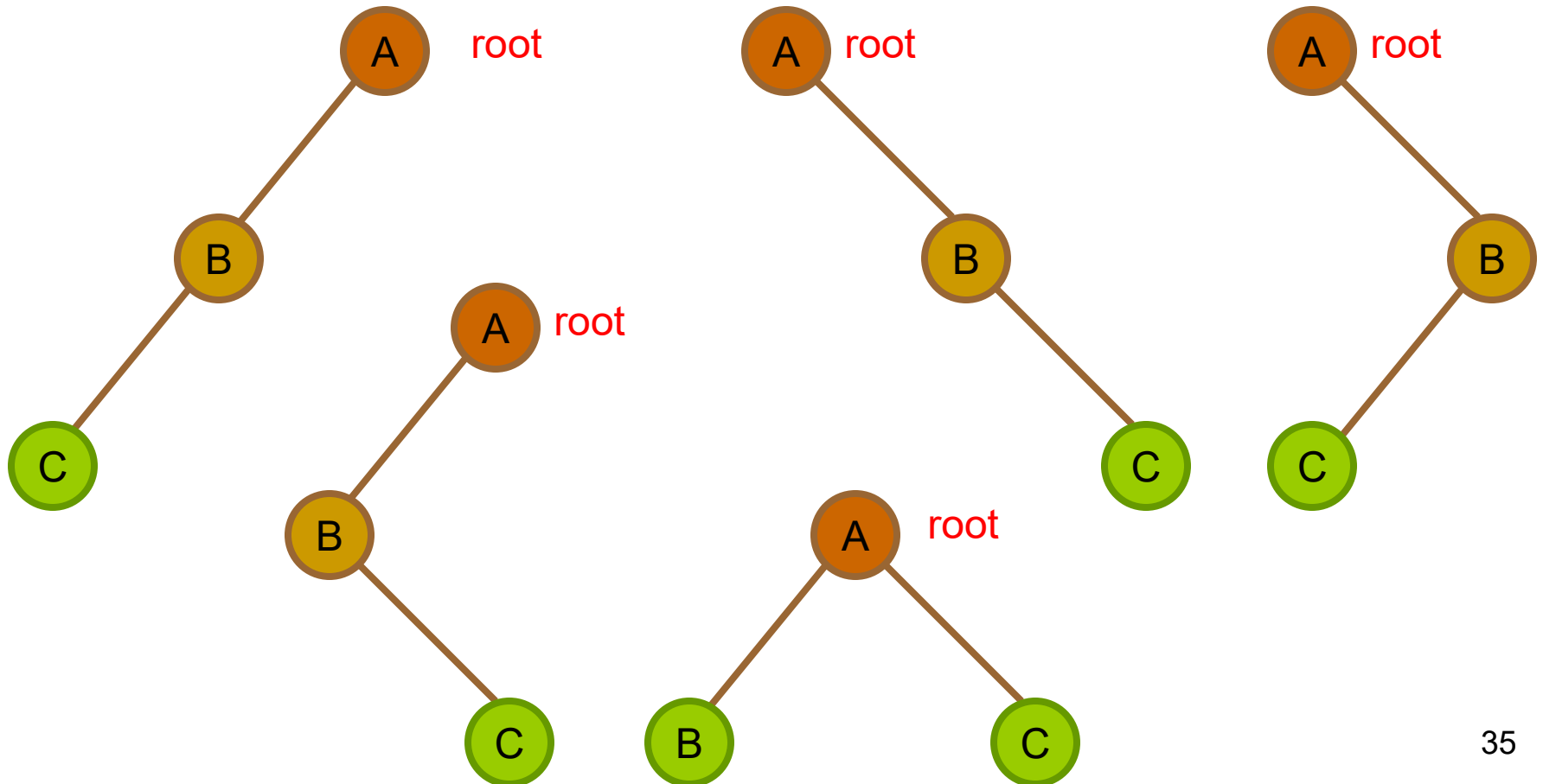
■ For $n = 2$, two combinations



Note: they are different!

Counting Binary Trees

■ For $n = 3$, five combinations



Counting Binary Trees

■ For $n = 4$, 14 combinations

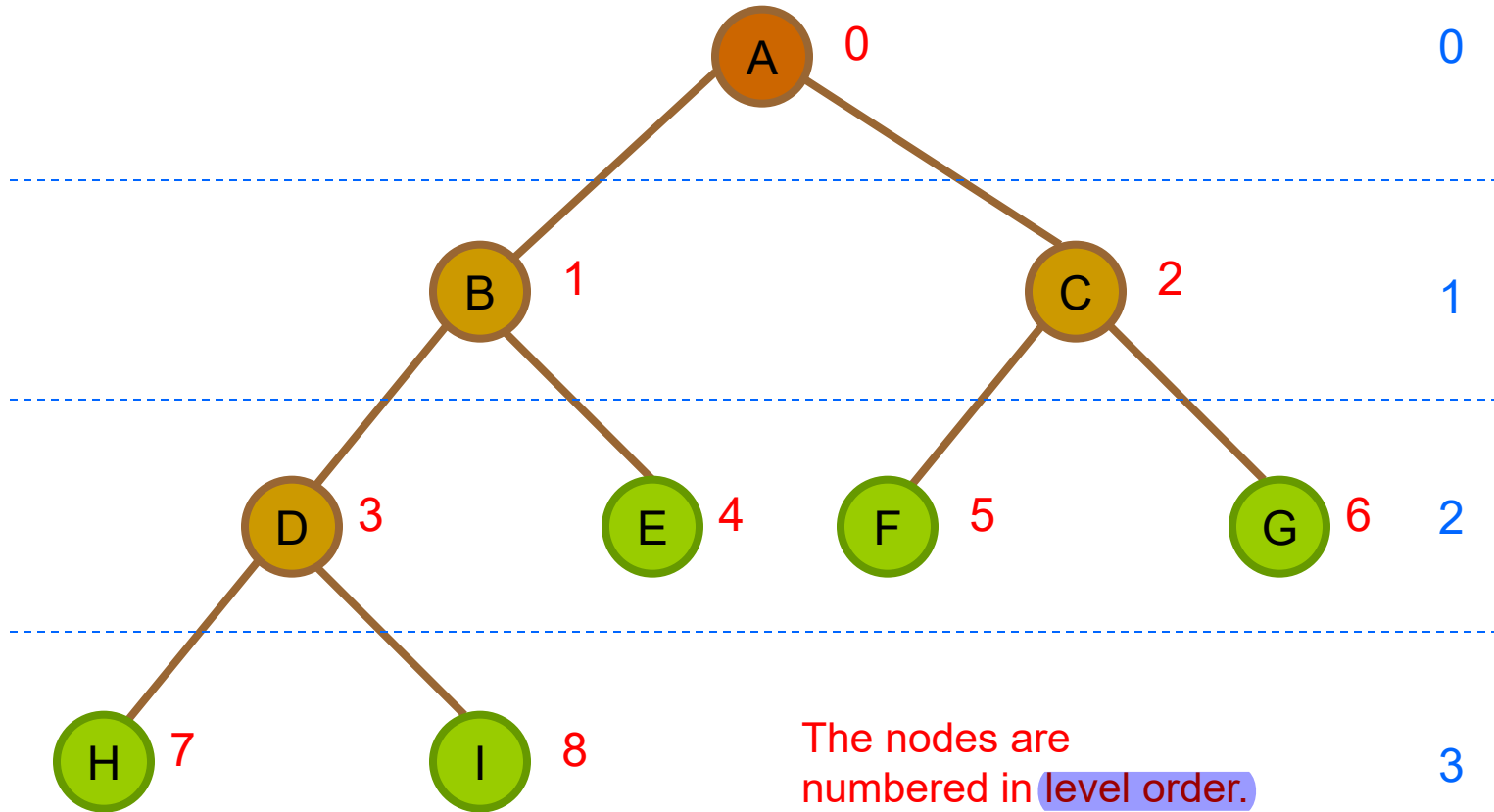
■ Try yourself

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

Array Implementation

Array Implementation

Depth



array:

[0] [1] [2] [3] [4] [5] [6] [7] [8]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|

Depth:

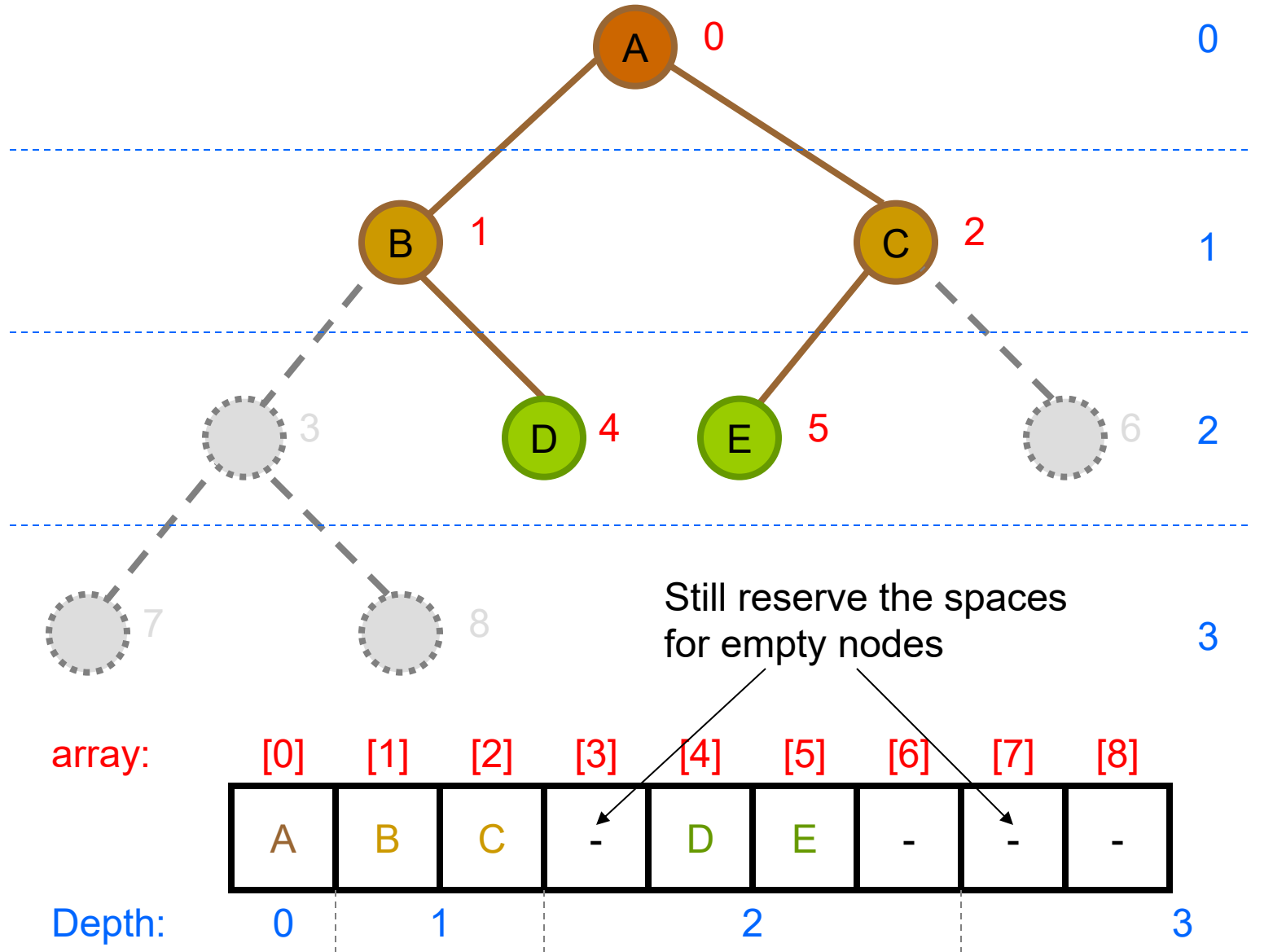
0

1

2

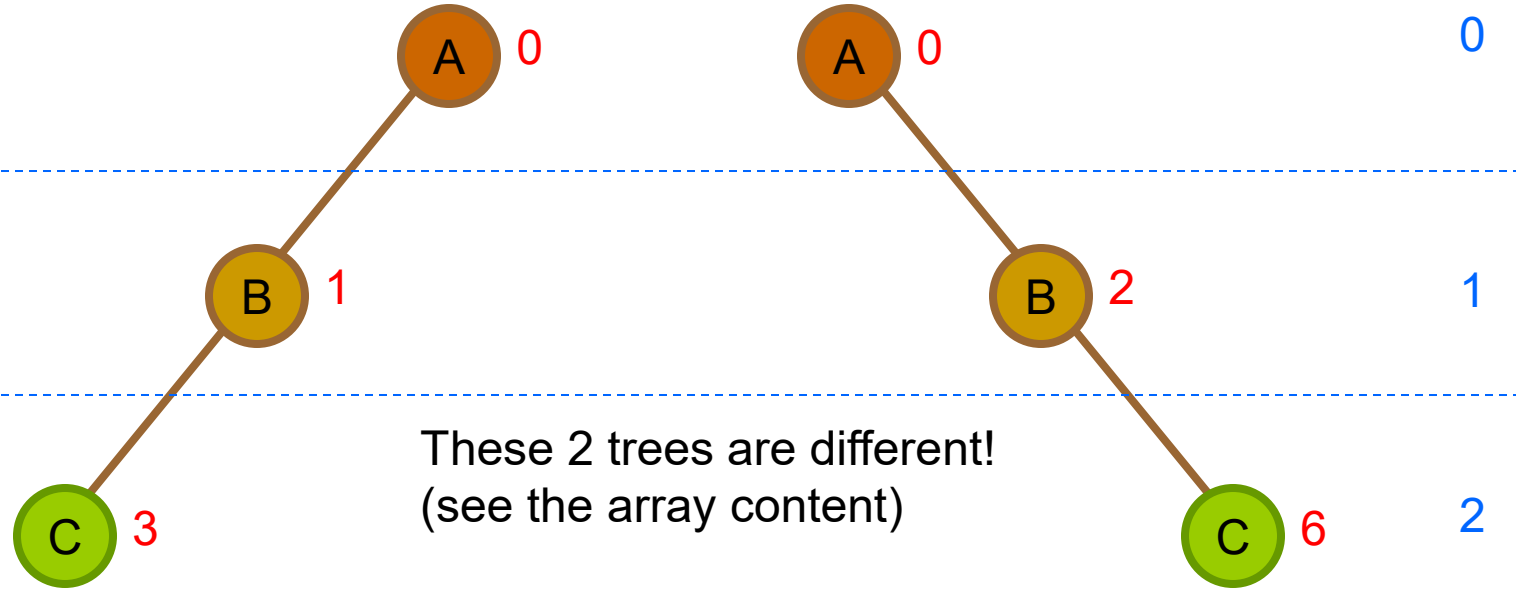
3

Array Implementation



Array Implementation

Depth



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 | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|

Create another
boolean array to
indicate the
unused node

additional
array:

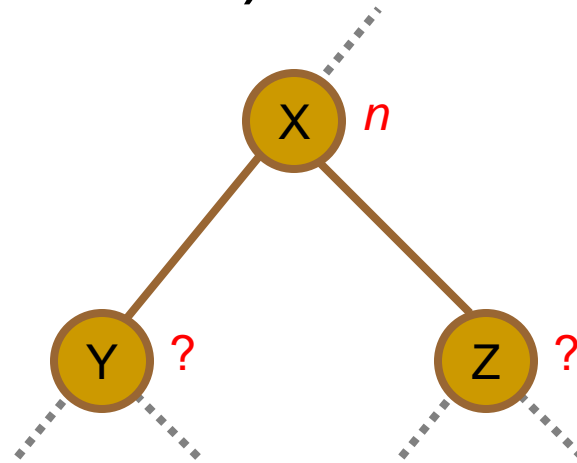
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|

Memory Efficiency

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

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)?

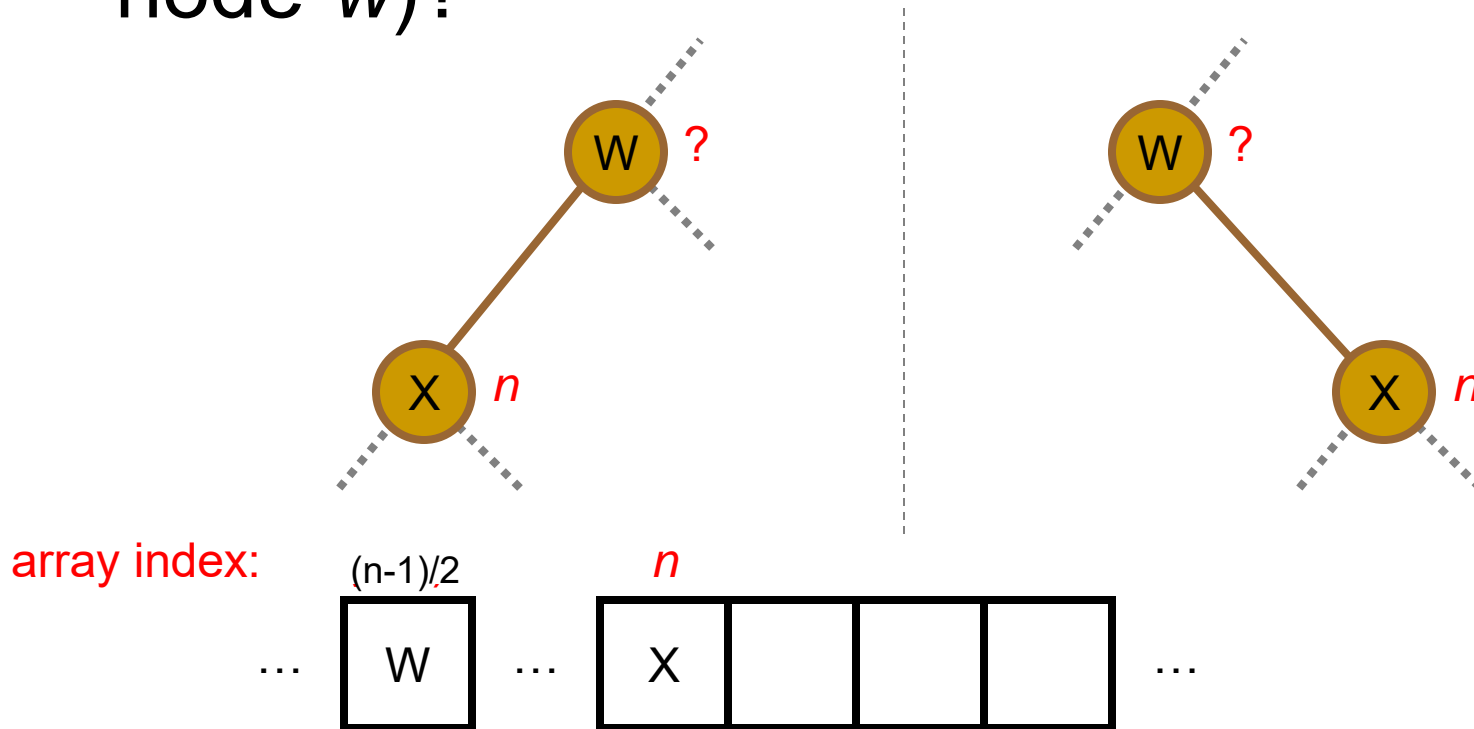


array index: $n-1$ n



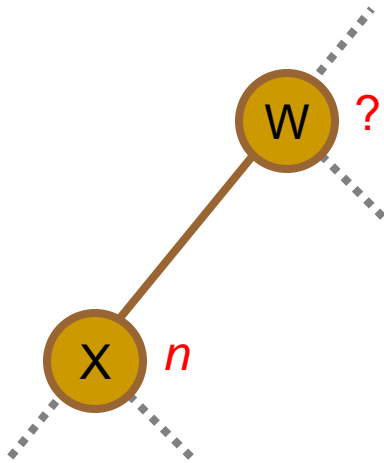
Determine the Index of Parent

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



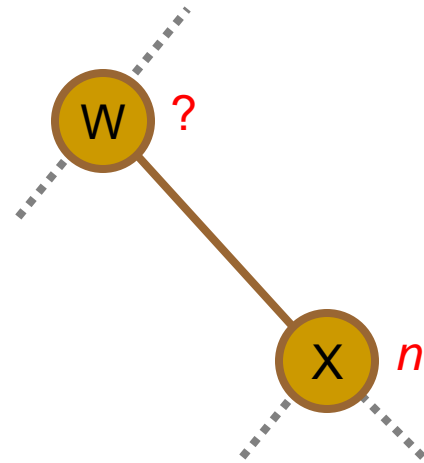
Left or Right Child?

- 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 $(n \% 2 == 1) \{ /* \text{left} */ \}$

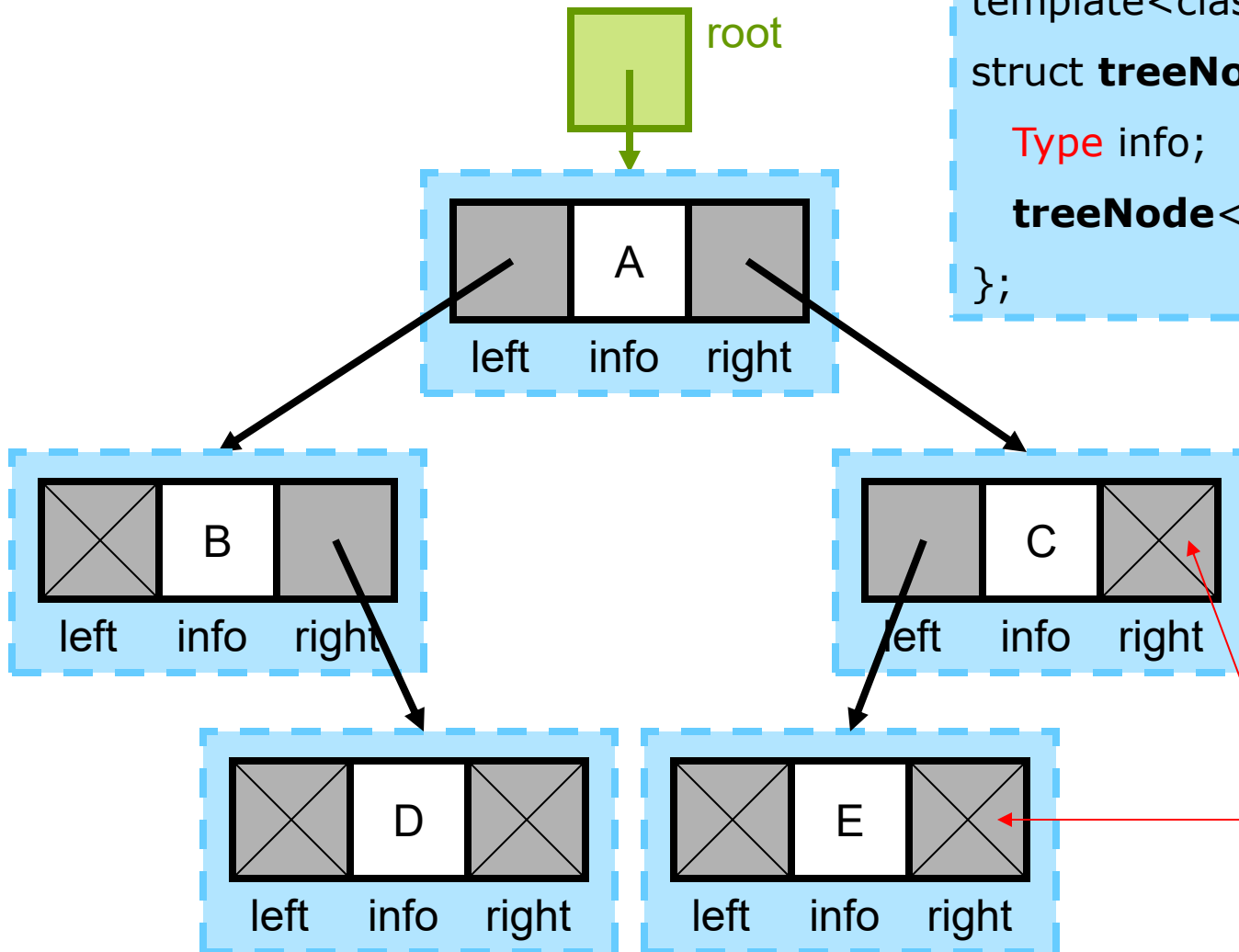


right child if n is even

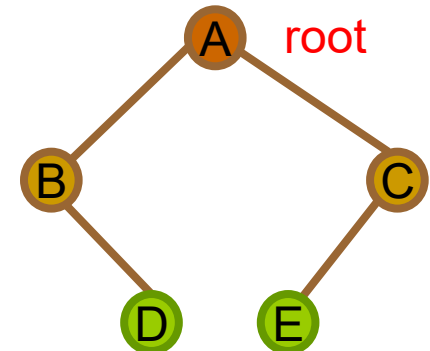
if $(n \% 2 == 0) \{ /* \text{right} */ \}$

