

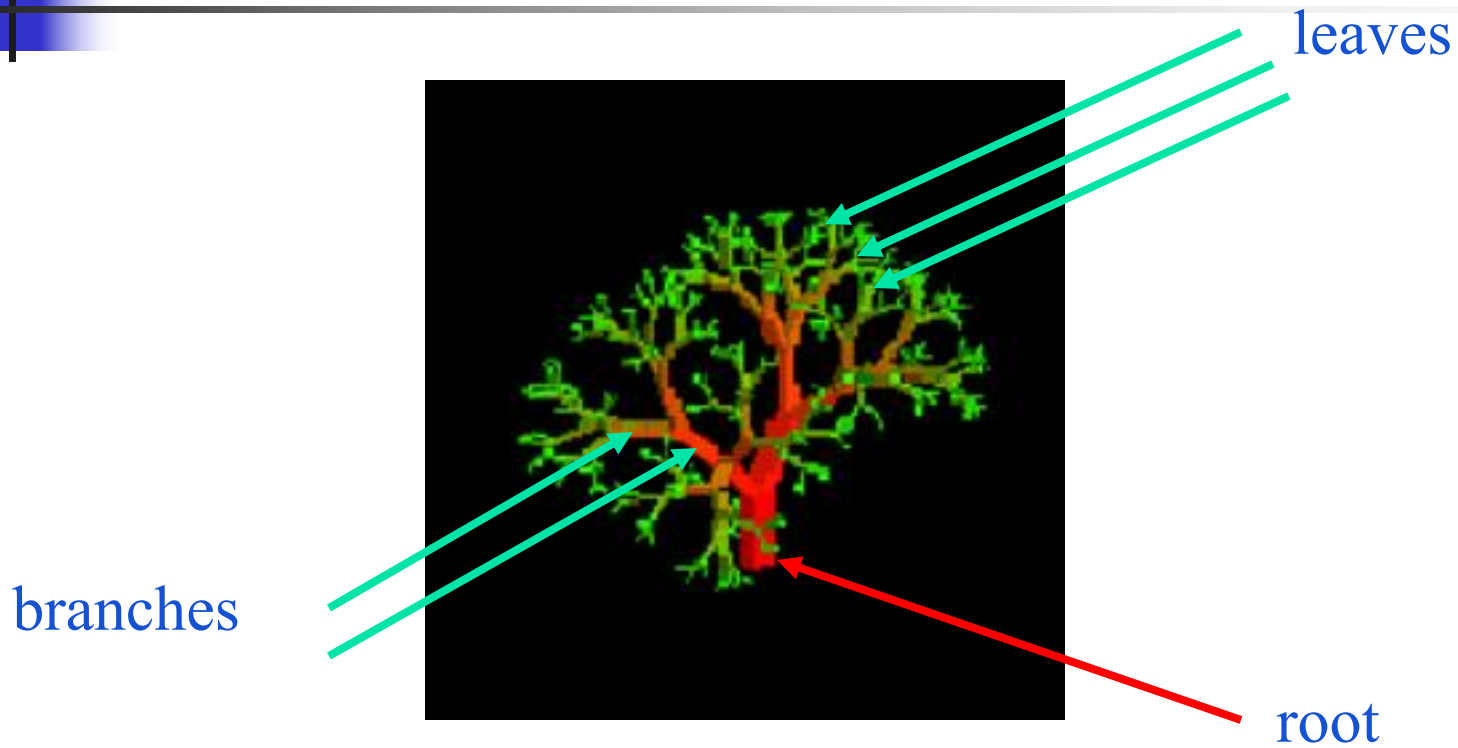
# **Tree And Its Applications**



---

**Alok Kumar Jagadev**

# Nature View of a Tree



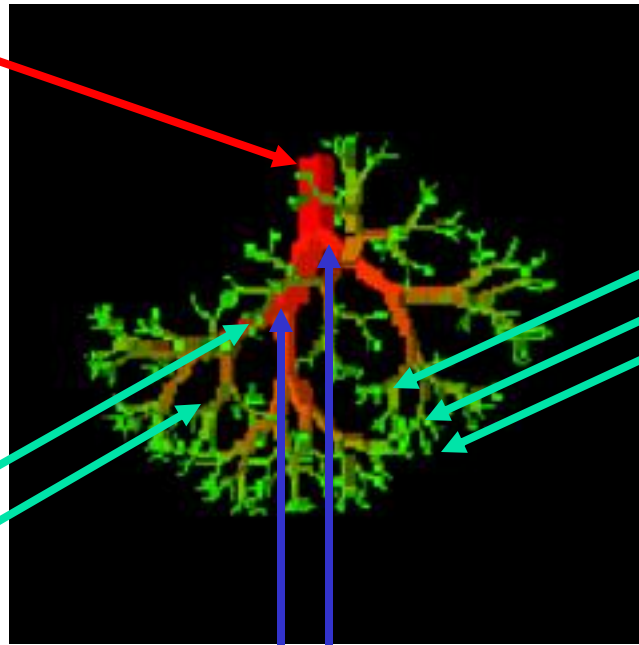
# Computer Scientist's View

root

branches

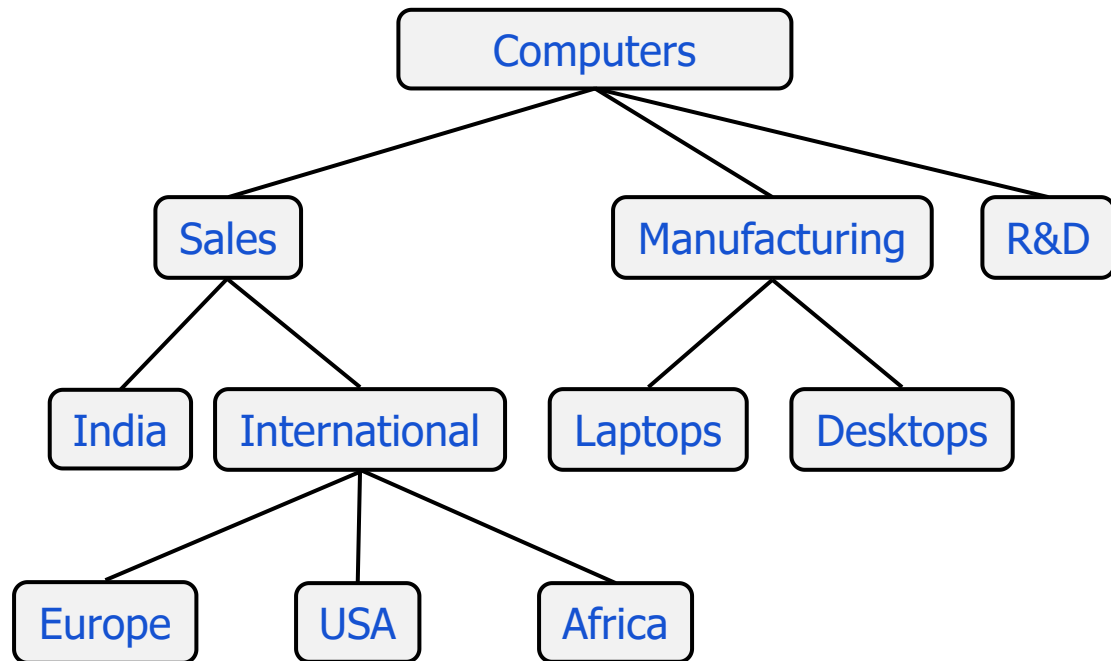
nodes

leaves



# What is a Tree

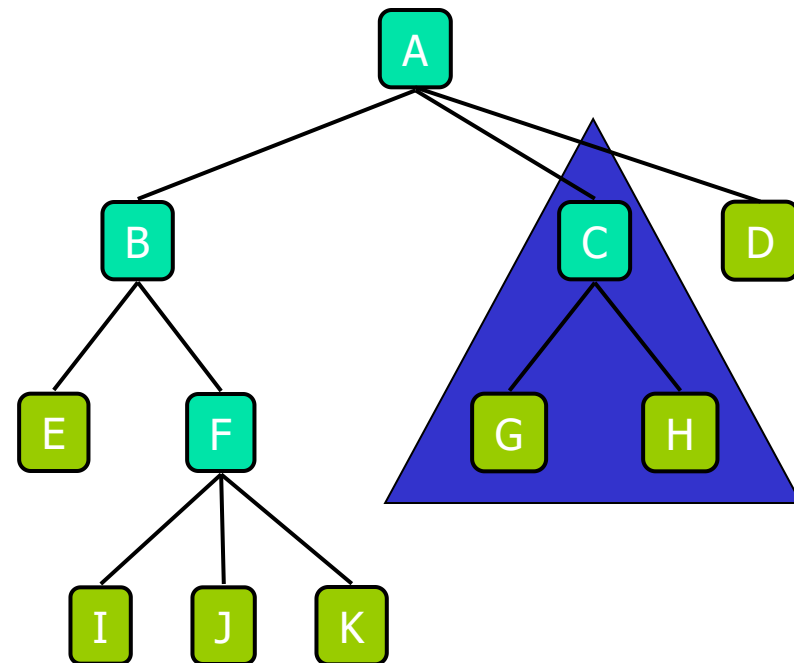
- A tree is a finite nonempty set of elements.
- An abstract model of a hierarchical structure.
  - consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems



# Tree Terminology

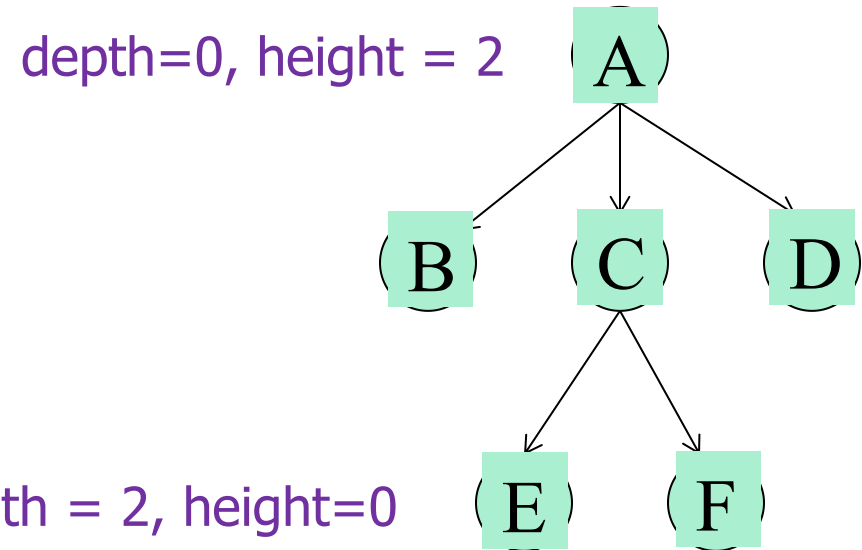
- **Root**: node without parent (A)
- **Siblings**: nodes having the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (**leaf/ terminal**): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Degree** of a node: the number of edges connected to the node
- **Degree** of a tree: highest degree of a node among all the nodes in the tree.

✚ **Subtree**: tree consisting of a node and its descendants



# Tree Terminology

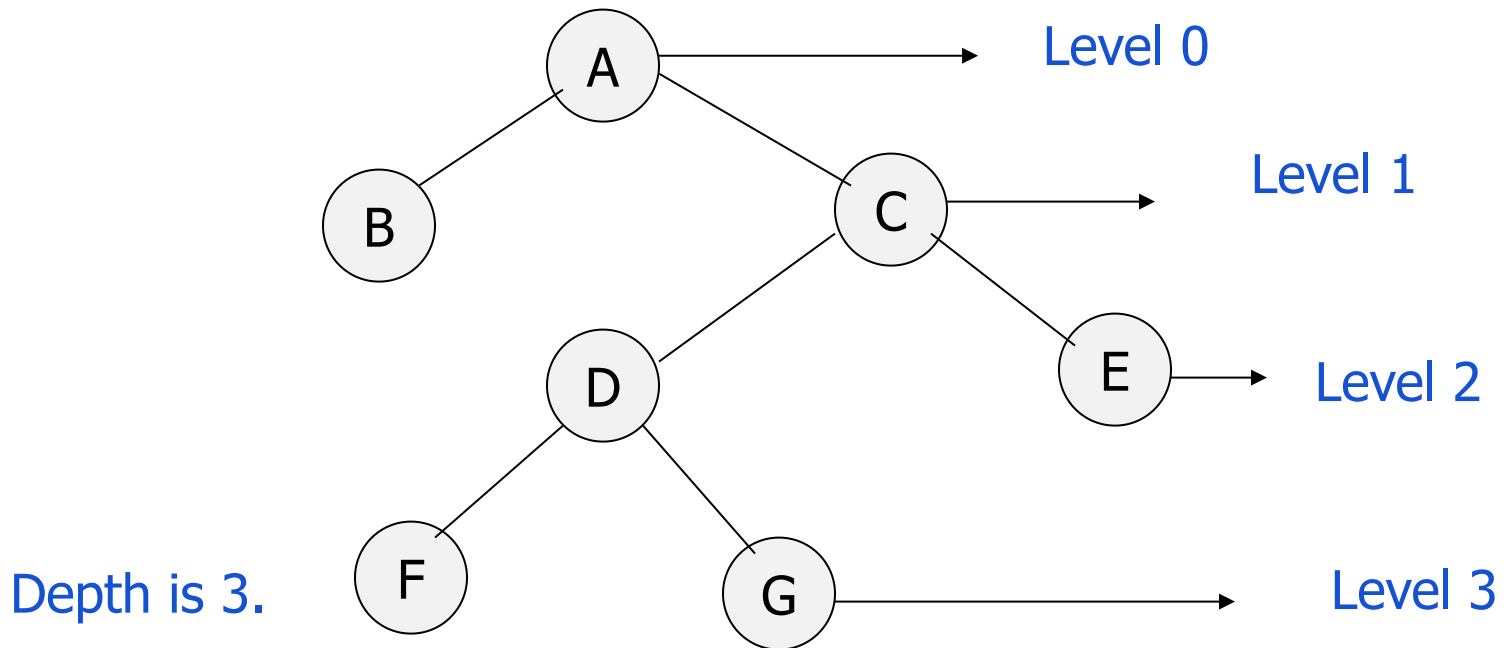
- Length of a path = number of edges
- Depth of a node N = length of path from root to N
- Height of node N = length of longest path from N to a leaf
- Depth and height of tree = height of root



# Level and depth of a binary Tree

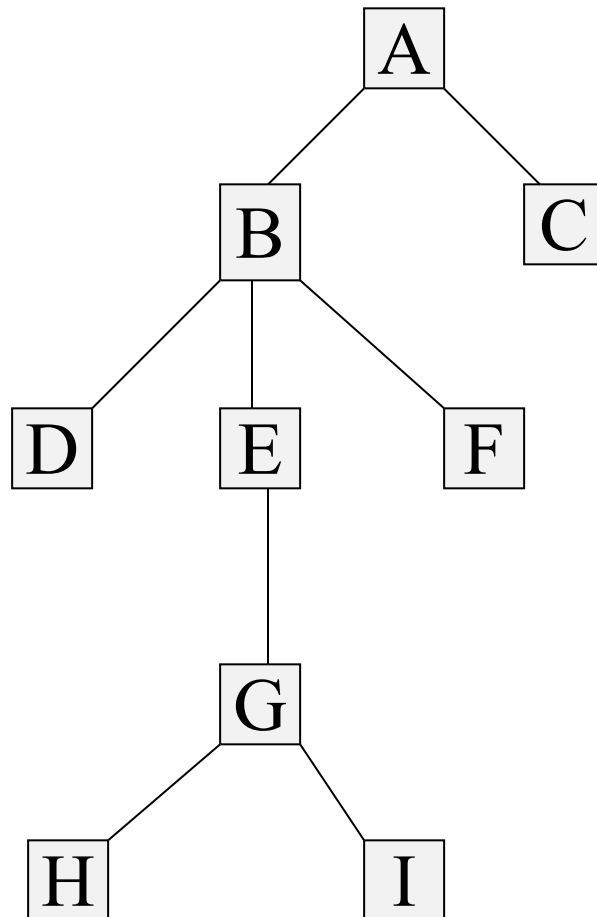
## Level of binary tree:

The root of the tree has level 0. And the level of any other node is one more than the level of its parent.





# Tree Properties



## Property

## Value

Number of nodes

9

Height

4

Root Node

A

Leaves

C, D, F, H, I

Interior nodes

B, E, G

Ancestors of H

A, B, E, G

Descendants of B

D, E, F, G, H, I

Siblings of E

D, F

Right subtree of A

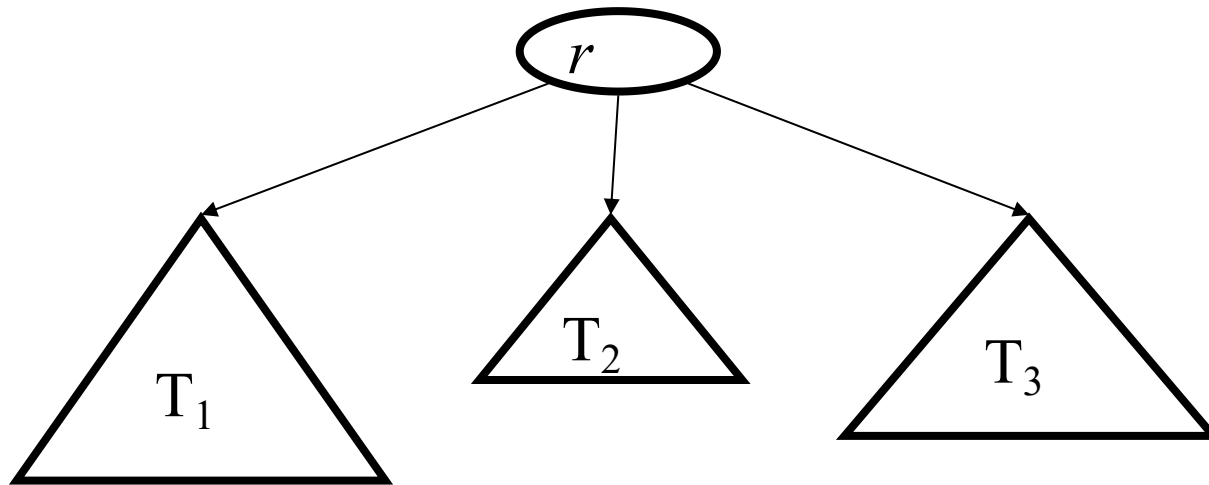
tree at C





# Recursive Definition of Tree

---



## Recursive Definition of a Tree:

A tree is a set of nodes that is

- a) an empty set of nodes, or
- b) has one node called the **root** from which zero or more **subtrees** connected.



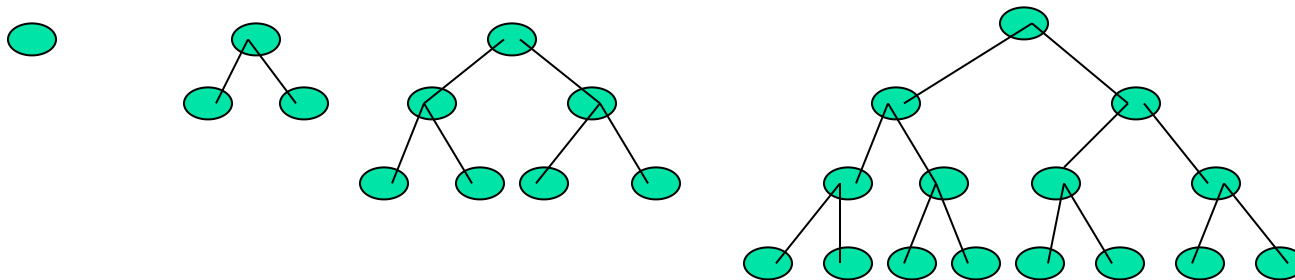
# Binary Tree: Definition

---

- A binary tree  $T$  is a finite set of nodes with one of the following properties:
  - $T$  is a tree if the set of nodes is empty. ([An empty tree is a tree](#))
  - The set consists of a root,  $R$ , and exactly two distinct binary trees, the [left subtree](#)  $T_L$  and the [right subtree](#)  $T_R$ .
    - The nodes in  $T$  consist of node  $R$  and all the nodes in  $T_L$  and  $T_R$ .
- A [binary tree](#) is a tree with the following properties:
  - Each internal node has at most two children ([degree of three](#))
- The children of an [internal node](#) are called [left child](#) and [right child](#)
- Applications:
  - arithmetic expressions
  - searching

# Other Kinds of Binary Trees (Full Binary Trees)

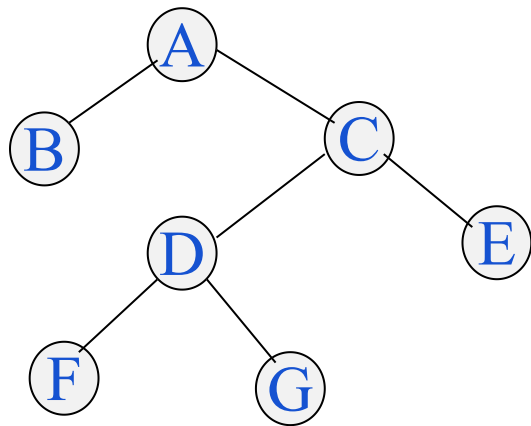
- Full Binary Tree: A binary tree  $T$  is full if each node is either a leaf or possesses exactly two children.
- The four full binary trees are:



# Other Kinds of Binary Trees (Strictly Binary Trees)

- If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree.
- A strictly binary tree with  $n$  leaves always contains  $2n - 1$  nodes.

In a strictly  $k$ -ary tree where every node has either 0 or  $k$  children  
 Sum of all degrees =  $2 \times (\text{\# of Edges})$



Sum of degrees of leaves +

Sum of degrees for Internal Node except root + Root's degree =  $2 \times (\text{\# of nodes} - 1)$  [ $\text{\# of Edges} = \text{\# of nodes} - 1$ ]

Putting values of above terms,

$$L + (I-1) \times (k+1) + k = 2 \times (L + I - 1) \quad (n \text{ nodes will } n-1 \text{ edges})$$

$$L + k \times I - k + I - 1 + k = 2 \times L + 2I - 2$$

$$L + k \times I + I - 1 = 2 \times L + 2 \times I - 2$$

$$k \times I + 1 - I = L$$

$$(k-1) \times I + 1 = L$$

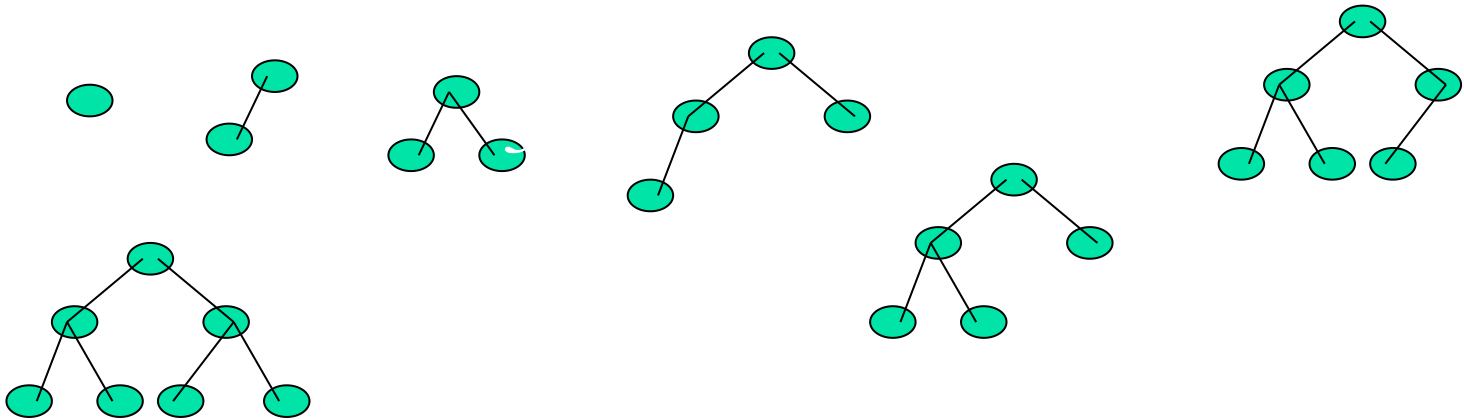
$$I = (L-1)/(k-1)$$



# Complete Binary Tree

---

Complete Binary Tree: A binary tree  $T$  with  $n$  levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.





# Binary Tree

---

The maximum number of nodes at level 'l' of a binary tree is  $2^l$ .



# Binary Tree

---

The maximum number of nodes at level 'l' of a binary tree  $2^l$ .

## Proof:

- Level of root is 0.
- This can be proved by [induction](#).
- For root,  $l = 0$ , number of nodes  $= 2^0 = 1$
- Assume that maximum number of nodes on level 'l' is  $2^l$
- Since in Binary tree every node has at most 2 children, next level ( i.e. in level  $l+1$ ) would have twice nodes, i.e.  $2 \times 2^l = 2^{l+1}$



# Binary Tree

---

Maximum number of nodes in a binary tree of height ' $h$ ' is  $2^{h+1} - 1$ .





# Binary Tree

---

Maximum number of nodes in a binary tree of height 'h' is  $2^{h+1} - 1$ .

## Proof:

- Here, height of a tree is maximum number of nodes on root to leaf path.  
So, Height of a tree with single node is considered as 0.
- A tree has maximum nodes if all levels have maximum nodes.
- So maximum number of nodes in a binary tree of height  $h$  is  $1 + 2 + 4 + \dots + 2^h$ .
  - the sum of this series is  $2^{h+1} - 1$ .



# Binary Tree

---

In a Binary Tree with  $n$  nodes, minimum possible height or minimum number of levels is?  $\log_2(n+1) - 1$  ?



# Binary Tree

---

In a Binary Tree with  $n$  nodes, minimum possible height or minimum number of levels is?  $\log_2(n+1) - 1$  ?

**Proof:**

$$\begin{aligned}\text{At height } h, \quad n &= 2^{h+1} - 1 \\ n+1 &= 2^{h+1} \\ 2^{h+1} &= n+1 \\ \log_2(2^{h+1}) &= \log_2(n+1) \\ h+1 &= \log_2(n+1) \\ h &= \log_2(n+1) - 1\end{aligned}$$

So, the *minimum possible height* becomes?  $\log_2(n+1) - 1$ .



# Binary Tree

---

A perfect binary tree of height  $h$  has  $2^h$  leaf nodes.



# Binary Tree

---

A perfect binary tree of height  $h$  has  $2^h$  leaf nodes.

## Proof:

Use induction on the height. When  $h = 0$ , there is  $2^0 = 1$  node and that node is a leaf node.

Assume a perfect tree of height  $h$  has  $2^h$  leaf nodes, then a perfect tree of height  $h+1$  has two subtrees, both of which are of height  $h$  and both of which have  $2^h$  leaf nodes.

Therefore, the perfect tree of height  $h + 1$  must have  $2 \cdot 2^h = 2^{h+1}$  leaf nodes.

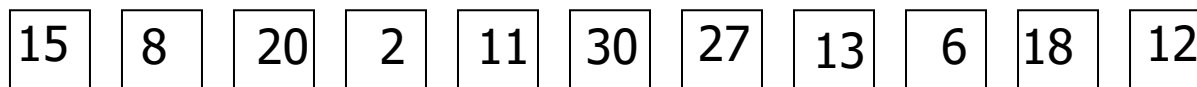
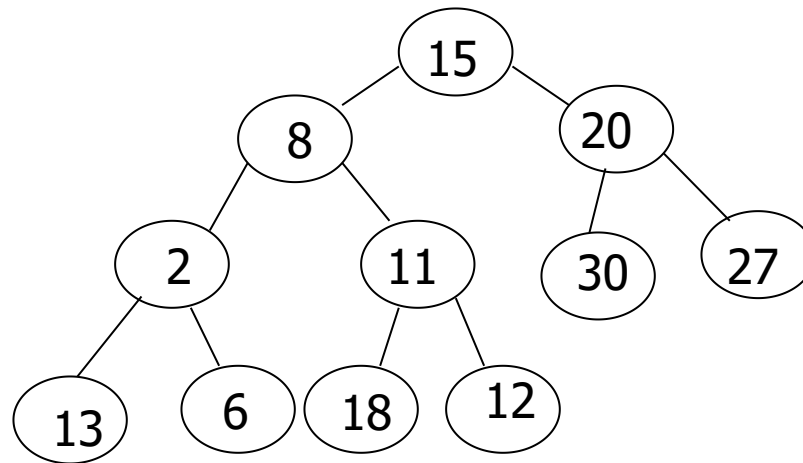


# Array Representation Trees

---

- A complete binary tree can be represented by an array  $T$  of the same length as the number of nodes
- $T[k]$  is identified with node
- That is,  $T[k]$  holds the data of node  $k$
- Advantage:
  - no need to store left and right pointers in the nodes → save memory
  - Direct access to nodes: to get to node  $k$ , access  $T[k]$

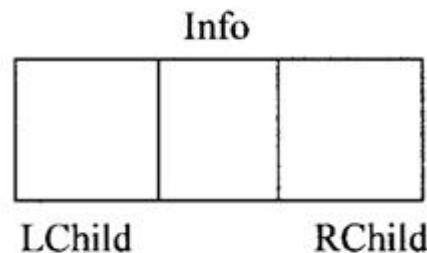
# Illustration of Array Representation



- **Notice:** Left child of  $T[4]$  (of data 11) is  $T[2*4+1]=T[9]$  (of data 18), and its right child is  $T[2*4+2]=T[10]$  (of data 12).
- Parent of  $T[3]$  is  $T[(3-1)/2]=T[1]$ , and parent of  $T[4]=T[(4-1)/2]=T[1]$

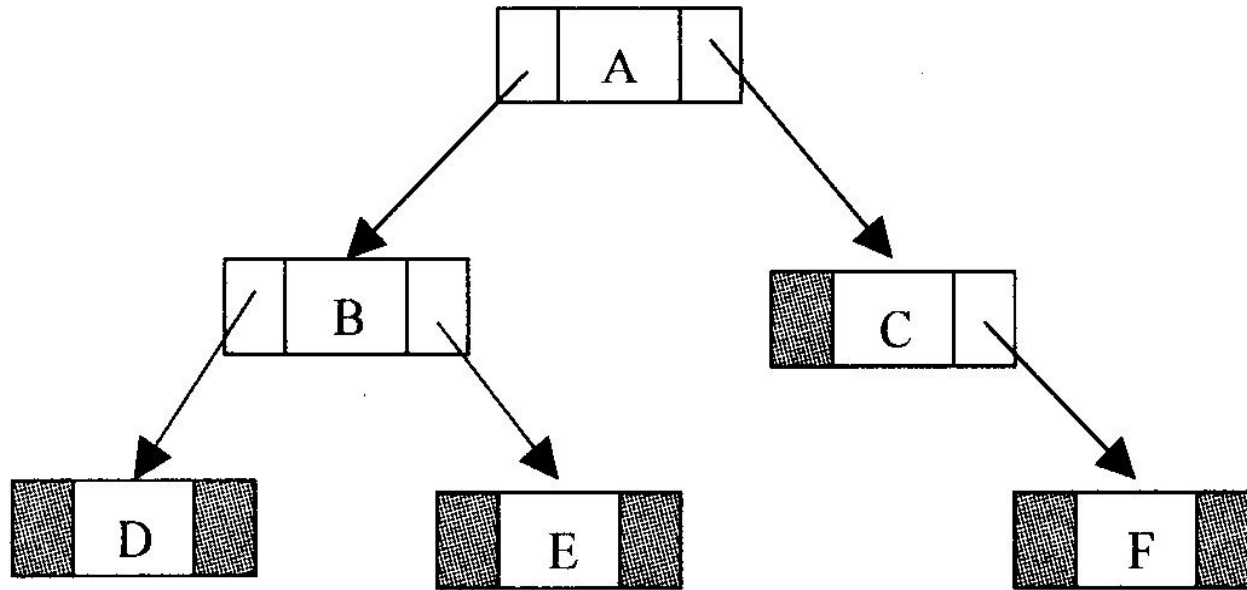
# Linked List Representation of Binary Tree

- The most popular and practical way of representing a binary tree is using **linked list** (or **pointers**). In linked list, every element is represented as nodes. A node consists of three fields such as :
  - (a) Left Child (LChild)
  - (b) Information of the Node (Info)
  - (c) Right Child (RChild)
- The LChild links points to the left child of the node, Info holds the information of the node and the RChild holds the address of right child node of the node. The structure of a binary tree node is shown below:





# Binary Tree Representations (Linked List)



If a node has no left or / and right node, corresponding LChild or RChild is assigned to **NULL**



# Binary Tree Traversal

---

- **Traversal** is the process of visiting every node once
- Visiting a node entails doing some processing at that node,
  - but when describing a **traversal strategy**, it is not required to concern with what that processing is.



# Binary Tree Traversal Techniques

---

Three recursive techniques for binary tree traversal

- In each technique,
  - the left subtree is traversed recursively,
  - the right subtree is traversed recursively, and
  - the root is visited.
- What distinguishes the techniques from one another is the order of those 3 tasks.



# Preoder, Inorder, Postorder

---

- In **Preorder**, the root is visited before traversals of both the subtrees.
- In **Inorder**, the root is visited in-between left and right subtree traversals.
- In **Postorder**, the root is visited after the traversals of both subtrees.

## Preorder Traversal:

1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

## Inorder Traversal:

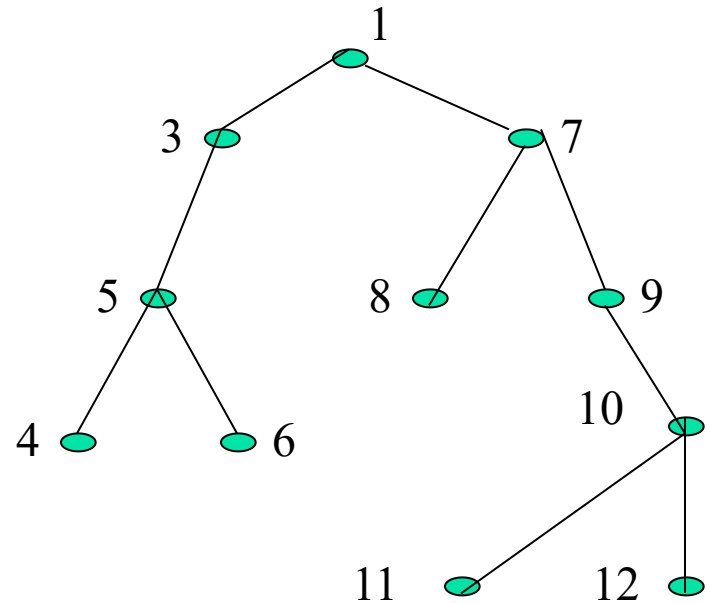
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

## Postorder Traversal:

1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

# Illustrations for Traversals

- Assume: visiting a node is printing its node number
- Preorder: 1 3 5 4 6 7 8 9 10 11 12
- Inorder: 4 5 6 3 1 8 7 9 11 10 12
- Postorder: 4 6 5 3 8 11 12 10 9 7 1

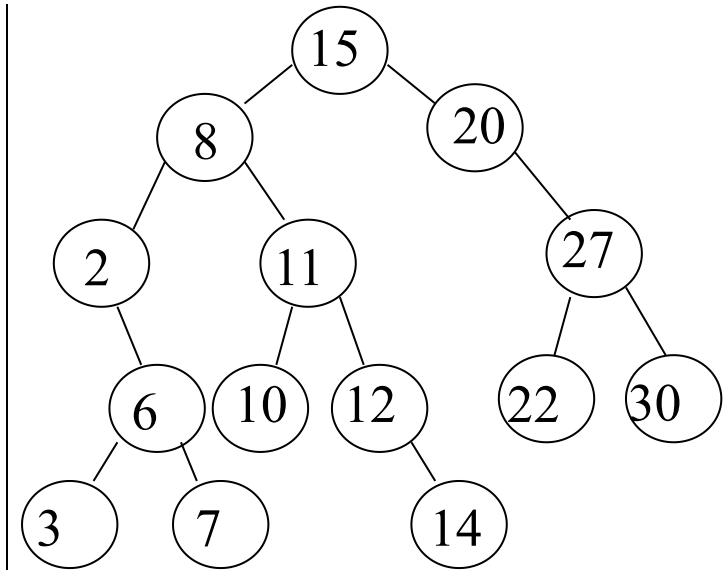




# Illustrations for Traversals (Contd.)

**Assume:** visiting a node is printing its data

- Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15





# Code for the Traversal Techniques

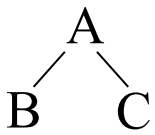
- The code for visit depends on the application
- A typical example for visit(...) is to print out the data part of its input node

```
typedef struct node {  
    int info;  
    struct node *left;  
    struct node *right;  
} Tree;
```

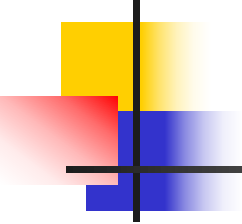
```
void preOrder(Tree *root){  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root->right);  
}
```

```
void inOrder(Tree *root){  
    if (root == NULL) return;  
    inOrder(root->left);  
    visit(root->info);  
    inOrder(root->right);  
}
```

```
void postOrder(Tree *root){  
    if (root == NULL) return;  
    postOrder(root->left);  
    postOrder(root->right);  
    visit(root->info);  
}
```



# Code for the Traversal Techniques



```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```

```
void preOrder(Tree *root) {  
    if (root == NULL) return;  
    visit(root->info);  
    preOrder(root->left);  
    preOrder(root-> right);  
}
```





# Nonrecursive Preorder Traversal: General Algorithm

---

```
curr = root; //traversal starts at root node
push(curr)
while(stack is nonempty)
    curr=pop()
    print(curr->info)
    if(curr->right != NULL)
        push(curr->right)
    if(curr->left != NULL)
        curr = curr->left
```



# Nonrecursive Inorder Traversal: General Algorithm

---

```
curr = root; //start traversing the binary tree at the root node
while(curr ≠ NULL or stack is nonempty)
    if(curr ≠ NULL)
        push(curr)
        curr = curr->left
    else
        curr = pop()
        print(curr->info)//visit the node
        curr = curr->right
        //move to the right child
```



# Nonrecursive Postorder Traversal: General Algorithm

---

```
curr = root; //start traversal at root node
while(curr!=NULL or stack is not empty)
    while (curr != NULL)
        if(curr->right != NULL)
            push(curr->right)
        push(curr)
        curr = curr->left
    curr=pop()
    if(curr->right != NULL and top() == curr->right)
        pop()
        push(curr)
        curr = curr->right
    else
        print(curr->info)
        curr = NULL
```



# Binary Tree Construction

---

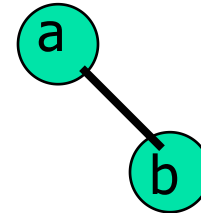
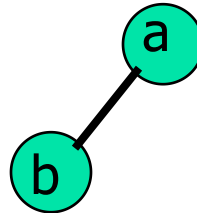
- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
  - When a traversal sequence has more than one element, the binary tree is not uniquely defined.
  - Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.



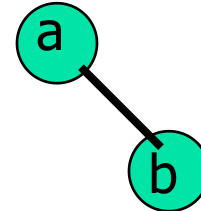
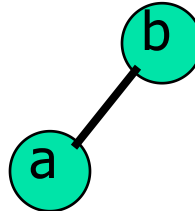
# Some Examples

---

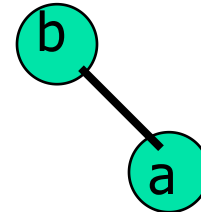
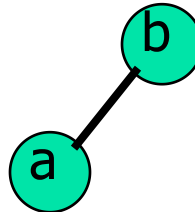
preorder = **ab**



inorder = **ab**



postorder = **ab**





# Binary Tree Construction

---

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

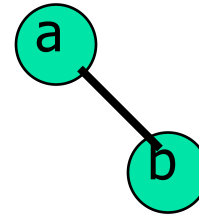
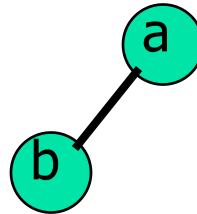


# Preorder And Postorder

---

preorder = **ab**

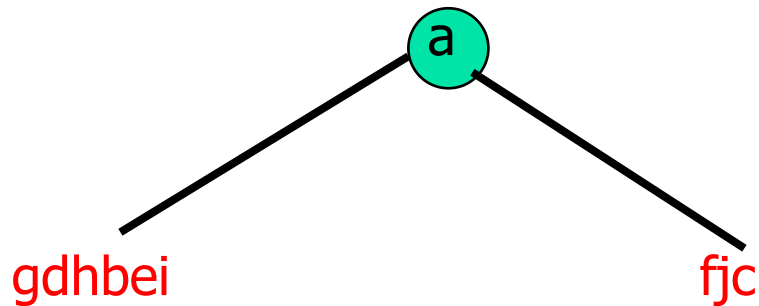
postorder = **ba**



- Preorder and postorder do not uniquely define a binary tree.

# Inorder And Preorder

- inorder: **g d h b e i a f j c**
- preorder: **a b d g h e i c f j**
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- **a** is the root of the tree: **gdhbei** are nodes in the left subtree: **fjc** are nodes in the right subtree.

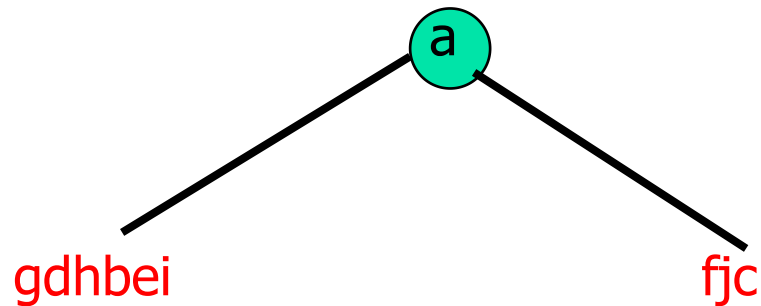




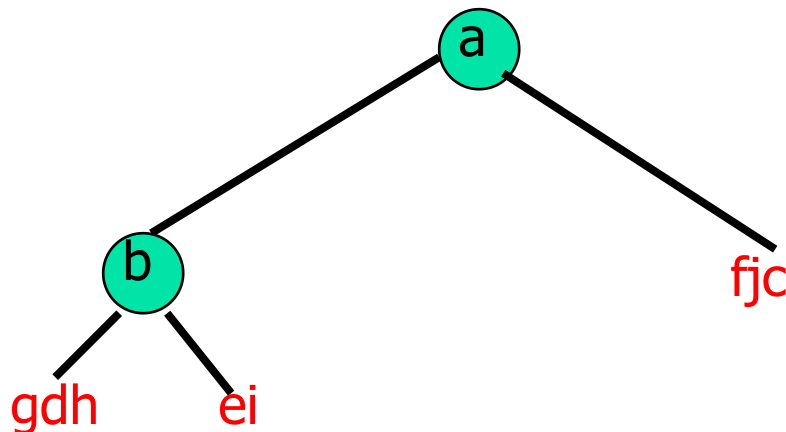


# Inorder And Preorder

---



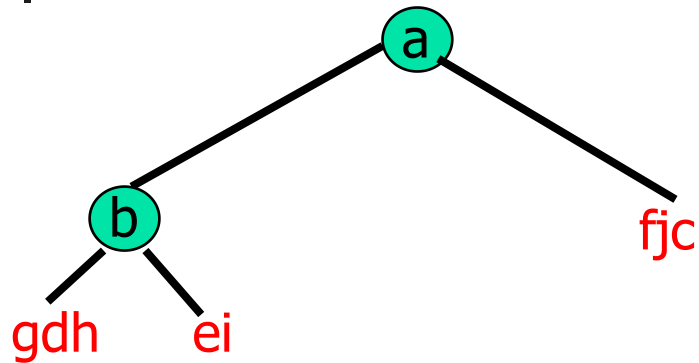
- preorder: **b** d g h e i c f j
- **b** is the next root; **gdh** are in the left subtree; **ei** are in the right subtree.



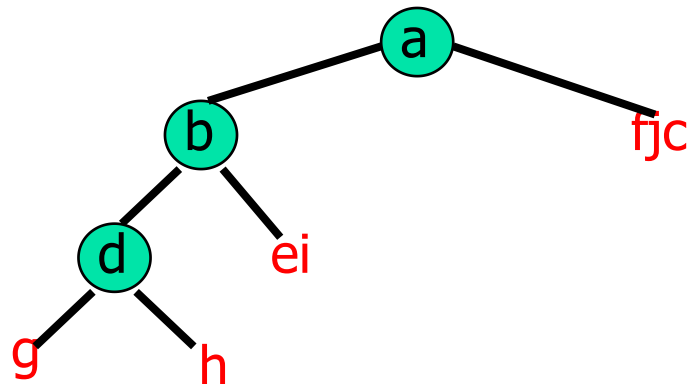


# Inorder And Preorder

---



- preorder: **d g h e i c f j**
- **d** is the next root; **g** is in the left subtree; **h** is in the right subtree.





# Inorder And Postorder

---

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder: **g d h b e i a f j c**
- postorder: **g h d i e b j f c a**
- Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.



# Construction of No. of Trees

---

Catalan Number:  $T(n) = (2^n)! / [(n+1)! n!]$

- # of Unlabeled Binary Tree:  $T(n) = (2^n)! / [(n+1)! n!]$
- # of Labelled Binary Tree: Every unlabeled tree with  $n$  nodes can create  $n!$  labeled trees by assigning different permutations of labels to all nodes. So,  
 $T(n) = [ (2^n)! / (n+1)! * n! ] * n!$



# Count Leaf nodes in a Binary Trees

---

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

```
/* Function to count of leaf nodes in a binary tree*/  
unsigned int leafCount(struct node* root) {  
    if(root == NULL) return 0;  
    if(root->left == NULL && root->right == NULL)  
        return 1;  
    else  
        return leafCount(root->left)+ leafCount(root->right);  
}
```



# Count non Leaf nodes in a Binary Trees

---

```
/* Computes the number of non-leaf nodes in a tree. */
int countNonleaf(struct node* root) {
    // Base cases.
    if (root == NULL || (root->left == NULL && root->right == NULL))
        return 0;
    // If root is Not NULL and its one of its child is also not NULL
    return 1 + countNonleaf(root->left) + countNonleaf(root->right);
}
```



# Count total nodes in a Binary Trees

---

```
unsigned int countNodes(struct node *root) {  
    if (root == NULL)  
        return 0;  
    else  
        return 1+ countNodes(root->left) + countNodes(root->right);  
}
```



## MCQ-1

---

Which of the following is a true about Binary Trees

- (A) Every binary tree is either complete or full.
- (B) Every complete binary tree is also a full binary tree.
- (C) Every full binary tree is also a complete binary tree.
- (D) No binary tree is both complete and full.
- (E) None of the above





## MCQ-1

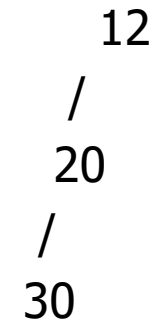
Which of the following is a true about Binary Trees

- (A) Every binary tree is either complete or full.
- (B) Every complete binary tree is also a full binary tree.
- (C) Every full binary tree is also a complete binary tree.
- (D) No binary tree is both complete and full.
- (E) None of the above

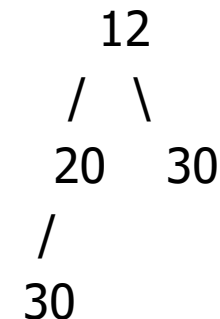
Answer: (E)

- Explanation: A full binary tree (sometimes **proper binary tree** or **2-tree** or **strictly binary tree**) is a tree in which every node other than the leaves has two children.
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A) is incorrect. For example, the following Binary tree is neither complete nor full



B) is incorrect. The following binary tree is complete but not full





## MCQ-1

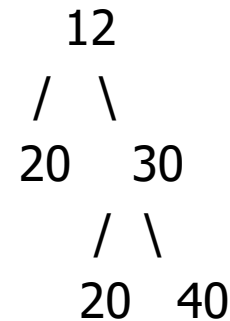
---

Which of the following is a true about Binary Trees

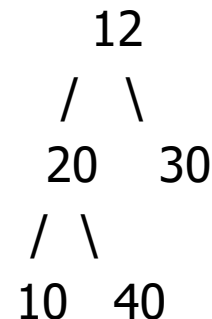
- (A) Every binary tree is either complete or full.
- (B) Every complete binary tree is also a full binary tree.
- (C) Every full binary tree is also a complete binary tree.
- (D) No binary tree is both complete and full.
- (E) None of the above

Answer: (E)

C) is incorrect. Following Binary tree is full, but not complete



D) is incorrect. Following Binary tree is both complete and full





## MCQ-2

---

Suppose the level of root is 1 and levels of left and right children of root is 2.  
The maximum number of nodes on level  $i$  of a binary tree is

- (A)  $2^{i-1}$
- (B)  $2^i$
- (C)  $2^{i+1}$
- (D)  $2^{(i+1)/2}$



## MCQ-2

---

Suppose the level of root is 1 and levels of left and right children of root is 2.  
The maximum number of nodes on level  $i$  of a binary tree is

- (A)  $2^{i-1}$
- (B)  $2^i$
- (C)  $2^{i+1}$
- (D)  $2^{(i+1)/2}$

Answer: (A)

**Explanation:** Number of nodes of binary tree will be maximum only when tree is full complete, therefore answer is  $2^{i-1}$

So, option (A) is true.



## MCQ-3

---

In a complete k-ary tree, every internal node has exactly k children or no child. What is the number of leaf nodes in such a tree with n internal nodes?

- (A)  $nk$
- (B)  $(n - 1)k + 1$
- (C)  $n(k - 1) + 1$
- (D)  $n(k - 1)$

## MCQ-3

In a complete k-ary tree, every internal node has exactly k children or no child. What is the number of leaf nodes in such a tree with n internal nodes?

- (A)  $nk$
- (B)  $(n - 1)k + 1$
- (C)  $n(k - 1) + 1$
- (D)  $n(k - 1)$

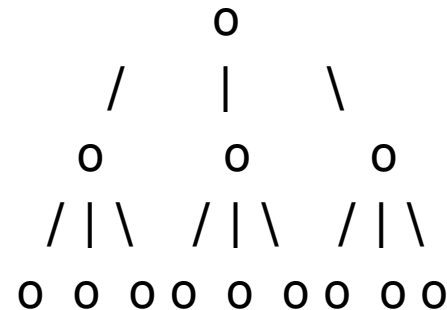
Answer: (C)

Explanation: For an k-ary tree where each node has k children or no child, following relation holds

$$L = (k-1)*n + 1$$

where L is the number of leaf nodes and n is the number of internal nodes. since it is a complete k tree, so every internal node will have k children.

**Example:**



$$k = 3$$

Number of internal nodes  $n = 4$

$$\begin{aligned}\text{Number of leaf nodes} &= (k-1)*n + 1 \\ &= (3-1)*4 + 1 = 9\end{aligned}$$



## MCQ-4

---

The maximum number of binary trees that can be formed with three unlabeled nodes is:

- (A) 1
- (B) 5
- (C) 4
- (D) 3

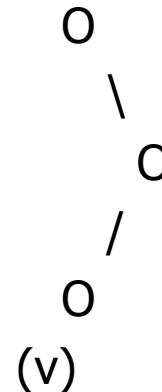
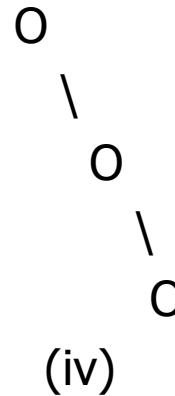
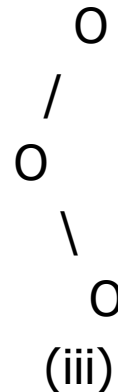
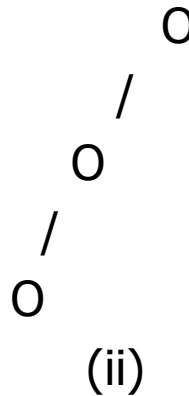
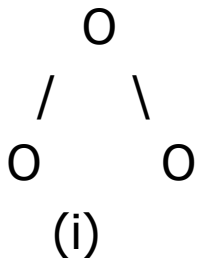
## MCQ-4

The maximum number of binary trees that can be formed with three unlabeled nodes is:

- (A) 1
- (B) 5
- (C) 4
- (D) 3

Answer: (B)

**Explanation:** Following are all possible unlabeled binary tree



**Note** that nodes are unlabeled. If the nodes are labeled, we get more number of trees.

We can find the number of binary tree by Catalan number number:

Here  $n = 3$

$$\begin{aligned}\text{Number of binary tree} &= \frac{(2^n C_n)}{(n+1)} \\ &= \frac{(2^3 C_3)}{(3+1)} \\ &= 5.\end{aligned}$$

So, option (B) is correct.

The first few **Catalan numbers** for  $n = 0, 1, 2, 3, \dots$  are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...





## MCQ-5

---

The number of leaf nodes in a rooted tree of  $n$  nodes, with each node having 0 or 3 children is:

- (A)  $n/2$
- (B)  $(n-1)/3$
- (C)  $(n-1)/2$
- (D)  $(2n+1)/3$



## MCQ-5

---

What is the number of leaf nodes in a rooted tree of  $n$  nodes, with each node having 0 or 3 children?

- (A)  $n/2$
- (B)  $(n-1)/3$
- (C)  $(n-1)/2$
- (D)  $(2n+1)/3$

Answer: (D)

**Explanation:** Let  $L$  be the number of leaf nodes and  $I$  be the number of internal nodes, then following relation holds for above given tree

$$L = (3-1)I + 1 = 2I + 1$$

Total number of nodes( $n$ ) is sum of leaf nodes and internal nodes

$$n = L + I \Rightarrow I = n - L \quad \text{Putting the value of } I \text{ in the above equation}$$

We get  $L = (2n+1)/3$



## MCQ-6

---

A complete  $n$ -ary tree is a tree in which each node has  $n$  children or no children. Let  $I$  be the number of internal nodes and  $L$  be the number of leaves in a complete  $n$ -ary tree. If  $L = 41$ , and  $I = 10$ , what is the value of  $n$ ?

- (A) 6
- (B) 3
- (C) 4
- (D) 5



## MCQ-6

---

A complete n-ary tree is a tree in which each node has n children or no children. Let I be the number of internal nodes and L be the number of leaves in a complete n-ary tree. If  $L = 41$ , and  $I = 10$ , what is the value of n?

- (A) 6
- (B) 3
- (C) 4
- (D) 5

Answer: (D)

**Explanation:** For an n-ary tree where each node has n children or no children, following relation holds

$$L = (n-1)*I + 1$$

Where L is the number of leaf nodes and I is the number of internal nodes.

Let us find out the value of n for the given data.

$$L = 41, I = 10$$

$$41 = 10*(n-1) + 1$$

$$(n-1) = 4$$

$$n = 5$$



## MCQ-7

---

A scheme for storing binary trees in an array  $X$  is as follows. Indexing of  $X$  starts at 1 instead of 0. the root is stored at  $X[1]$ . For a node stored at  $X[i]$ , the left child, if any, is stored in  $X[2i]$  and the right child, if any, in  $X[2i+1]$ . To be able to store any binary tree on  $n$  nodes the minimum size of  $X$  should be.

- (A)  $\log_2 n$
- (B)  $n$
- (C)  $2n + 1$
- (D)  $2^n - 1$

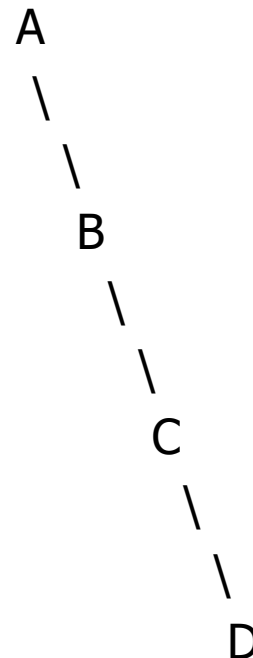
## MCQ-7

A scheme for storing binary trees in an array  $X$  is as follows. Indexing of  $X$  starts at 1 instead of 0. the root is stored at  $X[1]$ . For a node stored at  $X[i]$ , the left child, if any, is stored in  $X[2i]$  and the right child, if any, in  $X[2i+1]$ . To be able to store any binary tree on  $n$  nodes the minimum size of  $X$  should be.

- (A)  $\log_2 n$
- (B)  $n$
- (C)  $2n + 1$
- (D)  $2^n - 1$

Answer: (D)

**Explanation:** For a right skewed binary tree, number of nodes will be  $2^n - 1$ . For example, in below binary tree, node 'A' will be stored at index 1, 'B' at index 3, 'C' at index 7 and 'D' at index 15.





## MCQ-8

---

Consider the following nested representation of binary trees: (X Y Z) indicates Y and Z are the left and right sub stress, respectively, of node X. Note that Y and Z may be NULL, or further nested. Which of the following represents a valid binary tree?

- (A) (1 2 (4 5 6 7))
- (B) (1 (2 3 4) 5 6) 7)
- (C) (1 (2 3 4)(5 6 7))
- (D) (1 (2 3 NULL) (4 5))



## MCQ-8

Consider the following nested representation of binary trees: (X Y Z) indicates Y and Z are the left and right sub stress, respectively, of node X.

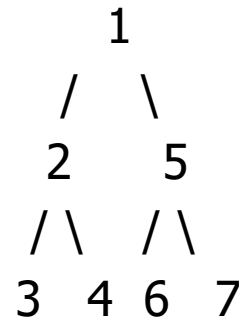
Note that Y and Z may be NULL, or further nested. Which of the following represents a valid binary tree?

- (A) (1 2 (4 5 6 7))
- (B) (1 (2 3 4) 5 6) 7)
- (C) (1 (2 3 4)(5 6 7))
- (D) (1 (2 3 NULL) (4 5))

Answer: (C)

**Explanation:** C is fine.

(1 (2 3 4)(5 6 7)) represents following binary tree



- A) (1 2 (4 5 6 7)) is not fine as there are 4 elements in one bracket.
- B) (1 (2 3 4) 5 6) 7) is not fine as there are 2 opening brackets and 3 closing.
- D) (1 (2 3 NULL) (4 5)) is not fine one bracket has only two entries (4 5)





## MCQ-9

---

Consider a node X in a Binary Tree. Given that X has two children, let Y be Inorder successor of X. Which of the following is true about Y?

- (A) Y has no right child
- (B) Y has no left child
- (C) Y has both children
- (D) None of the above



## MCQ-9

---

Consider a node X in a Binary Tree. Given that X has two children, let Y be Inorder successor of X. Which of the following is true about Y?

- (A) Y has no right child
- (B) Y has no left child
- (C) Y has both children
- (D) None of the above

Answer: (B)

**Explanation:** Since X has both children, Y must be leftmost node in right child of X.



## MCQ-10

---

The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are

- (A) 63 and 6, respectively
- (B) 64 and 5, respectively
- (C) 32 and 6, respectively
- (D) 31 and 5, respectively



## MCQ-10

---

The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are

- (A) 63 and 6, respectively
- (B) 64 and 5, respectively
- (C) 32 and 6, respectively
- (D) 31 and 5, respectively

Answer: (A)

**Explanation:**

Number of nodes is maximum for a perfect binary tree.

A perfect binary tree of height  $h$  has  $2^{h+1} - 1$  nodes

Number of nodes is minimum for a skewed binary tree.

A perfect binary tree of height  $h$  has  $h+1$  nodes.



## MCQ-11

---

A binary tree  $T$  has 20 leaves. The number of nodes in  $T$  having two children is

- (A) 18
- (B) 19
- (C) 17
- (D) Any number between 10 and 20



## MCQ-11

---

A binary tree T has 20 leaves. The number of nodes in T having two children is

- (A) 18
- (B) 19
- (C) 17
- (D) Any number between 10 and 20

Answer: (B)

### Explanation:

Sum of all degrees =  $2 * |E|$ .

Here considering tree as a k-ary tree :

Sum of degrees of leaves + Sum of degrees for Internal Node except root + Root's degree =  $2 * (\text{No. of nodes} - 1)$ .

Putting values of above terms,

$$L + (I-1)*(k+1) + k = 2 * (L + I - 1)$$

$$L + k*I - k + I - 1 + k = 2*L + 2I - 2$$

$$L + K*I + I - 1 = 2*L + 2*I - 2$$

$$K*I + 1 - I = L$$

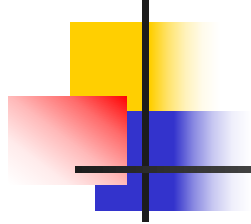
$$(K-1)*I + 1 = L$$

Given  $k = 2, L=20$

$$\implies (2-1)*I + 1 = 20$$

$$\implies I = 19$$

$\implies$  T has 19 internal nodes which are having two children.



---

An application of binary trees:  
**Expression Trees**



# A Binary Expression Tree

---

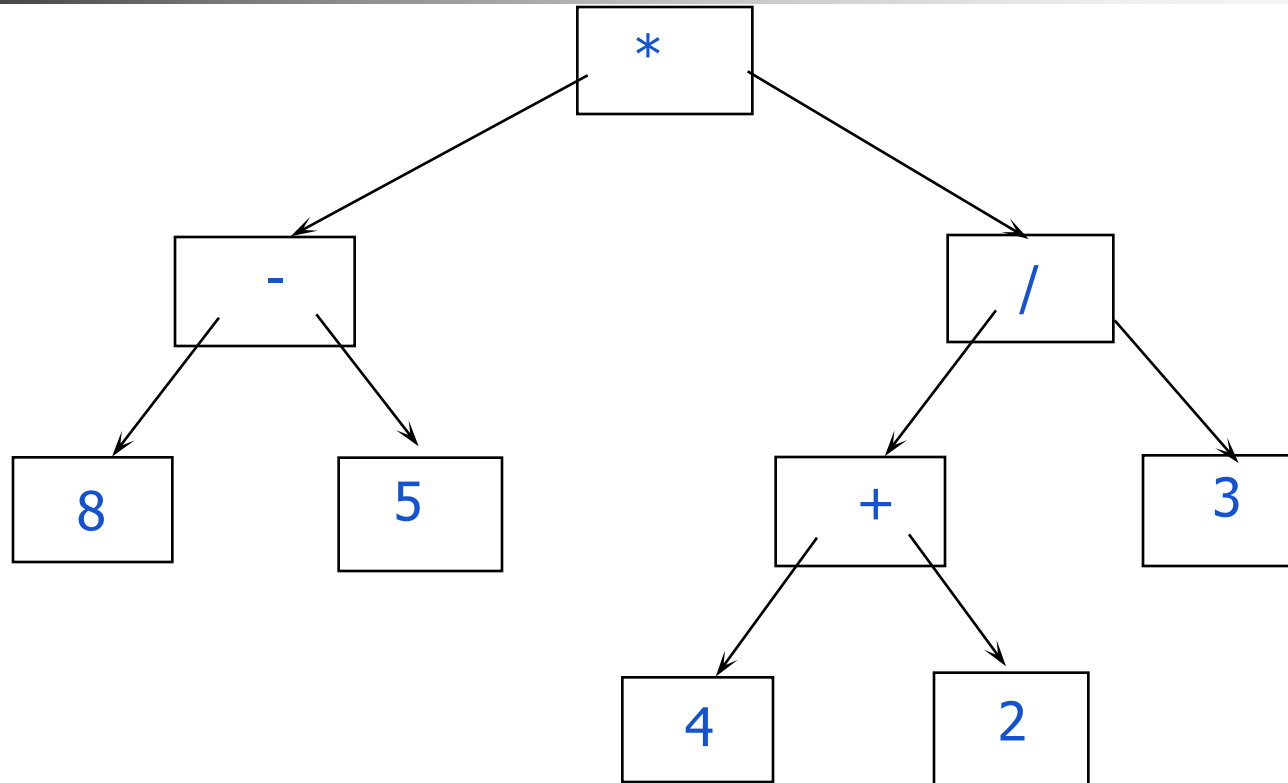
A special kind of binary tree in which:

1. Each **leaf node** contains a single operand
2. Each **nonleaf node** contains a single binary operator
3. The left and right subtrees of an operator node represent **subexpressions** that must be evaluated **before** applying the operator at the root of the subtree.





# A Four-Level Binary Expression





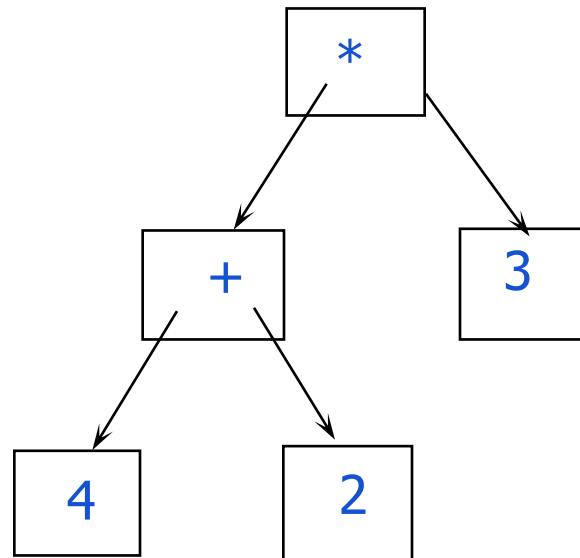
# Levels Indicate Precedence

---

- The levels of the nodes in the tree indicate their relative precedence of evaluation (**Do not need parentheses to indicate precedence**).
- Operations at higher levels of the tree are evaluated later than those below them.
- The operation at the root is always the last operation performed.



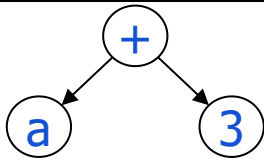
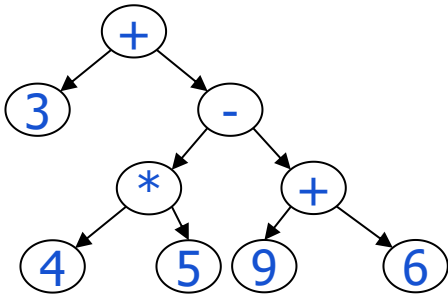
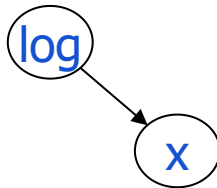
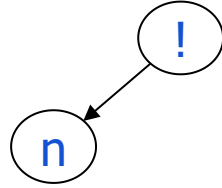
# A Binary Expression Tree



What value does it have?

$$(4 + 2) * 3 = 18$$

# Expression Tree Examples

Inorder Traversal Result	Expression Tree	Expression
$a + 3$	 <pre> graph TD     A((+)) --&gt; B((a))     A --&gt; C((3))           </pre>	$(a+3)$
$3+4*5-9+6$	 <pre> graph TD     A((+)) --&gt; B((3))     A --&gt; C((-))     C --&gt; D((*))     C --&gt; E((+))     D --&gt; F((4))     D --&gt; G((5))     E --&gt; H((9))     E --&gt; I((6))           </pre>	$3+(4*5-(9+6))$
$\log x$	 <pre> graph TD     A((log)) --&gt; B((x))           </pre>	$\log(x)$
$n !$	 <pre> graph TD     A((!)) --&gt; B((n))           </pre>	$n!$



# Why Expression trees?

---

- Expression trees are used to remove ambiguity in expressions.
- Consider the algebraic expression  $2 - 3 * 4 + 5$ .
- Without the use of precedence rules or parentheses, different orders of evaluation are possible:
  - $((2-3)*(4+5)) = -9$
  - $((2-(3*4))+5) = -5$
  - $(2-((3*4)+5)) = -15$
  - $((2-3)*4)+5 = 1$
  - $(2-(3*(4+5))) = -25$
- The expression is ambiguous because it uses infix notation:  
each operator is placed between its operands.



# Why Expression trees? (contd.)

---

- Storing a **fully parenthesized expression**, such as  $((x+2)-(y*(4-z)))$ , is **wasteful**, since the parentheses in the expression need to be stored to properly evaluate the expression.
- A compiler will read an expression in a language, and transform it into an **expression tree**.
- Expression trees impose a hierarchy on the operations in the expression.
  - Terms deeper in the tree get evaluated first.
  - This allows the establishment of the correct precedence of operations without using parentheses.



# Why Expression trees? (contd.)

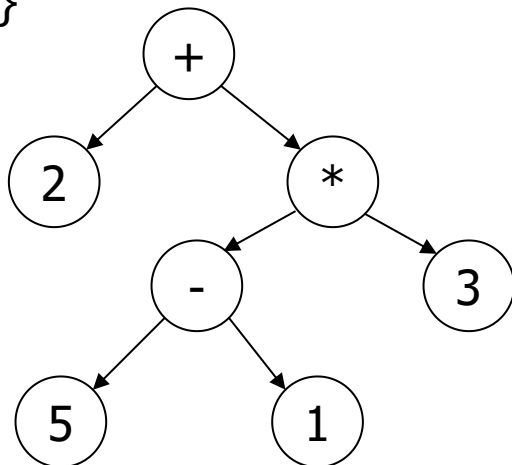
---

- Expression trees can be very useful for:
  - Evaluation of the expression.
  - Generating correct compiler code to actually compute the expression's value at execution time.
  - Performing symbolic mathematical operations (such as differentiation) on the expression.

# Evaluating an Expression tree

Assuming that  $t$  is a valid expression tree, a pseudo code algorithm for evaluating the expression tree is:

```
evalExpr(ExpressionTree t){  
    if(t->left == NULL && t->right == NULL)    // node t is a leaf node  
        return (t->info)    // which is an operand  
    else {  
        op = t->info  
        opnd1 = evalExpr(t->left)  
        opnd2 = evalExpr(t->right) ;  
        return(eval(opnd1, op, opnd2))  
    }  
}
```



Order of evaluation: 3 1 2

$(2 + ((5 - 1) * 3))$



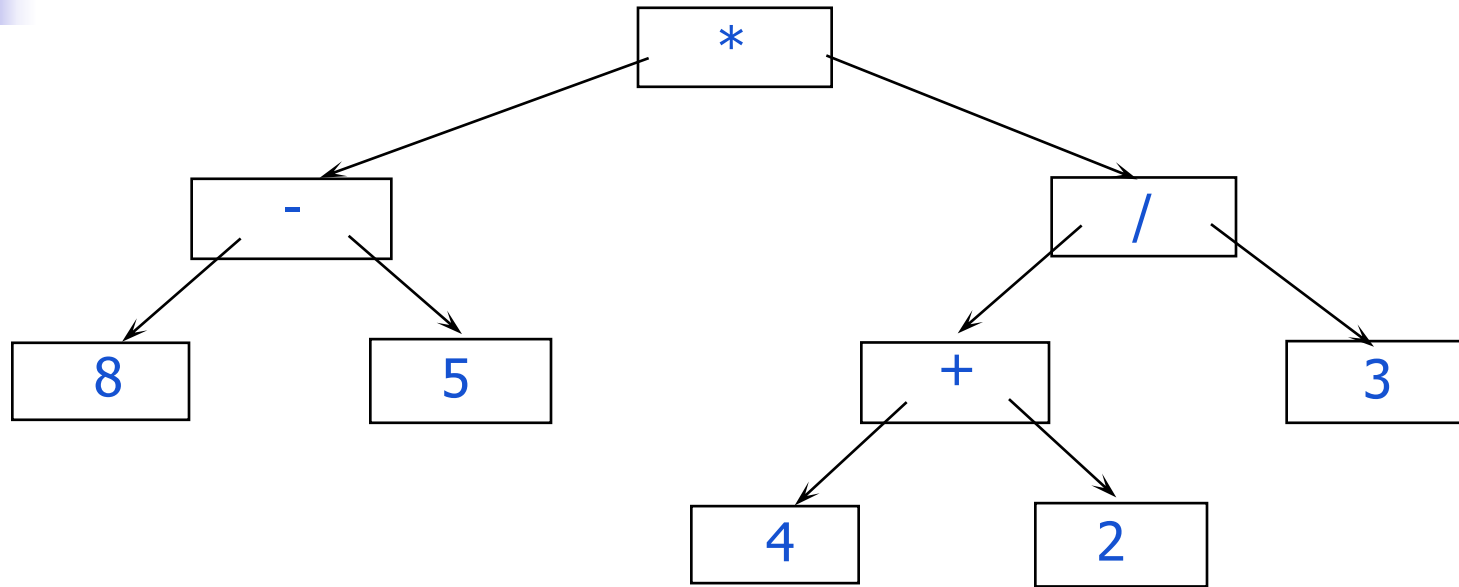


# Constructing an expression tree from a postfix expression

The pseudo code algorithm to convert a valid postfix expression, containing binary operators, to an expression tree:

```
while(not the end of the expression) {  
    if(symbol == operand) {  
        create a node for the operand (named node)  
        push(node)  
    }  
    if(symbol == operator) {  
        create a node for the operator (named node)  
        node->right = pop()  
        node->left=pop()  
        push(node)  
    }  
}
```

# Easy to generate the infix, prefix, postfix expressions (how?)

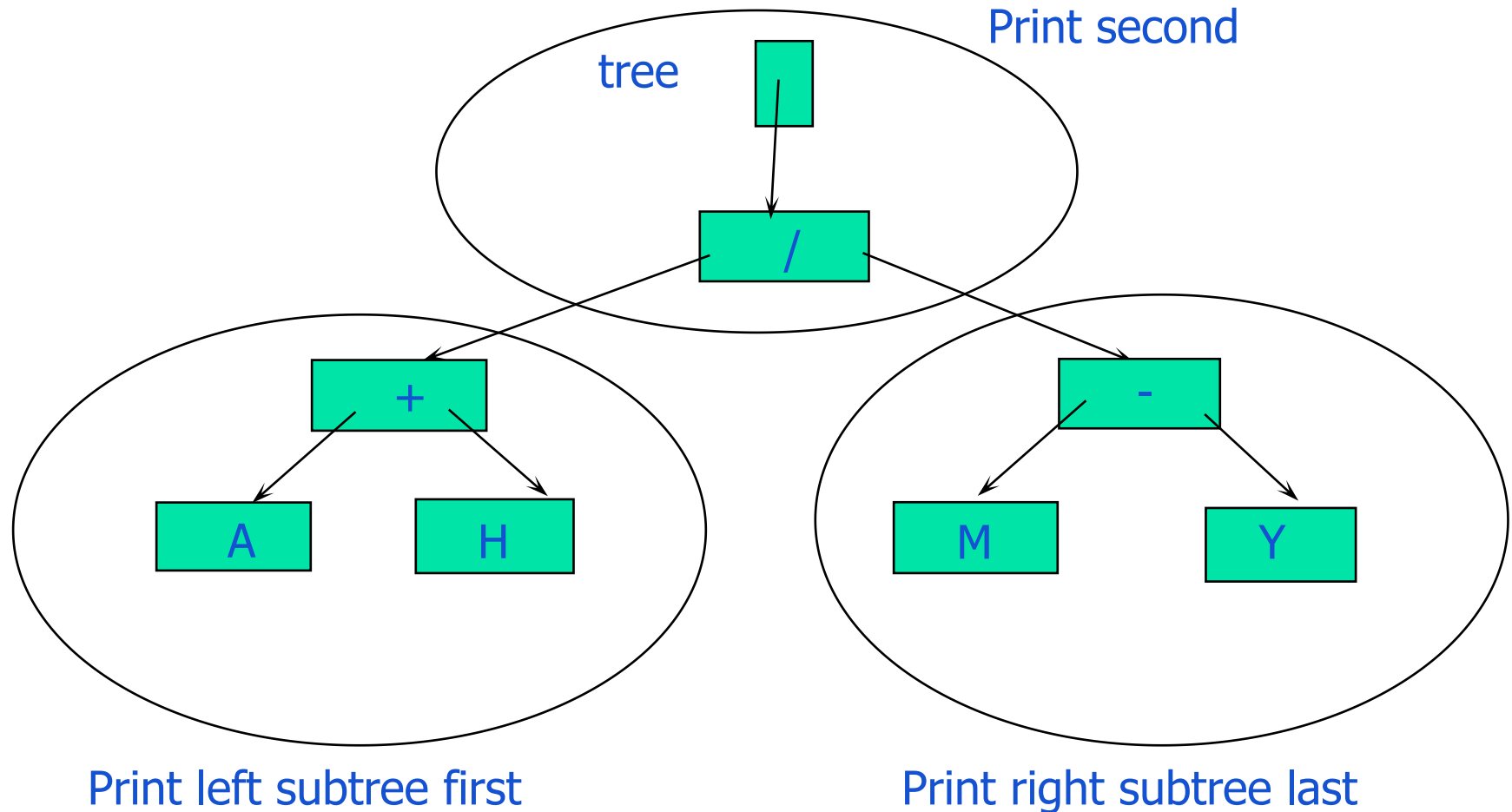


Infix:  $((8 - 5) * ((4 + 2) / 3))$

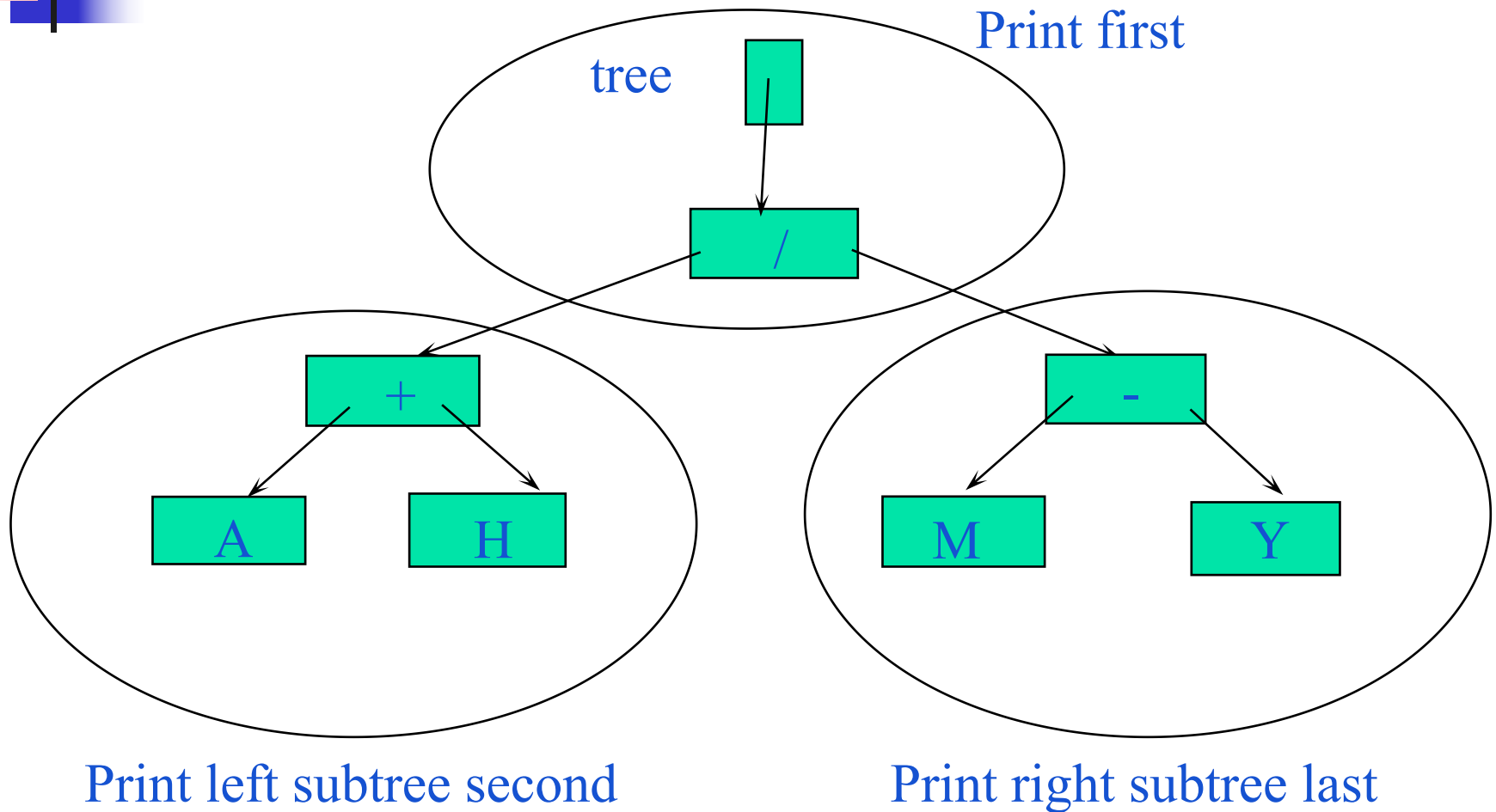
Prefix:  $* - 8 5 / + 4 2 3$

Postfix:  $8 5 - 4 2 + 3 / *$

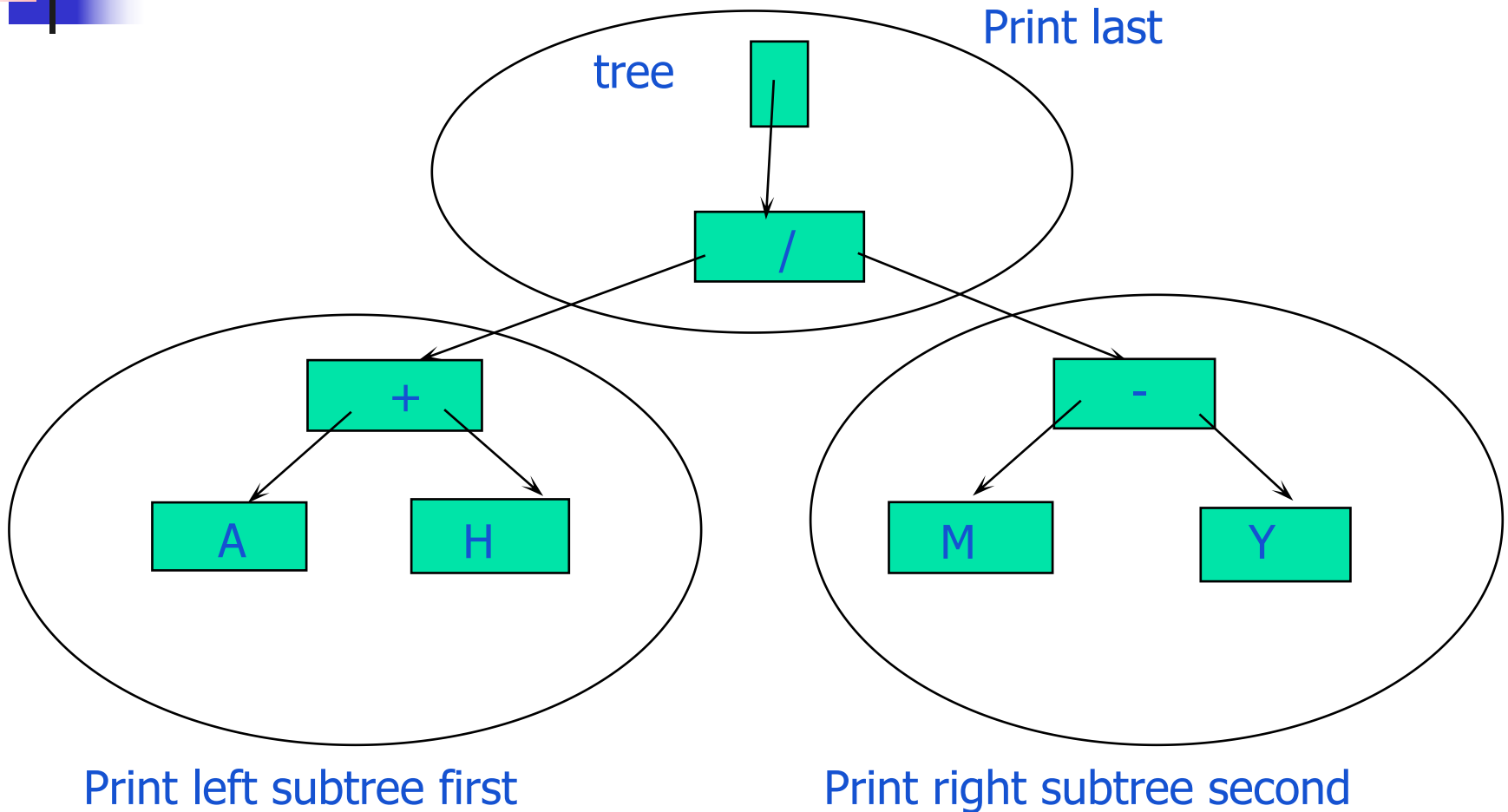
# Inorder Traversal: $(A + H) / (M - Y)$



# Preorder Traversal: / + A H - M Y



# Postorder Traversal: A H + M Y - /





# Binary Search Trees

---

- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.



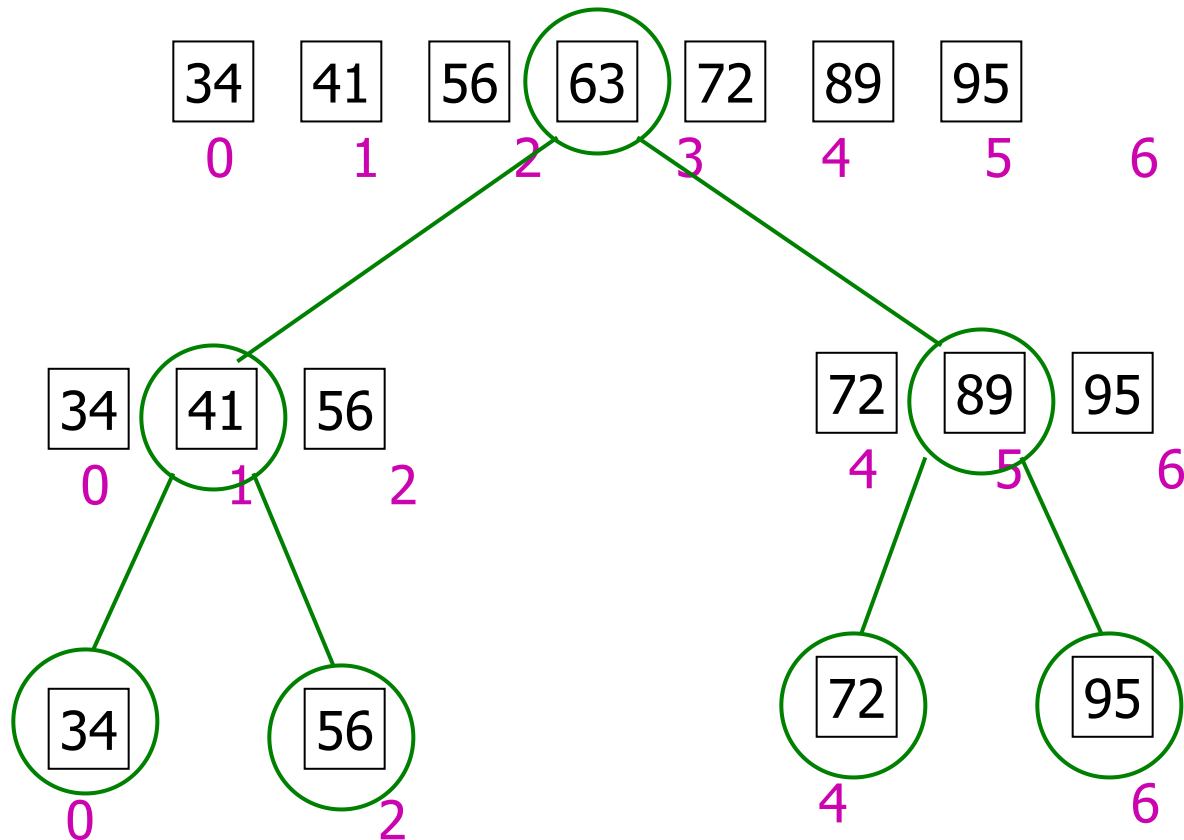
# Binary Search Trees

---

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- This data organization leads to  $O(\log n)$  complexity for searching, insertion and deletion in certain types of the BST (balanced trees).
  - $O(h)$  in general

# Binary Search Algorithm

Binary Search algorithm of an array of sorted items reduces the search space by one half after each comparison



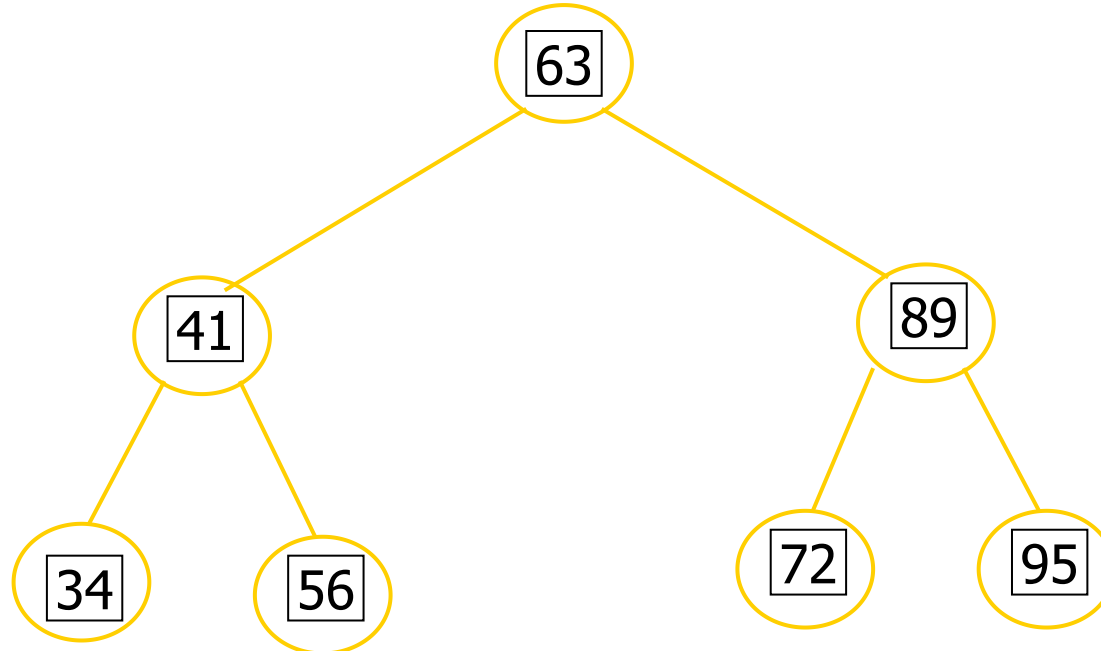




# Organization Rule for BST

---

- values at all nodes in the left subtree of a node are less than the node value
- values in all nodes at the right subtree of a node are greater than the node values





# Binary Tree

---