Linked List Implementation

Linked List Implementation

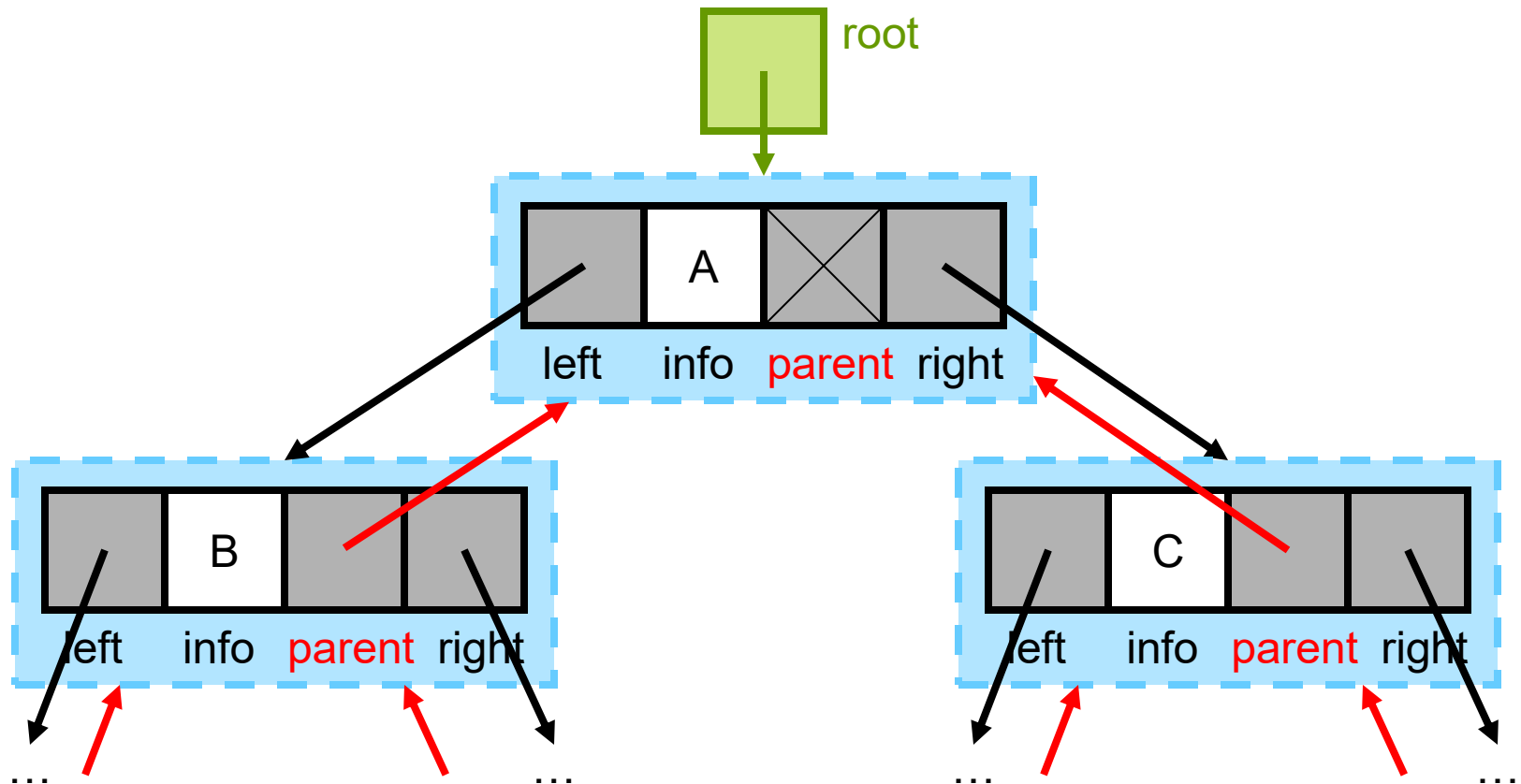


```
template<class Type>
struct treeNode {
    Type info;
    treeNode<Type> *left, *right;
};
```



NULL indicating no subtrees

Possible Variations



Each node has 3 references:

left, right and parent

Common Operations

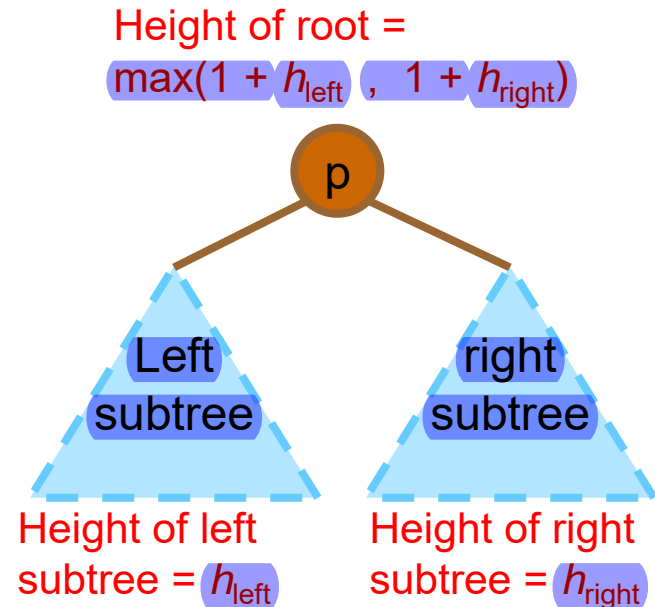
Compute the Height

```
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;
}
```



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>
void copyTree(treeNode<Type>*& copiedTree, treeNode<Type> *other) {
    if (other == NULL)
        copiedTree = NULL;
    else {
        copiedTree = new treeNode<Type>;
        copiedTree->info = other->info;
        copyTree(copiedTree->left, other->left);    // copy left subtree
        copyTree(copiedTree->right, other->right);   // copy right subtree
    }
}
```

Copy Binary Tree (Alternative)

```
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

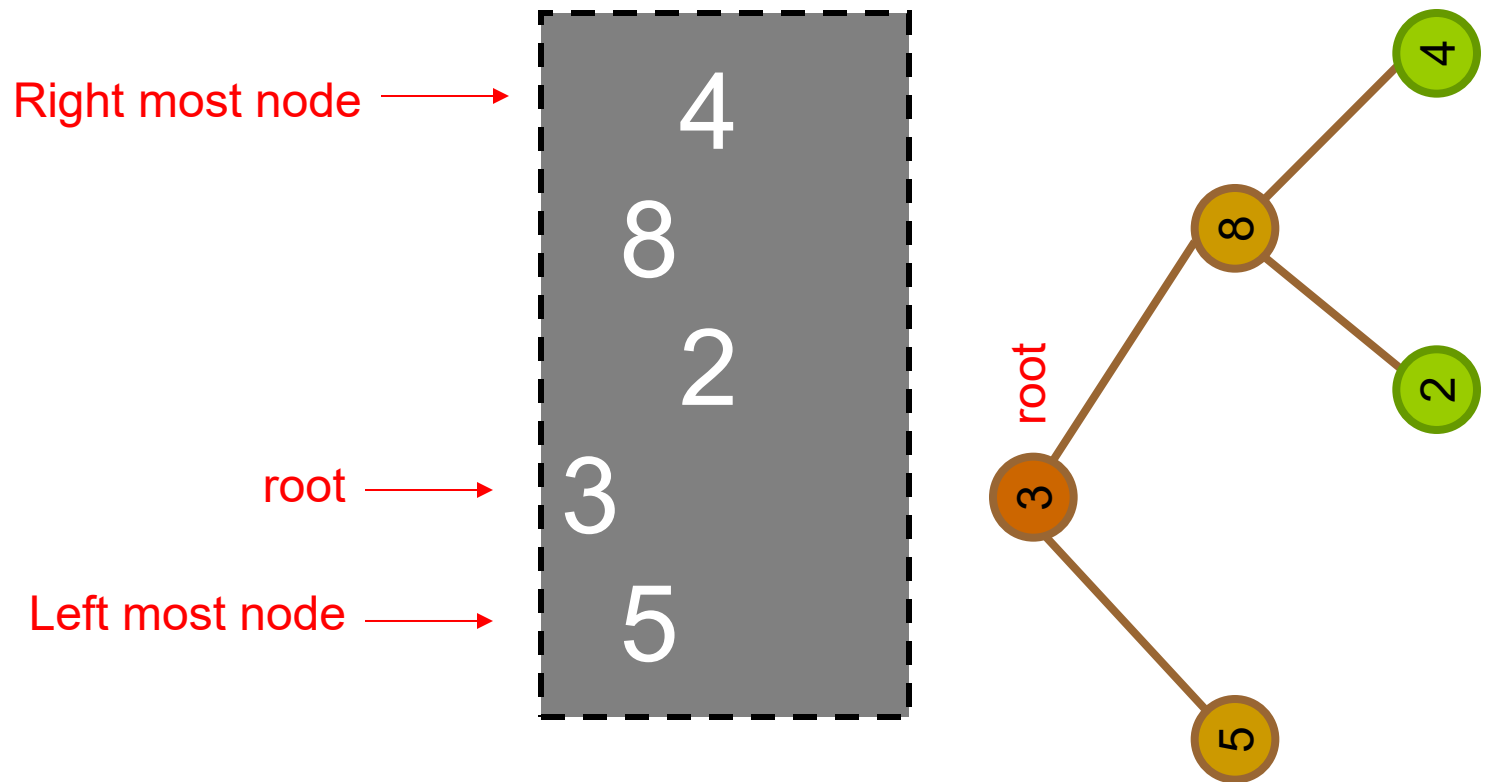
- The two binary trees are identical iff
 - Their root nodes are equal;
 - their left subtrees are equal and;
 - their right subtrees are equal.

```
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

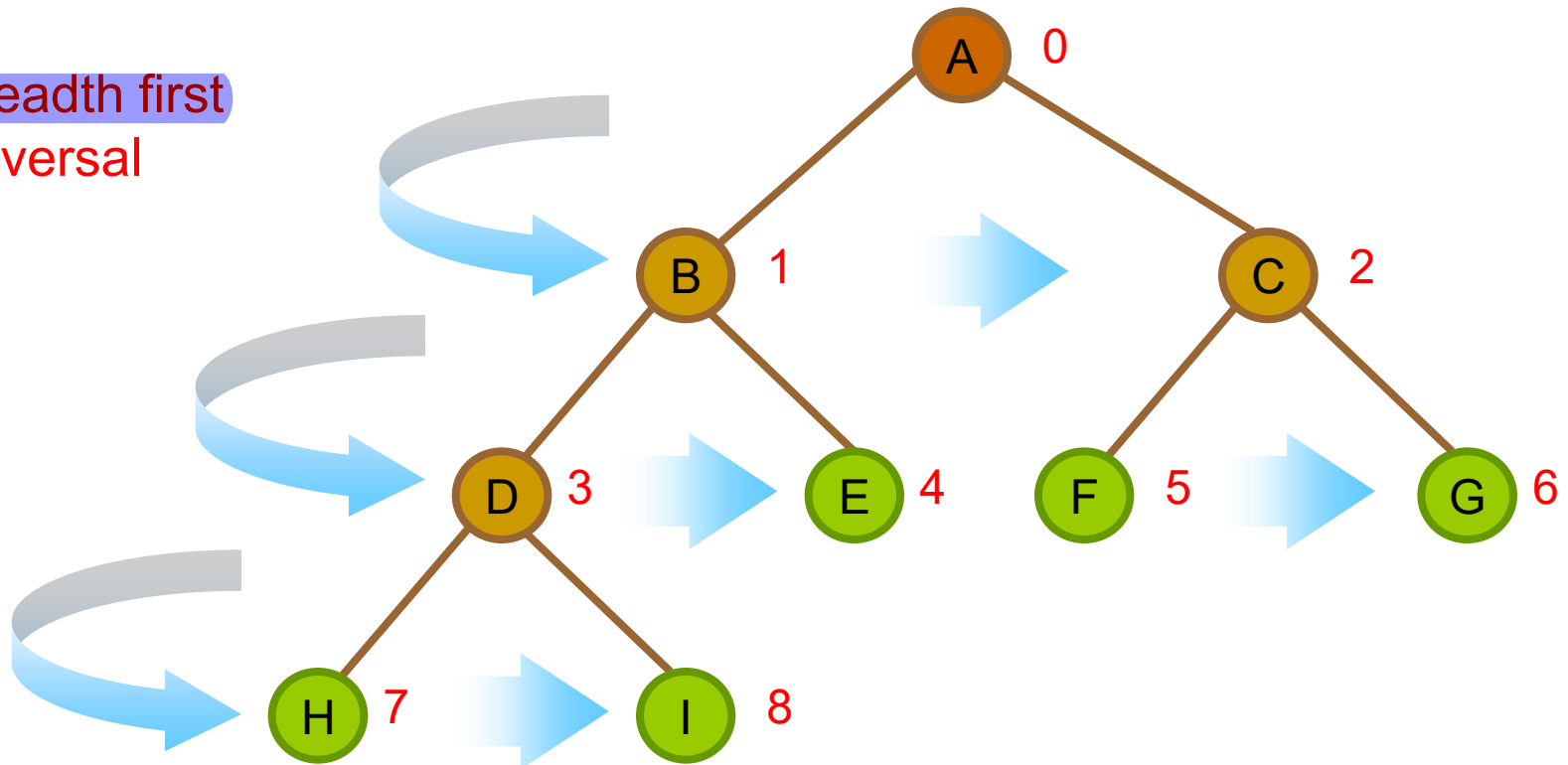
```
#include <iomanip>                //setw(), set width

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



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)

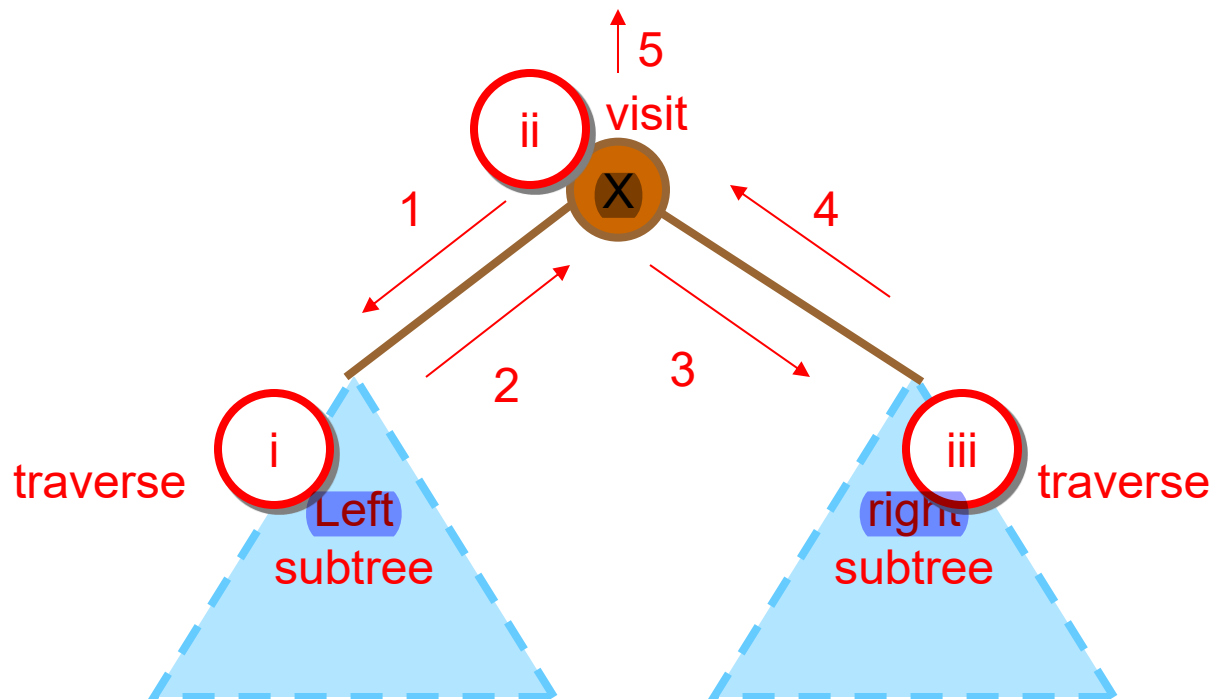
RVL -> not common (in order)

Depth first traversal

- Which kind of traversal does backtracking use?

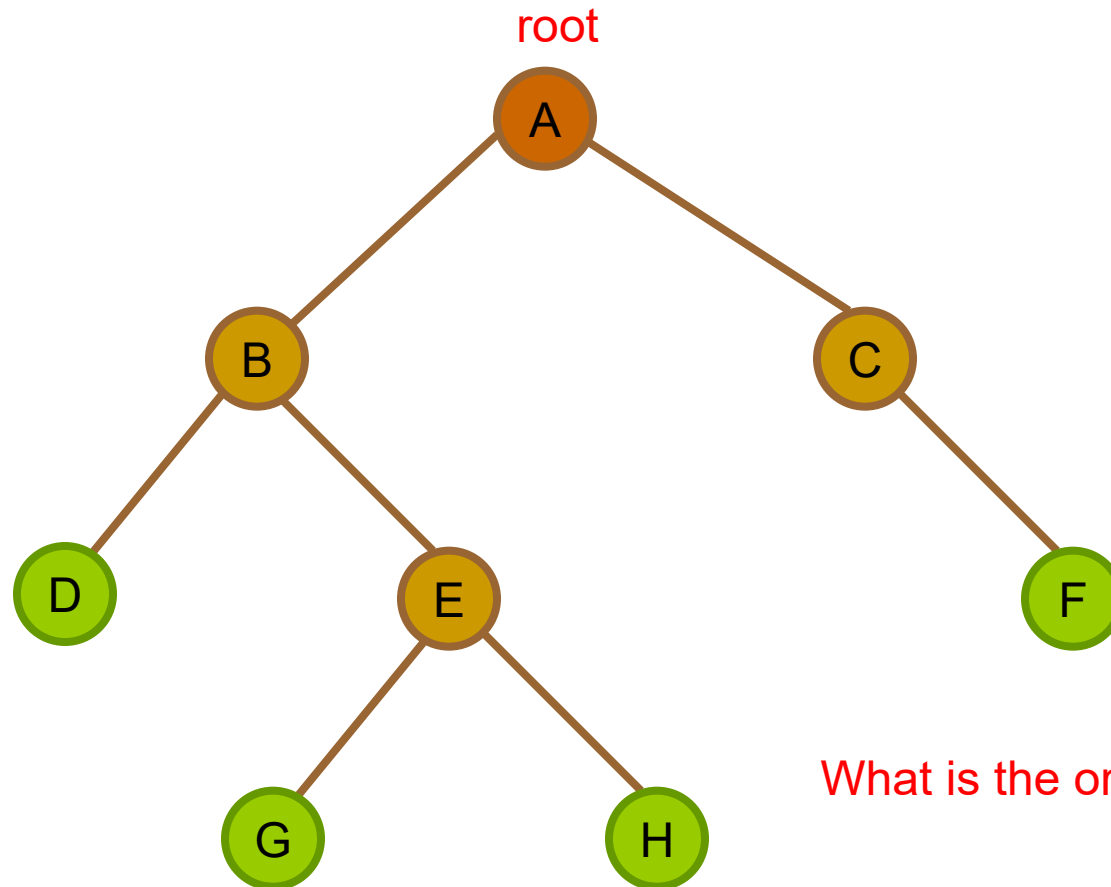
Example: LVR

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



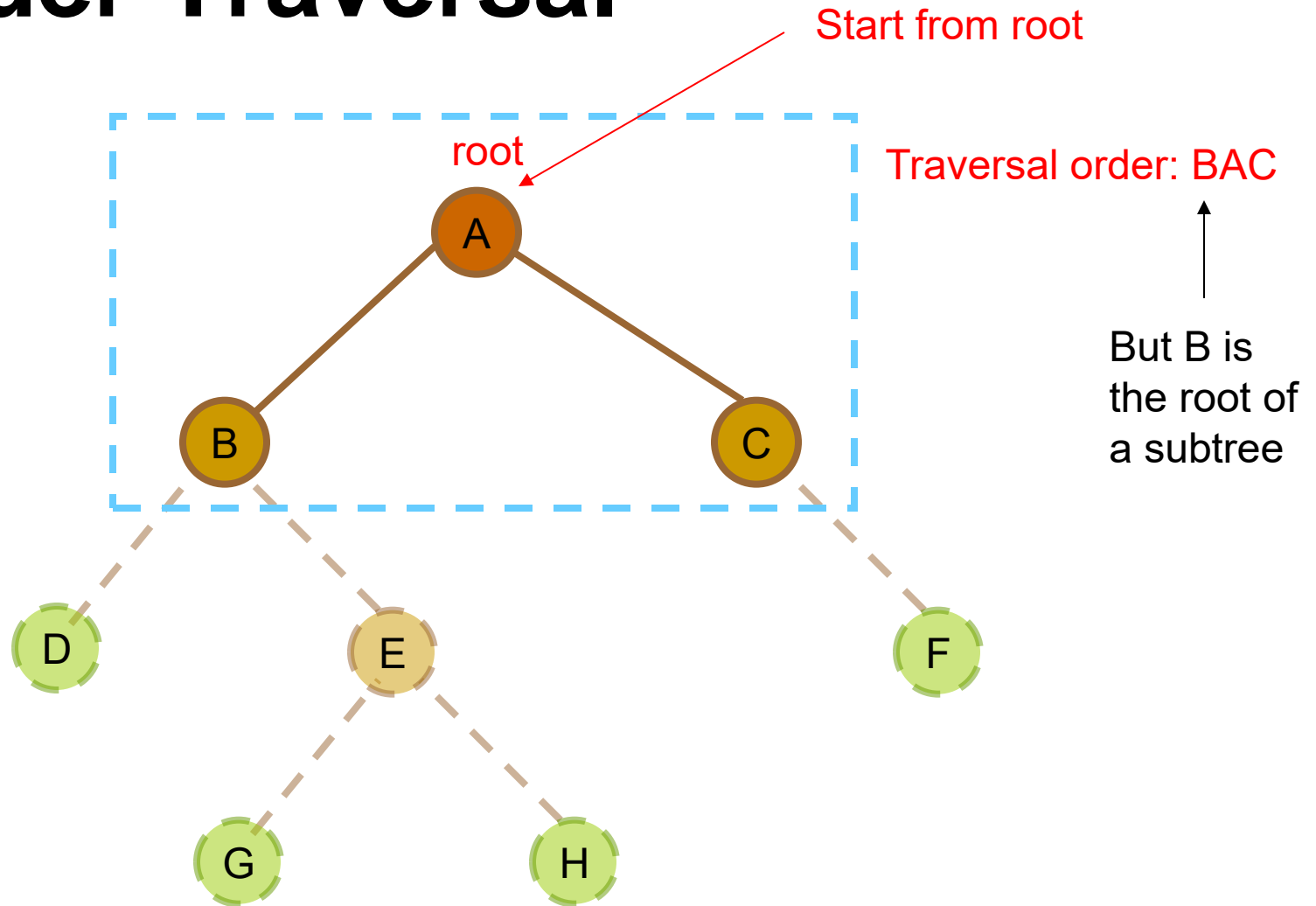
Inorder Traversal

■ LVR

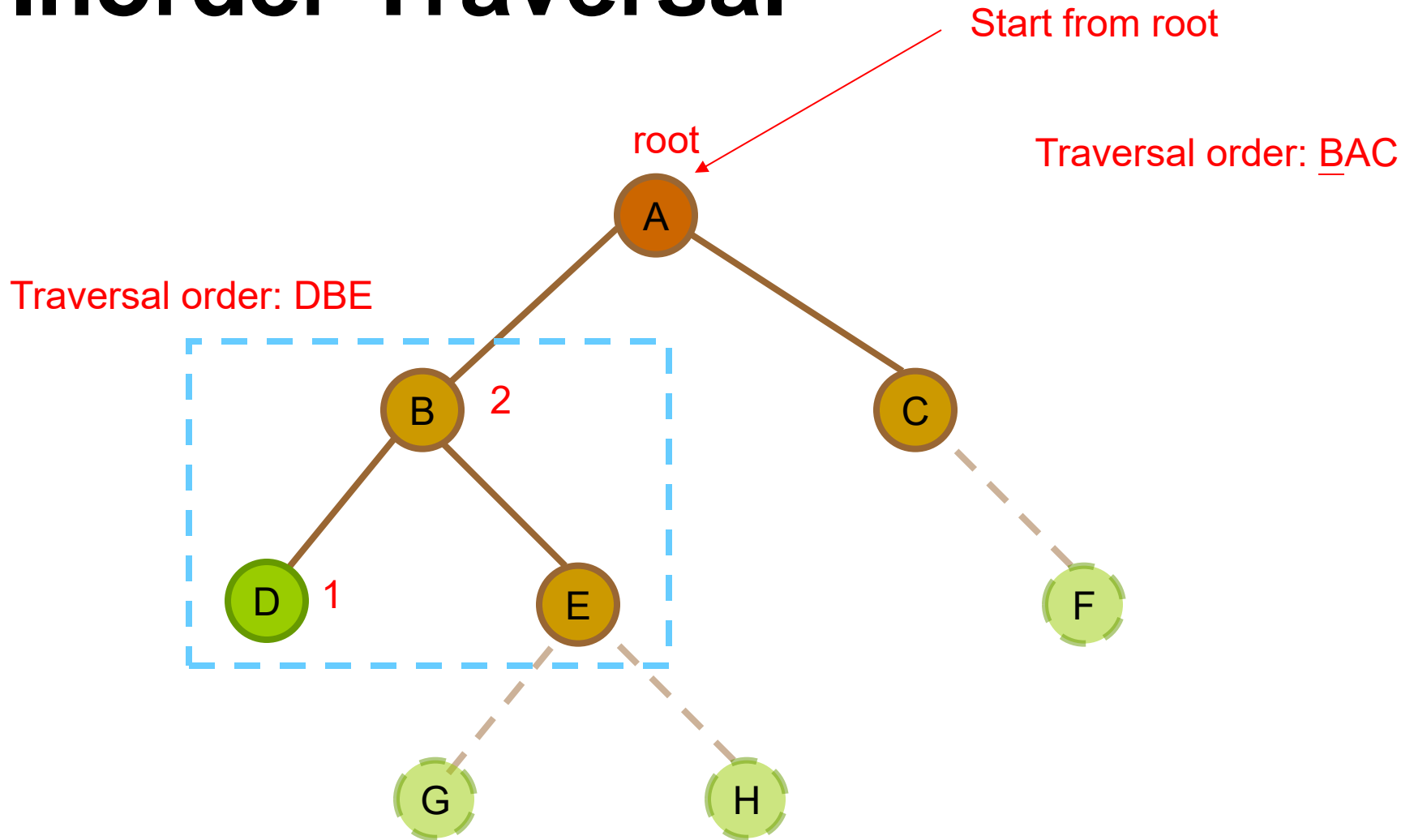


What is the order of traversal?