```
typedef struct node {  
    int  info;  
    struct node *left;  
    struct node *right  
} *treeNode;
```



# BST Operations: Search

---

Searching in the BST

method search(key)

- implements the binary search based on comparison of the items in the tree
- the items in the BST must be comparable (e.g integers, strings, etc.)
- The search starts from the root node.
- It probes down, comparing the values in each node with the target, till it finds the node value equal to the target.
  - Returns this node address or null if there is none (i.e. search is unsuccessful).



# Search in a BST: Non-Recursive code

---

```
treeNode iterativeSearch(treeNode root, int key) {  
    // Traverse untill root reaches to terminal node  
    while (root != NULL) {  
        if (key > root->info) // pass right subtree as new tree  
            root = root->right;  
        else if (key < root->info) // pass left subtree as new tree  
            root = root->left;  
        else  
            return root; // if the key is found return 1  
    }  
    return NULL;  
}
```



# Search in a BST: Recursive code

---

```
treeNode search(treeNode root, int key) {  
    // Base Cases: root is null or key is present at root  
    if (root == NULL || root->info == key)  
        return root;  
  
    // Key is greater than root's key  
    if (root->info < key)  
        return search(root->right, key);  
  
    // Key is smaller than root's key  
    return search(root->left, key);  
}
```



# BST Operations: Insertion

---

method insert(key)

- find the place for the new item near the frontier of the BST **while retaining its organization of data**:
  - **starting at the root** it probes **down** the tree till it finds a node whose left or right pointer is **NULL** and that is a logical place for the new value to be inserted

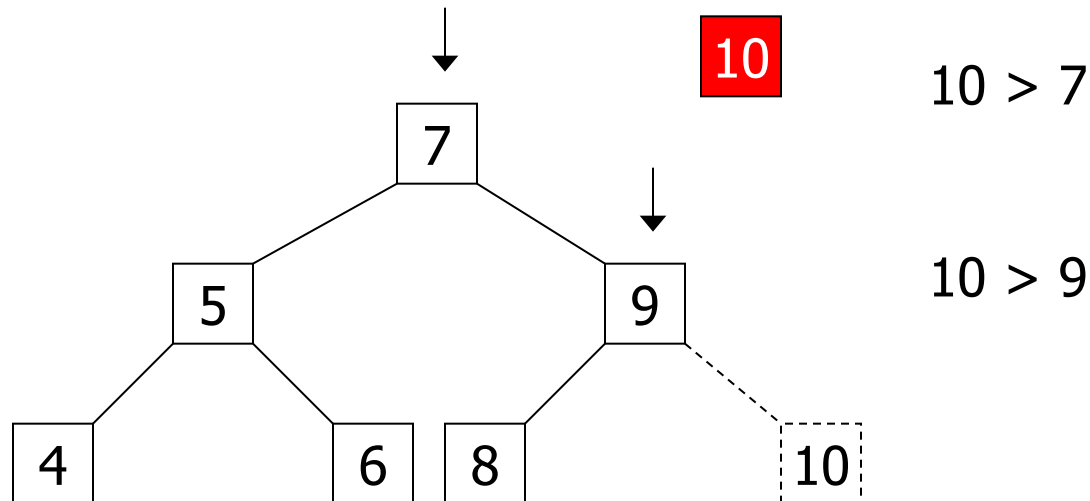
# Operations: Insertion

## Case 1: The Tree is Empty

- Set the root to a new node containing the item

## Case 2: The Tree is Not Empty

- Call a recursive function to insert the item





# BST Insertion: Pseudocode

---

```
if tree is empty
    create a root node with the new key
else
    if key = node->info
        // replace the node with the new value (unique node values)
    else if key > node->info // compare key with the right subtree
        if subtree is empty create a leaf node
            add key in right subtree
        else if key < node->info // compare key with the left subtree
            if the subtree is empty create a leaf node
                add key to the left subtree
```





# BST Insertion: Nonrecursive Code

---

```
void insert (treeNode root, int key) {
    treeNode newNode, ptr;
    ptr = findPos(root, key);
    if (ptr || root == NULL) {
        newNode = (treeNode) malloc(sizeof(struct node));
        if (newNode == NULL) {
            printf("Out of memory space");
            return;
        }
        newNode->info = key;
        newNode->left = newNode->right = NULL;
        if (root != NULL) {
            if (key < ptr->info) ptr->left = newNode;
            else if (key > ptr->info) ptr->right = newNode;
        }
        else root = newNode;
    }
}
```



# BST Insertion: Nonrecursive Code

---

```
treeNode findPos(treeNode root, int key) {  
    while (root != NULL) {  
        if (key > root->info) root = root->right;  
        else if (key < root->info) root = root->left;  
        else return root;  
    }  
    return NULL;  
}
```



# BST Insertion: Recursive

---

```
treeNode insert(treeNode root, int key) {  
    if(root==NULL) {  
        root=(treeNode) malloc(sizeof(struct node));  
        root->left=root->right = NULL;  
        root->info=key;  
    }  
    else {  
        if(key<root->info)  
            root->left=insert(root->left, key);  
        else if(key>root->info)  
            root->right=insert(root->right, key);  
    }  
    return root;  
}
```



# BST Shapes

---

- The order of supplying the data determines where it is placed in the BST, which determines the shape of the BST
- Create BSTs from the same set of data presented each time in a different order:
  - a) 17 4 14 19 15 7 9 3 16 10
  - b) 9 10 17 4 3 7 14 16 15 19
  - c) 19 17 16 15 14 10 9 7 4 3



# BST Operations: Removal

---

- **removes** a specified item from the BST and **adjusts** the tree
- uses a binary search to locate the target item:
  - starting at the root it **probes down** the tree till it finds the target or reaches a leaf node (i.e. target not in the tree)
- removal of a node must not leave a '**gap**' in the tree,



# Removal in BST - Pseudocode

---

method remove (key)

- I if the tree is empty return false
- II Attempt to locate the node containing the target using the binary search algorithm
  - if the target is not found return false
  - else the target is found, so remove the node:

Case 1: if the node has 2 empty subtrees  
replace the link in the parent with NULL

Case 2: if the node has a left and a right subtree

- replace the node's value with the max(min) value in the left(right) subtree
- delete the max(min) node in the left(right) subtree



# Removal in BST - Pseudocode

---

Case 3: if the node has no left child

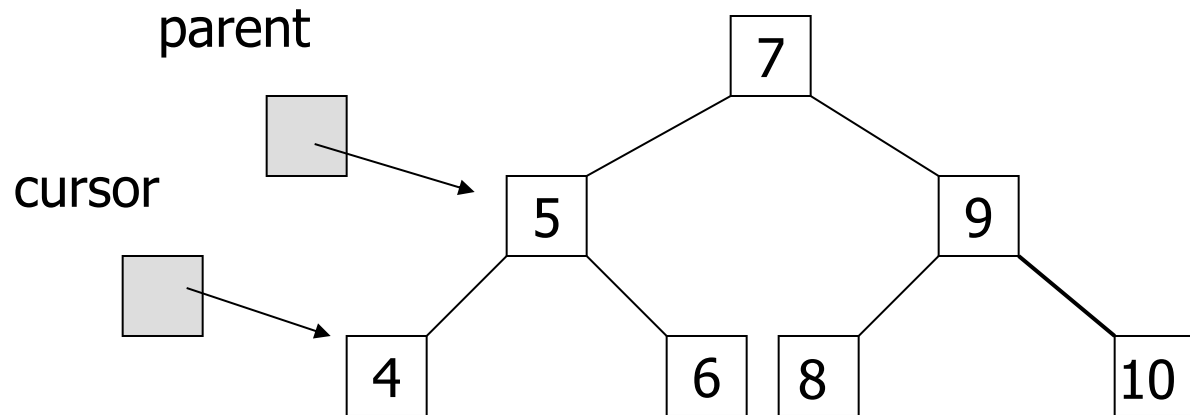
- link the parent of the node to the right (non-empty) subtree

Case 4: if the node has no right child

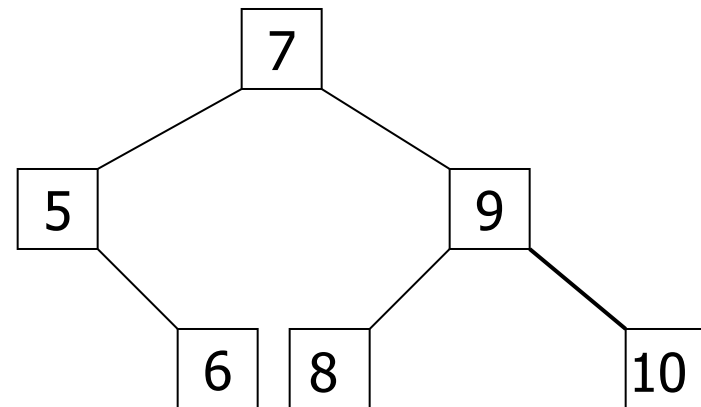
- link the parent of the target to the left (non-empty) subtree

# Removal in BST: Example

Case 1: removing a node with 2 empty subtrees



Removing 4  
replace the link in the  
parent with NULL





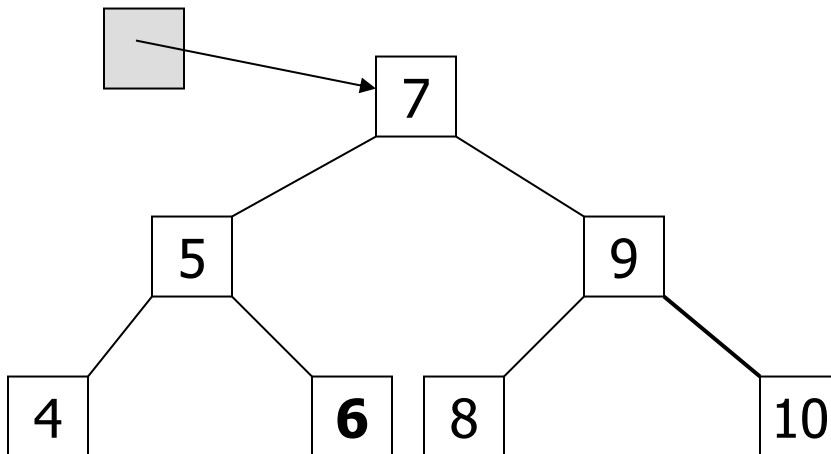
# Removal in BST: Example

## Case 2: removing a node with 2 subtrees

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

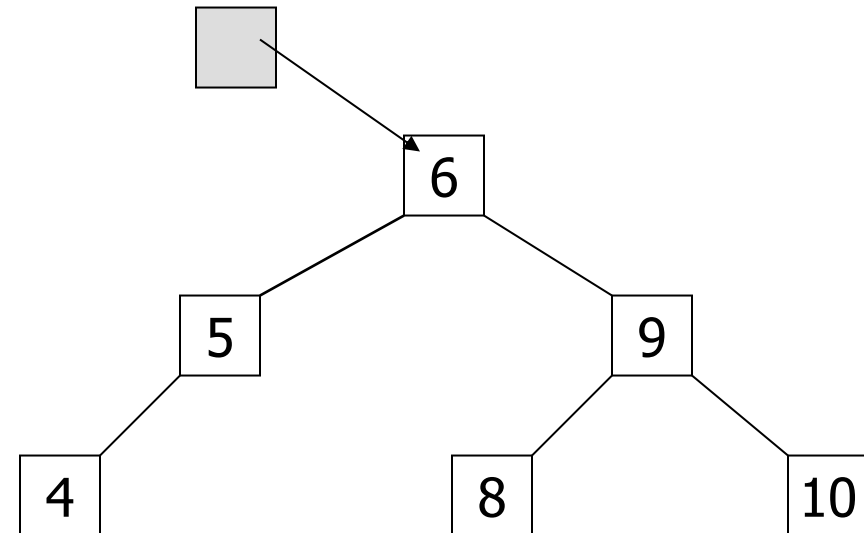
### Removing 7

cursor



What other element  
can be used as  
replacement?

cursor



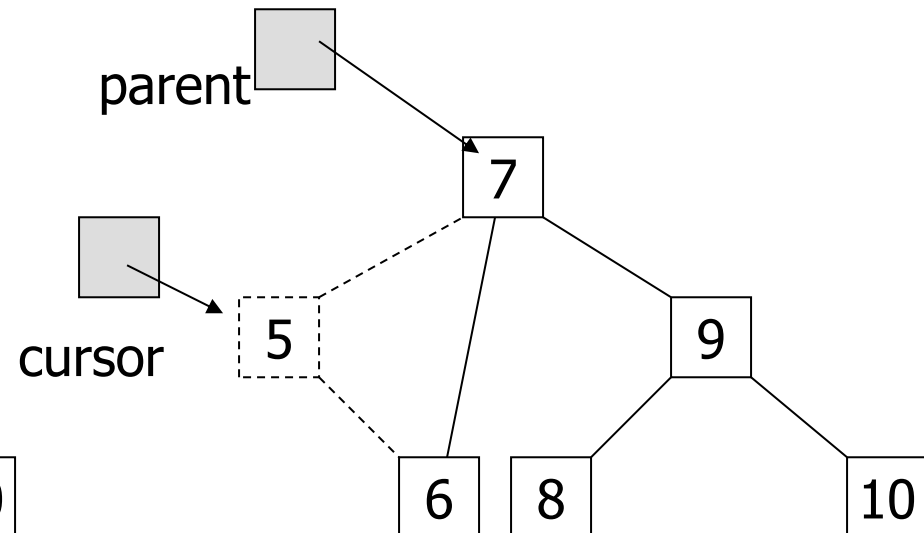
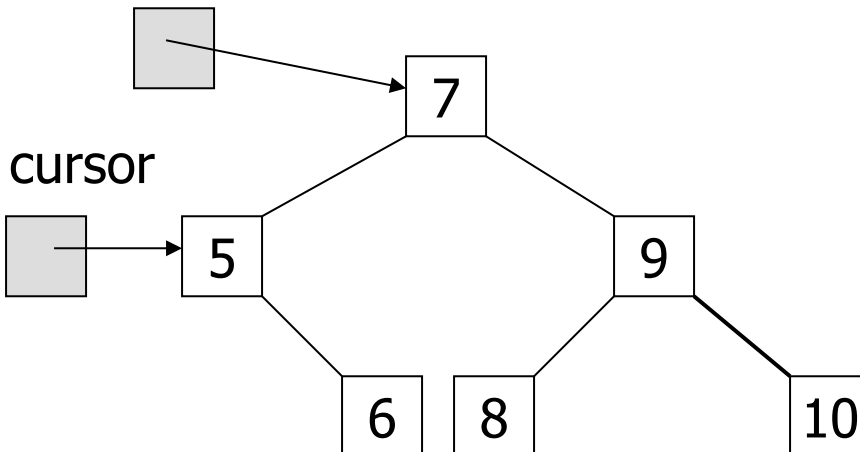
# Removal in BST: Example

Case 3: removing a node with 1 empty subtree

the node has no left child:

link the parent of the node to the right (non-empty) subtree

parent



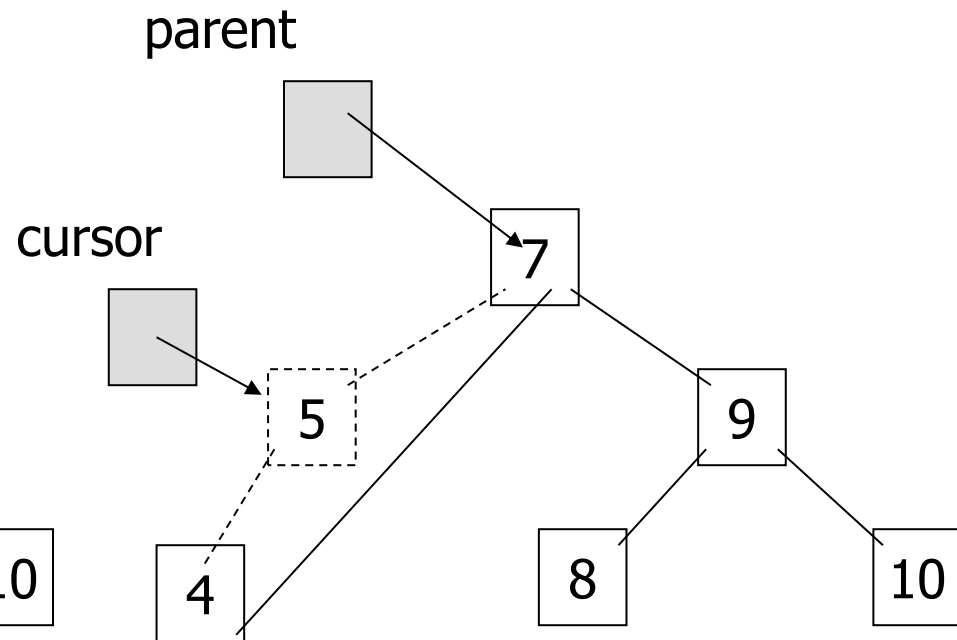
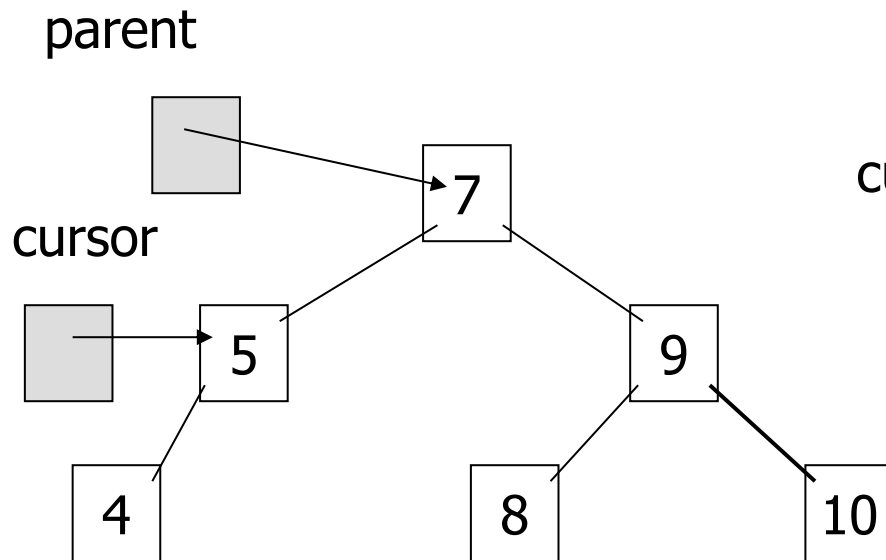
# Removal in BST: Example

Case 4: removing a node with 1 empty subtree

the node has no right child:

link the parent of the node to the left (non-empty) subtree

Removing 5





# Removal in BST: C code

---

```
treeNode * Delete(treeNode *node, int key) {
    treeNode *temp;
    if(node==NULL) printf("Empty Tree");
    else if(key < node->info)
        node->left = Delete(node->left, key);
    else if(key > node->info)
        node->right = Delete(node->right, key);
    else {
        if(node->right && node->left) {
            temp = FindMin(node->right);
            node -> info = temp->info;
            node -> right = Delete(node->right,
                                   temp->info);
        }
        else {
            temp = node;
            if(node->left == NULL)
                node = node->right;
```

```
            else if(node->right == NULL)
                node = node->left;
            free(temp);
        }
    }
    return node;
}
```

```
treeNode * FindMin(treeNode *node, int key) {
    if(node==NULL) return NULL;
    if(key > node->info)
        return FindMin(node->right, key);
    else if(key < node->info)
        return FindMin(node->left, data);
    else
        return node;
}
```



# Analysis of BST Operations

---

- The complexity of operations **search**, **insert** and **remove** in BST is  $O(h)$ , where  $h$  is the height.
- $O(\log n)$  when the tree is balanced. The updating operations cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become  $O(n)$



# Best Case

```
BST *root;
```

```
insert (root, "E");  
insert (root, "C");  
insert (root, "D");  
insert (root, "A");  
insert (root, "H");  
insert (root, "F");  
insert (root, "K");
```

**Output:**

**>>>> Items in advantageous order:**

	<b>K</b>
<b>H</b>	
	<b>F</b>
<b>E</b>	
	<b>D</b>
<b>C</b>	
	<b>A</b>



# Worst Case

---

```
BST *root;  
for (int i = 1; i <= 8; i++)  
    insert(root, i);
```

Output:

>>>> Items in worst order:

8  
7  
6  
5  
4  
3  
2  
1



# Random Case

---