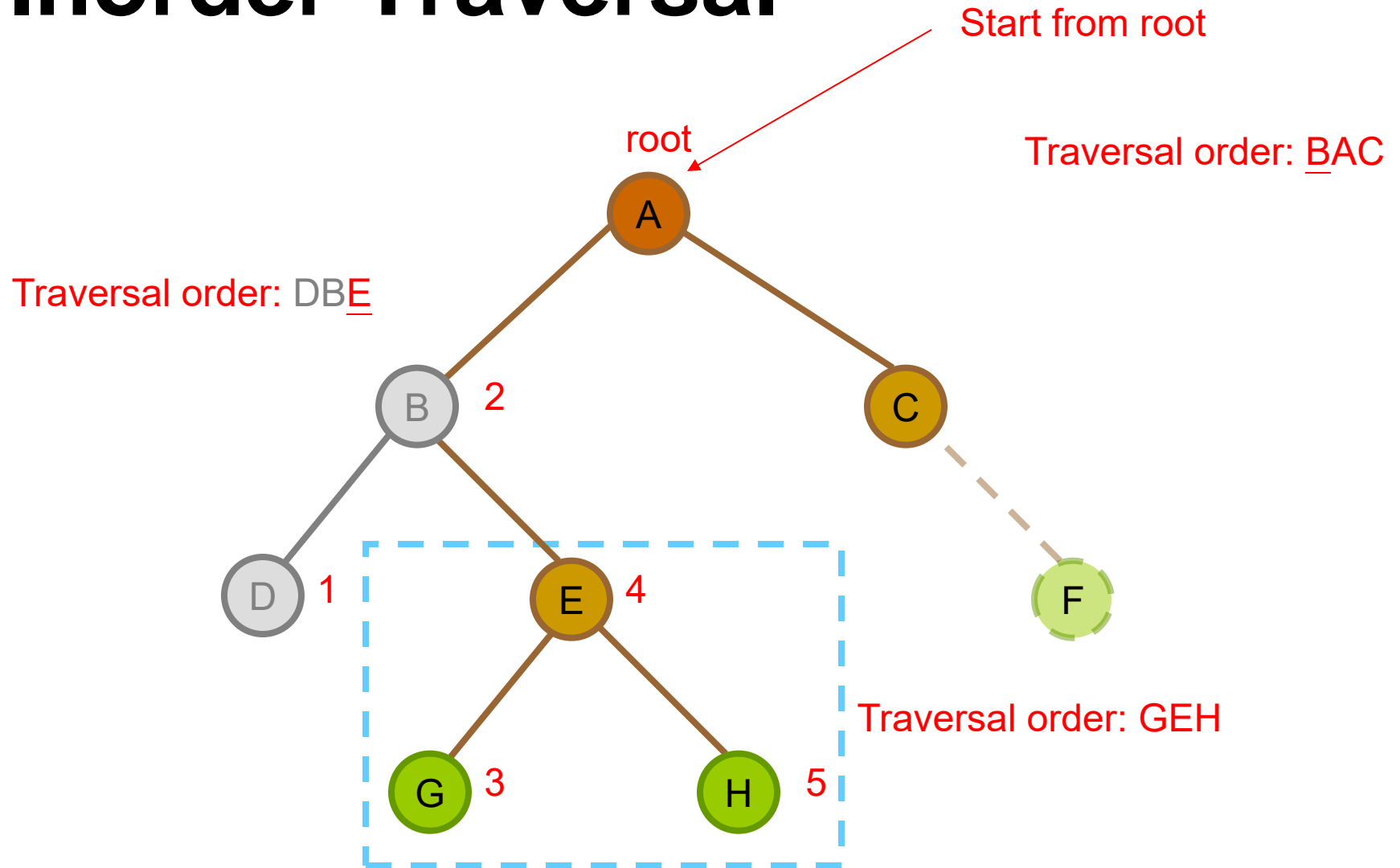
Inorder Traversal



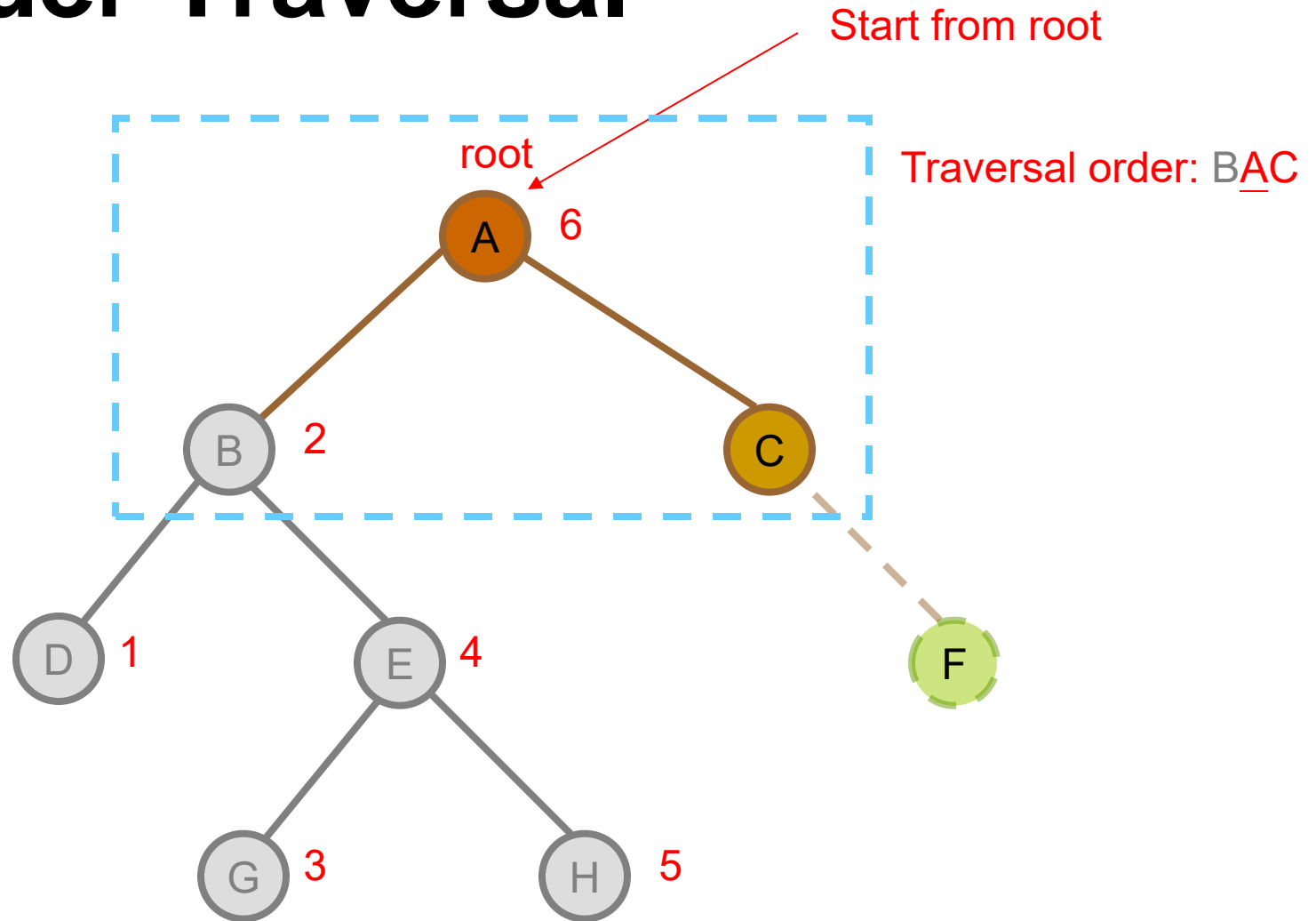
Inorder Traversal



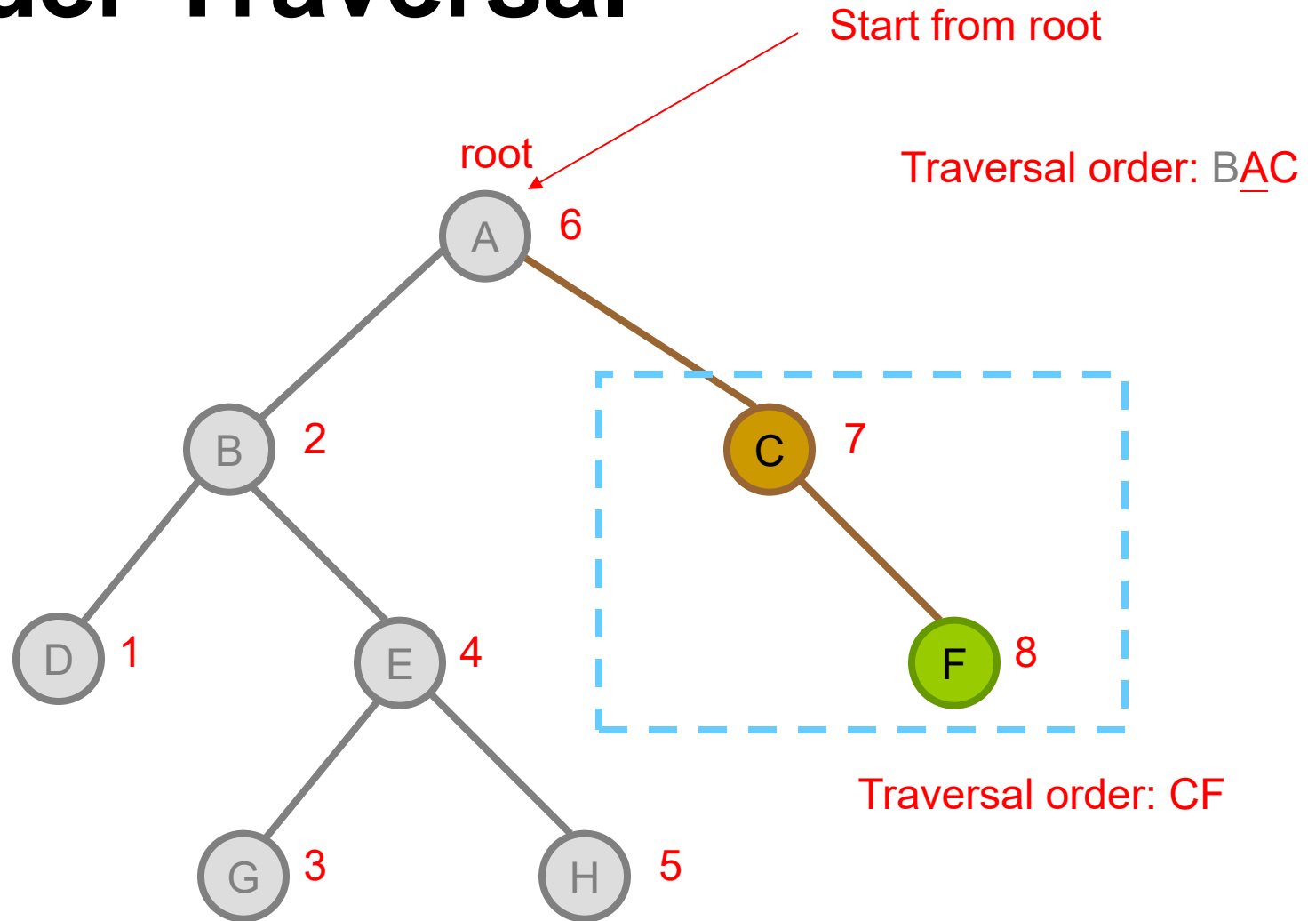
Inorder Traversal



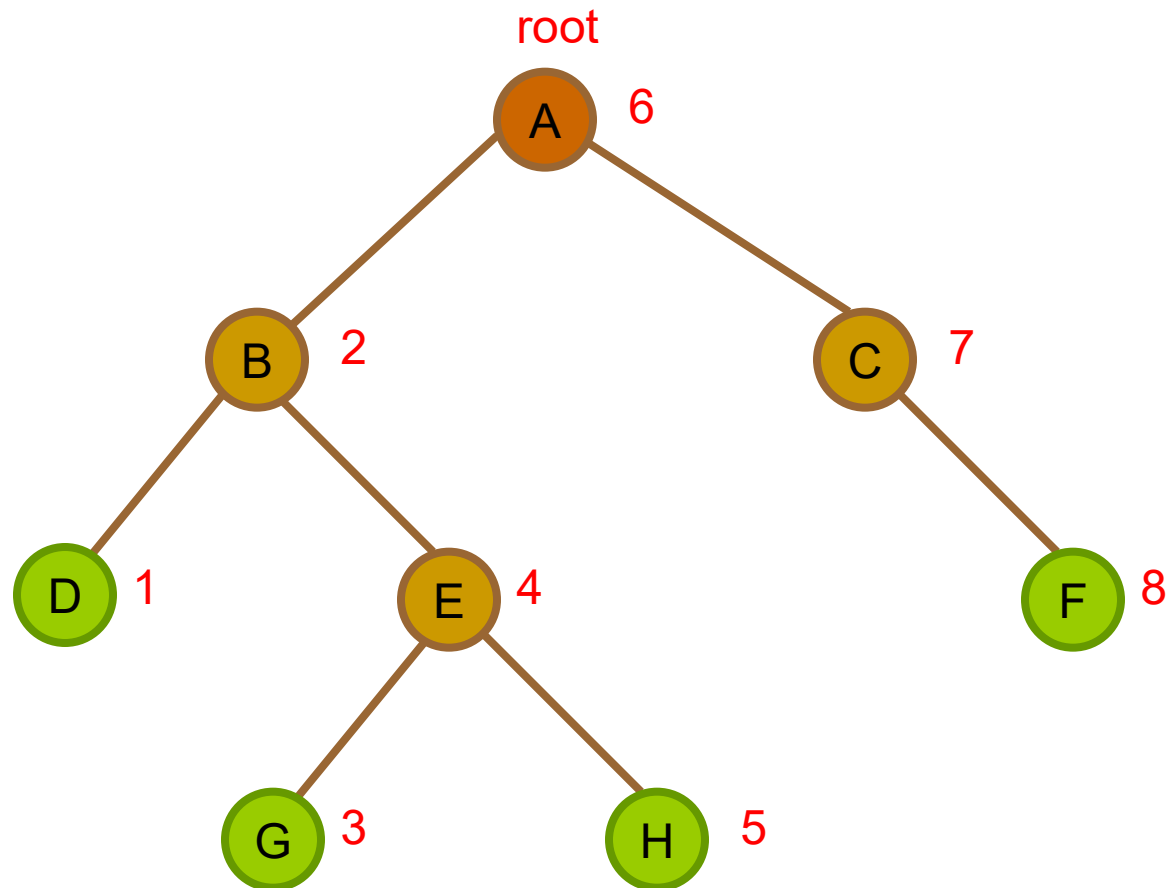
Inorder Traversal



Inorder Traversal



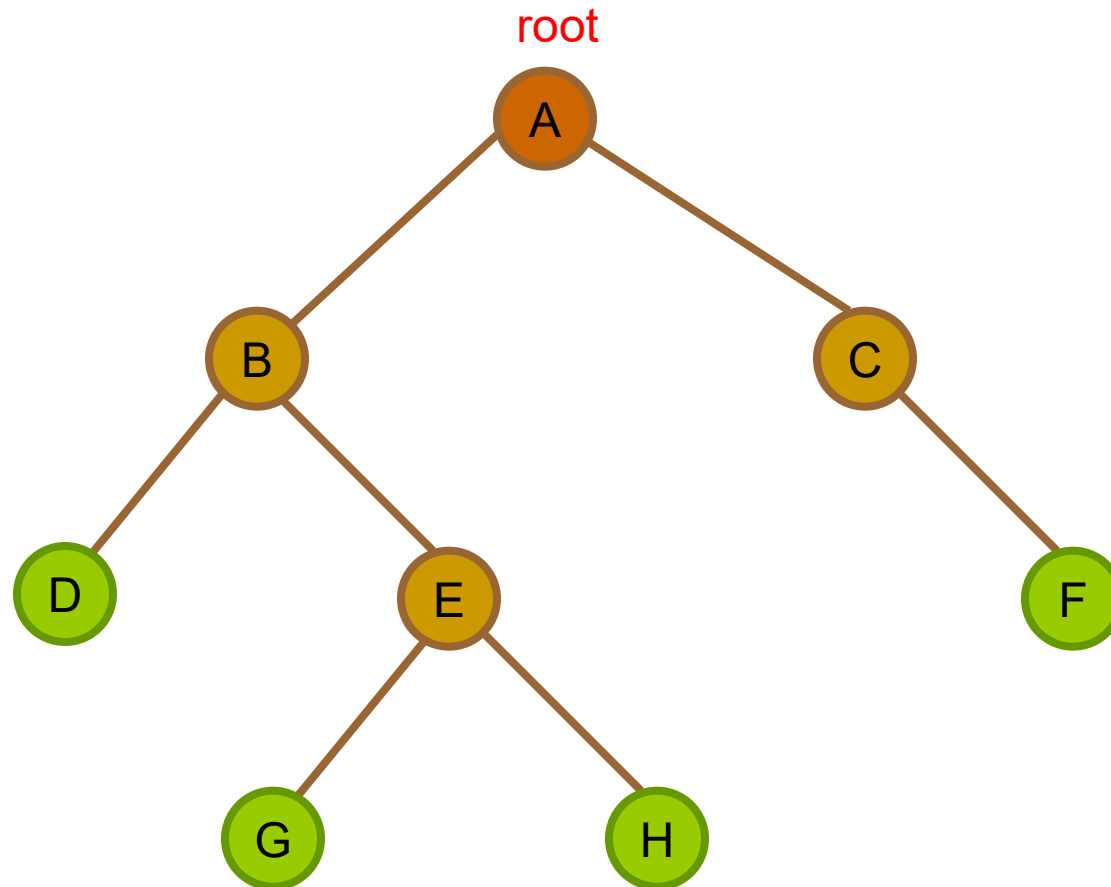
Inorder Traversal



The final sequence: DBGEHACF

Postorder Traversal

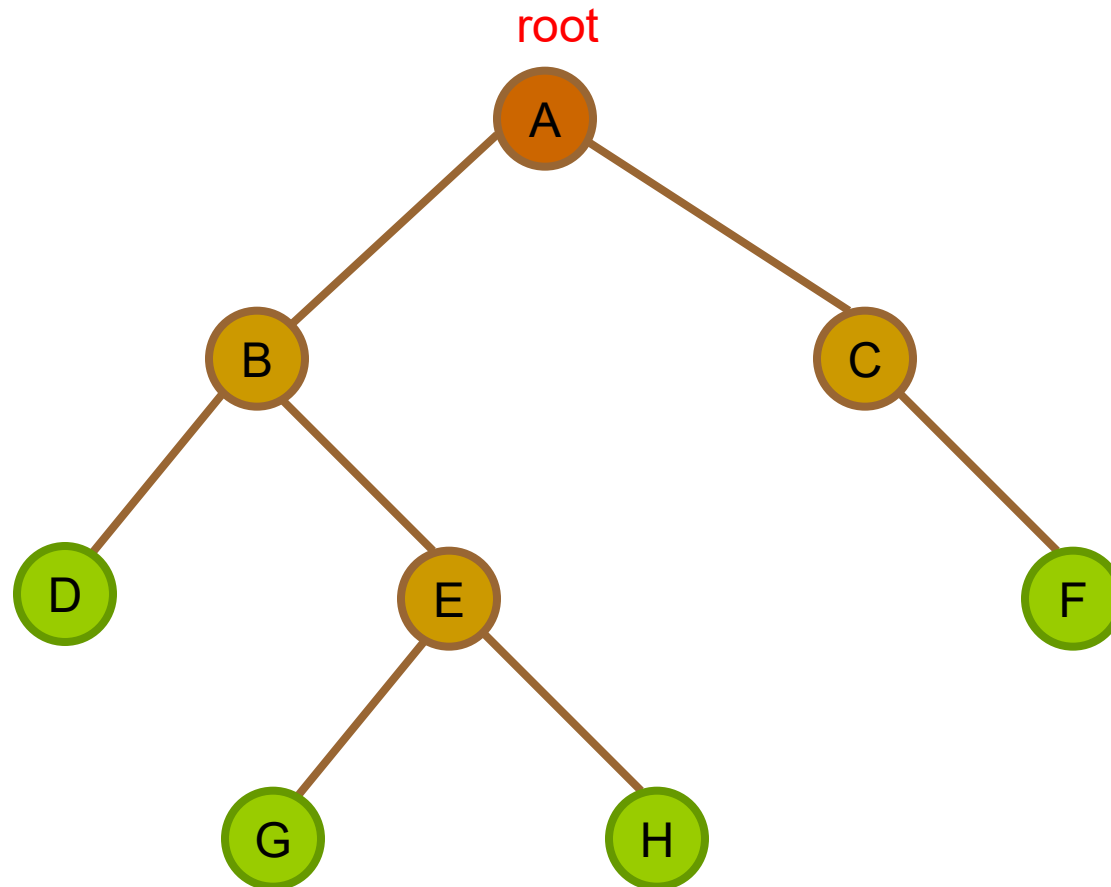
■ LRV



The final sequence: DGHEBFCA

Preorder Traversal

■ VLR




The final sequence: ABDEGHCF

Preorder Traversal

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

//visit the node
// visit left subtree
// visit right subtree



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

Inorder & Postorder Traversal

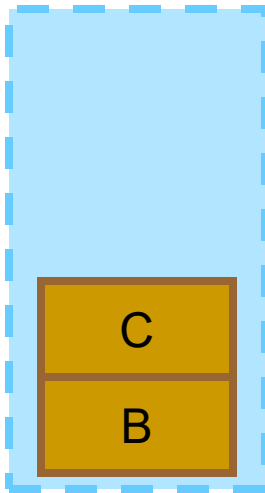
```
template<class Type>
void inorder(treeNode<Type> *p) {
    if (p != NULL) {
        inorder(p->left);
        cout << p->info << " ";           //visit the node
        inorder(p->right);
    }
}
```

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

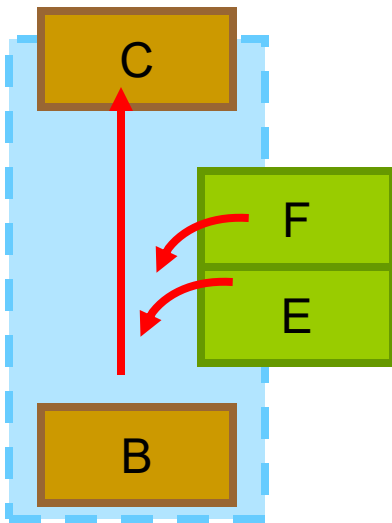
Non-Recursive Tree Operations

- Recursion algorithm intrinsically uses the internal **Call Stack** to buffer tree nodes for further processing
- We may also explicitly use **stack** and **queue** for this purpose and implement tree operations with **iterative approach**
- Use **stack** for **depth first** traversal / search
 - Inorder / preorder / postorder traversal are depth first traversal
 - Traversals are go along the left subtree or right subtree until meeting the leaf nodes
- Use **queue** for **breadth first** traversal / search
 - Breadth first traversal is along the levels
- Now rewrite the function
 - Count number of nodes (i.e. the size of tree)
 - How to do it in a **non-recursive** way?

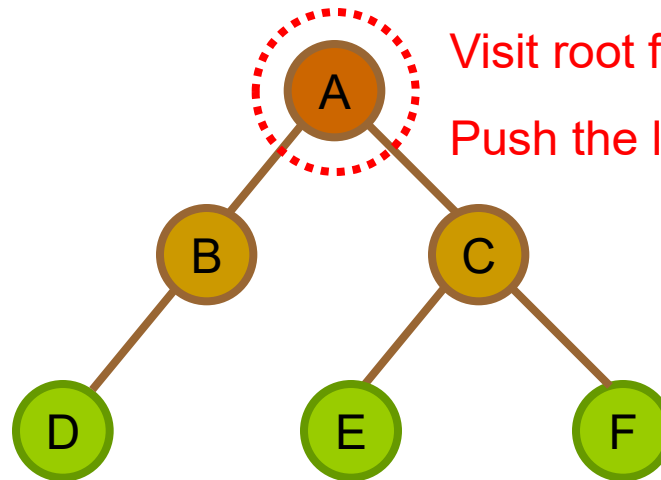
Count Nodes (Iterative Approach)



Stack

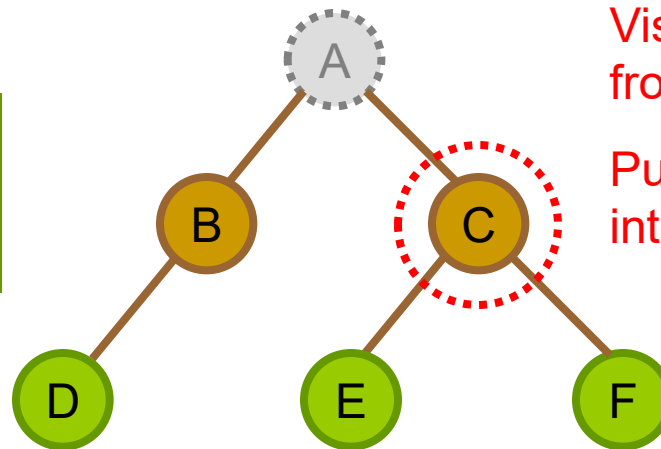


Stack



Visit root first, count + 1

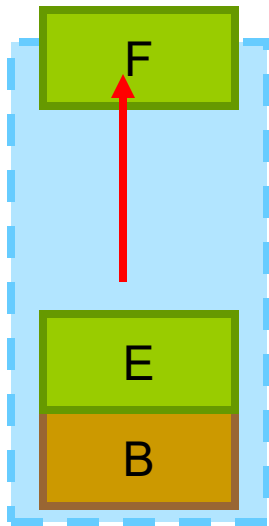
Push the left child and right child into stack!



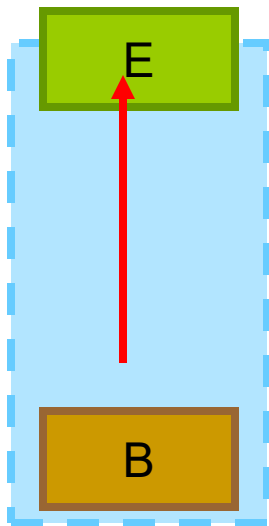
Visit the next node (the top element from stack), count + 1

Push left child and right child of C into stack

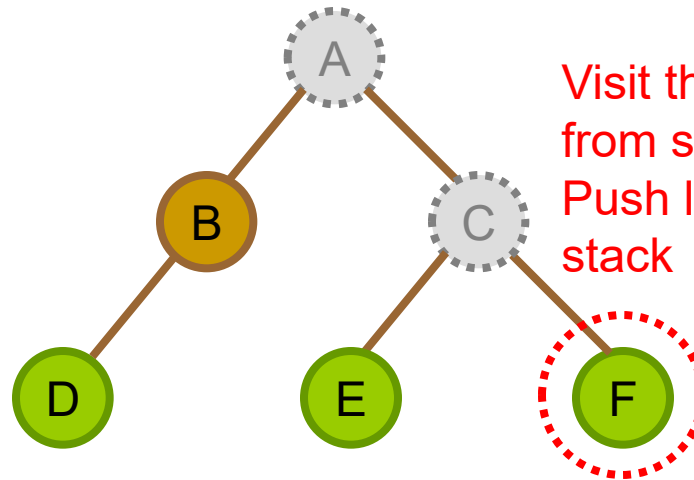
Count Nodes (Iterative Approach)



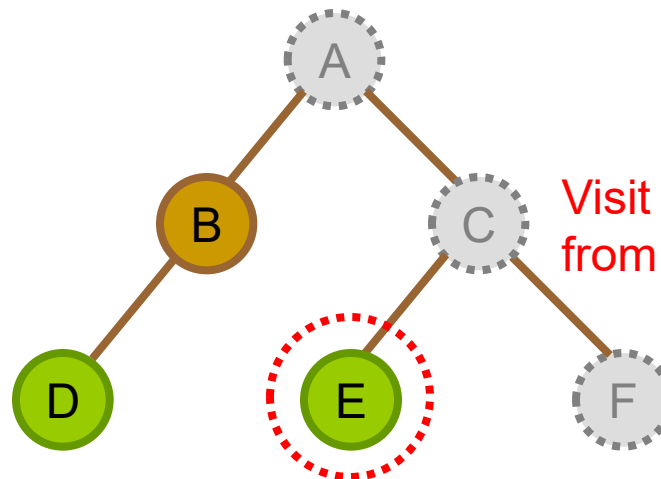
Stack



Stack

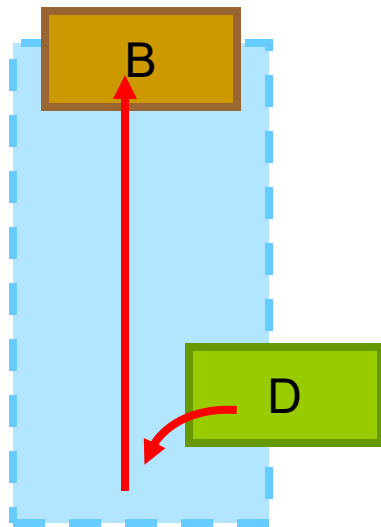


Visit the next node (the top element from stack), count + 1
Push left child and right child of F into stack

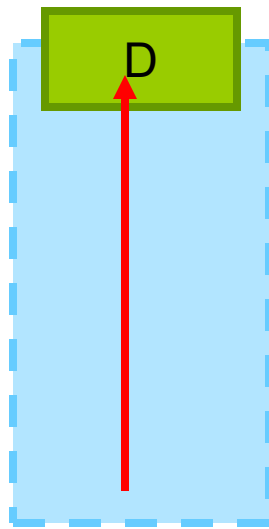


Visit the next node (the top element from stack), count + 1

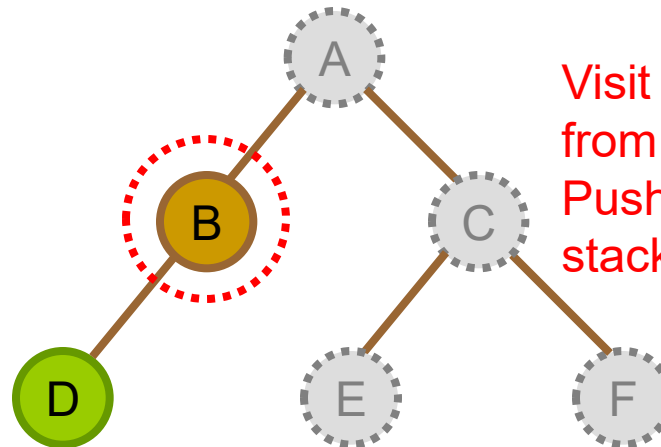
Count Nodes (Iterative Approach)



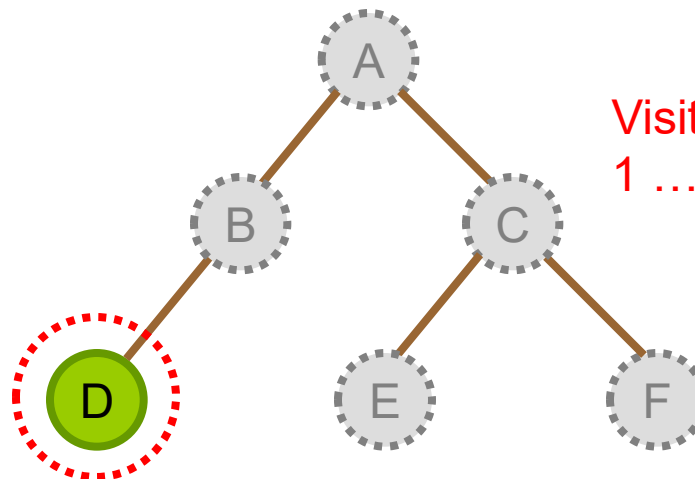
Stack



Stack



Visit the next node (the top element from stack), count + 1
Push left child and right child of B into stack



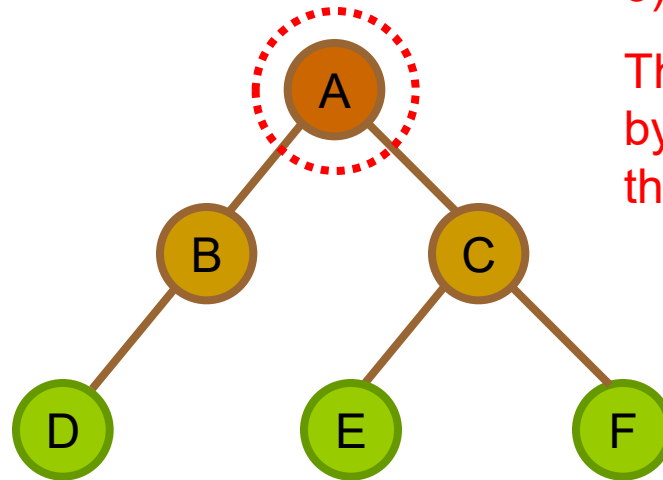
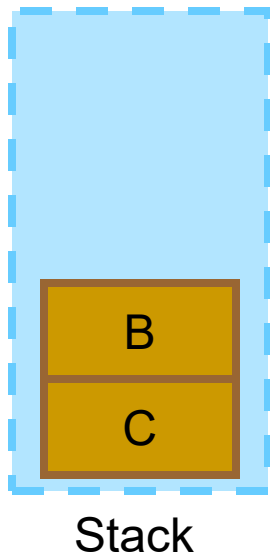
Visit the next node, count + 1 ...until the stack is empty

Non-Recursive Tree Traversal

■ Preorder traversal using stack

- 1) Visit root first
- 2) Push right child
- 3) Push left child

Then iterate the next node by popping from stack until the stack is empty



Non-Recursive Inorder Traversal Using Stack

LVR

```
#include <stack>

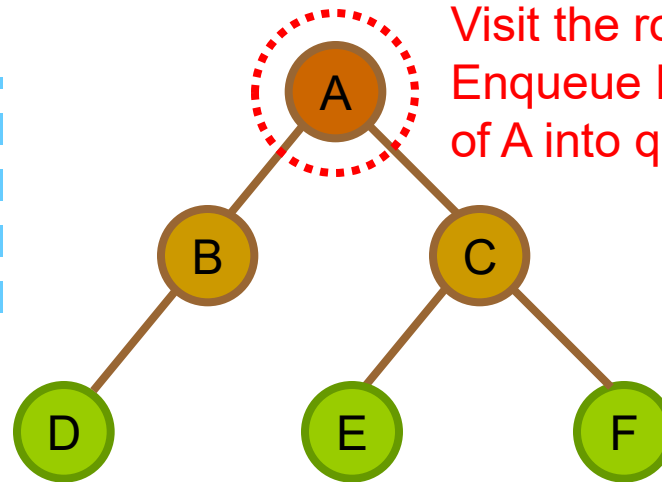
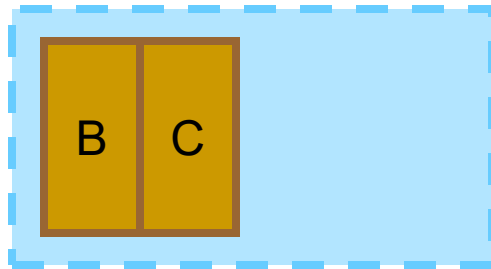
template<class Type>
void traverseLeft(treeNode<Type> *p, stack<treeNode<Type>*>& S) {
    while (p != NULL) {
        S.push(p);
        p = p->left;
    }
}

template<class Type>
void inorder_2(treeNode<Type> *tree) {
    Stack<treeNode<Type>*> S;           // store pointer only
    traverseLeft(tree, S);             // reach leftmost node

    while (!S.empty()) {               //there are nodes not yet visited
        treeNode<Type>* p = S.top();
        S.pop();
        cout << p->info << " ";
        traverseLeft(p->right, S);
    }
}
```

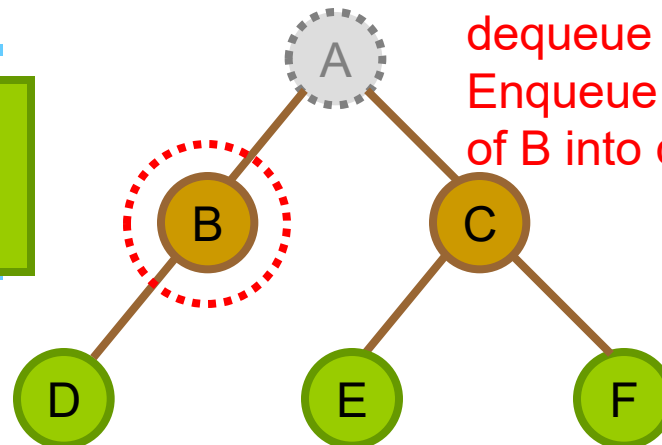
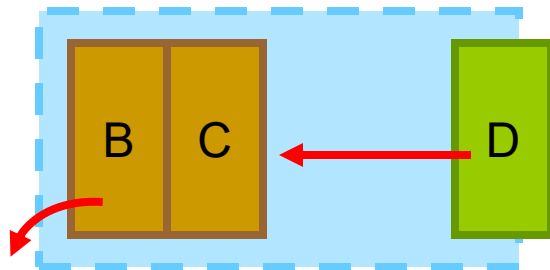
Breadth First Traversal

Queue



Visit the root node
Enqueue left child and right child
of A into queue

Queue



Visit the next node by calling
dequeue
Enqueue left child and right child
of B into queue

Level Order Traversal Using Queue

```
#include <queue>

template<class Type>
void levelTrav(treeNode<Type> *tree) {
    queue<treeNode<Type>*> Q;           // store pointer only

    if (tree != NULL)
        Q.push(tree);

    while (!Q.empty()) {                //there are nodes not yet visited
        treeNode<Type>* p = Q.front();
        Q.pop();

        cout << p->info << " ";

        if (p->left != NULL)
            Q.push(p->left);

        if (p->right != NULL)
            Q.push(p->right);
    }
}
```

Reconstruction of Binary Tree

Class Exercise

- 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

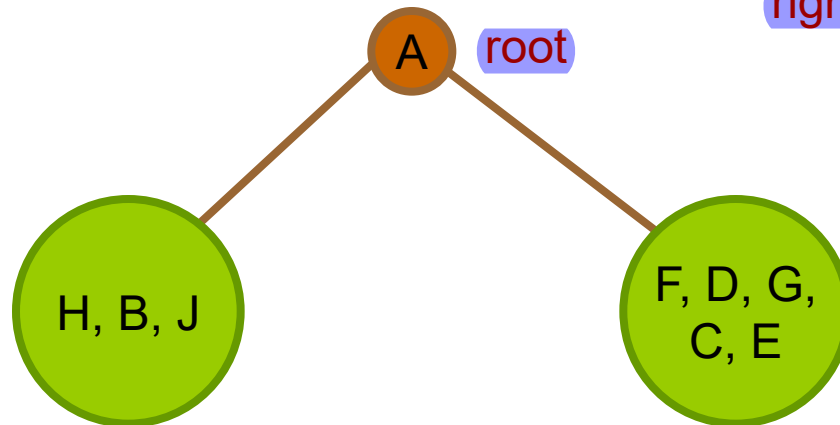
- Step 1) Determine the root node, left and right subtrees
 - From **postorder**, the **last node** is the **root**
 - e.g. node A
 - Then from **inorder**, the nodes on the **left hand side** of node A belongs to the **left subtree** of node A, nodes on the **right hand side** belongs to its **right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

First Determine the Root

■ Postorder: H J B F G D E C **A**

■ Inorder: **H B J** **A** **F D G C E**

left subtree



root

right subtree

root node

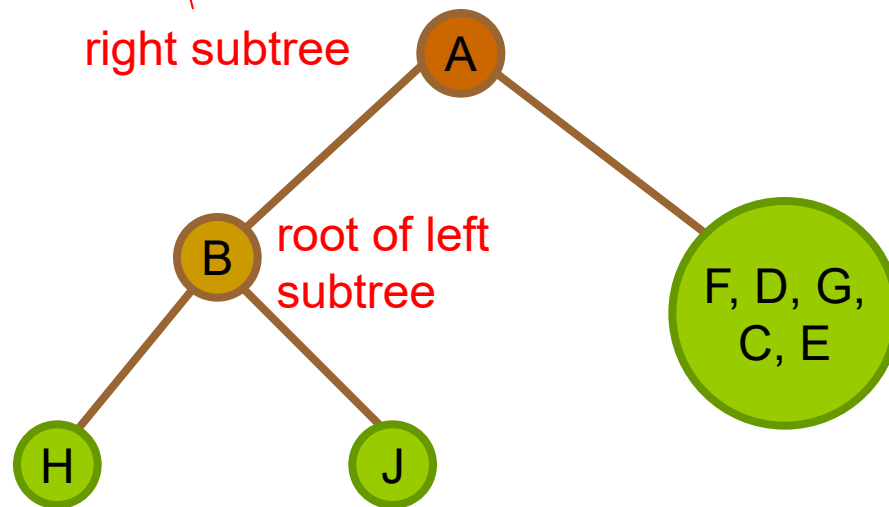
Consider Left Subtree of Root

■ Postorder: H J **B** F G D E C A

■ Inorder: **H B J** A F D G C E

left subtree

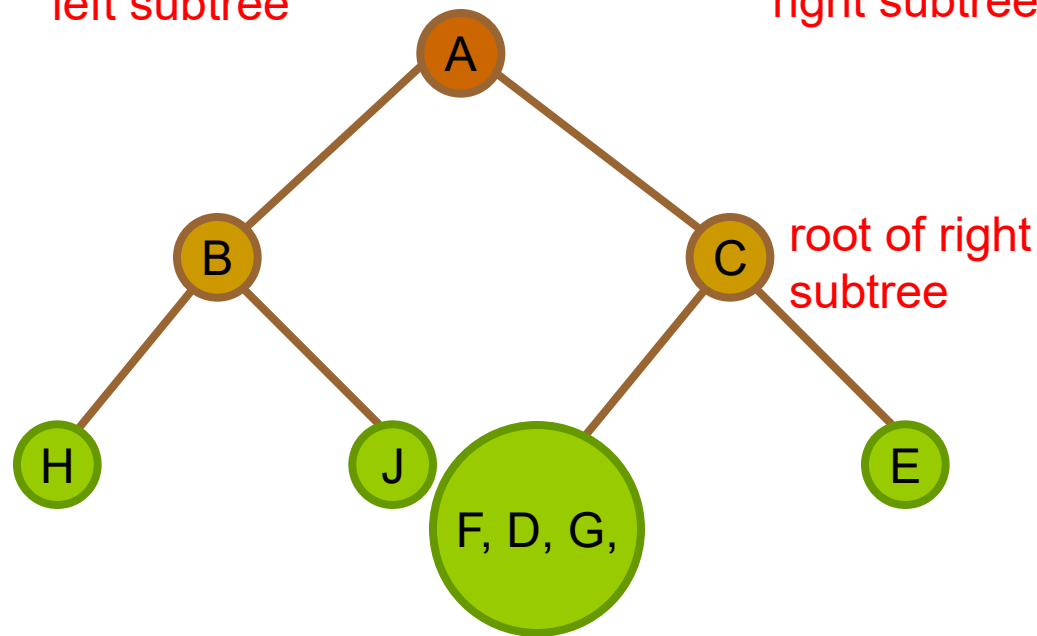
right subtree



Consider Right Subtree of Root

■ Postorder: ~~H~~ ~~J~~ ~~B~~ F G D E **C** ~~A~~ root node of the subtree

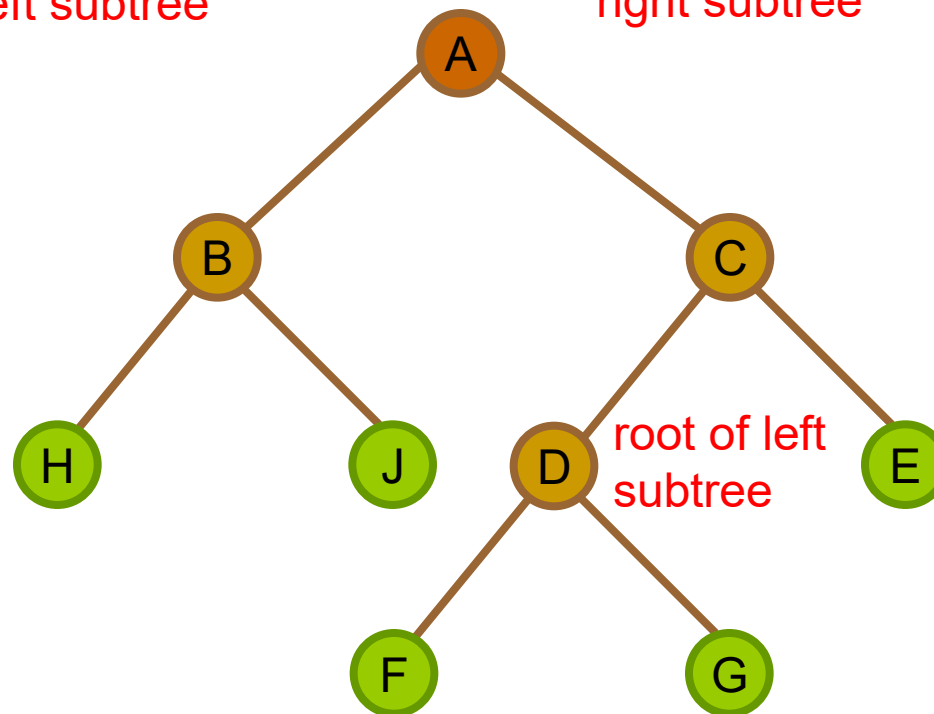
■ Inorder: ~~H~~ ~~B~~ ~~J~~ ~~A~~ **F D G C** E left subtree right subtree



Consider Left Subtree of C

■ Postorder: ~~H~~ ~~J~~ ~~B~~ F G **D** ~~E~~ ~~C~~ ~~A~~ root node of the subtree

■ Inorder: ~~H~~ ~~B~~ ~~J~~ ~~A~~ **F** **D** **G** ~~C~~ ~~E~~ left subtree right subtree



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 Tree Implementation



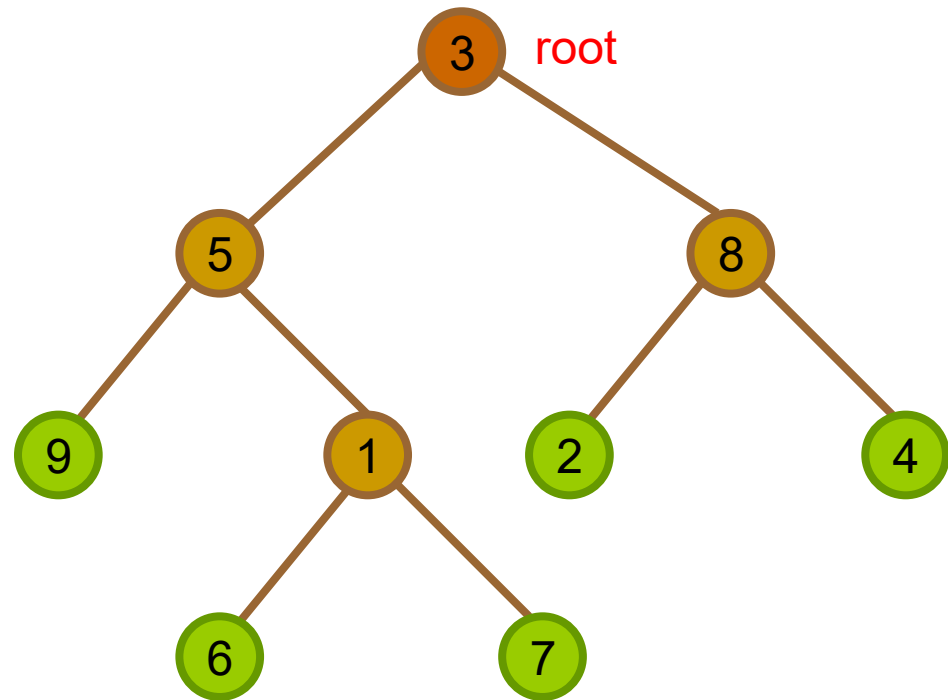
- To avoid the tedious details, only the implementation of some selected member functions will be given.
- A binary tree is a **container** (i.e. it is used to hold a collection of items).
- We need to provide one or more types of **iterator** such that the external user can use it to traverse the elements in the tree one at a time.
- The implementation of the iterator class given below only serves to illustrate the conceptual idea.
- Different implementation methods are used in the C++ STL.

Binary Search Tree (BST)

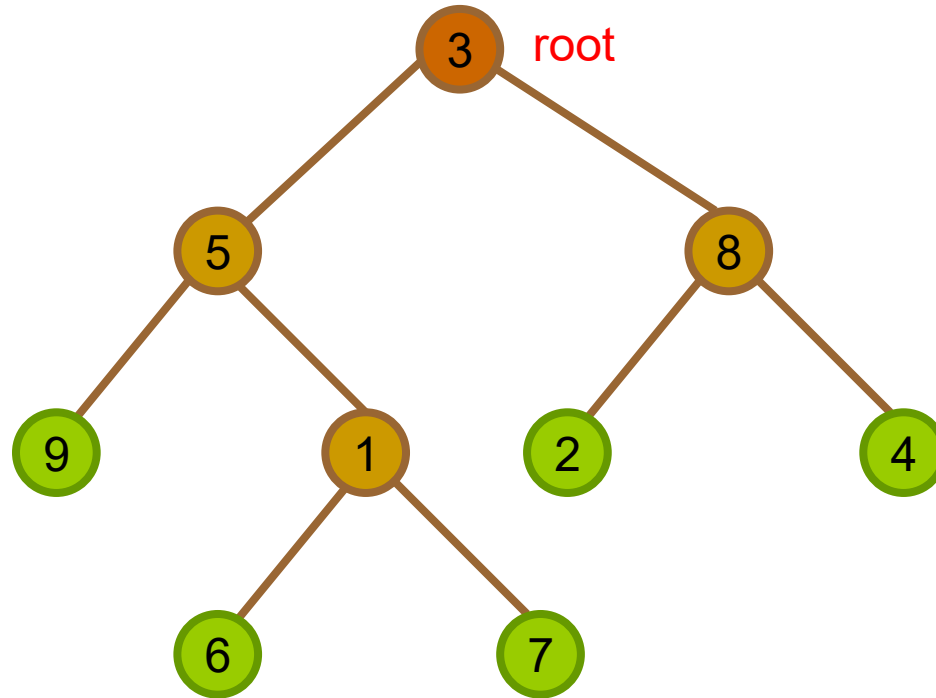
and its operations

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?



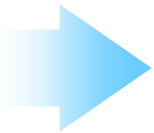
How to Search a Tree?



array:

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

for-loop

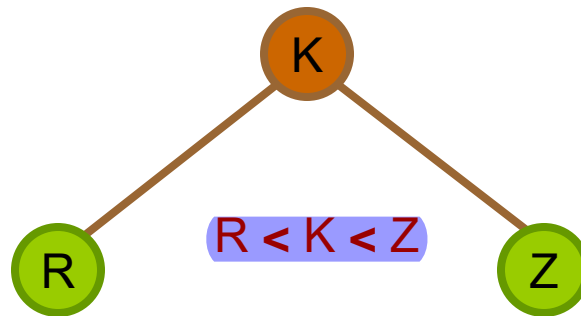


In order to find the max. node, min. node or a node equal to particular value, you have to visit the entire tree once

How about linked list implemented tree?