```
BST *root;  
for (int i = 1; i <= 8; i++)  
    insert(root, random());
```

Output:

>>>> Items in random order:

X  
U  
P  
O  
H  
F  
B





## MCQ-1

---

Postorder traversal of a given binary search tree, T produces the following sequence of keys

10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?

- (A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95
- (B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29
- (C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95
- (D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29



## MCQ-1

Postorder traversal of a given binary search tree, T produces the following sequence of keys

10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?

- (A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95
- (B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29
- (C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95
- (D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

Answer: (A)

**Explanation:** Inorder traversal of a BST always gives elements in increasing order. Among all four options, a) is the only increasing order sequence.



# Better Search Trees

---

Prevent the degeneration of the BST:

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of ST which maintain the optimal performance:
  - splay trees
  - **AVL trees**
  - 2-4 Trees
  - Red-Black trees
  - **B-trees**



# Binary Search Tree - Best Time

---

- All BST operations are  $O(d)$ , where  $d$  is tree depth
- minimum  $d$  is for a binary tree with  $N$  nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is  $O(\log_2 N)$

$$d = \lfloor \log_2 N \rfloor$$



# Binary Search Tree - Worst Time

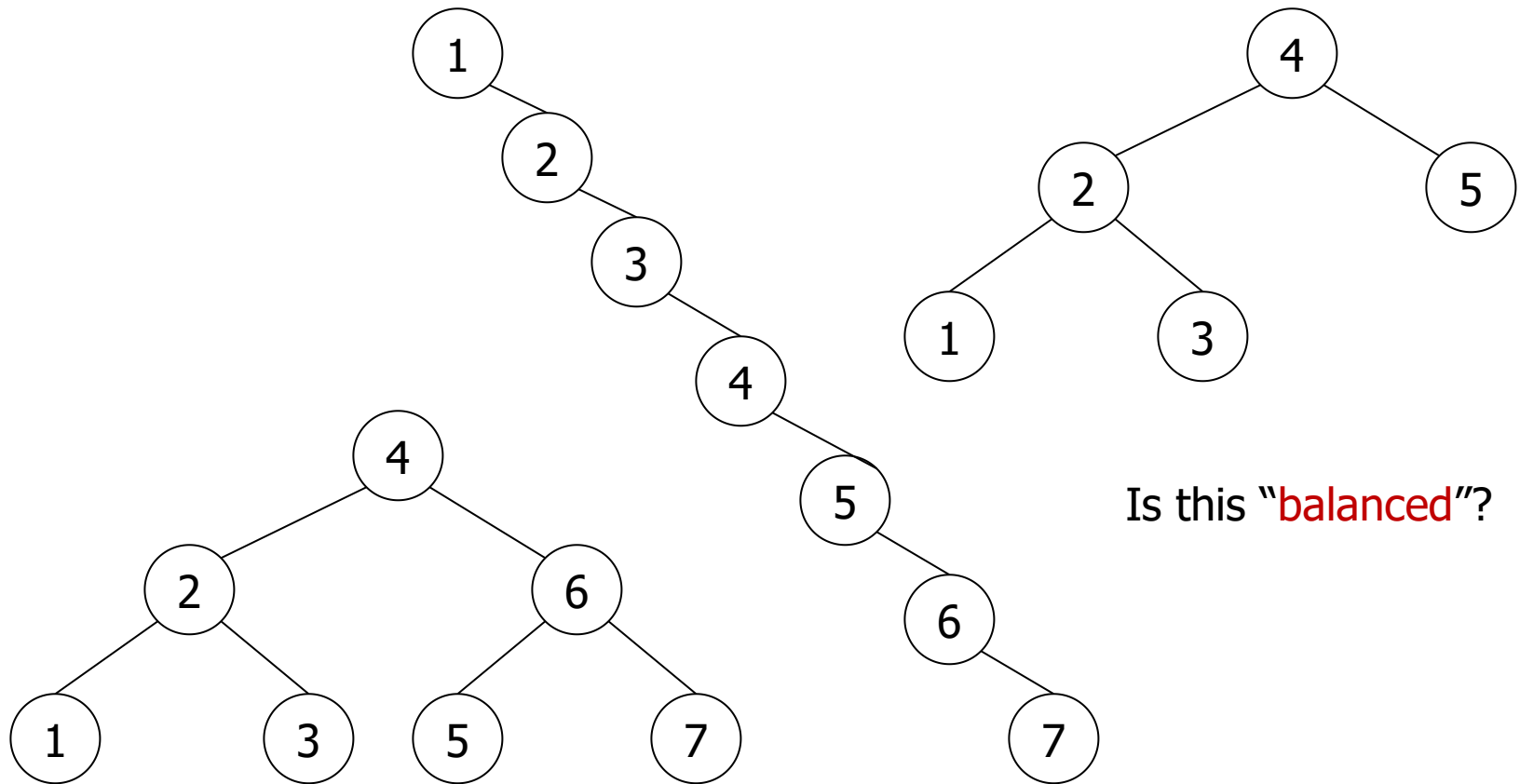
---

- Worst case running time is  $O(N)$ 
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree



# Balanced and unbalanced BST

---

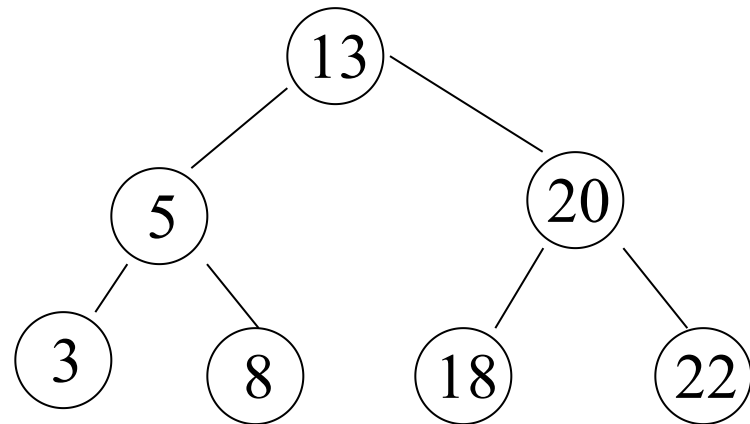
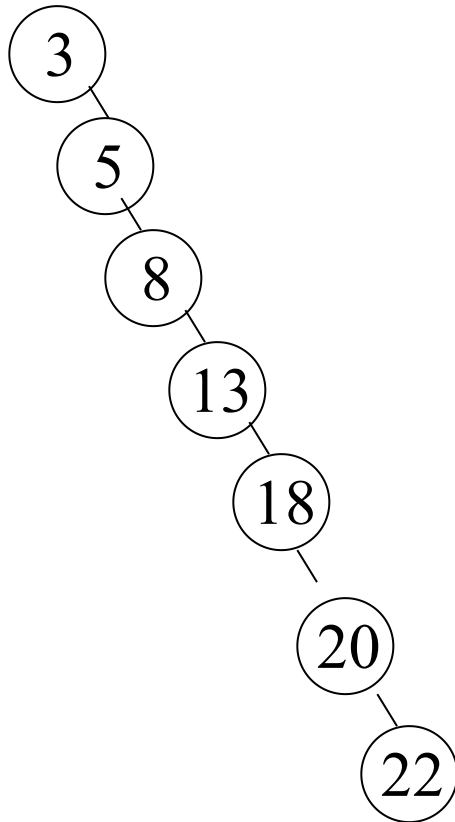


Is this "balanced"?



# Motivation

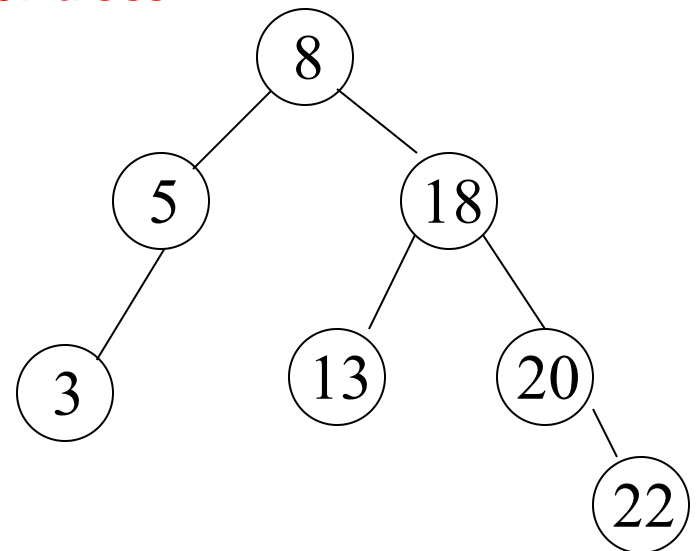
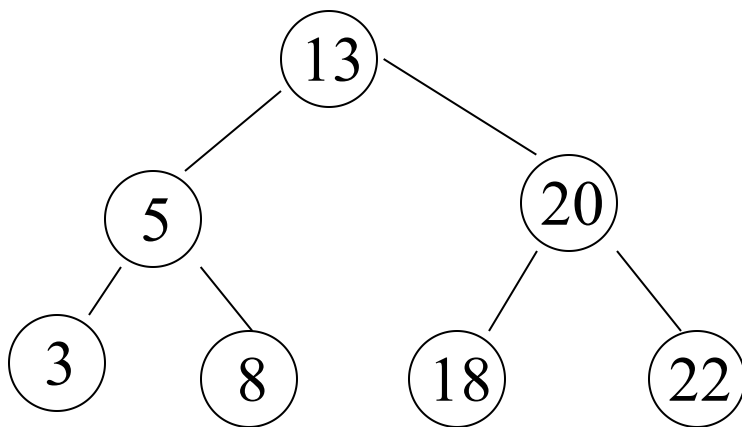
- When building a binary search tree, what type of trees would we like? **Example:** 3, 5, 8, 20, 18, 13, 22



# Motivation

Complete binary tree is **hard to build** when we allow **dynamic insert and remove**.

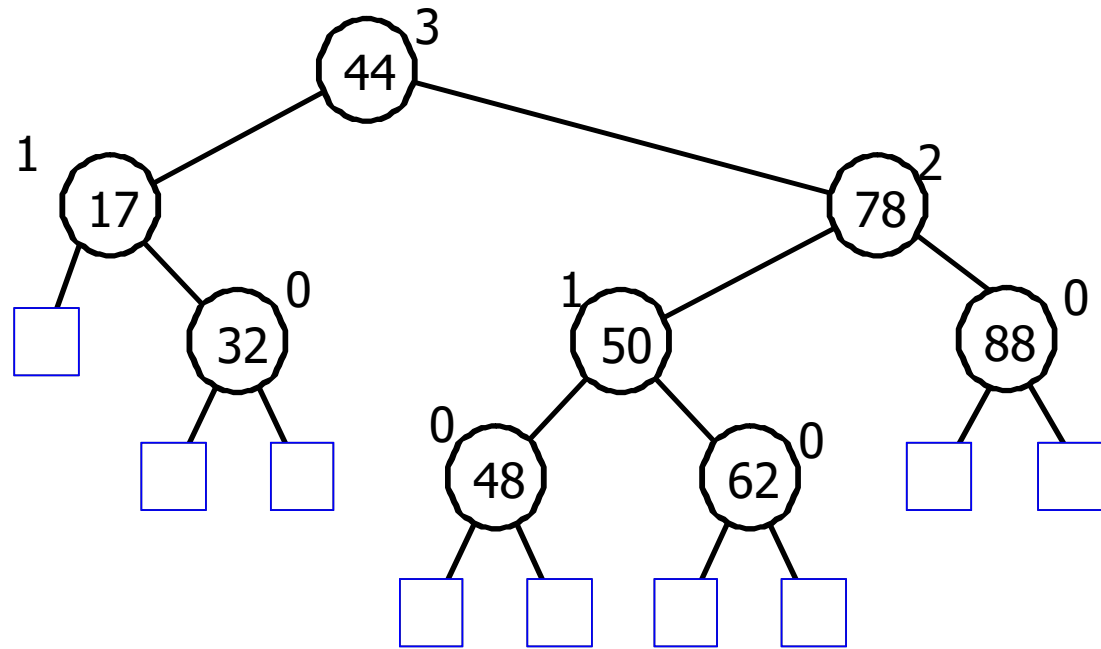
- We want a tree that has the following properties
  - Tree height =  $O(\log(N))$
  - allows dynamic insert and remove with  $O(\log(N))$  time complexity.
- The AVL tree is one of this kind of trees.





# AVL (Adelson-Velskii and Landis) Trees

- An AVL Tree is a binary search tree such that for every internal node, v of T, the heights of the children of v can differ by at most 1.



An example of an AVL tree where the heights are shown next to the nodes:



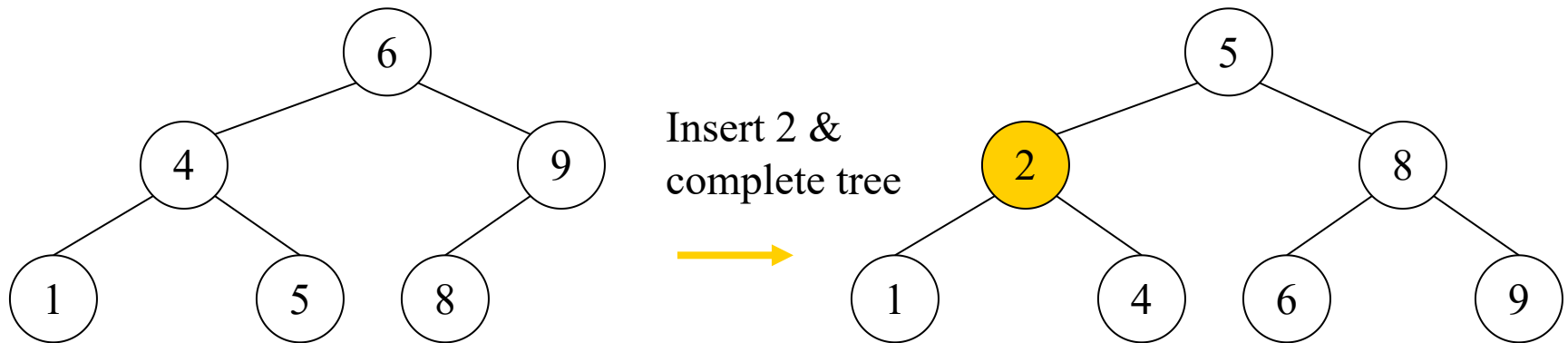
# AVL (Adelson-Velskii and Landis) Trees

---

- AVL tree is a binary search tree with **balance condition**
  - To ensure depth of the tree is  $O(\log(N))$
  - And consequently, search/insert/remove complexity bound  $O(\log(N))$
- **Balance condition**
  - For **every node** in the tree, height of left and right subtree can differ by at most 1

# Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree





# AVL - Good but not Perfect Balance

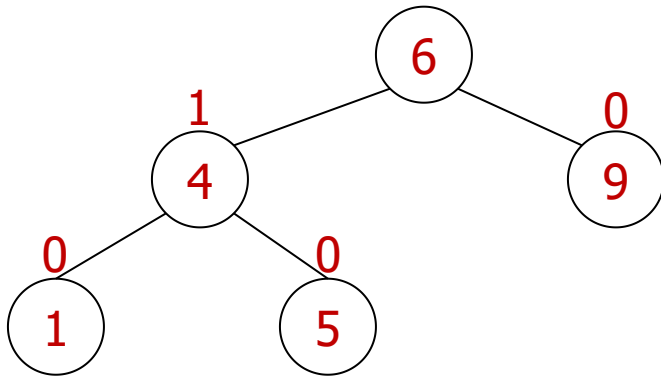
---

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

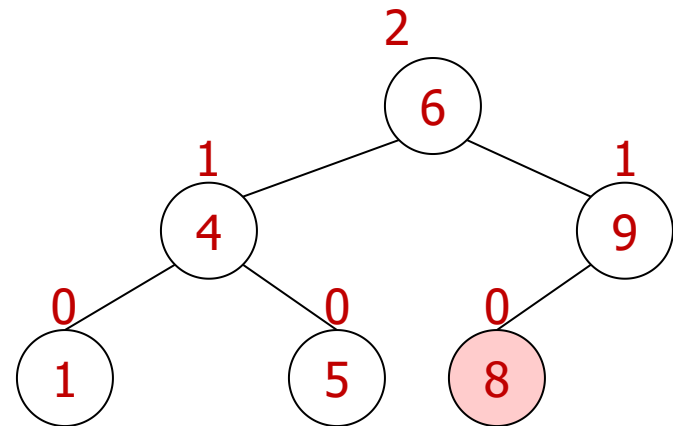
# Node Heights

Tree A (AVL)

height=2 BF=1-0=1



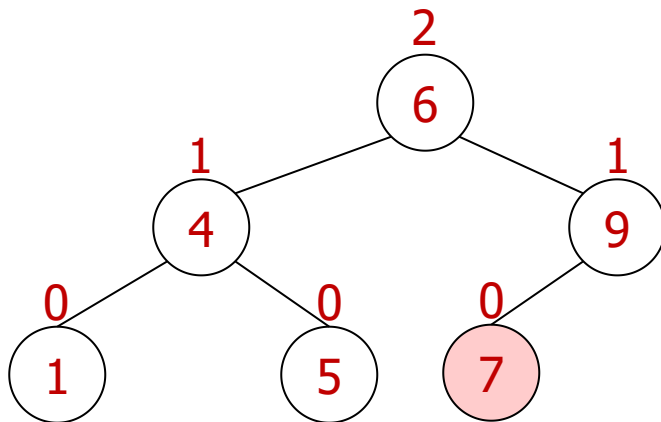
Tree B (AVL)



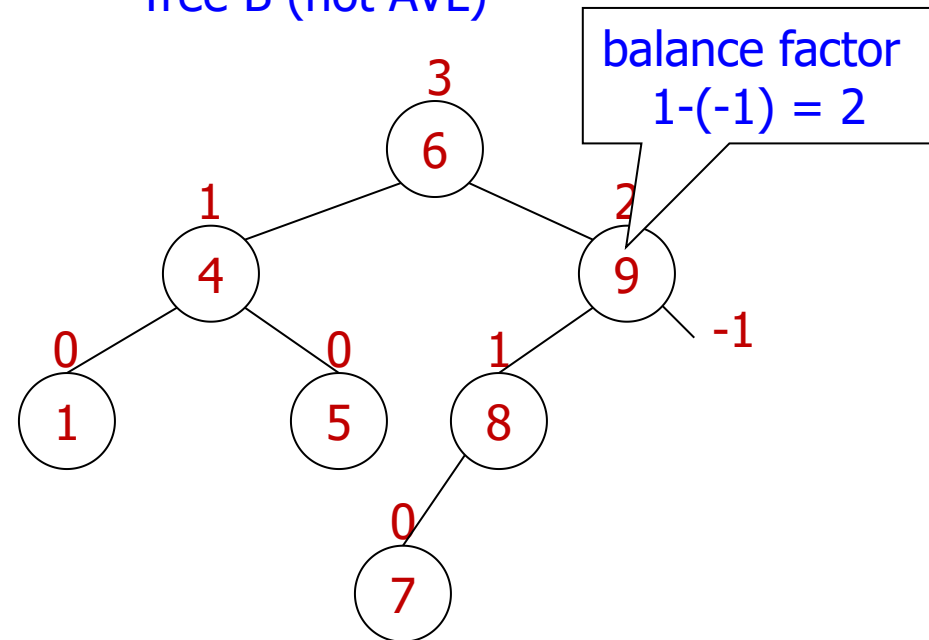
height of node =  $h$   
balance factor =  $h_{\text{left}} - h_{\text{right}}$   
empty height = -1

# Node Heights after Insert 7

Tree A (AVL)



Tree B (not AVL)



height of node = h  
balance factor =  $h_{\text{left}} - h_{\text{right}}$   
empty height = -1

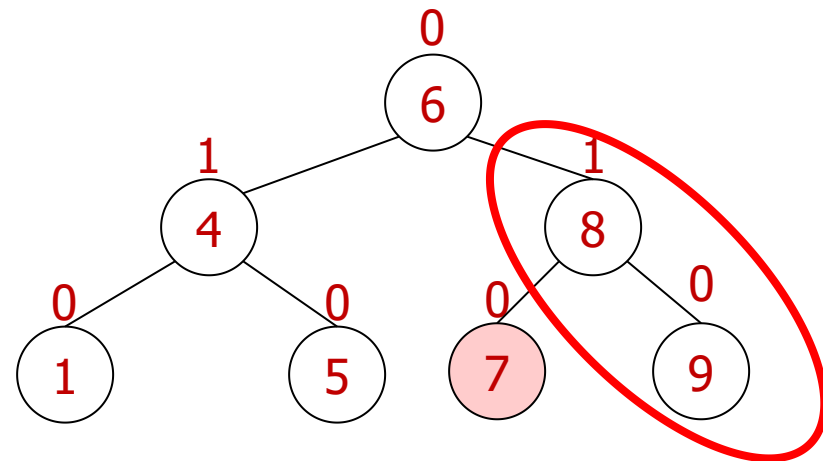
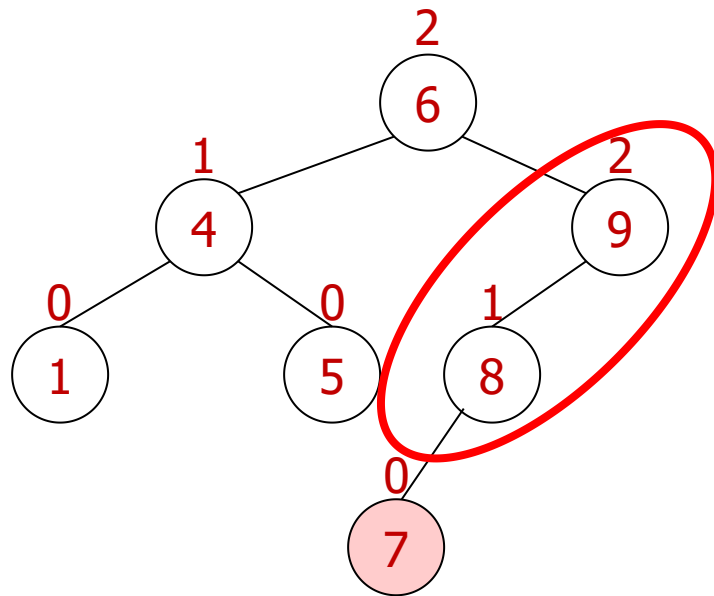


# Insert and Rotation in AVL Trees

---

- Insert operation may cause balance factor to become 2 or -2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference  $h_{\text{left}} - h_{\text{right}}$ ) is 2 or -2, adjust tree by *rotation* round the node

# Single Rotation in an AVL Tree







# Insertions in AVL Trees

---

Let the node that needs rebalancing be  $\alpha$ .

There are four cases:

Require single rotation:

Insertion into left subtree of left child of  $\alpha$ .

Insertion into right subtree of right child of  $\alpha$ .

Require double rotation

Insertion into right subtree of left child of  $\alpha$ .

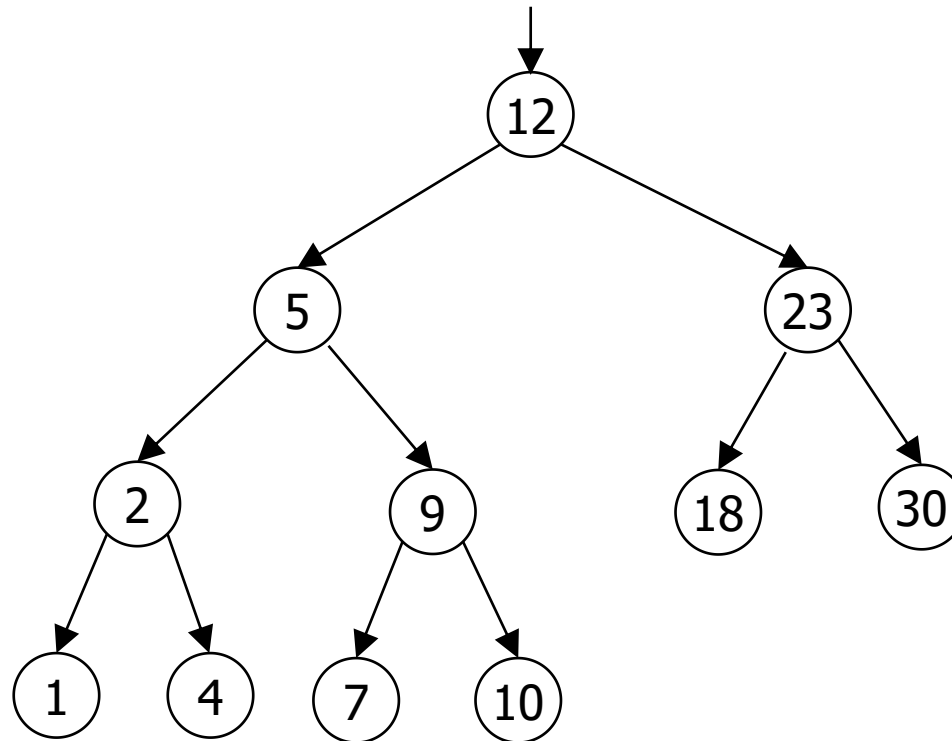
Insertion into left subtree of right child of  $\alpha$ .

The rebalancing is performed through four separate rotation algorithms.

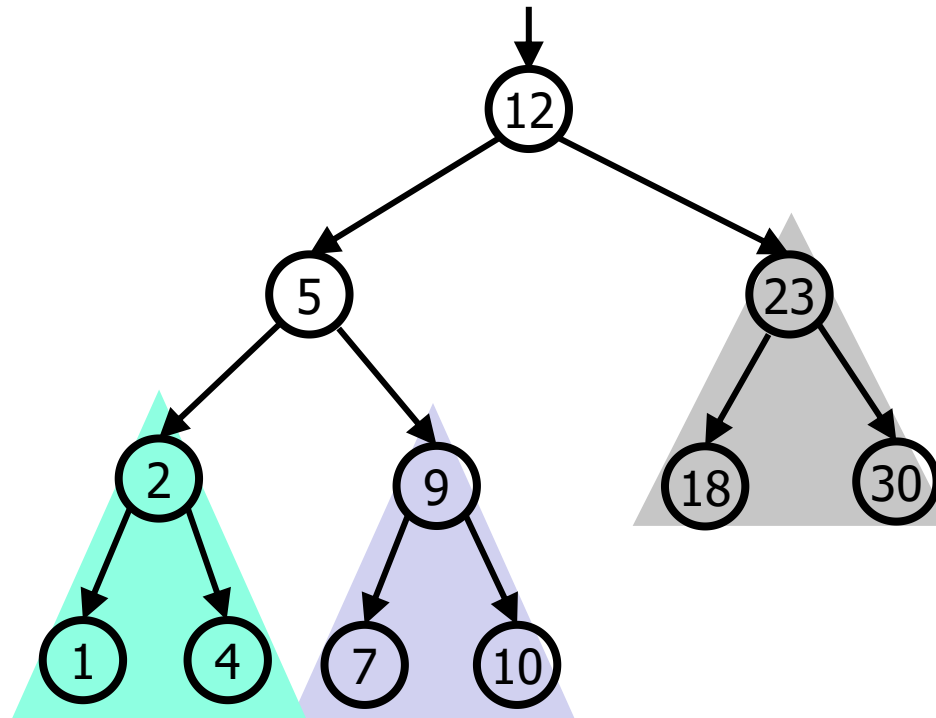


# Generalizing the Example

---



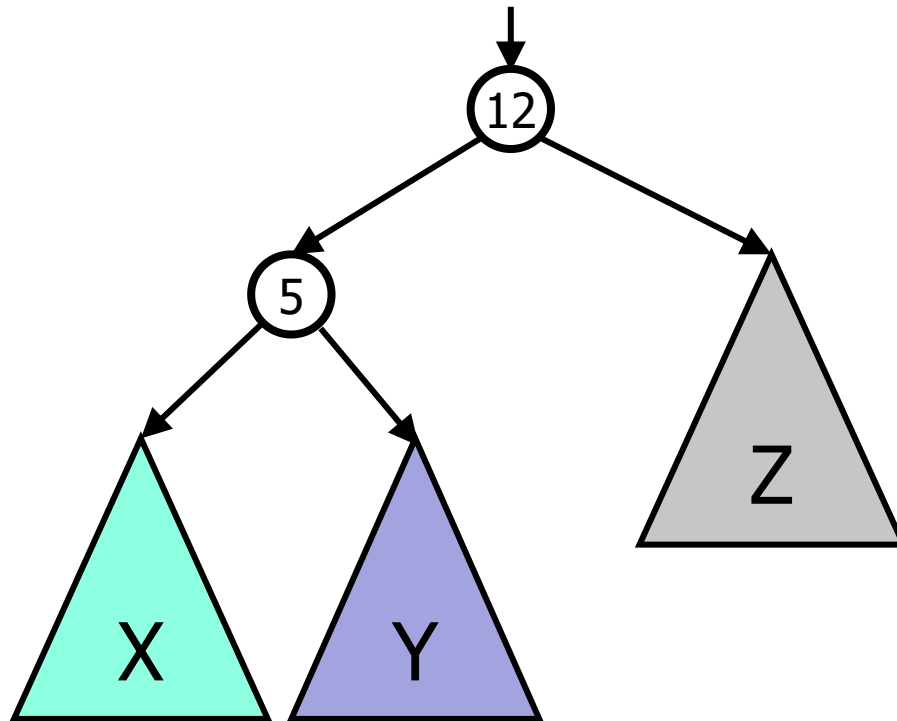
# Generalizing the Example





# Generalizing our example

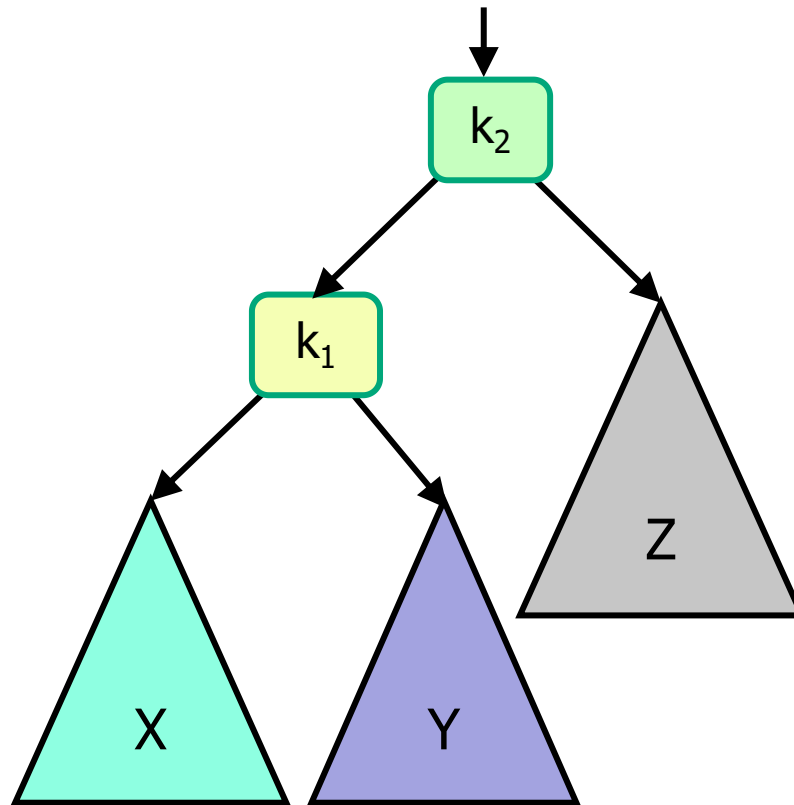
---





# Generalized Single Rotation

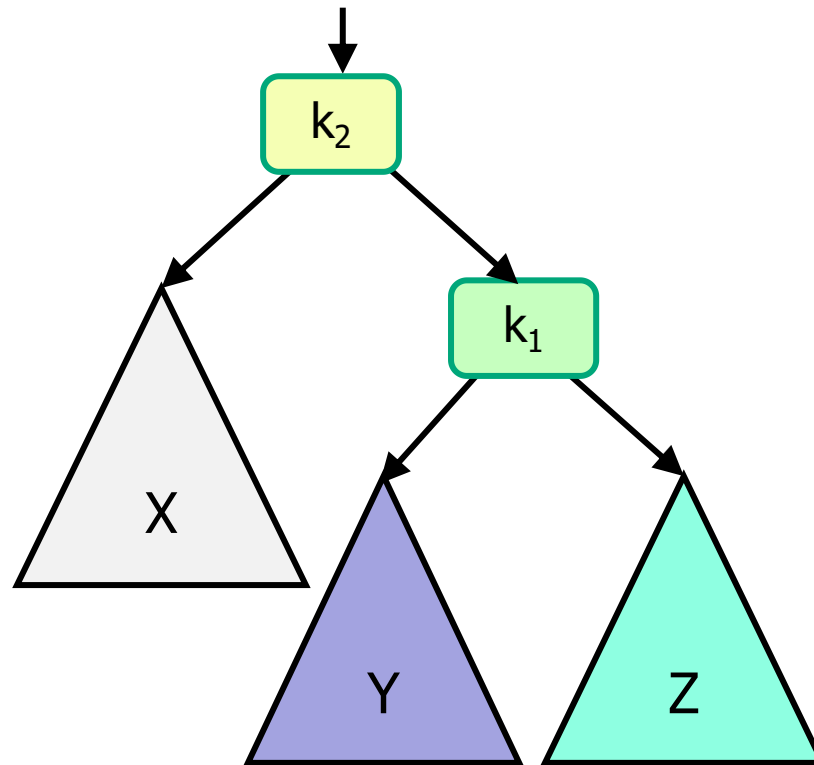
---



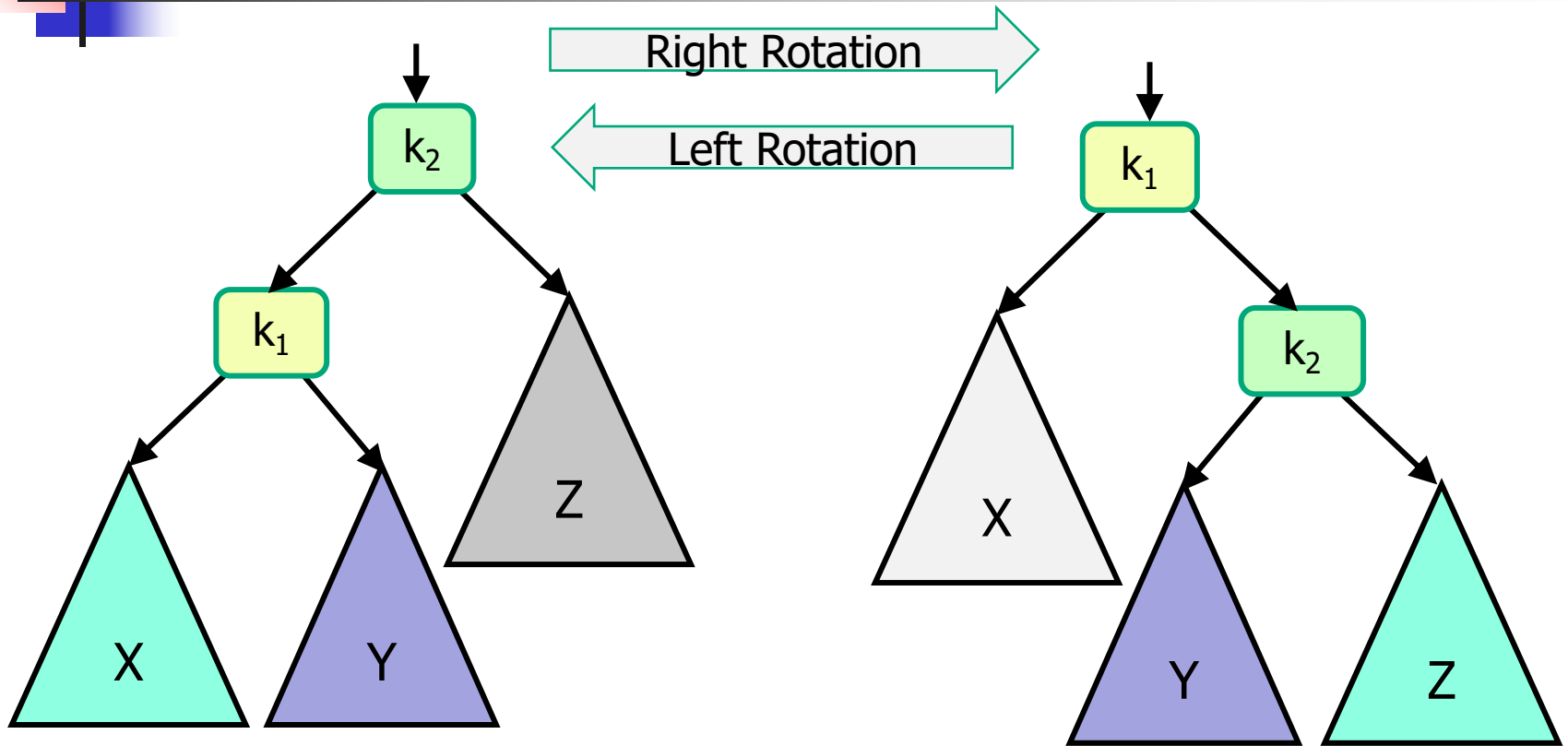


# Generalized Single Rotation

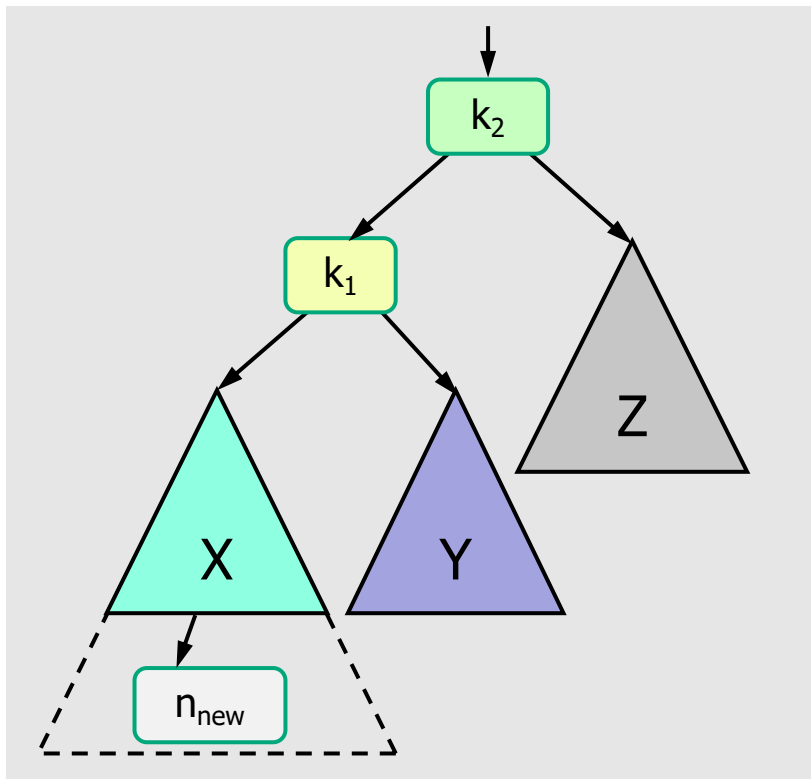
---



# Single Rotations



# Single Rotations: LL

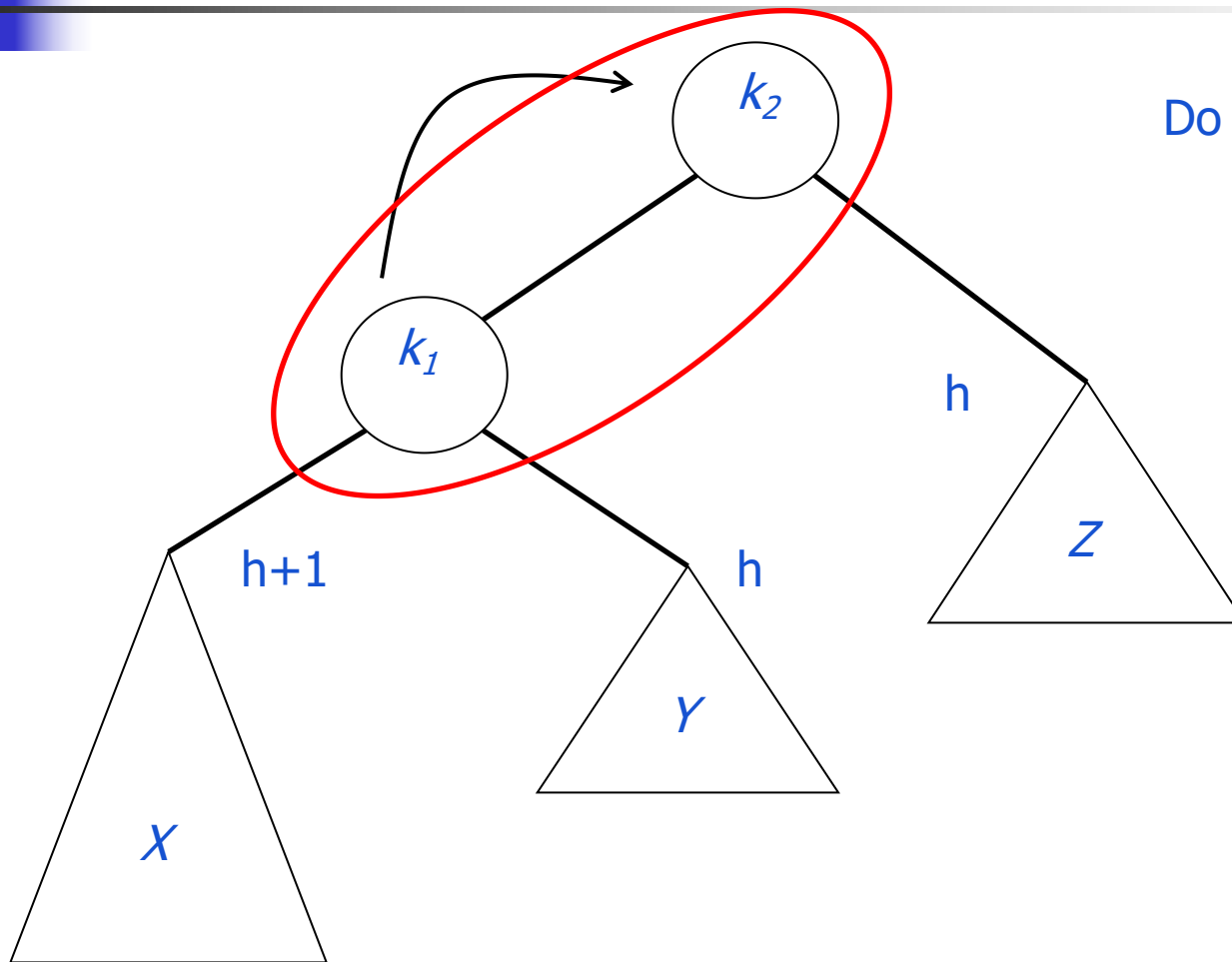


Inserting into subtree  $X$   
destroys the AVL  
property at node  $k_2$

- Replace node  $k_2$  by node  $k_1$
- Set node  $k_2$  to be right child of node  $k_1$
- Set subtree  $Y$  to be left child of node  $k_2$



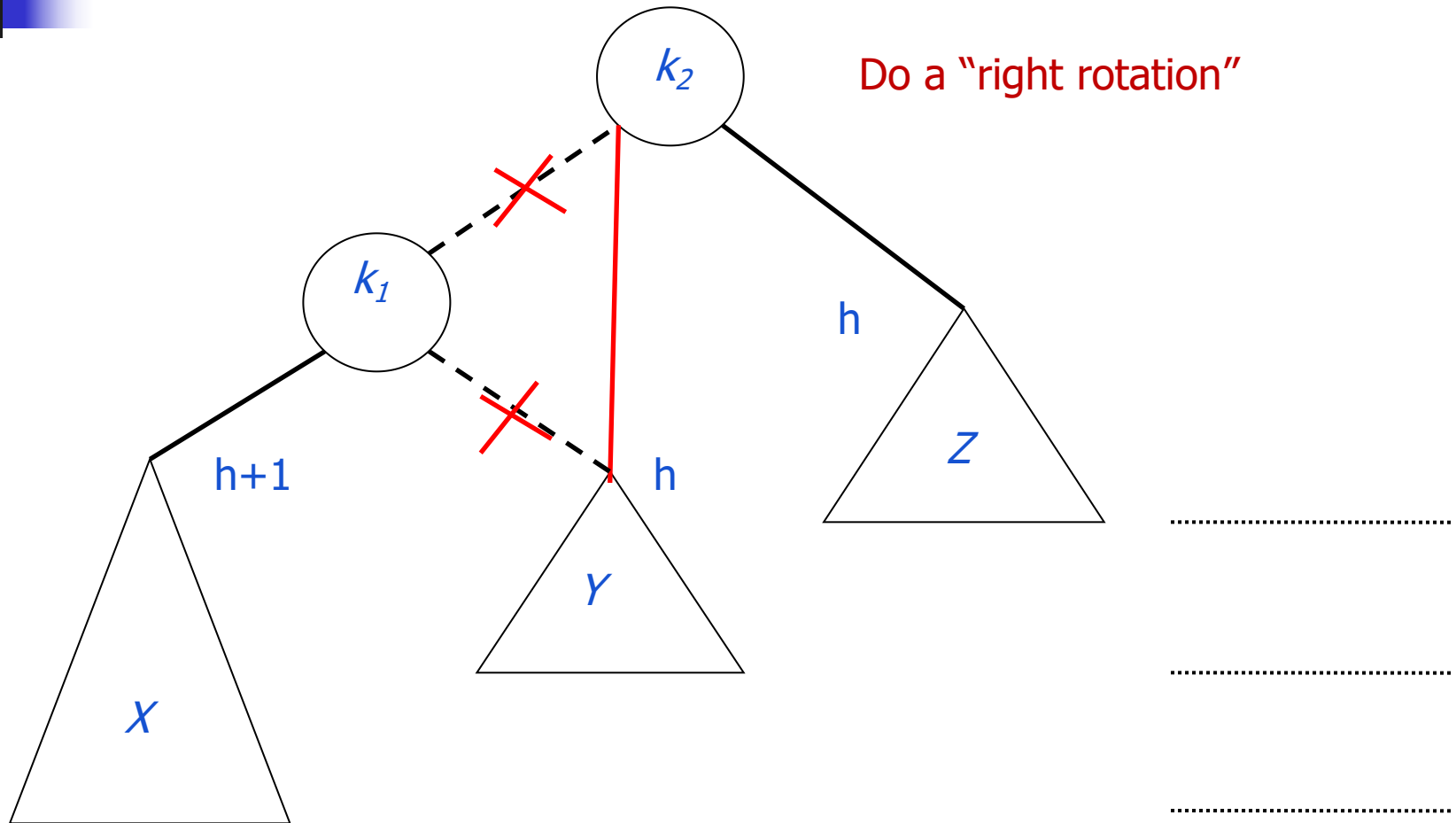
# AVL Insertion: Single Rotation



Do a "right rotation"

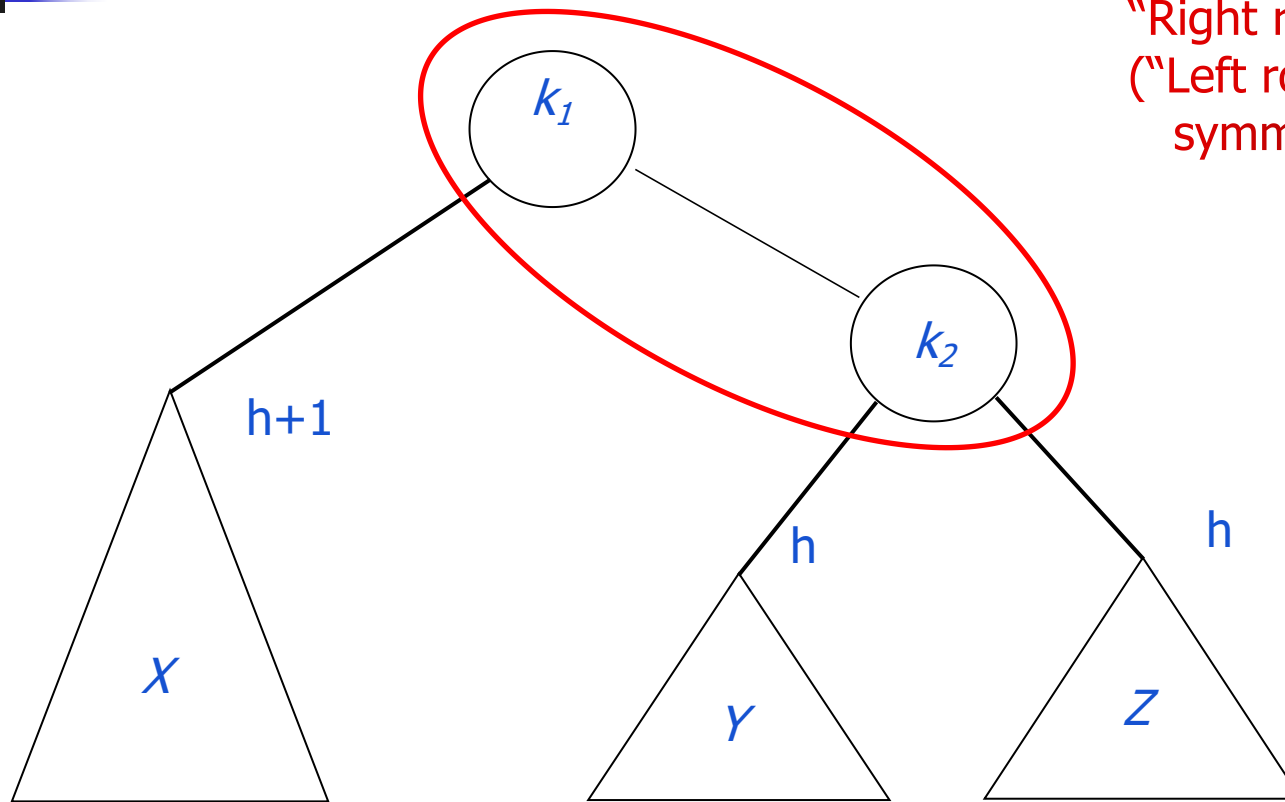
.....  
.....  
.....

# Single right rotation





# Single Rotation Completed

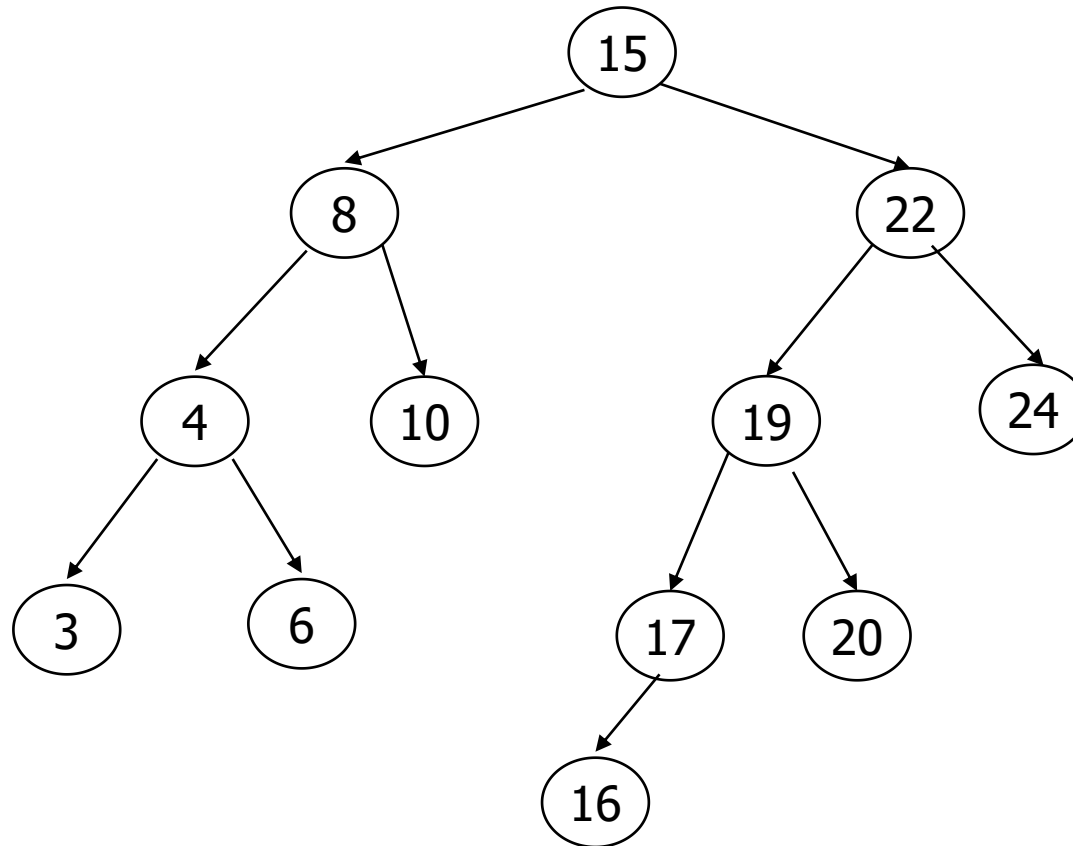


"Right rotation" done!  
("Left rotation" is mirror  
symmetric)

AVL property has been restored!

# Example: Single Rotation: LL

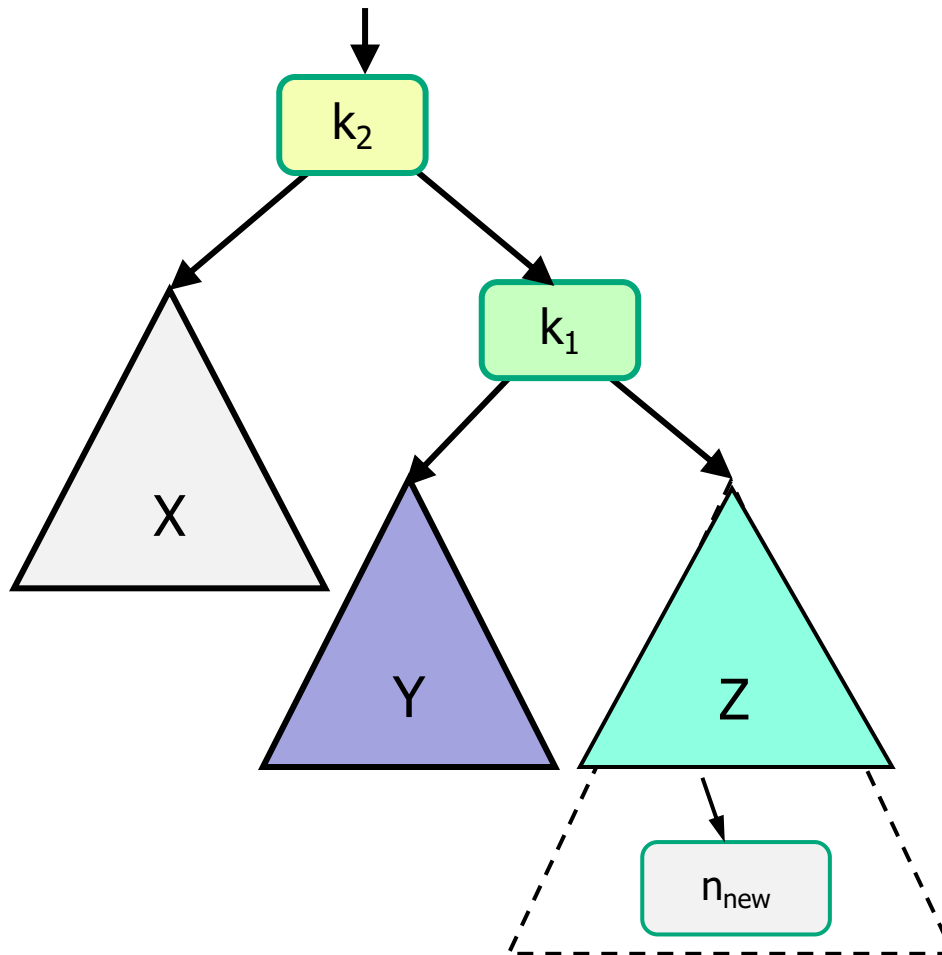
## insert(16)





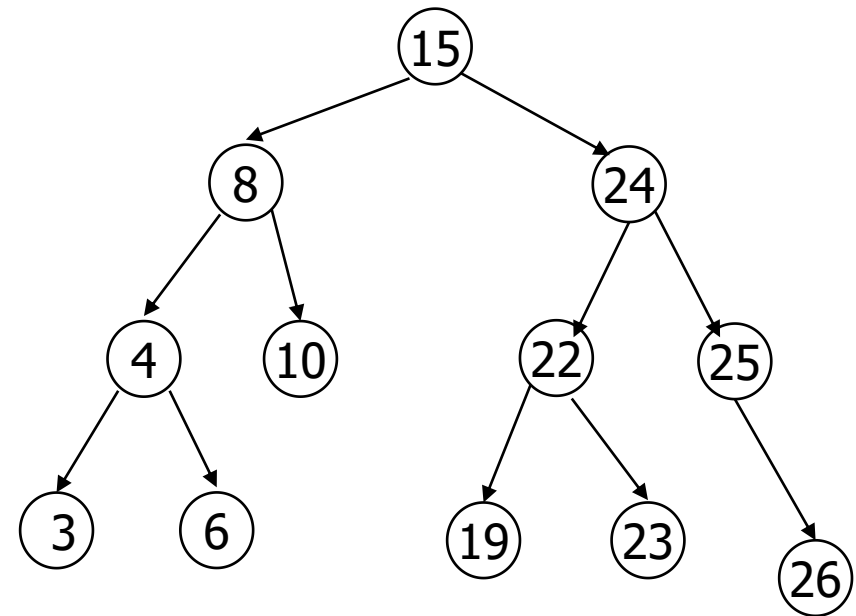
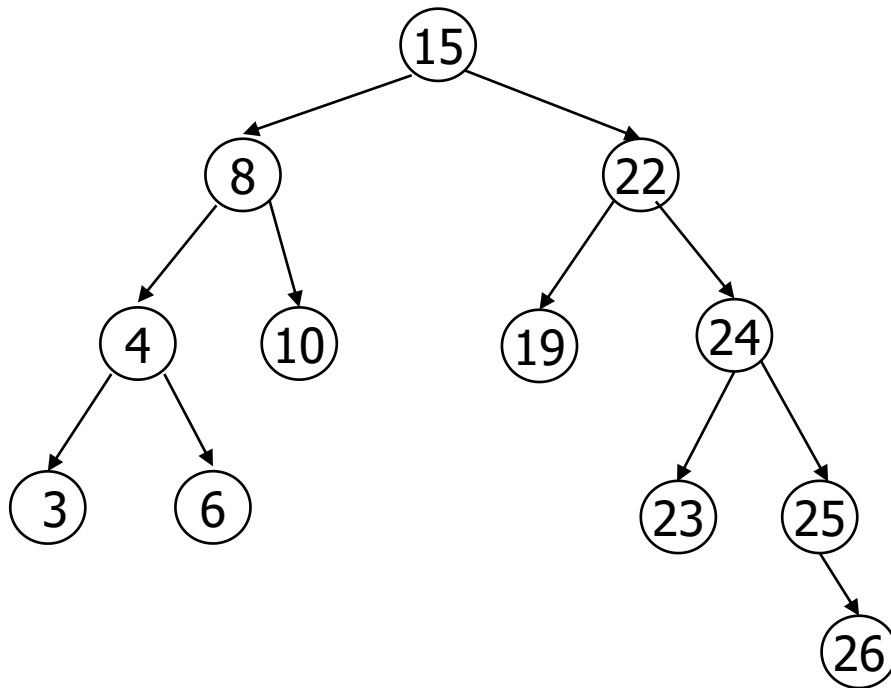
# Single Rotations: RR

---



# Example: Single Rotation: RR

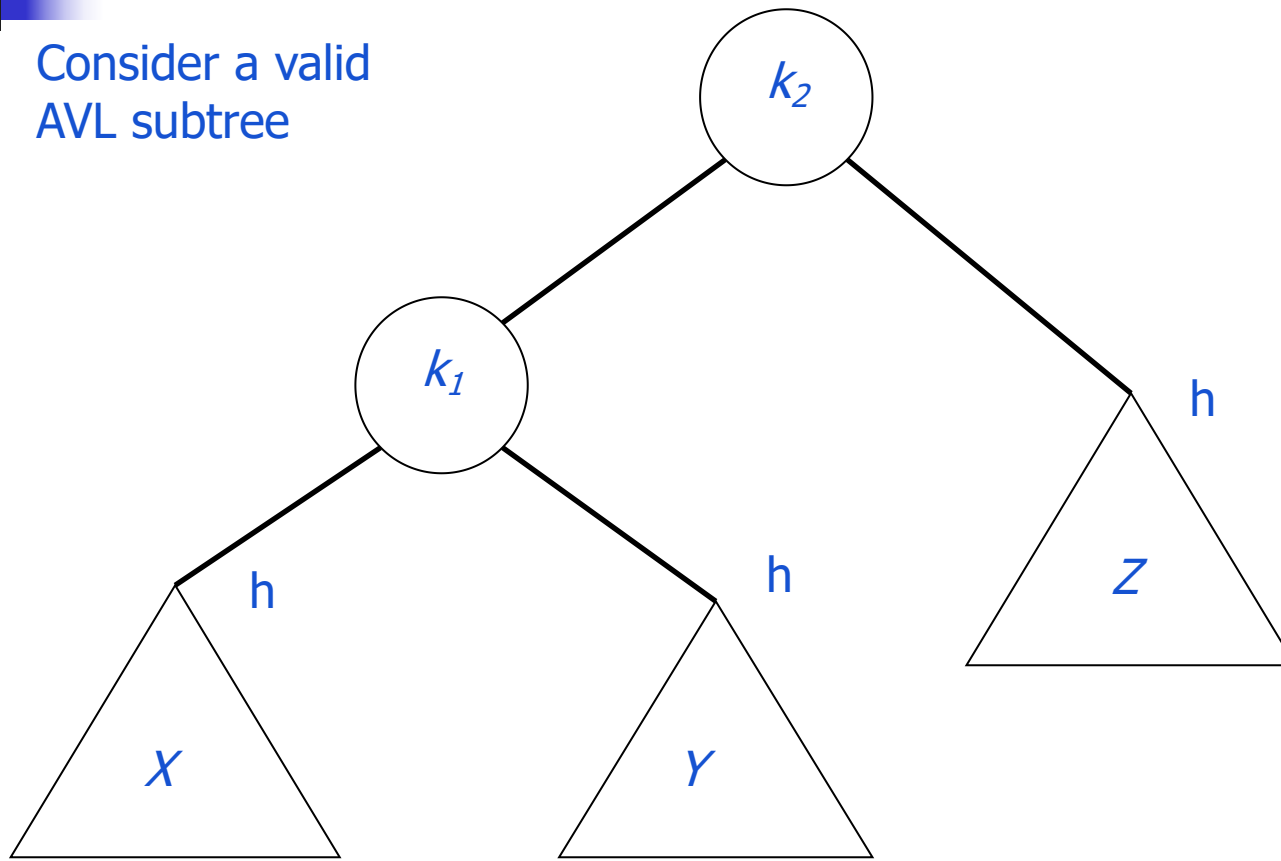
## insert(26)





# AVL Insertion: Double Rotation

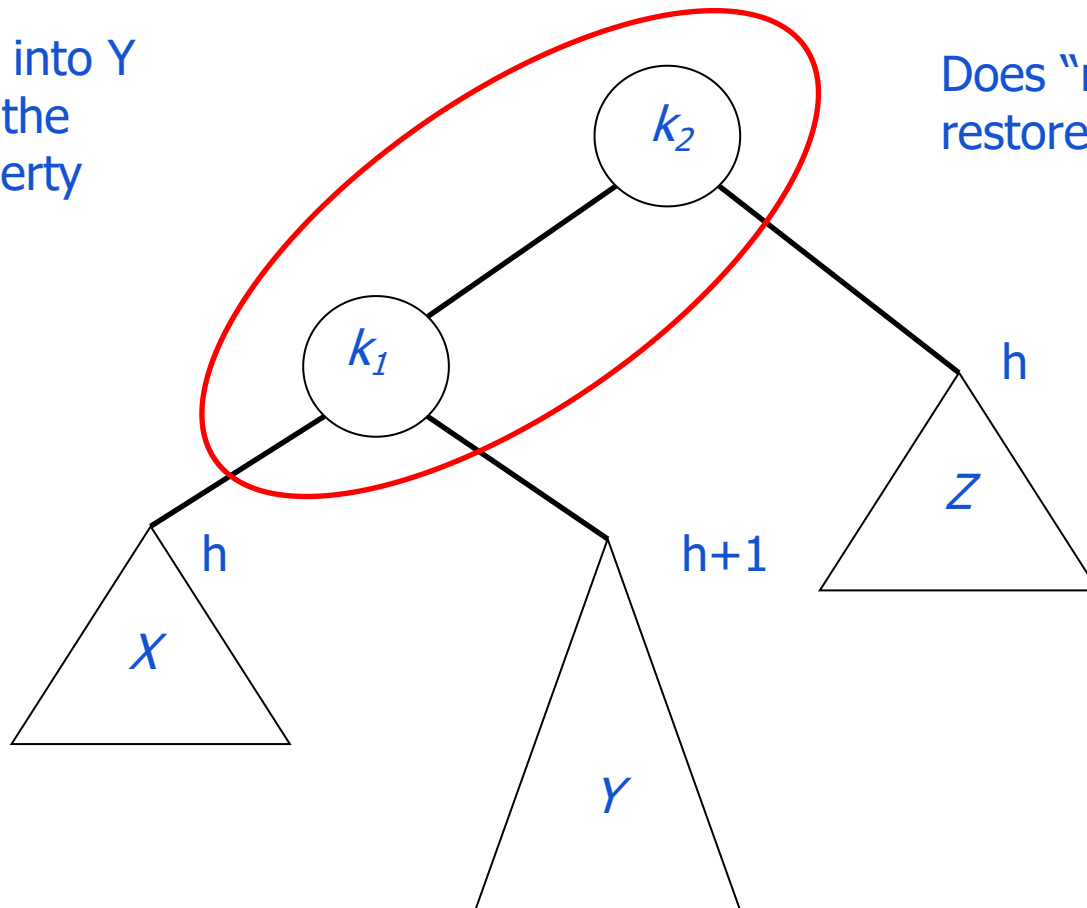
Consider a valid  
AVL subtree



# AVL Insertion: Double Rotation

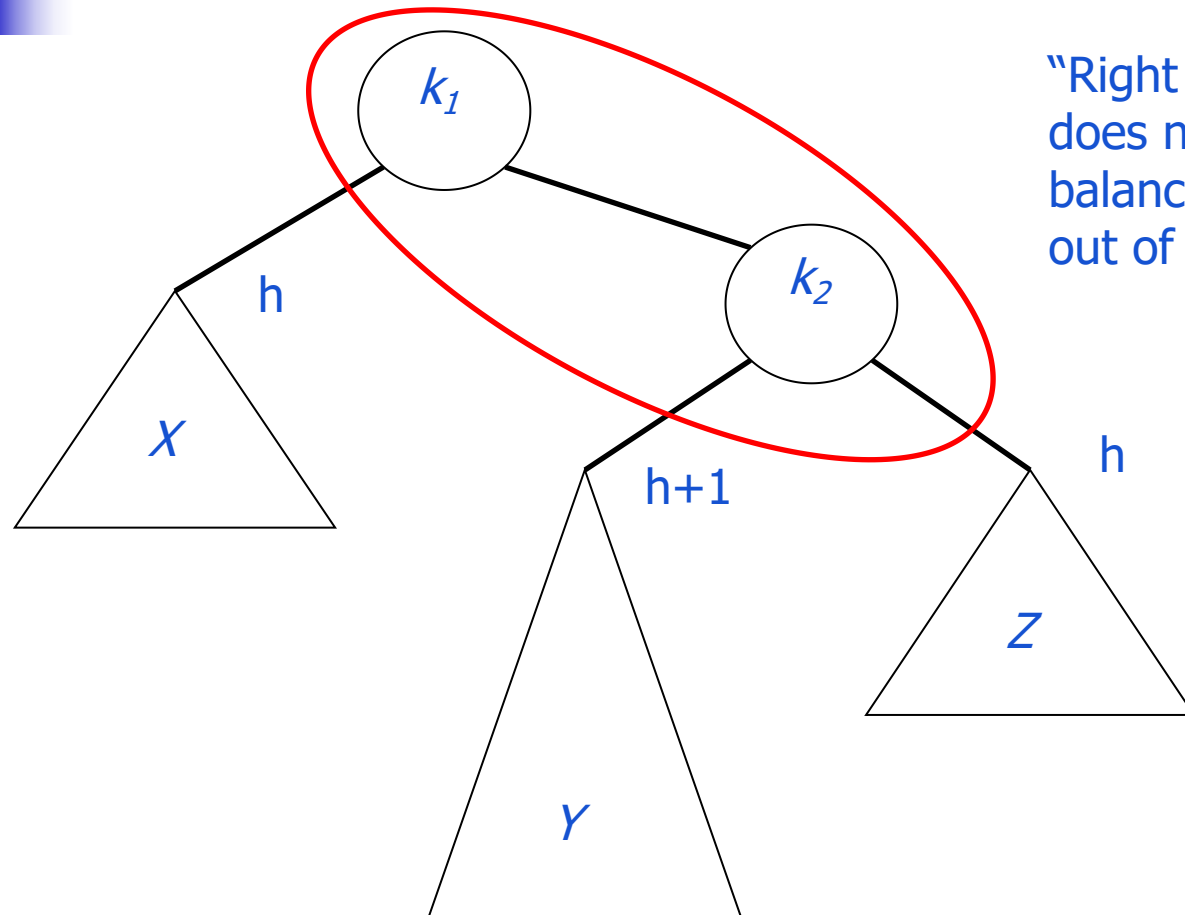
Inserting into Y  
destroys the  