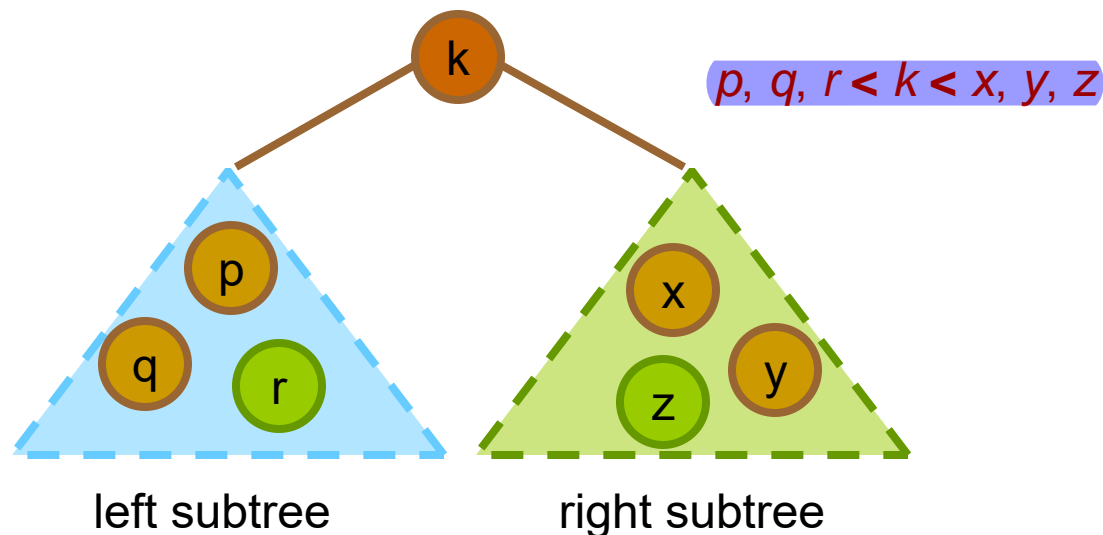
Pre-sort Tree

- 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.

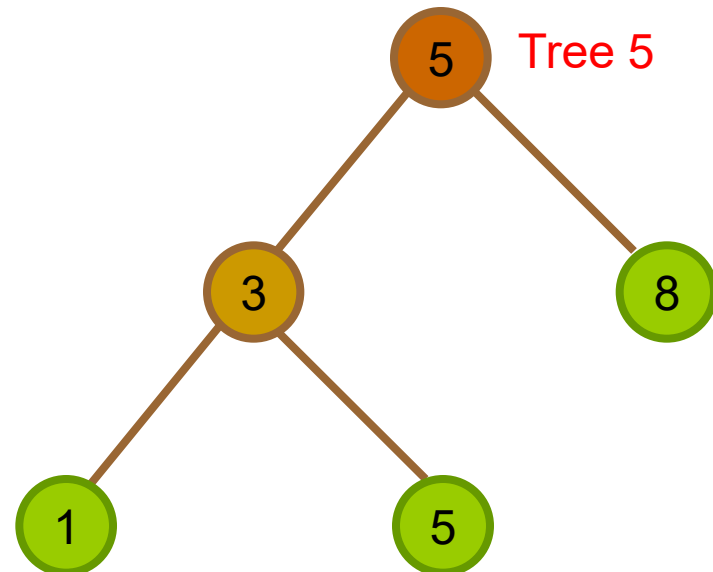
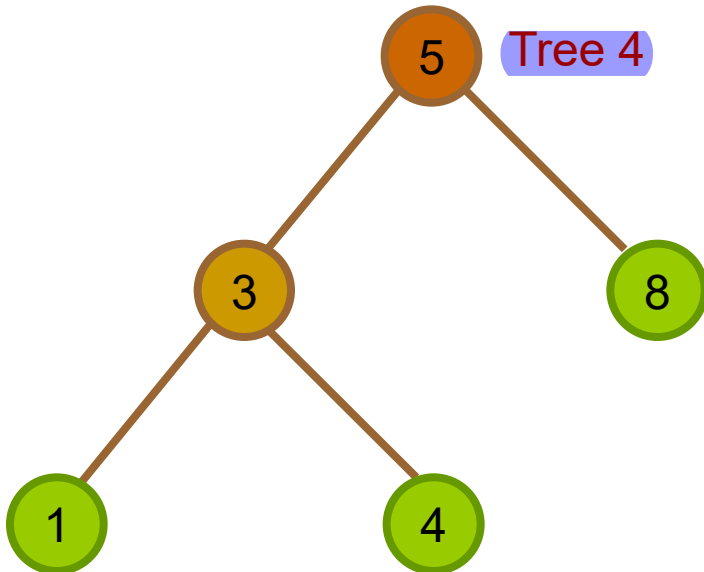
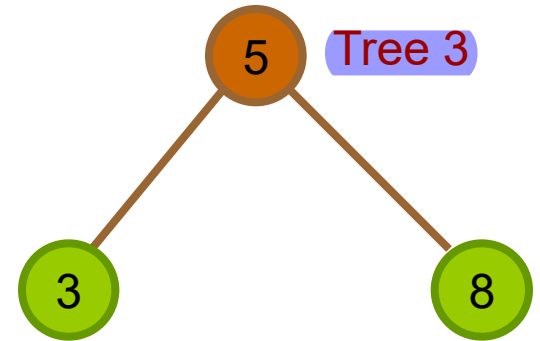
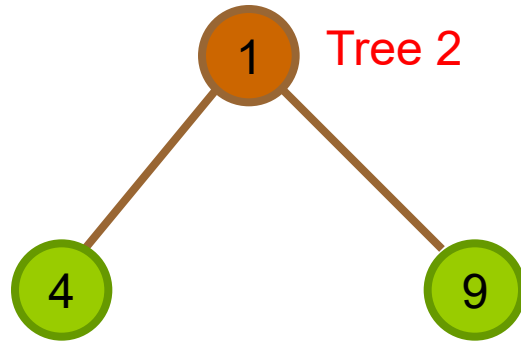
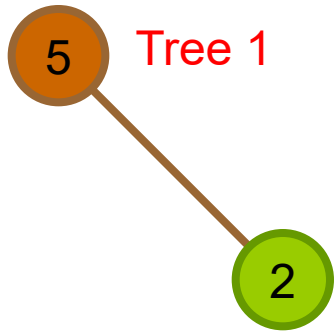


Binary Search Tree (BST)

- 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).

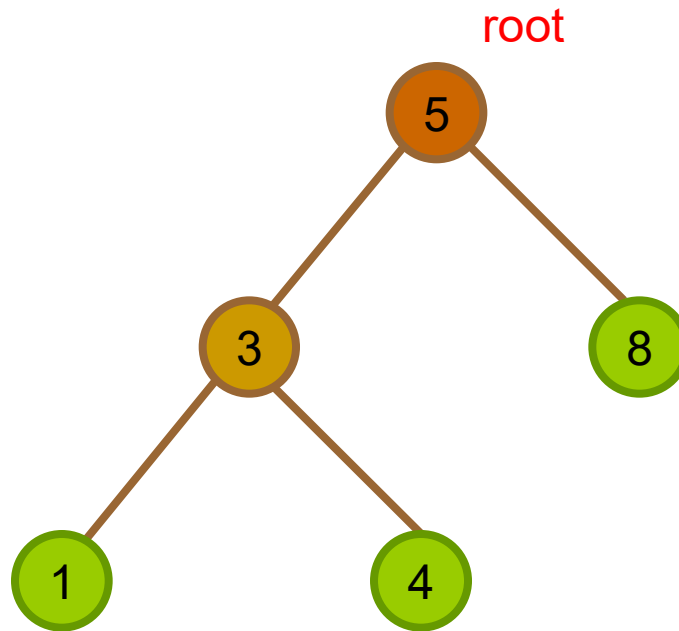


Exercise: Are They BST?



Find a Node in BST

■ How to find a node with value = k ?

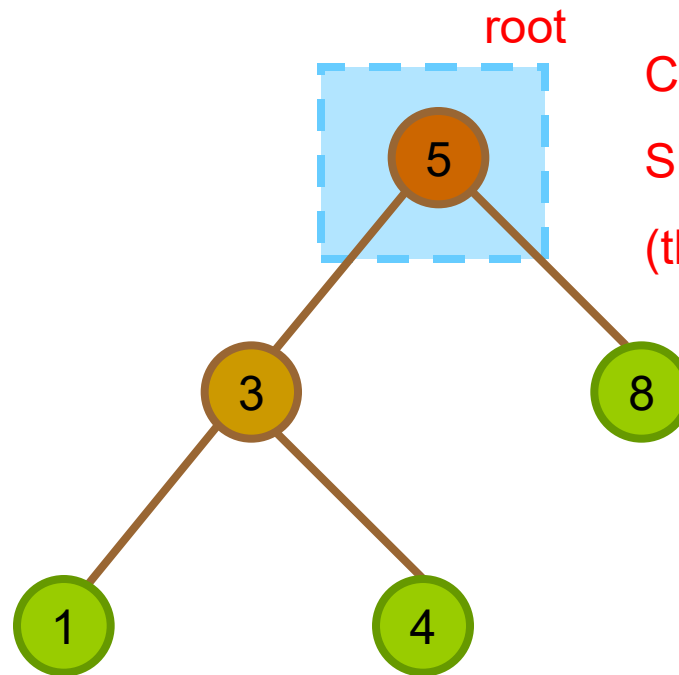


Find a node in BST

- Compare k with the value of root
 - If value of $\text{root} == k$, the answer is root!
 - If value of $\text{root} > k$, go to the left subtree
 - If value of $\text{root} < k$, go to the right subtree
-
- Continue to compare recursively until it meets a leaf node

Find a Node in BST

■ e.g. $k = 1$



Compare k with 5

Since $k < 5$, so go to left subtree
(the right subtree will be ignored)

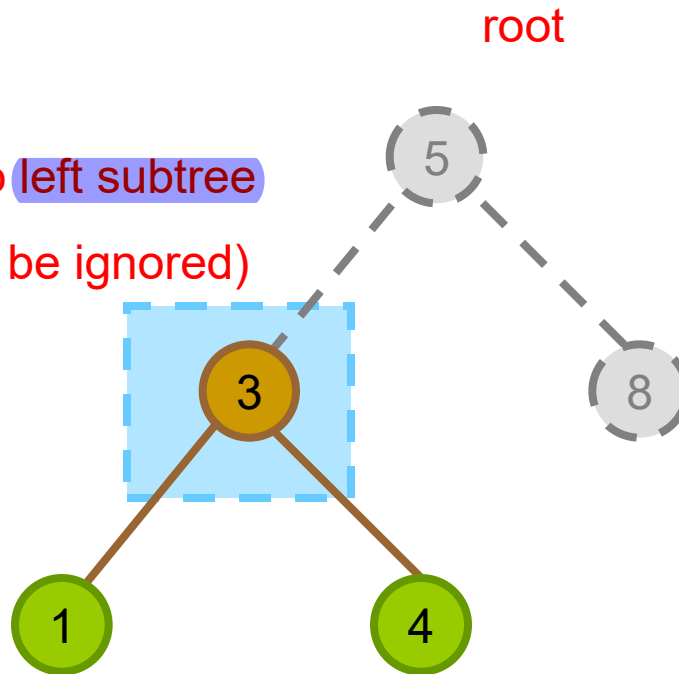
Find a node in BST

■ e.g. $k = 1$

Compare k with 3

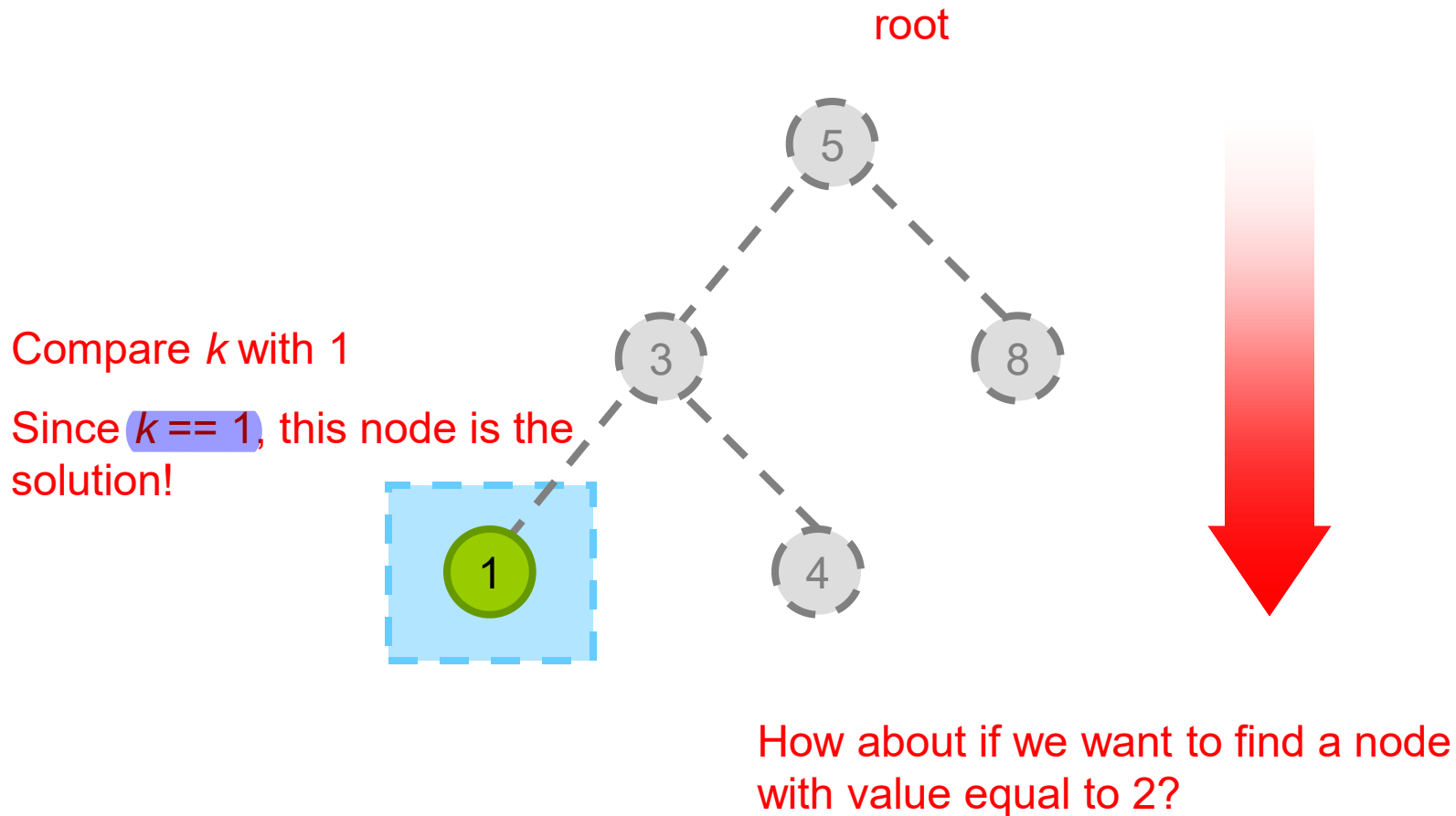
Since $k < 3$, so go to left subtree

(its right subtree will be ignored)



Find a node in BST

■ e.g. $k = 1$



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

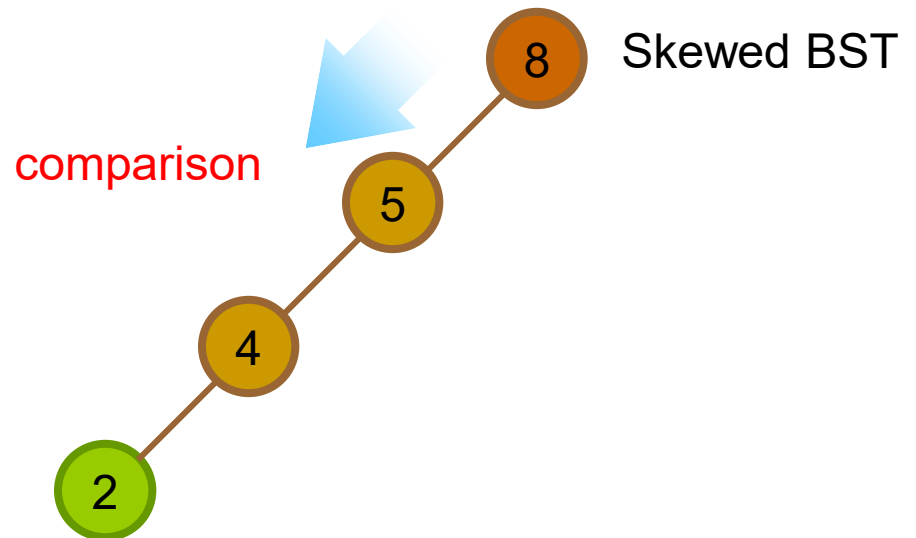
Complete BST

- 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 each comparison
 - The **depth** of the tree is **$\text{floor}(\log_2 n)$**
- **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

- If search() is a public member function of class BST, it should return an **iterator** that refers to the node containing x instead of a node pointer to x so as to prevent exposing internal structure.

```
// 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

- If this is implemented as a member function, it should be defined as a private function. Note that public member functions should not require any private member variables as input parameters.

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

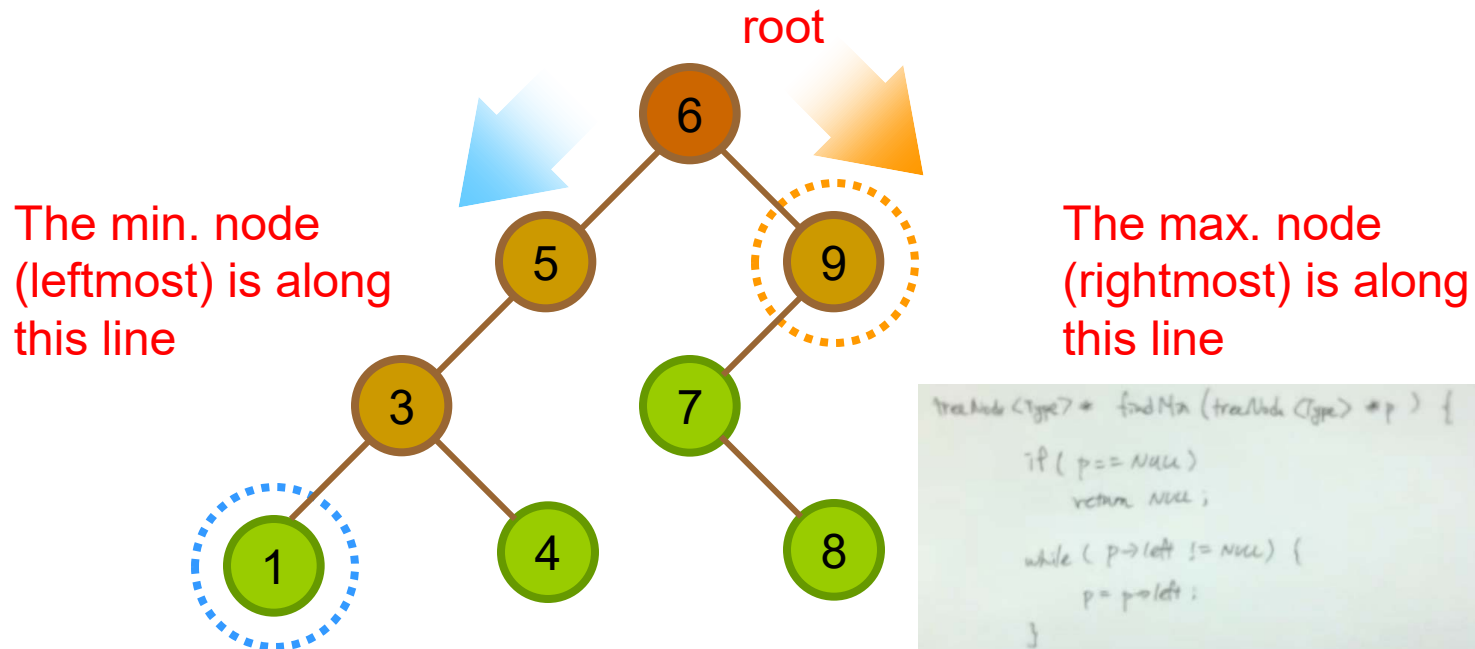
    if (x == p->info)
        return p;

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


Class Exercise:

Min & Max Node of BST

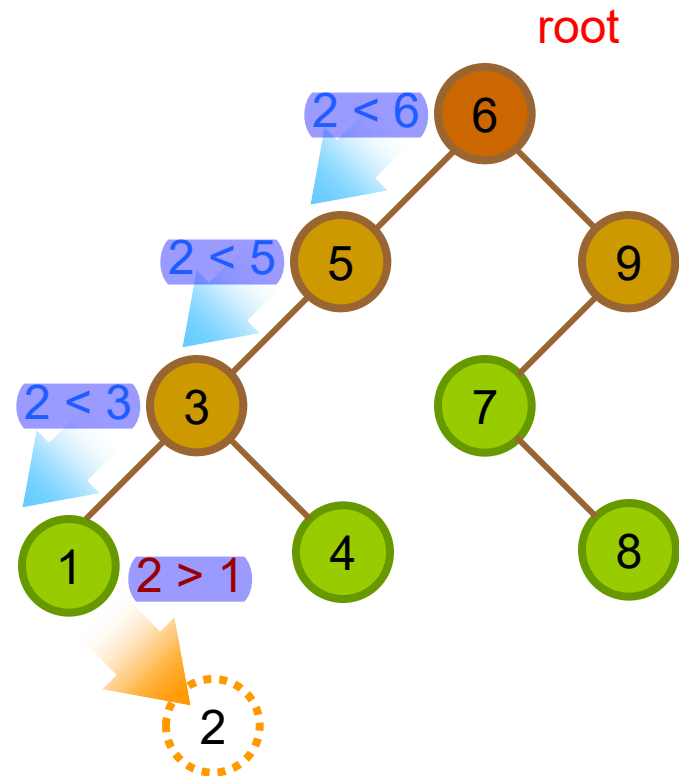
- Exercise: write the code to find the min and max node (using recursion/iteration)



```
TreeNode<Type> * findMin (TreeNode<Type> *p) {  
    if ( p == NULL )  
        return NULL ;  
    while ( p->left != NULL ) {  
        p = p->left ;  
    }  
    return p ;  
}
```

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 exist in the BST
 - Determine the point of insertion



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

Step 1

root

(empty tree)

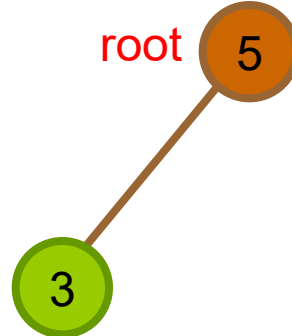
Step 2

root



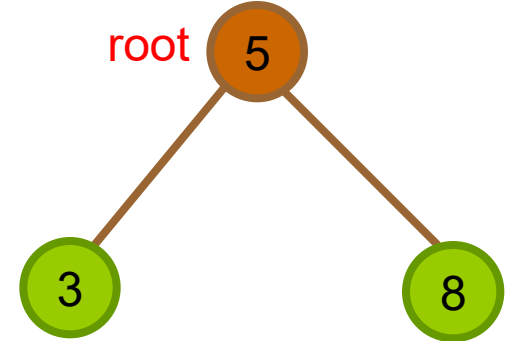
Step 3

root



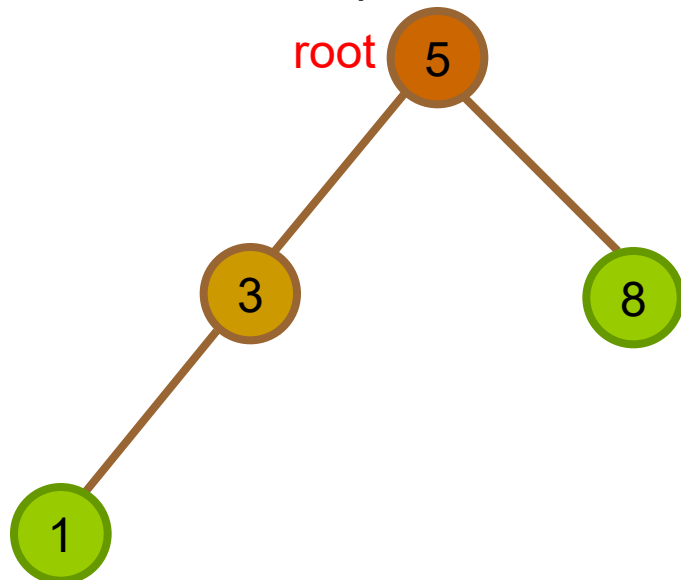
Step 4

root



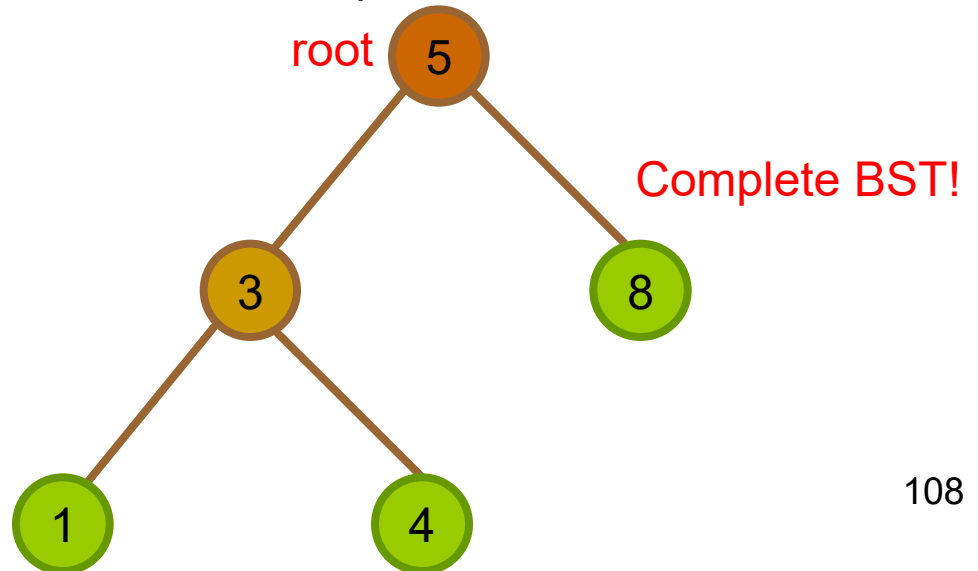
Step 4

root



Step 5

root



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

Step 1

root

(empty tree)

Step 2



root

Step 3



root

Step 4



root

Step 4



root

Step 5



root

Skewed BST!

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;
    }
}
```

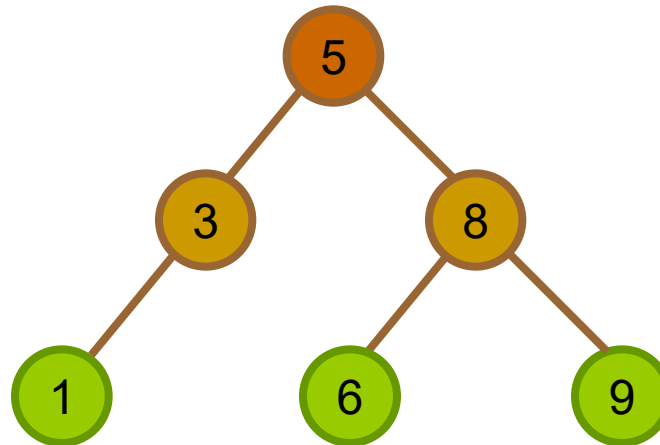
```
treeNode<Type> *v = new treeNode<Type>;
v->info = x;
v->left = v->right = NULL;

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

return v;
}
```

Delete a Node in BST

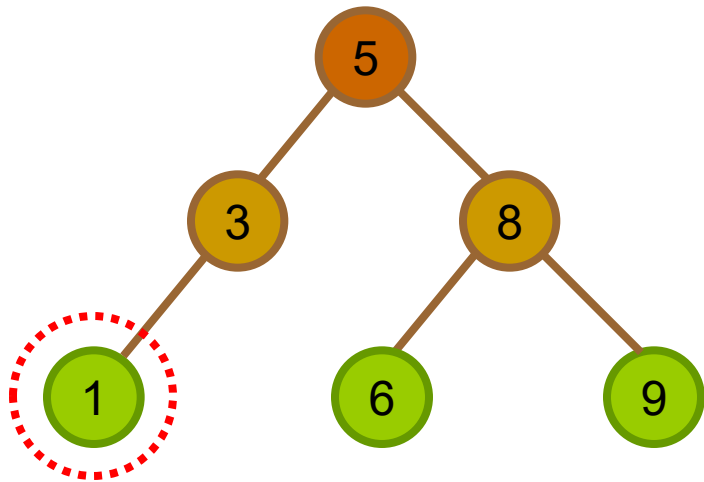
- The property of BST must be **preserved** after deletion
- We have to consider 3 different cases
 - The node to be deleted is:
 - 1) A leaf node (e.g. node 1)
 - 2) A node has only one child (e.g. node 3)
 - 3) A node has two children (e.g. node 5)



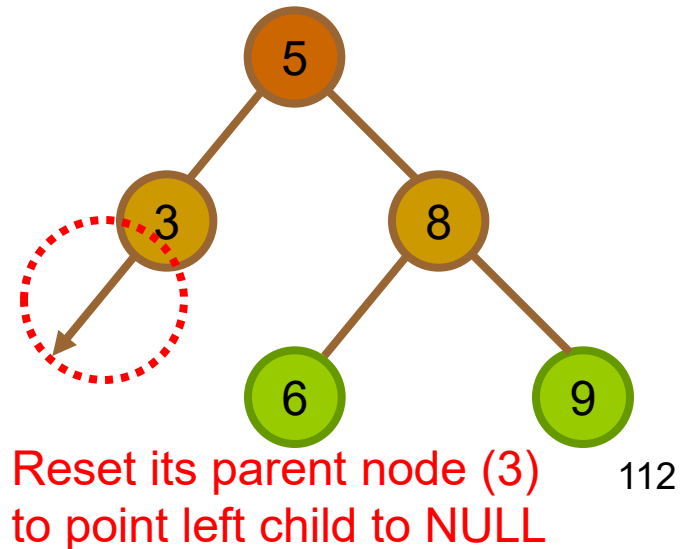
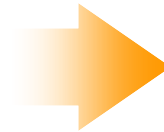
Delete Node: Case 1

■ Degree 0 Node (leaf node)

- Just delete it
- Then reset the reference of its parent node



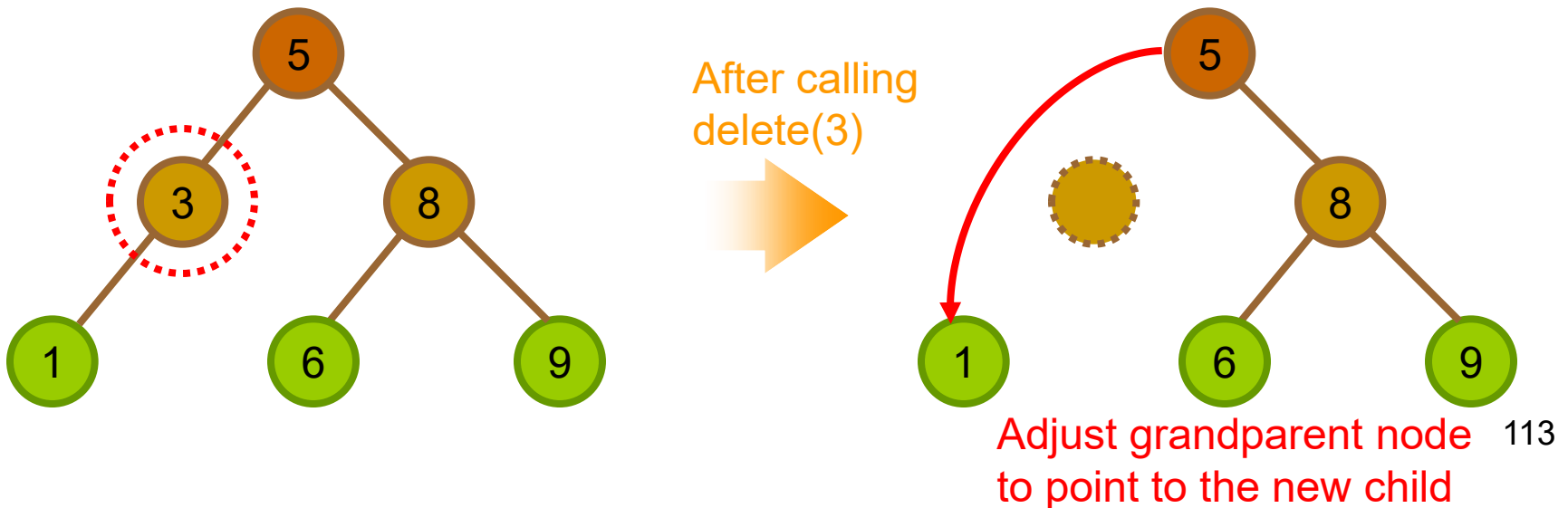
After calling
delete(1)



Delete Node: Case 2

■ Degree 1 Node (with 1 child)

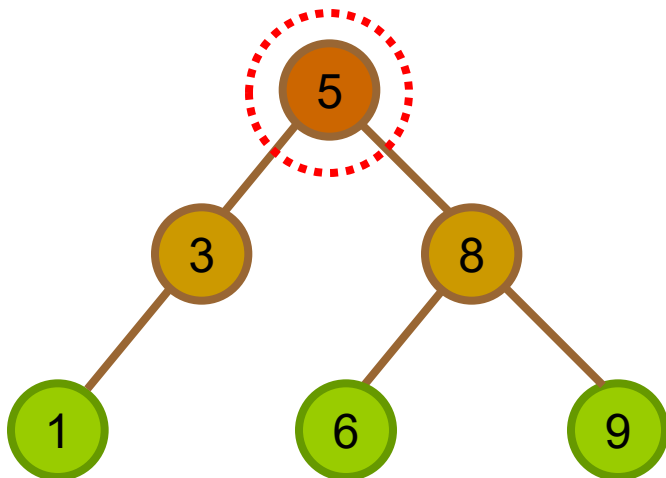
- **Before** deletion, adjust the pointer of parent to point to the grandson
- Then simply delete it



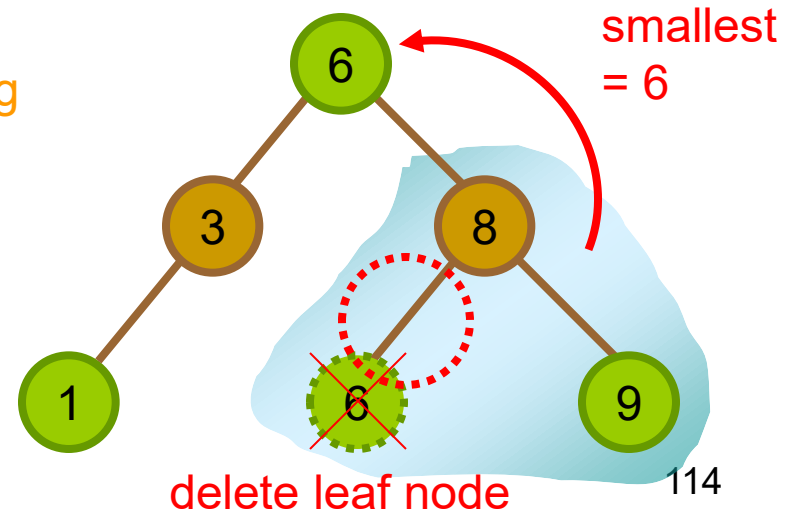
Delete Node: Case 3

■ 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)

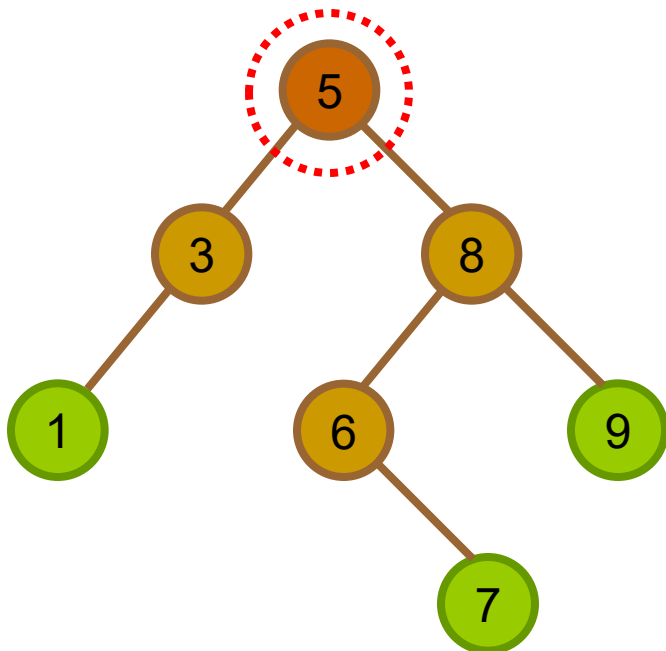


After calling
delete(5)

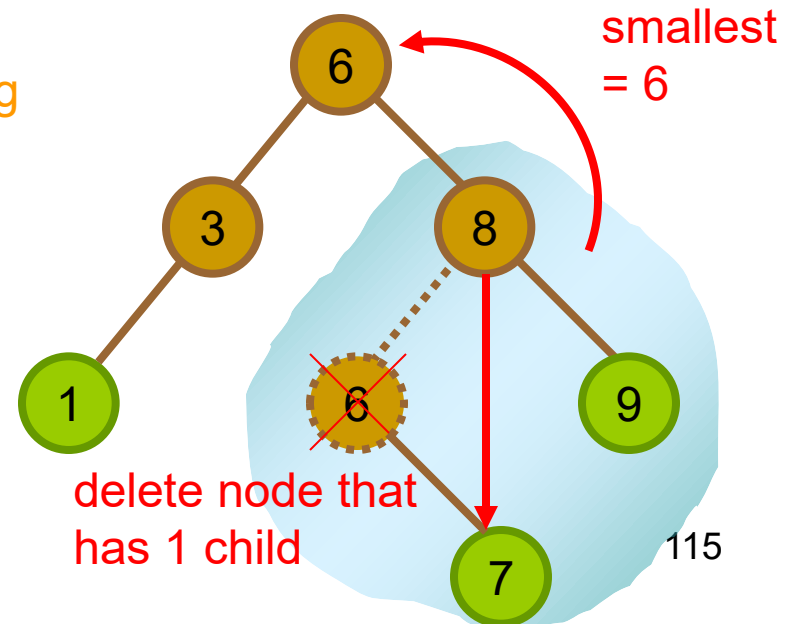
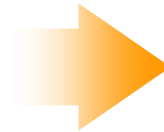


Delete Node: Case 3

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



After calling
delete(5)



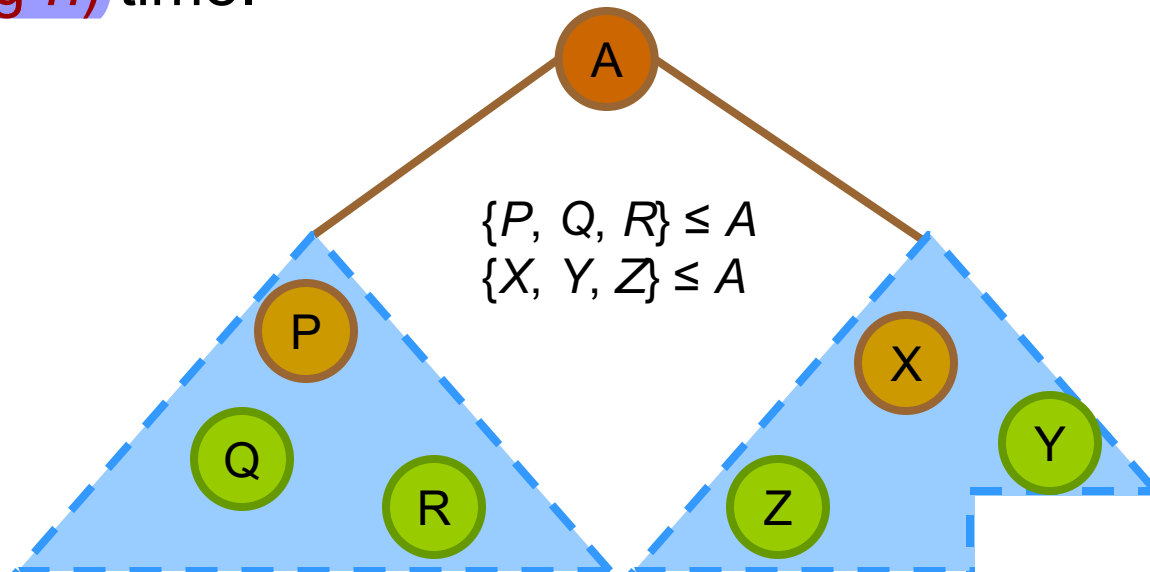
Heap

What is a Heap?

- A **max tree** is a tree in which the key value in **each node** is **no smaller than** the key values in its **children** (if any).
- Similarly, a **min tree** is a tree in which the key value in each node is **no bigger than** the key values in its **children** (if any).
- A **max heap** (descending heap) is a **complete binary tree** that is also a **max tree**.
- A **min heap** (ascending heap) is a **complete binary tree** that is also a **min tree**.

Using Heap as Priority Queue

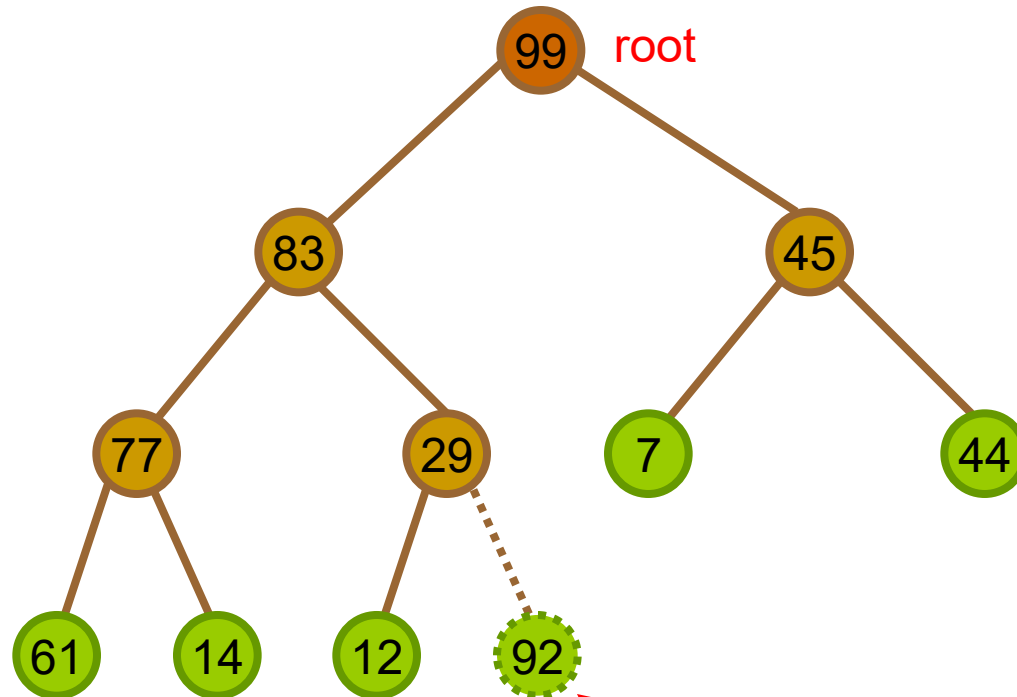
- In **Priority Queue**, the element to be deleted (dequeue) is the one with the highest priority.
- **Max Heap** always has the largest element in root
- Both the insert and delete operations on a heap require $O(\log n)$ time.



Insert Node Into Heap

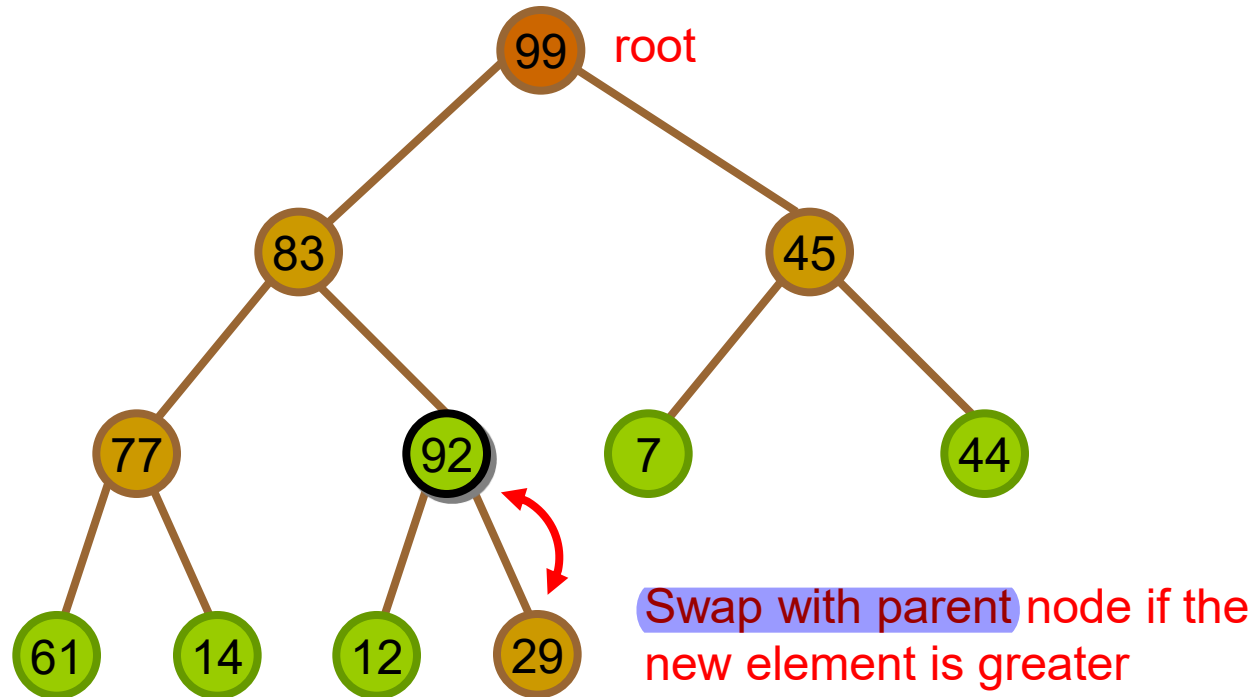
- Step 1) Insert the new element in the next bottom leftmost place
- Step 2) **Percolate up**
 - Swap with its parent node recursively until it satisfy the property of heap

Example: Percolate Up

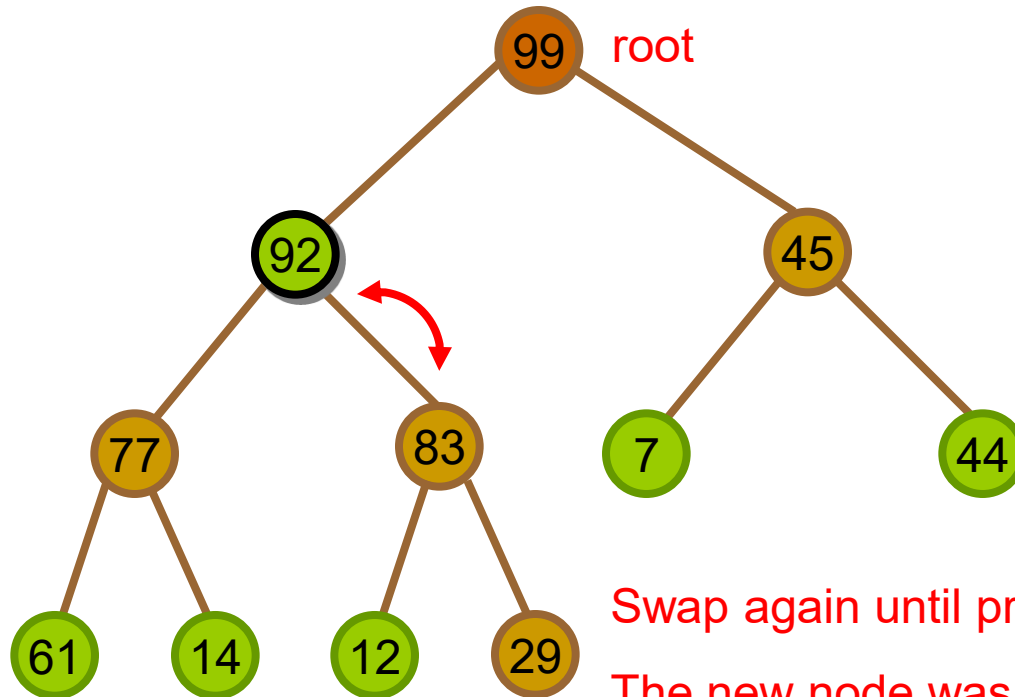


A new element was added here
(and its noted that property 2
has been violated!)