AVL property  
at node j

Does "right rotation"  
restore balance?





# AVL Insertion: Inside Case

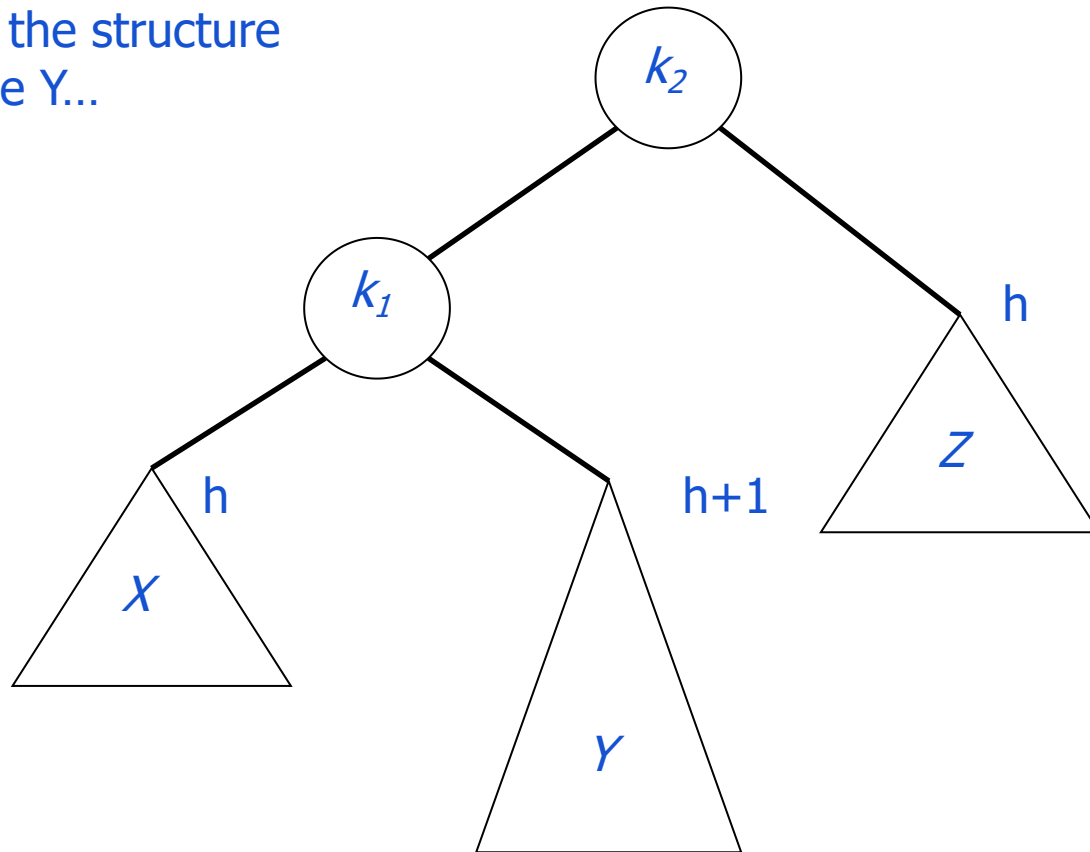


“Right rotation”  
does not restore  
balance... now  $k_1$  is  
out of balance



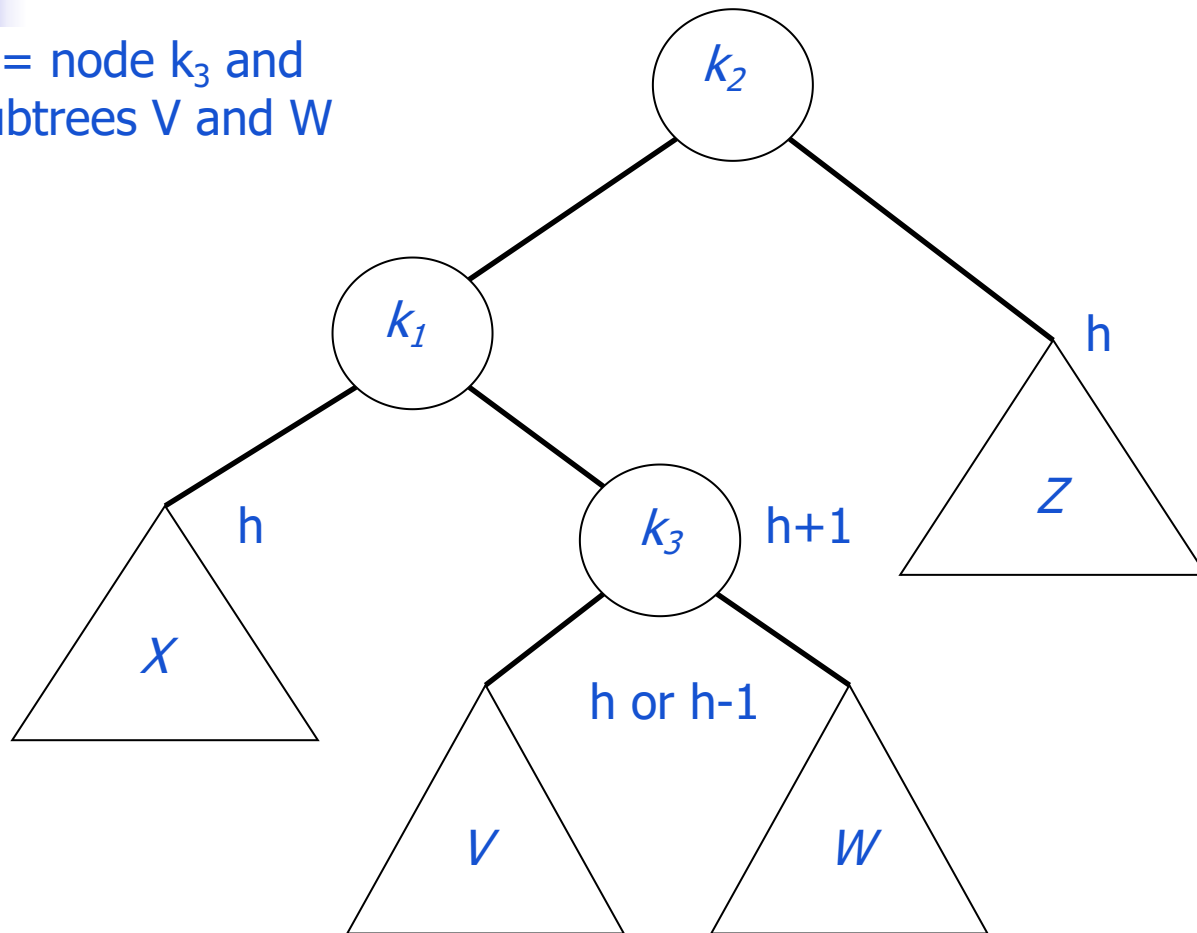
# AVL Insertion: Double Rotation

Consider the structure of subtree Y...

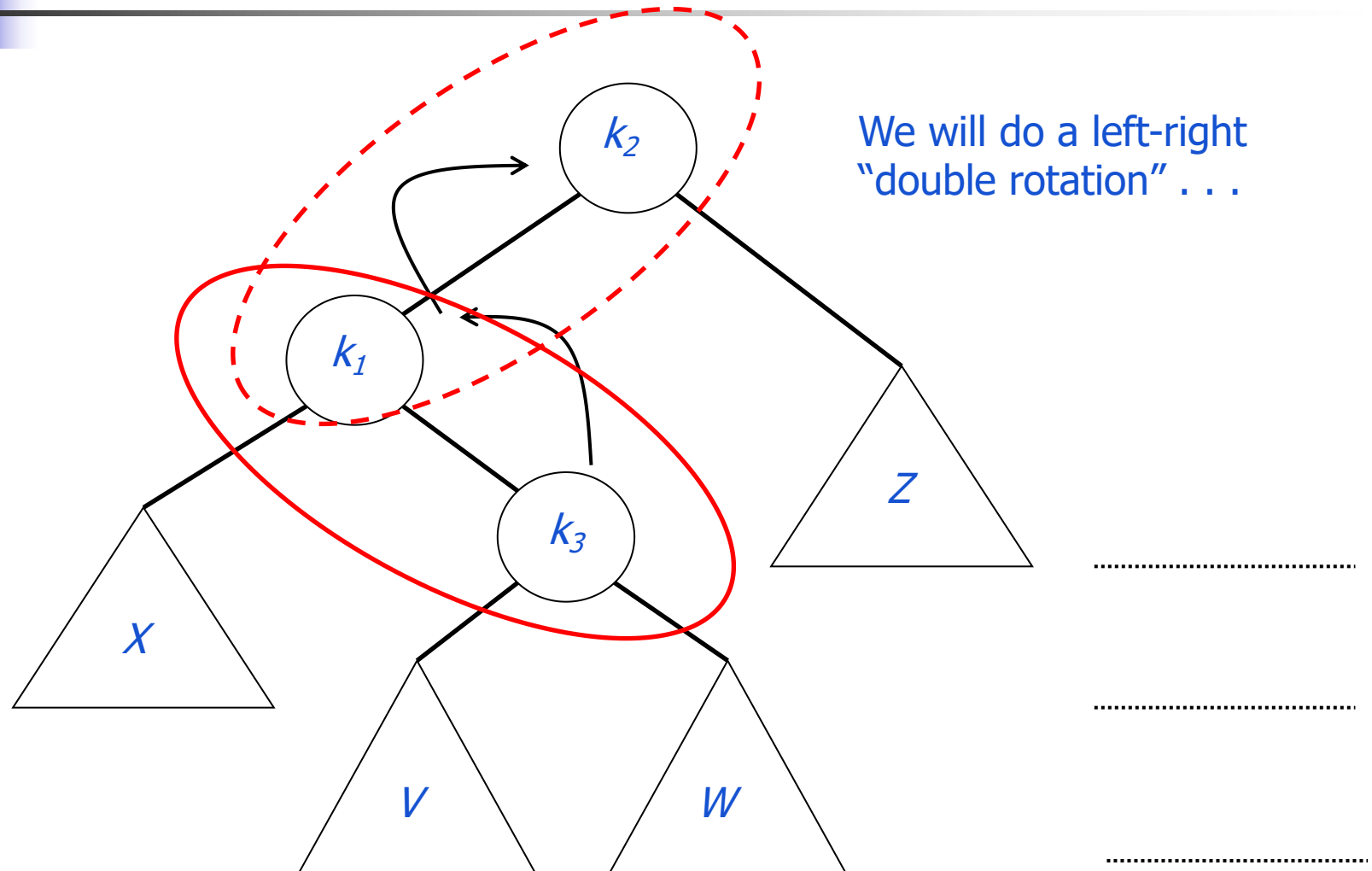


# AVL Insertion: Double Rotation

Y = node  $k_3$  and  
subtrees V and W

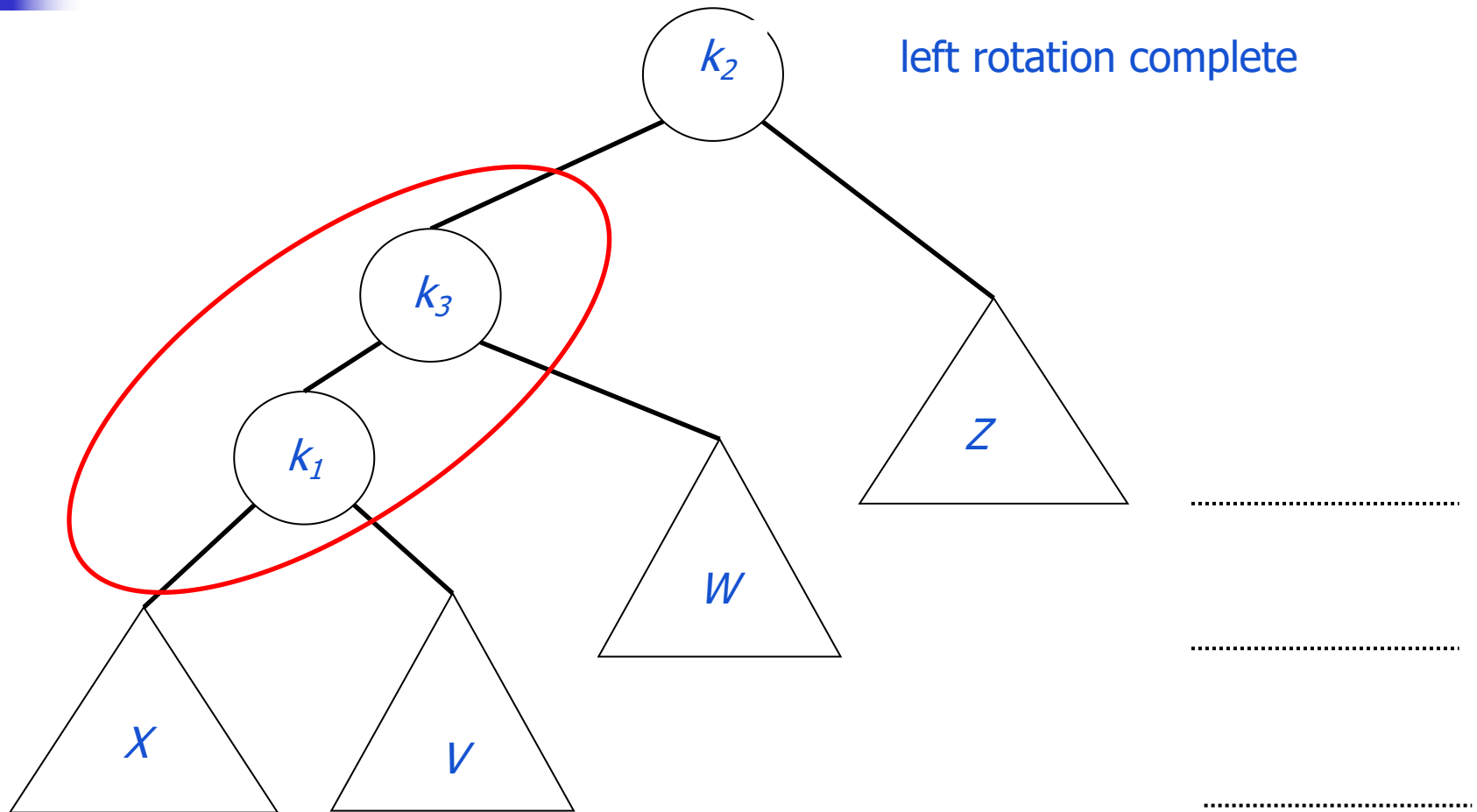


# AVL Insertion: Double Rotation



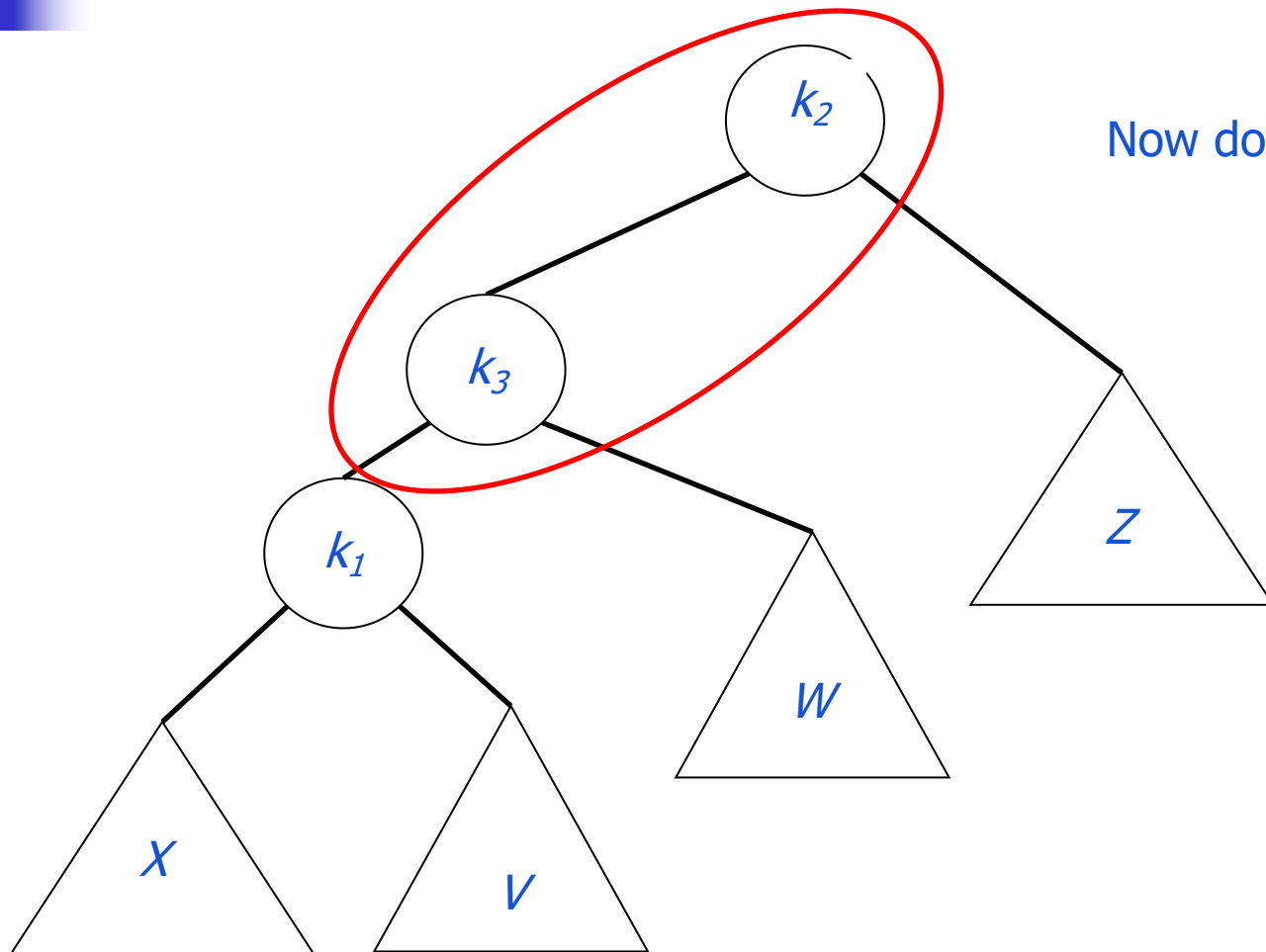


# Double Rotation: First Rotation





# Double rotation: Second Rotation



Now do a right rotation

.....

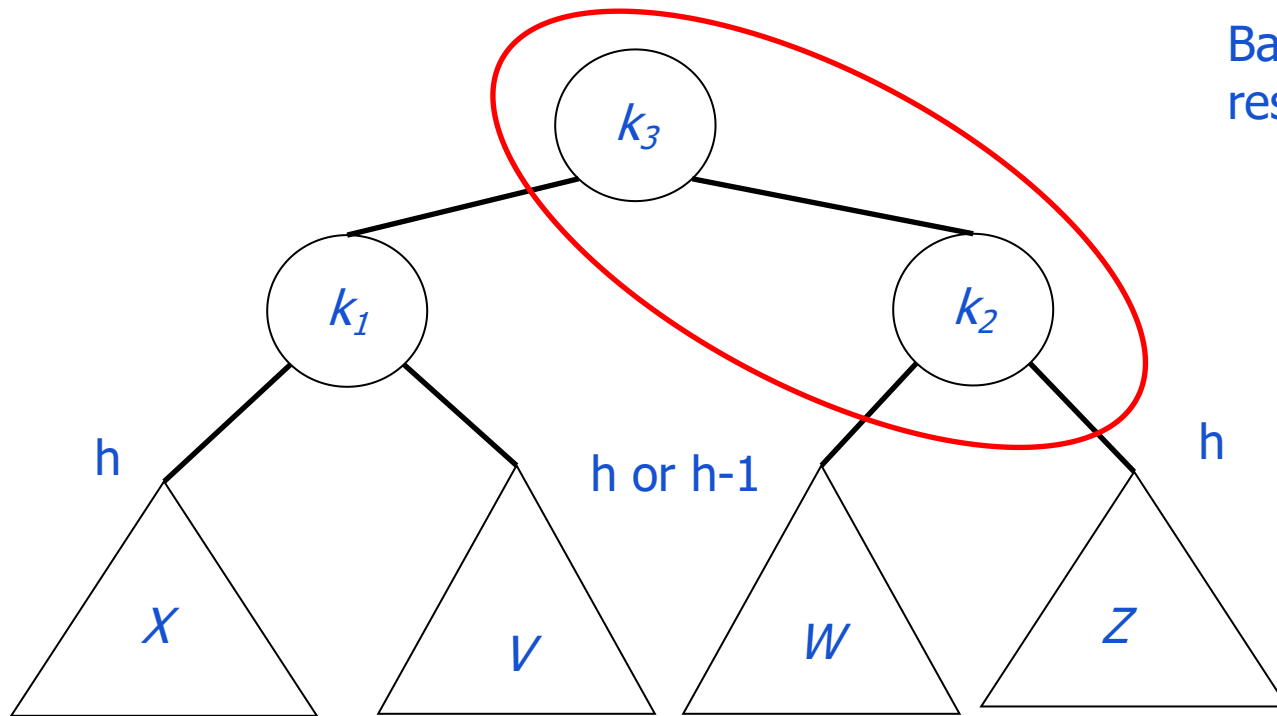
.....

.....

# Double Rotation: Second Rotation

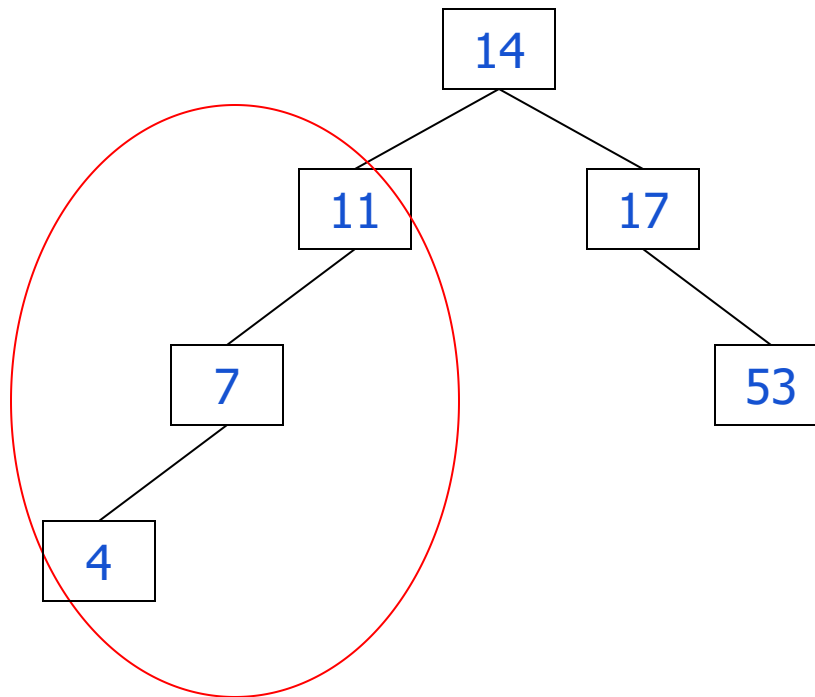
right rotation complete

Balance has been restored



## AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

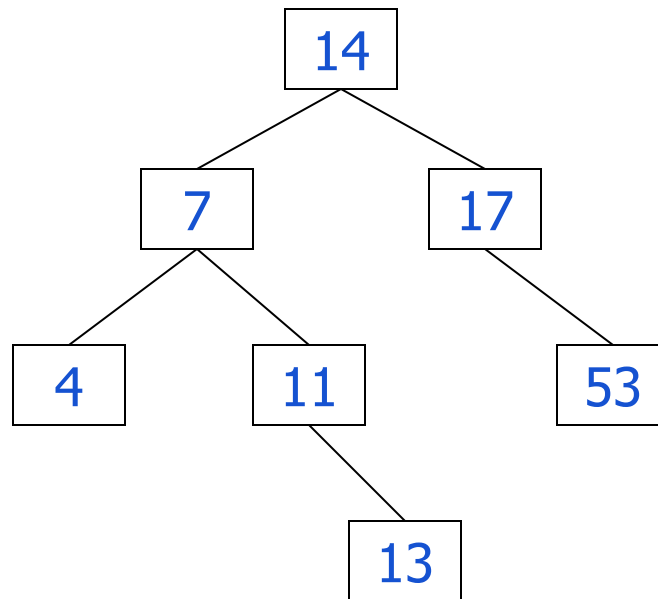






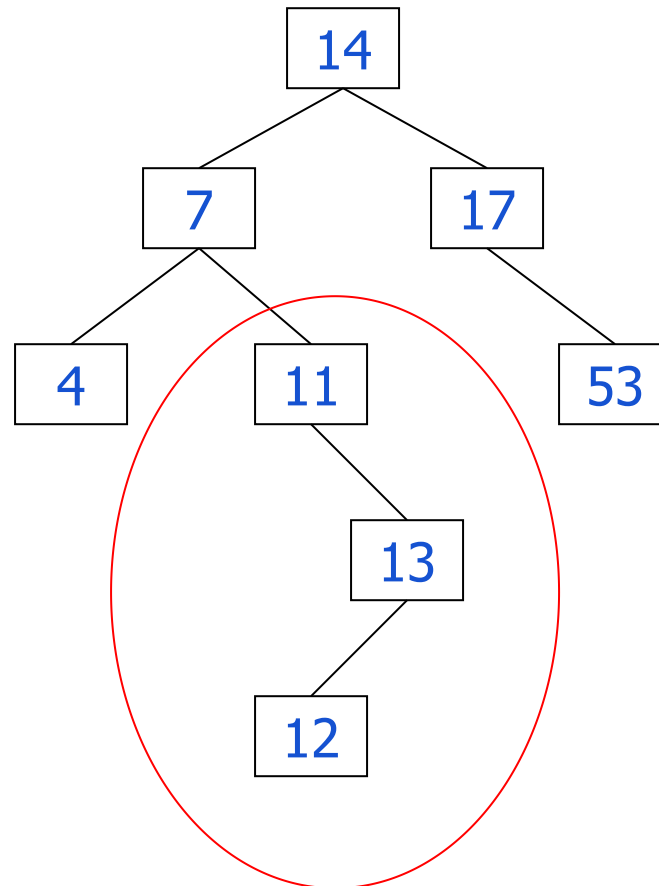
## AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



## AVL Tree Example:

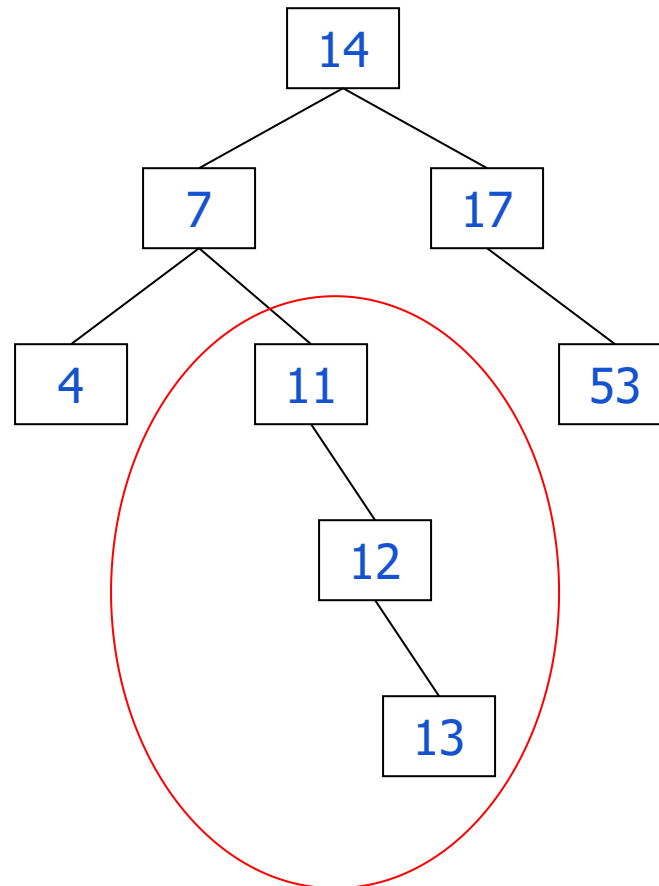
- Now insert 12





## AVL Tree Example:

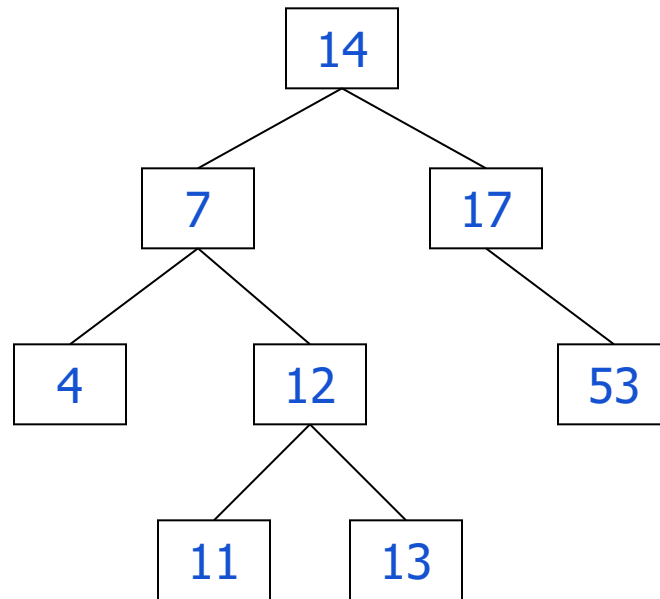
- Now insert 12





## AVL Tree Example:

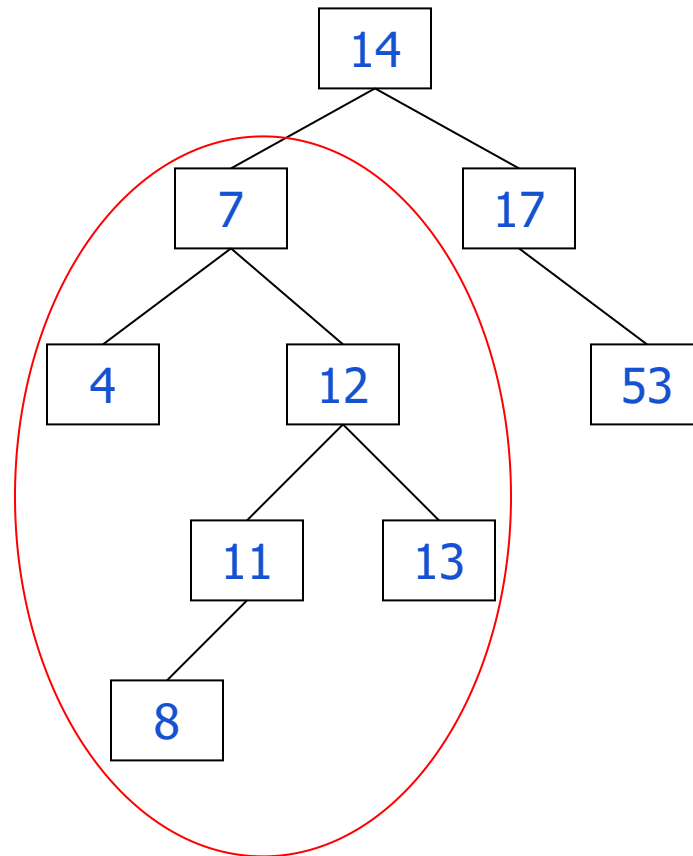
- Now the AVL tree is balanced.





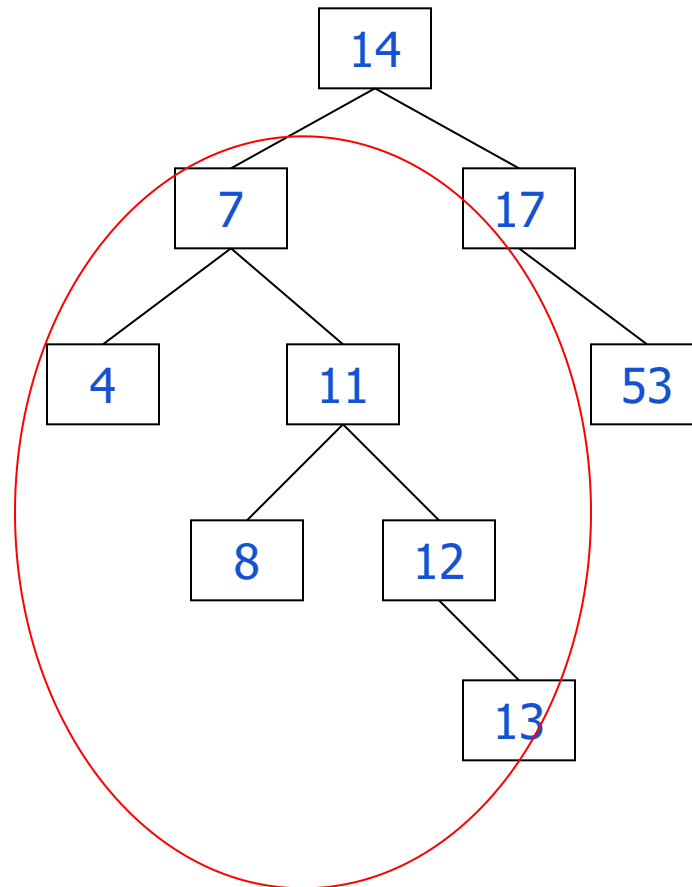
## AVL Tree Example:

- Now insert 8



## AVL Tree Example:

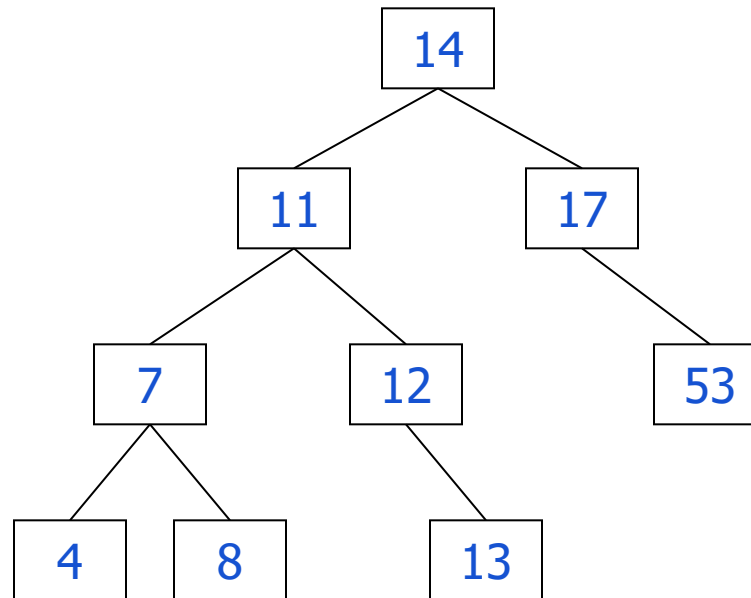
- Now insert 8





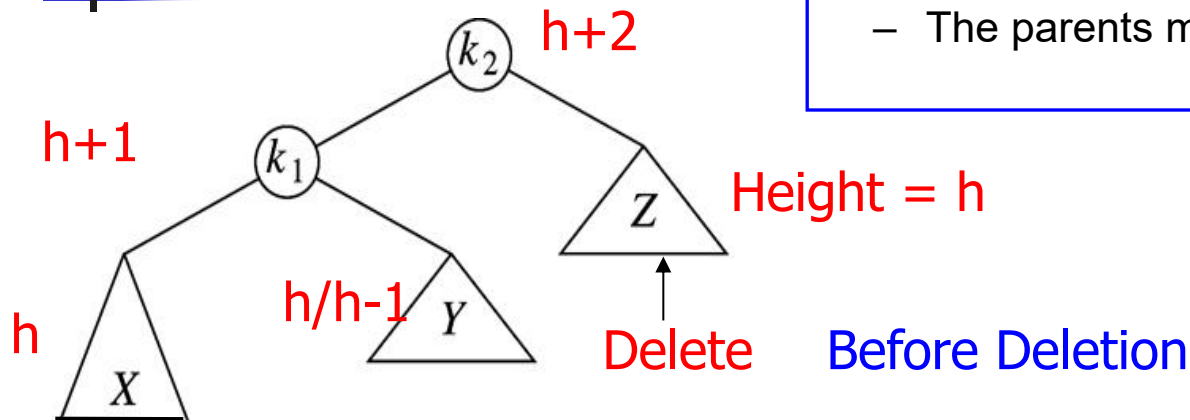
## AVL Tree Example:

- Now the AVL tree is balanced.
- 

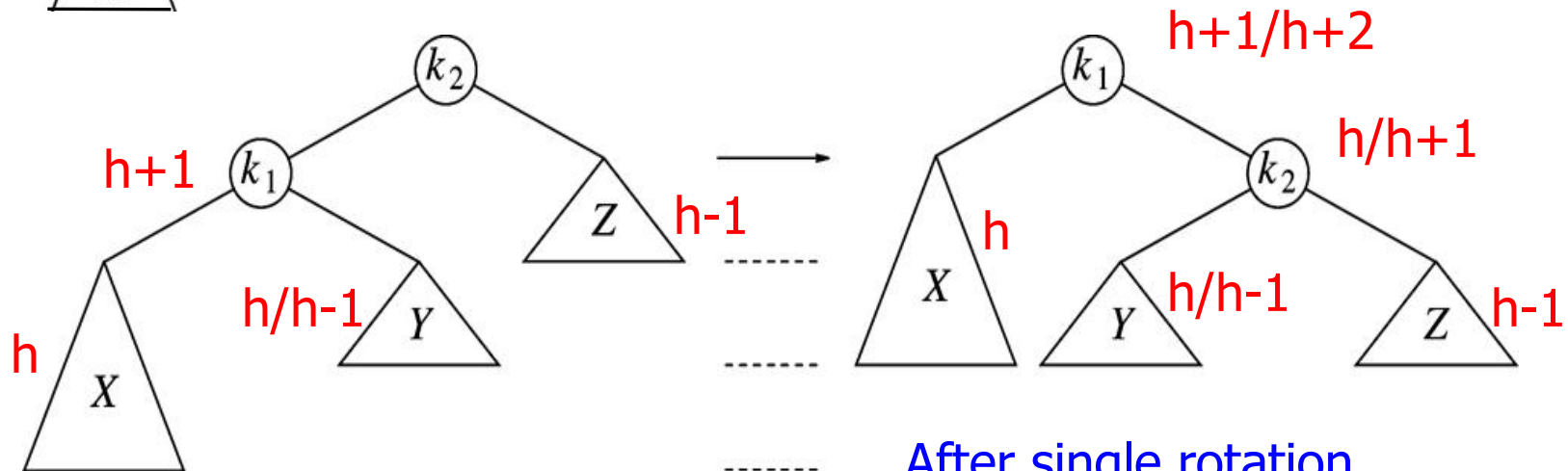


# Delete: Single Rotation

- Consider deepest unbalanced node
  - Case 1: Left child's left side is too high
  - Case 4: Right child's right side is too high
  - The parents may need to be recursively rotated



Before Deletion



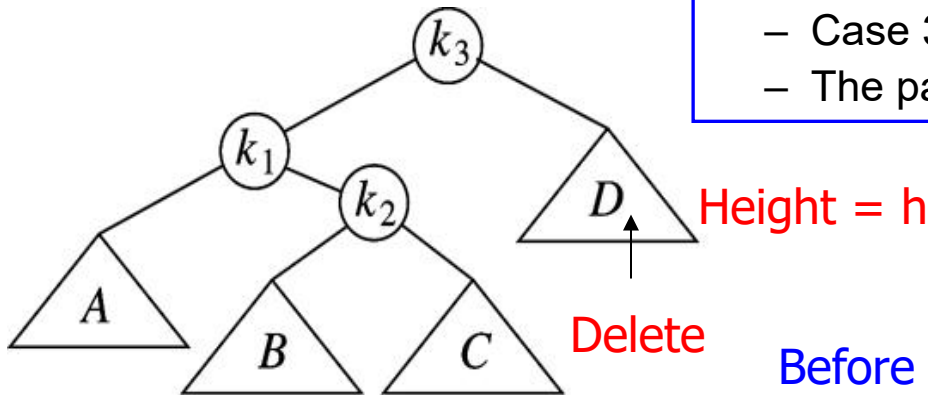
After single rotation

After delete

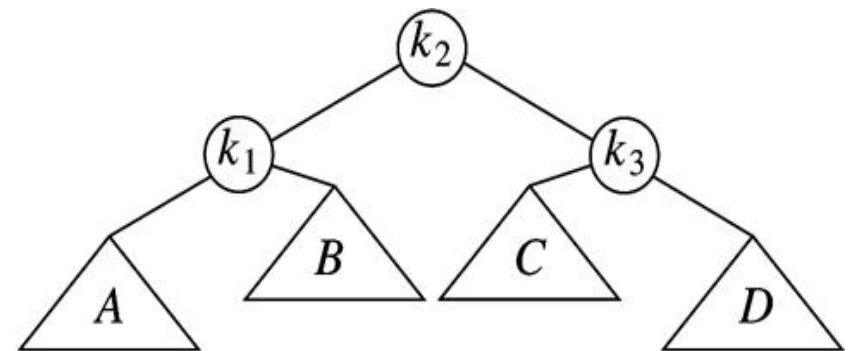
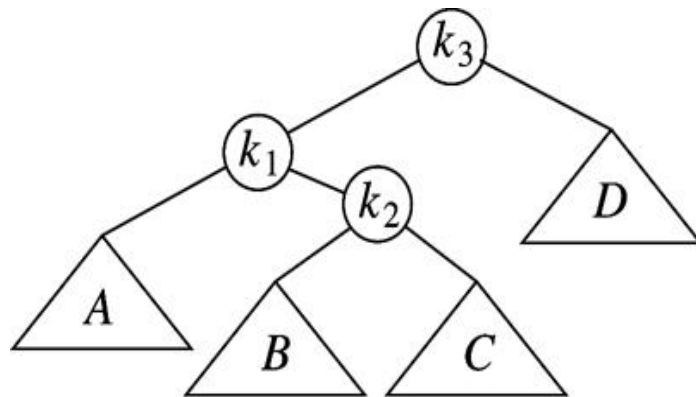


# Delete: Double Rotation

- Consider deepest unbalanced node
  - Case 2: Left child's right side is too high
  - Case 3: Right child's left side is too high
  - The parents may need to be recursively rotated



Determine all heights

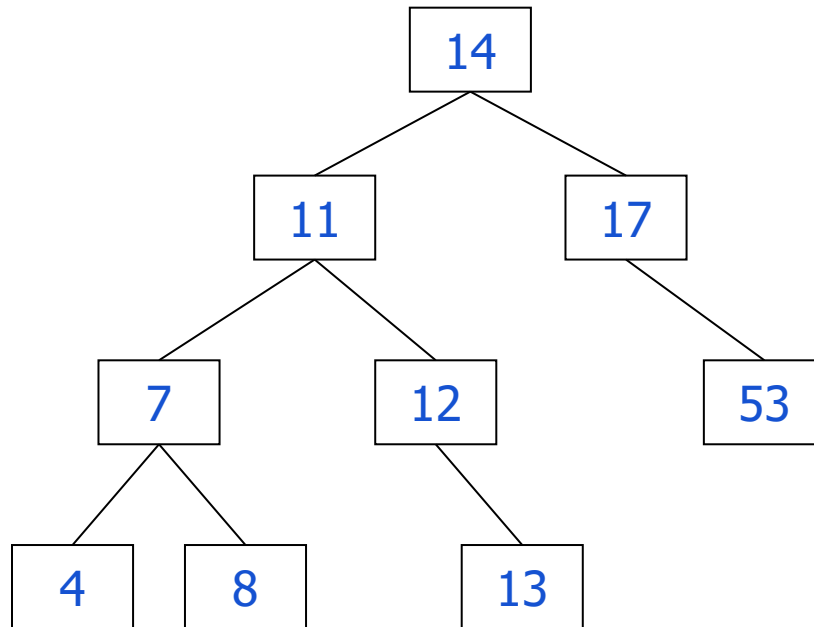




# AVL Tree Example

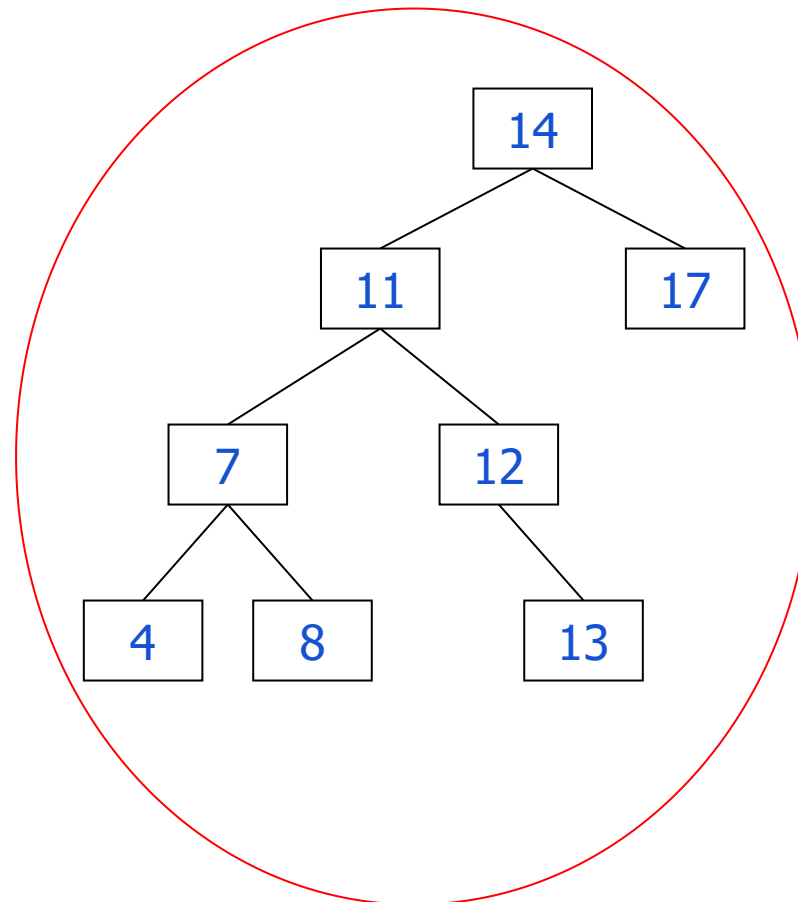
---

Now remove 53



# AVL Tree Example

Now remove 53, unbalanced

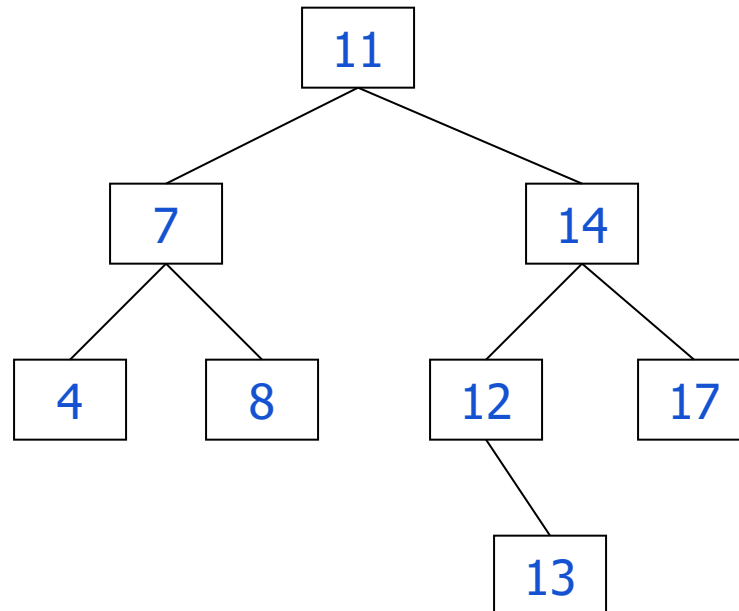




# AVL Tree Example

---

Balanced!    Remove 11

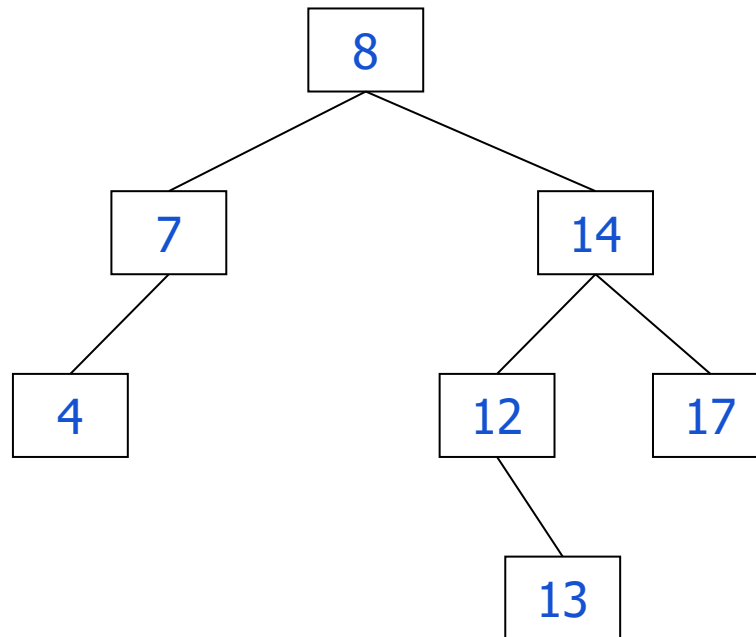




# AVL Tree Example

---

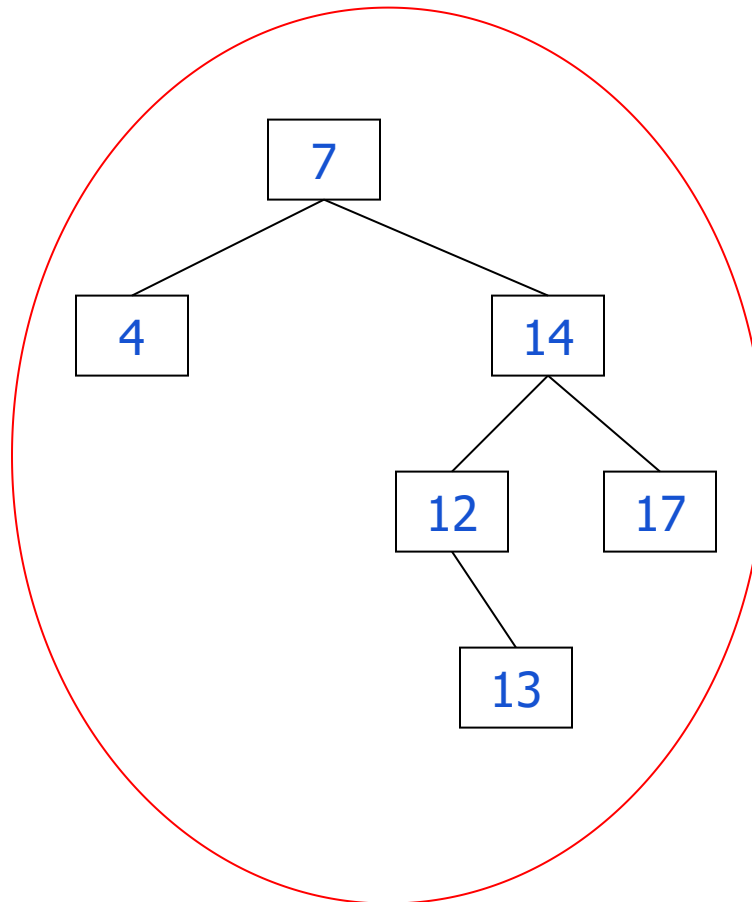
Remove 11, replace it with the largest in its left branch





# AVL Tree Example

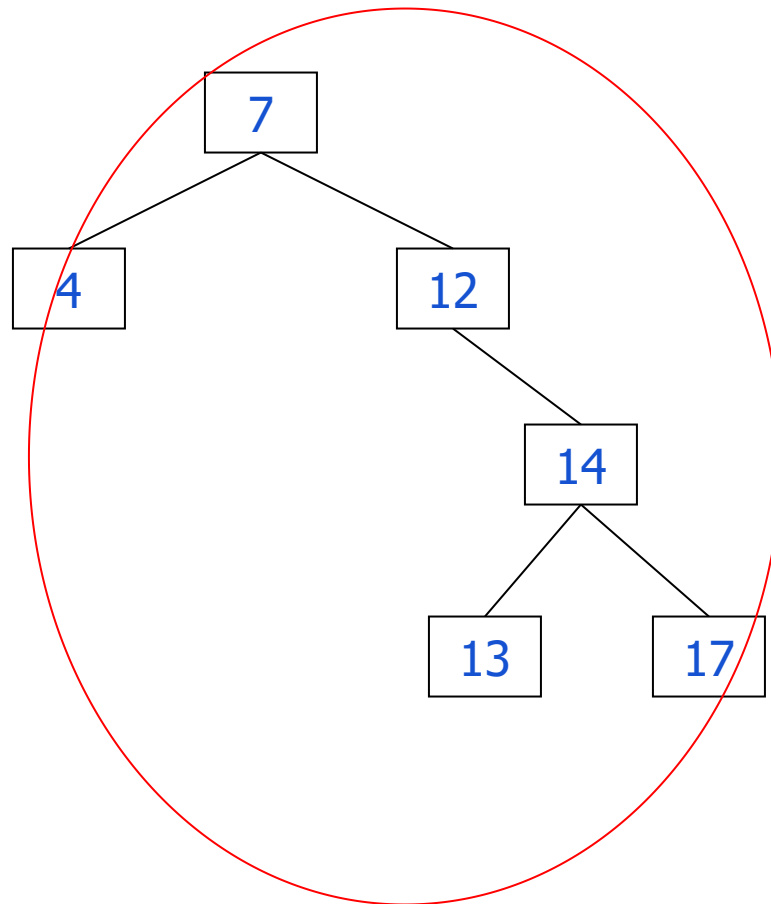
Remove 8, unbalanced





# AVL Tree Example

Remove 8, unbalanced

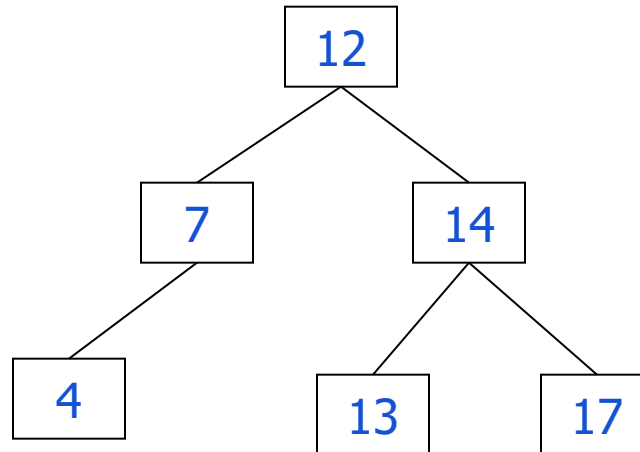




# AVL Tree Example

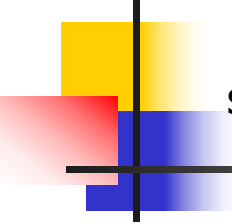
---

Balanced!!





# C-code to Implement AVL Tree



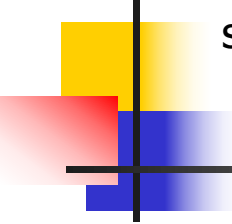
```
struct Node {
    int info;
    struct node *left;
    struct node *right;
    int ht;
};