Example: Percolate Up



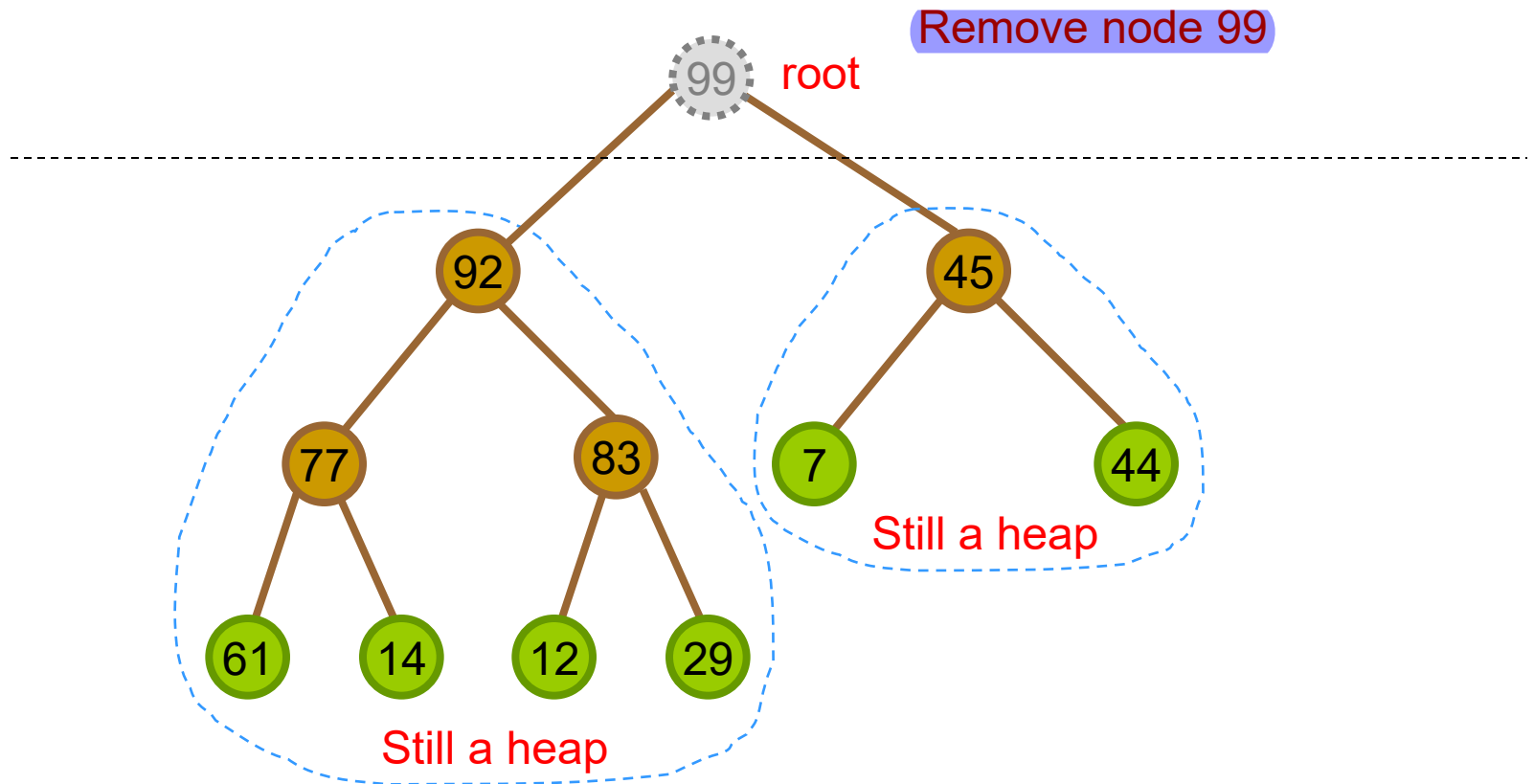
Example: Percolate Up



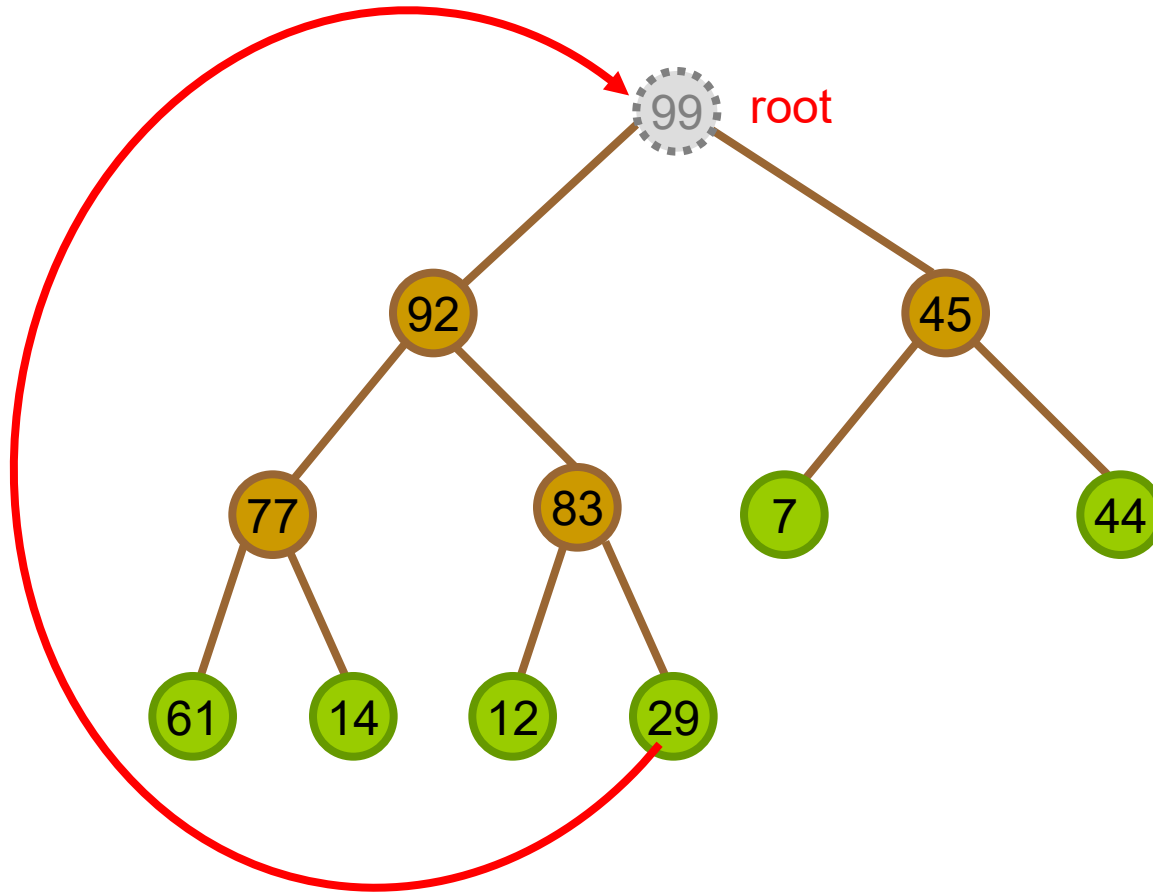
Remove Node From Heap

- In heap, nodes are always remove the **root** position (the largest element)
- Step 1) **Replace** the **root** node with the **bottom rightmost element**
- Step 2) **Percolate down**
 - **Swap with the its greater child** node recursively until it satisfies the heap property

Example: Percolate Down



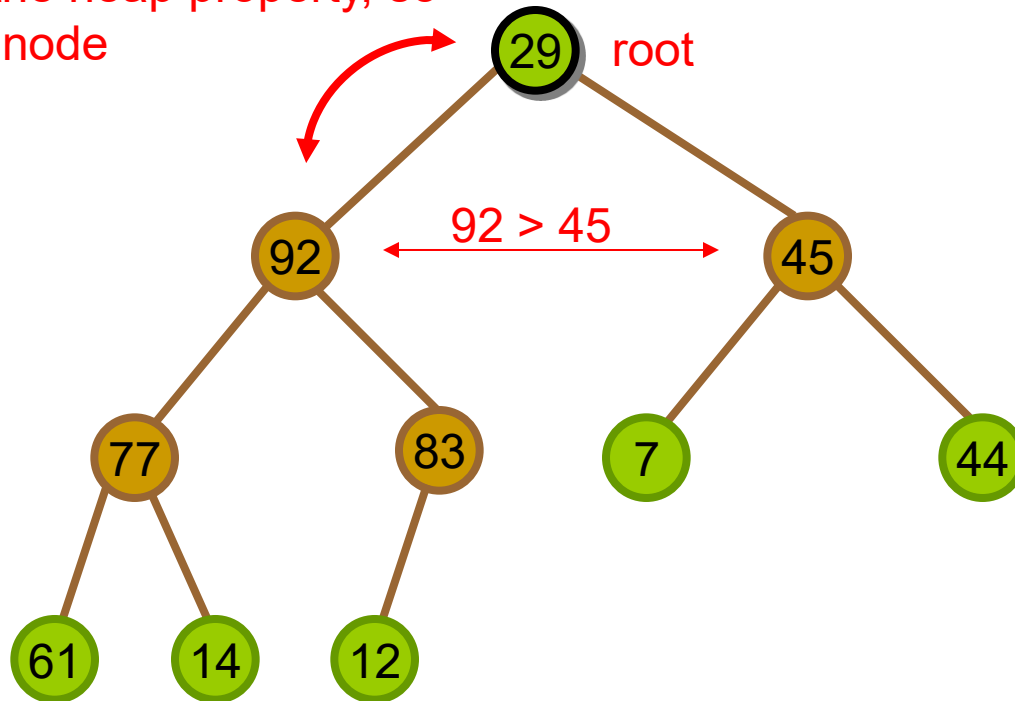
Example: Percolate Down



move node 29 to replace root

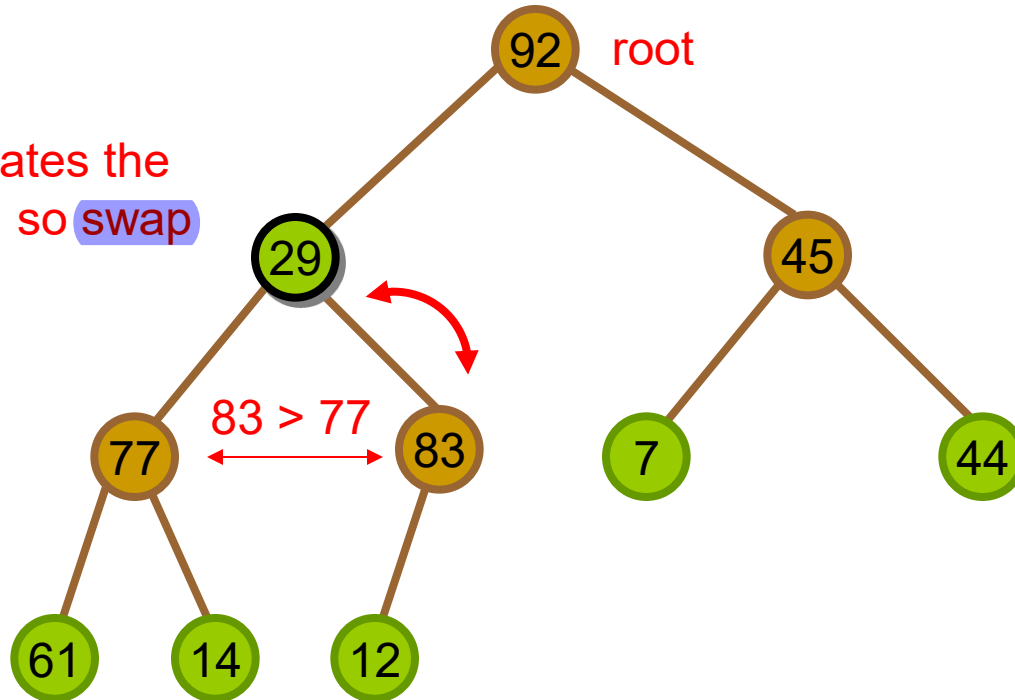
Example: Percolate Down

Now it violates the heap property, so
swap with child node



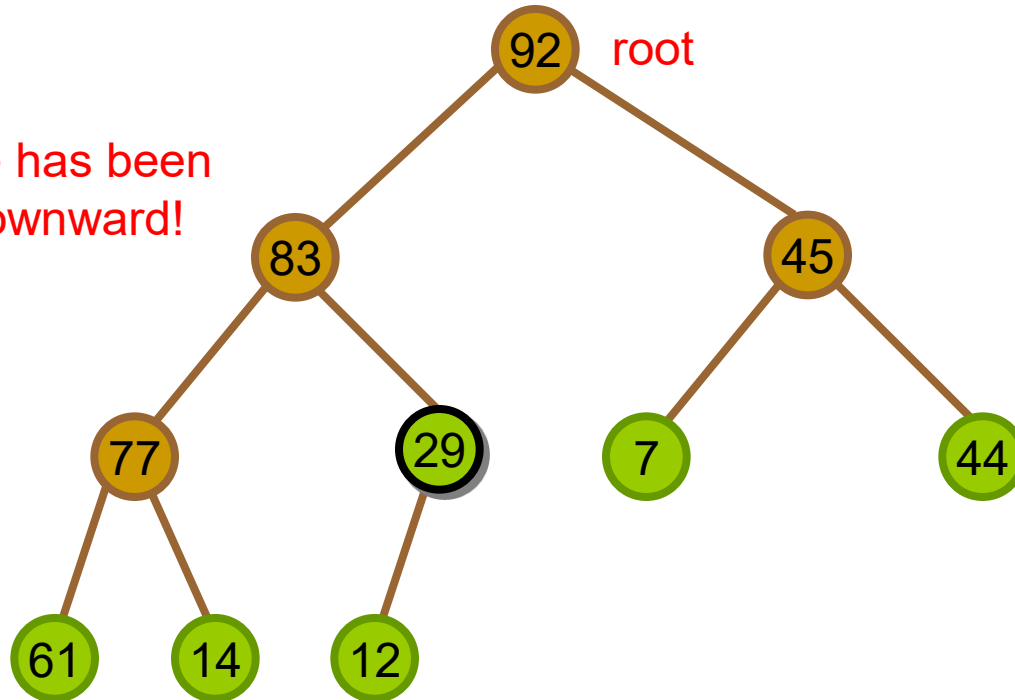
Example: Percolate Down

Now it still violates the heap property, so swap again



Example: Percolate Down

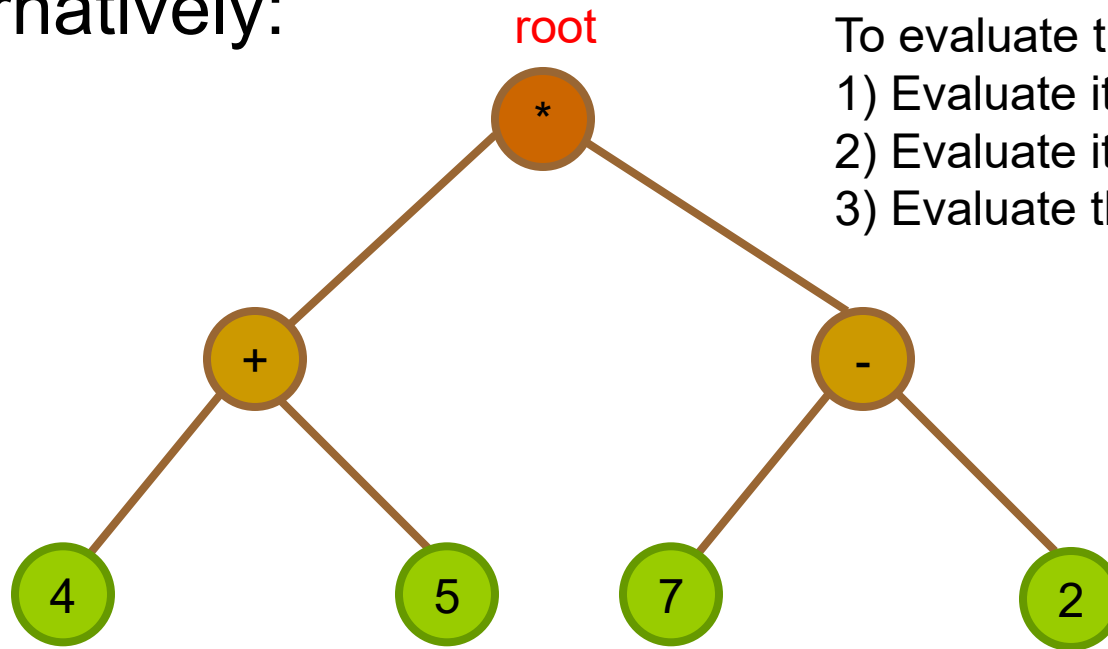
The new node has been propagated downward!



Applications

1st Applications: Infix & Postfix

- We learnt to use stack to convert infix to postfix, alternatively:



To evaluate the tree:

- 1) Evaluate its left subtree,
- 2) Evaluate its right subtree,
- 3) Evaluate the root

Inorder (LVR): $(4 + 5) * (7 - 2)$ ← = infix!

Postorder (LRV): $4 5 + 7 2 - *$ ← = postfix!

Evaluate Arithmetic Expression Using a Binary Tree

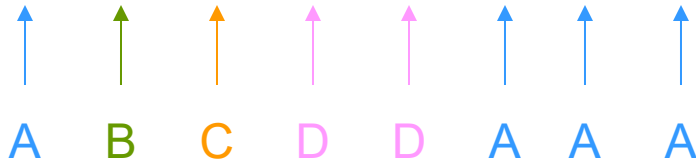
```
#define operand 0
#define operator 1

struct infoRecord {
    char dataType;
    union {                //all members in union share the same physical space in memory
        char opr;
        double val;
    };
};

// Precondition: the expression tree is nonempty and has no
// syntax error. The algorithm is based on postorder traversal.
double evalExprTree(treeNode<infoRecord> *tree) {
    if (tree->info.dataType == operand)
        return tree->info.val;
    else {
        double d1 = evalExprTree(tree->left);
        double d2 = evalExprTree(tree->right);
        char symb = tree->info.opr;
        return evaluate(symb, d1, d2);    // compute the result, not shown here
    }
}
```

2nd Application: Huffman Tree

- To encode and decode a message using shorter length
- e.g. the original message is “ABCD DAAA”
- We use 00 to represent A, 01 to represent B, 10 to represent C, 11 to represent D
- The message can be encoded as “0001101111000000” (16 bits)



Not Optimal

- But we found that the previous encoding method is not optimal
- Since character **A repeated many times**
- It is not wise to use the same no. of bits to represent as other characters
- Variable length codeword
 - **frequently** appeared character should use **fewer** bits!

New Encoding Scheme

- 0 to represent A
- 100 to represent B
- 101 to represent C
- 11 to represent D

| Code | Symbol |
|------|--------|
| 0 | 'A' |
| 100 | 'B' |
| 101 | 'C' |
| 11 | 'D' |

- The message can be encoded as

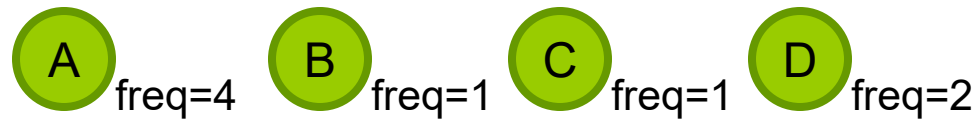
“01001011111000” (14 bits only!)

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
A B C D D A A A

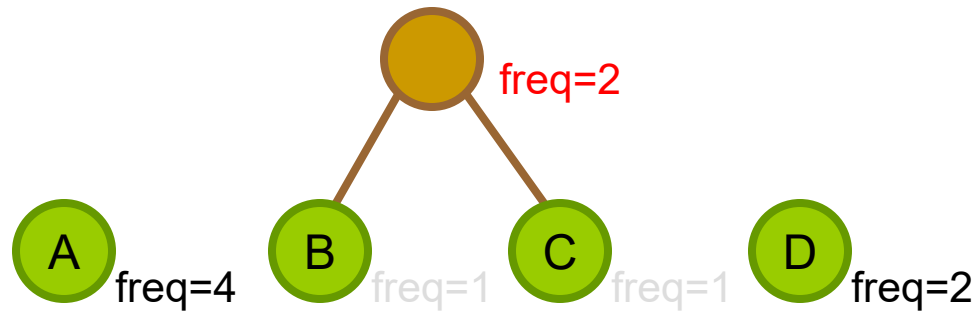
How to Determine the Code Table?

- Solution: Huffman Tree
- The original message is “ABCD DAAA”
- Count the frequency of each character
 - A: 4
 - B: 1
 - C: 1
 - D: 2
- Build the Huffman tree by **recursively** grouping the smallest two nodes together

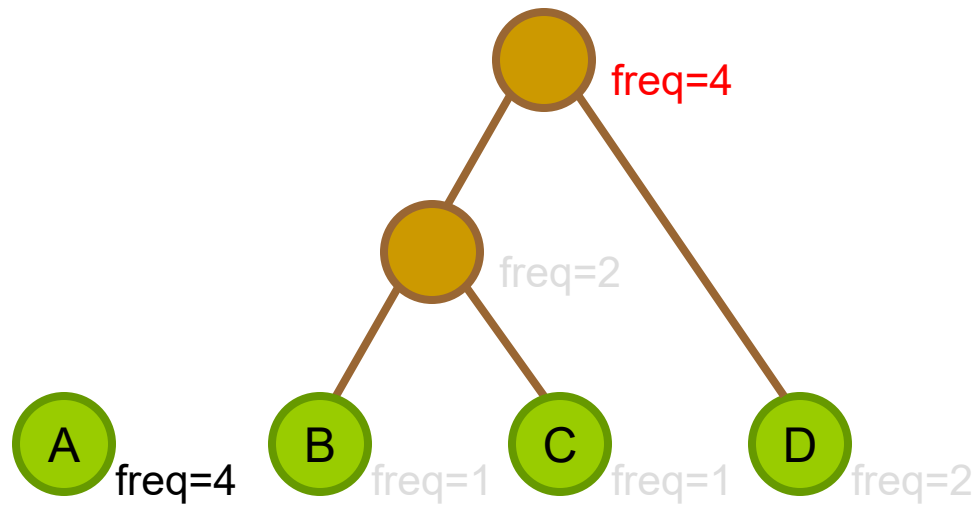
Combine Two Nodes Whose Frequency are Smallest



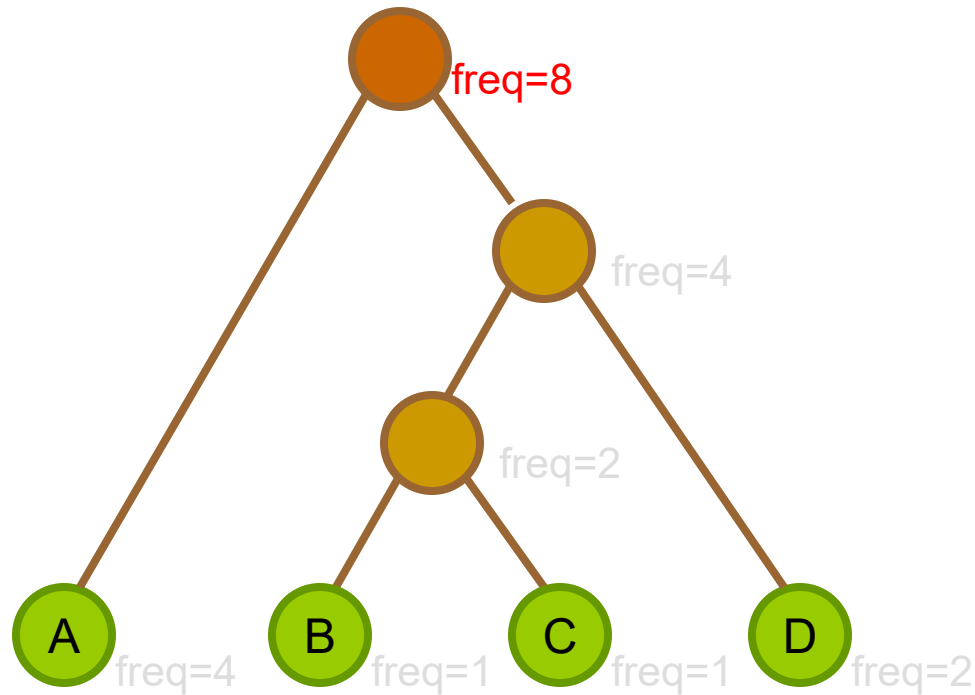
Combine and Update the Frequency



Combine Again



Combine Again Until...

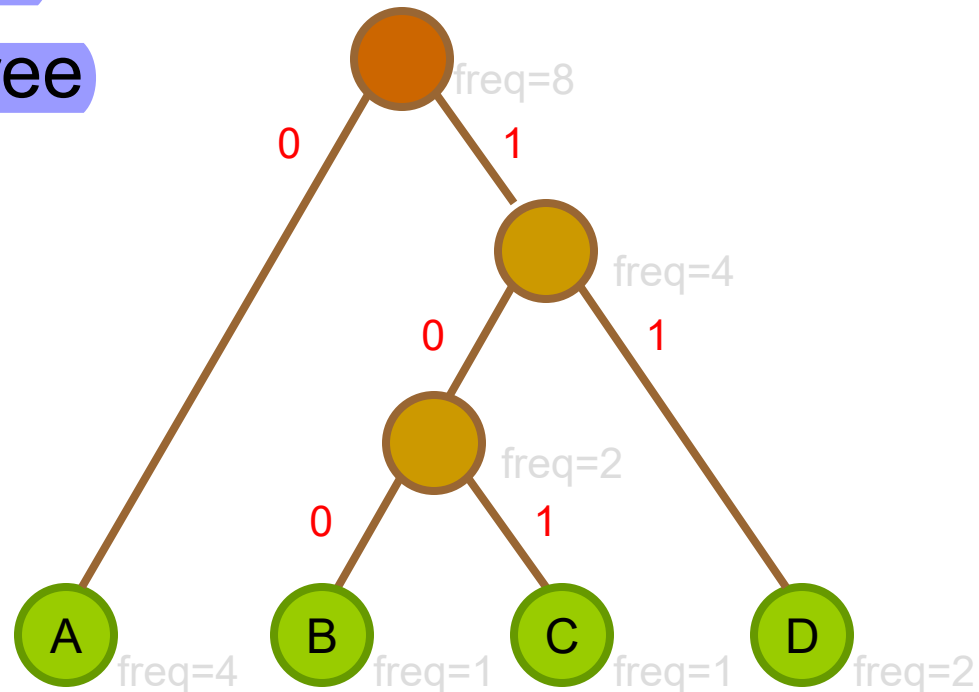


Assign Values On Each Subtree

■ By convention:

■ 0: left subtree

■ 1: right subtree



The final Huffman code:

0

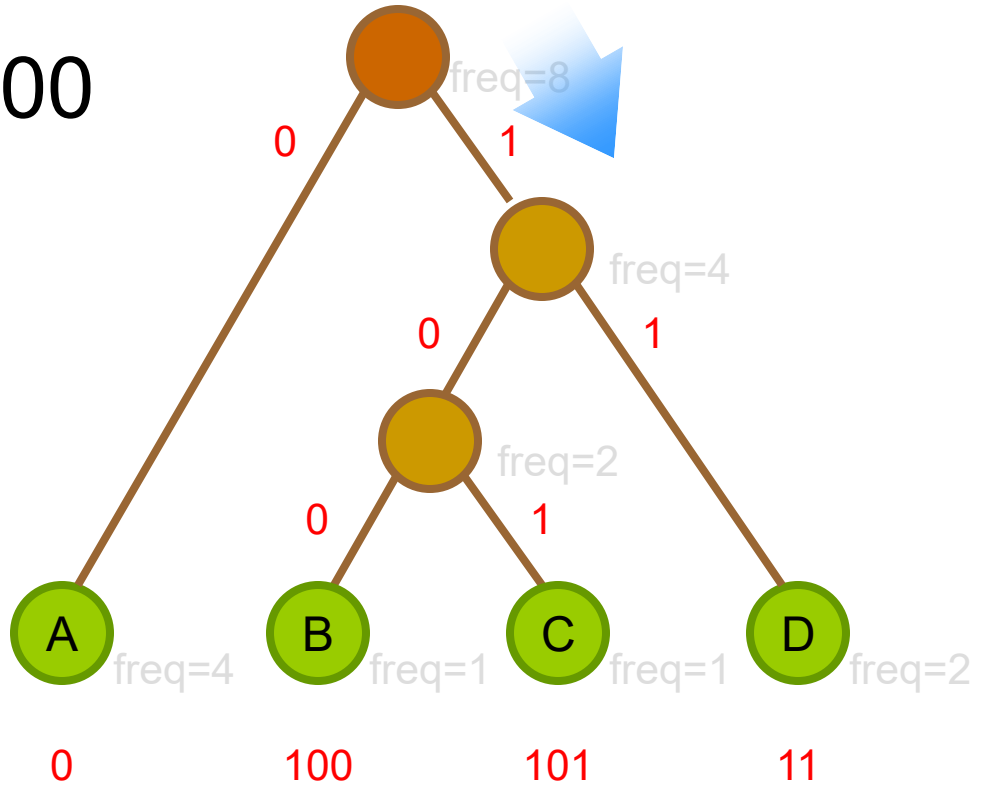
100

101

11

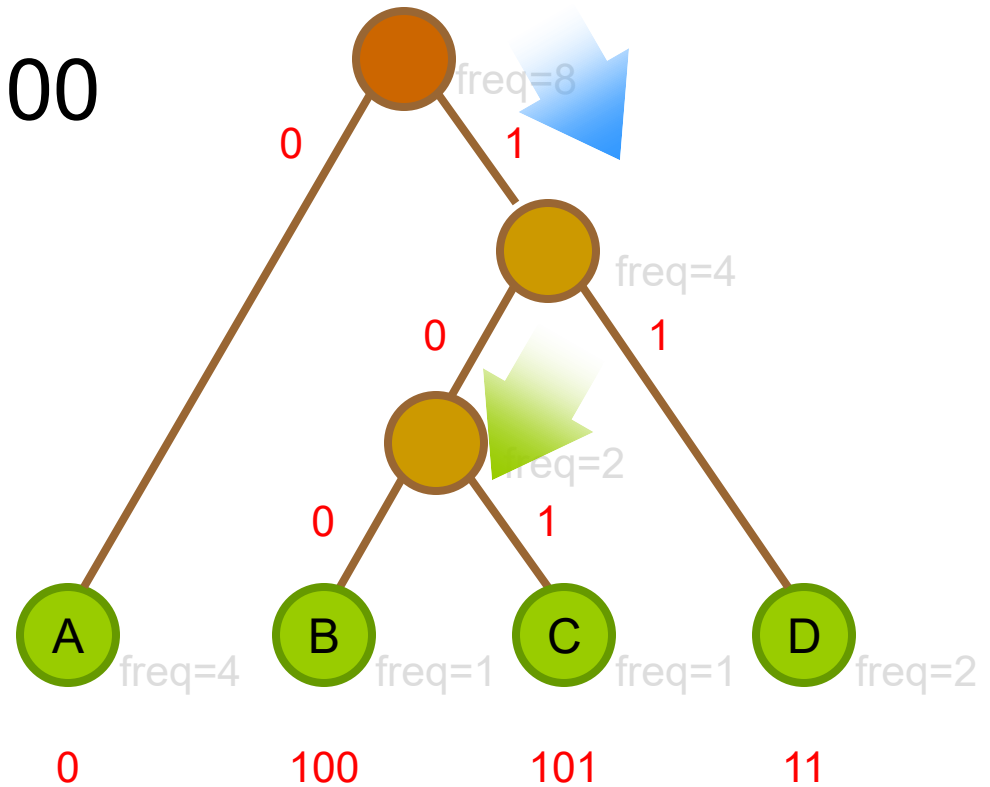
How to Decode This Message?

10111010001011100



Traverse The Tree Node by Node

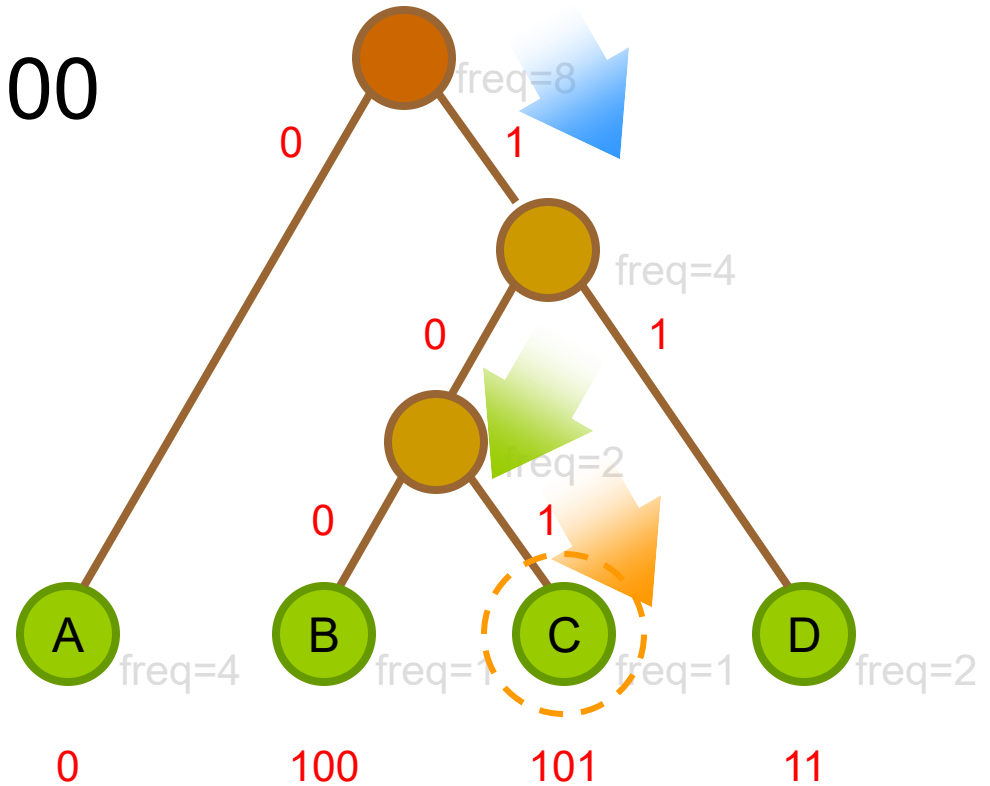
■ 10111010001011100



Until a Leaf Has Been Reached

■ 10111010001011100

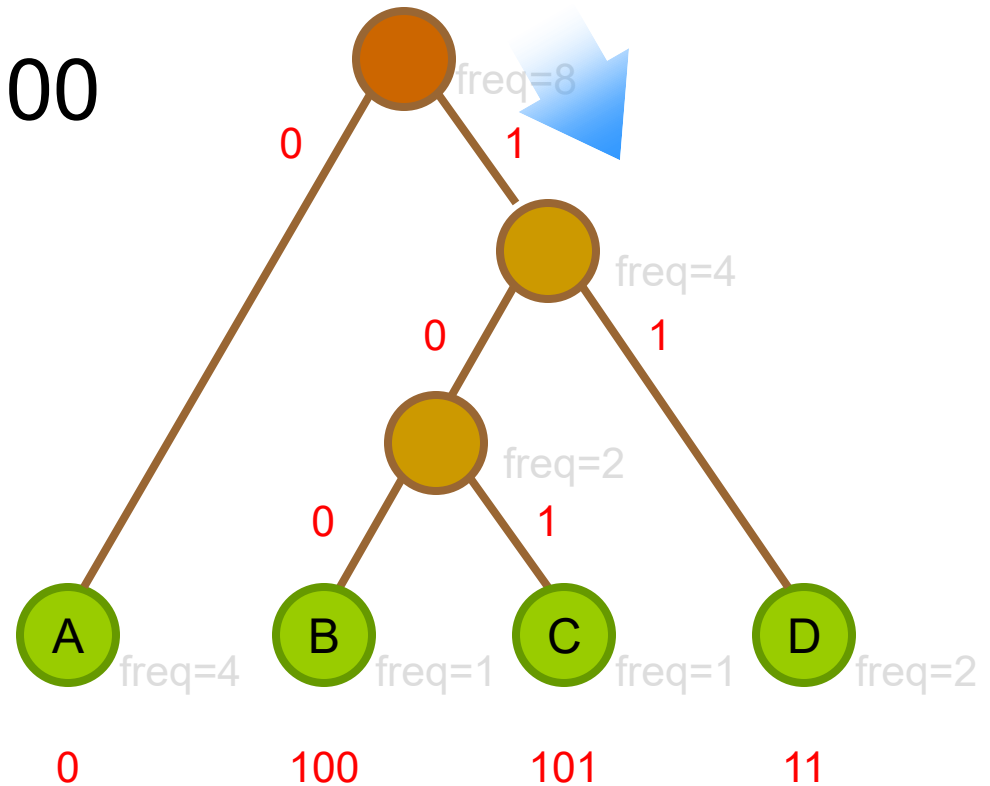
■ C



Restart From Root Again

■ 10111010001011100

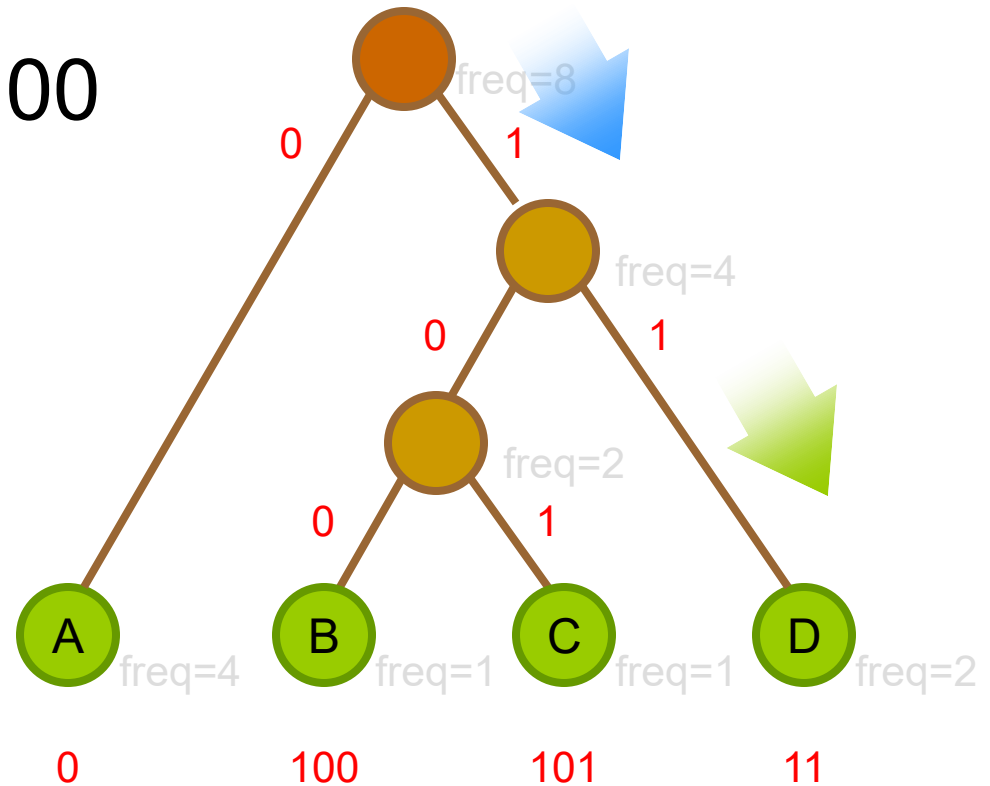
■ C



Reach Another Leaf Node

■ 10111010001011100

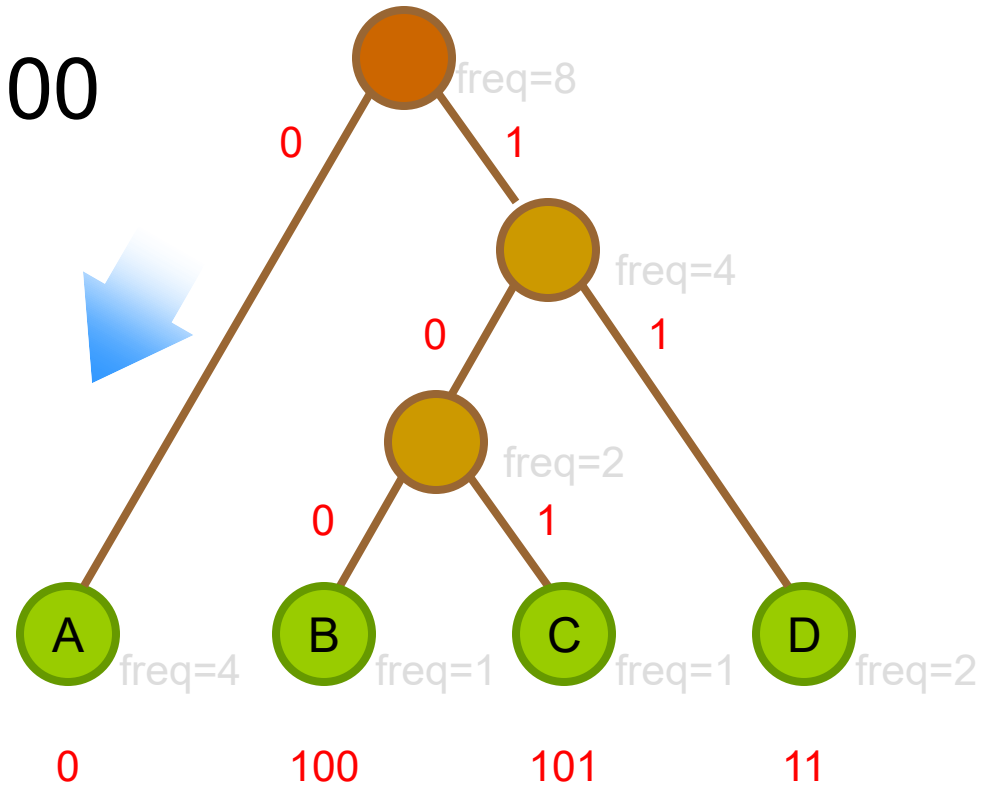
■ CD



Decode the Remaining by Similar Method

■ 10111010001011100

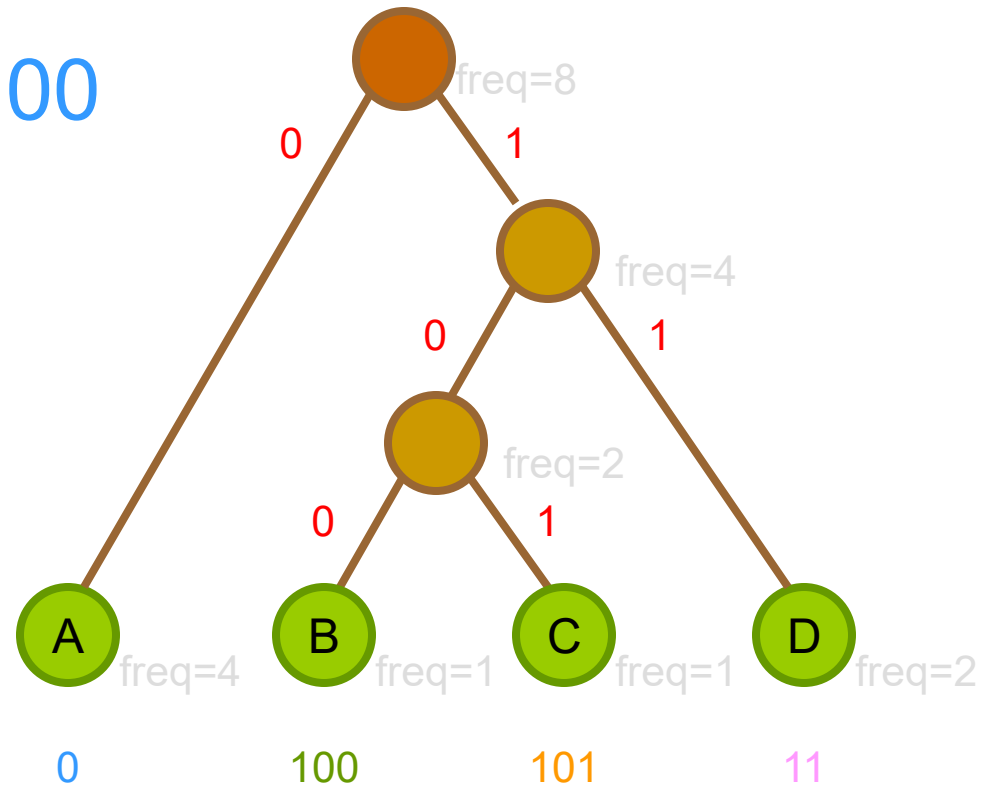
■ CDAA



Finally Obtain the Decoded Message

■ 10111010001011100

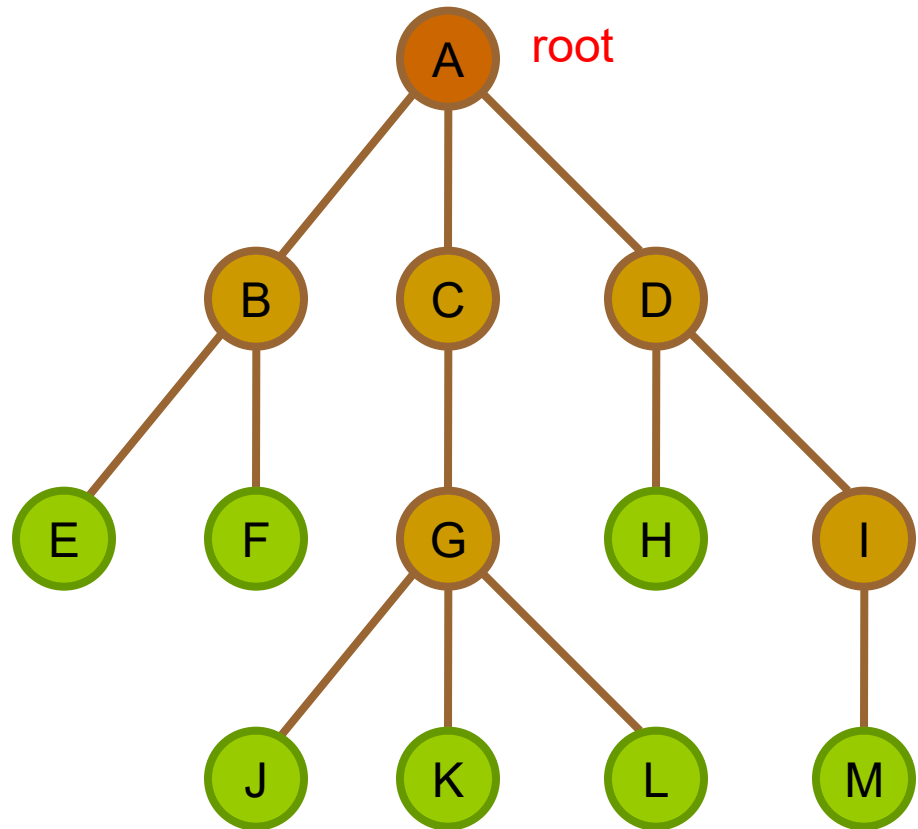
■ CDABACDAA



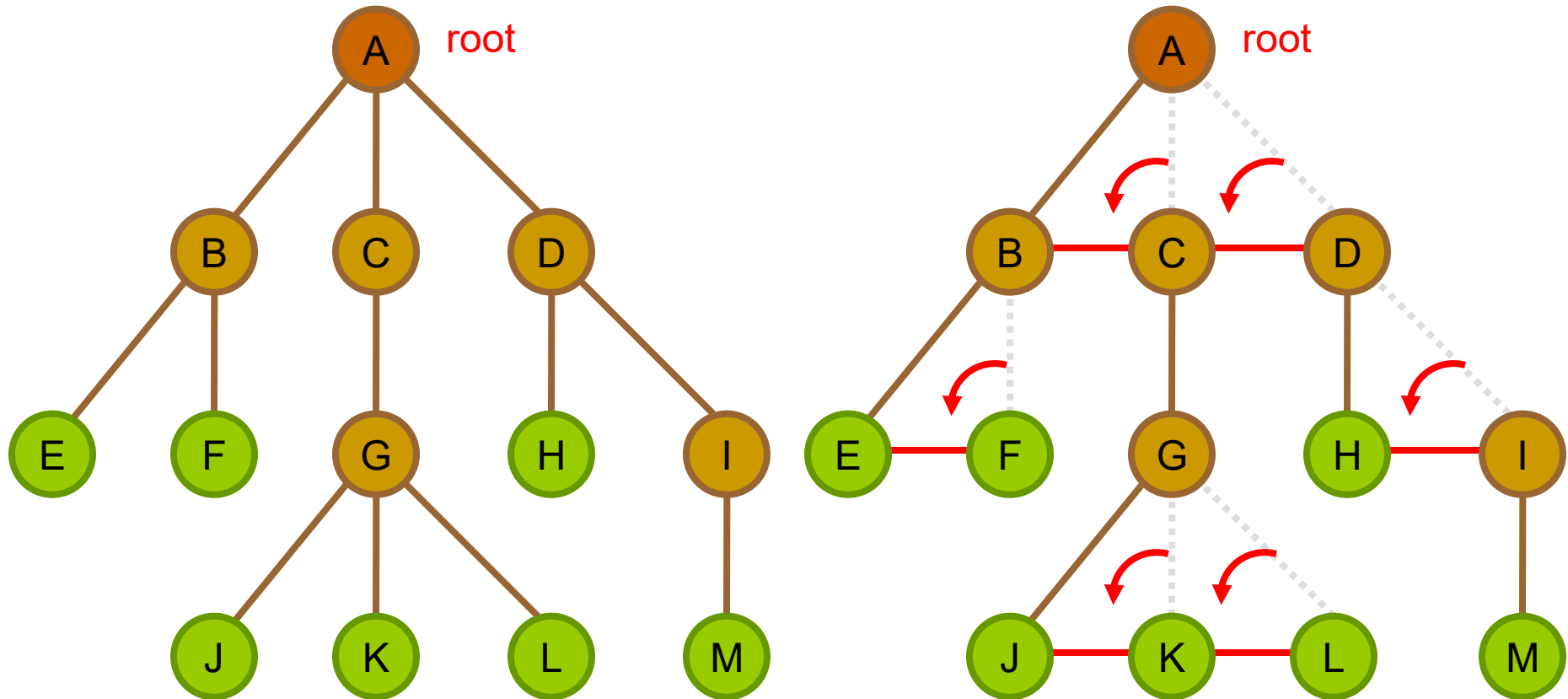
General Tree to Binary Tree Conversion

General tree

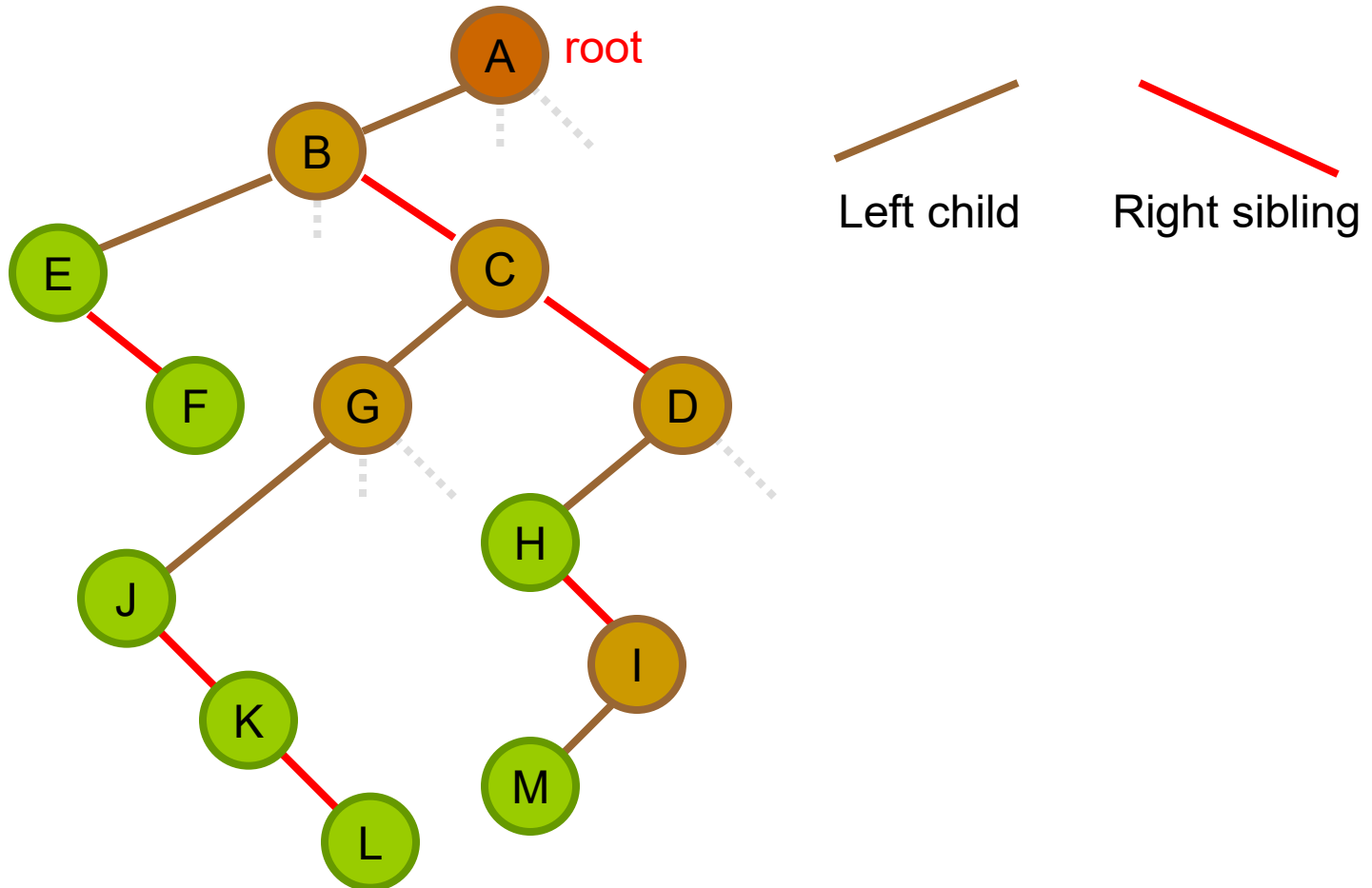
- We go back to the very beginning problem
- How to represent a general tree using binary tree?
 - Left Child-Right Sibling Representation



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 = 0;
    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;
}
```