int height(struct Node *root) {
    if (root == NULL) return -1;
    return root->ht;
}

int max(int x, int y) {
    return (x > y)? x : y;
}

struct Node* createNode(int key) {
    struct node *newNode;
    newNode=(struct Node*) malloc(sizeof(struct Node));
    newNode->info=key;
    newNode->left=NULL;
    newNode->right=NULL;
    newNode->ht=0;
    return(newNode);
}
```

# C-code to Implement AVL Tree

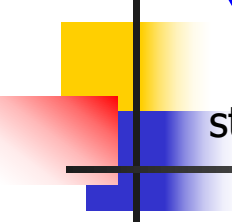


```
struct Node *rightRotate(struct Node *root) {
    struct Node *k1=root->left;
    struct Node *k2=k1->right;
    k1->right = root;
    root->left = k2;
    root->ht=max(height(root->left), height(root->right))+1;
    k1->ht = max(height(k1->left), height(k1->right))+1;
    return k1;
}

struct Node *leftRotate(struct Node *root) {
    struct Node *k1=root->right;
    struct Node *k2 = k1->left;
    k1->left=root;
    root->right=k2;
    root->ht=max(height(root->left), height(root->right))+1;
    k1->ht=max(height(k1->left), height(k1->right))+1;
    return k1;
}

int getBalance(struct Node *root) {
    if (root == NULL) return 0;
    return height(root->left)-height(root->right);
}
```

# C-code to Implement AVL Tree



```
struct Node* insert(struct Node* root, int key) {  
    int bal;  
    if (root == NULL) return(createNode(key));  
    if (key < root->info)  
        node->left=insert(root->left, key);  
    else  
        root->right=insert(root->right, key);  
    root->ht=max(height(root->left), height(root->right)) + 1;  
    bal=getBalance(root);  
    if (bal > 1 && key < root->left->info)  
        return rightRotate(root);  
    if (bal < -1 && key > root->right->info)  
        return leftRotate(root);  
    if (bal > 1 && key > root->left->info) {  
        root->left=leftRotate(root->left);  
        return rightRotate(root);  
    }  
    if (bal < -1 && key < root->right->info) {  
        root->right=rightRotate(root->right);  
        return leftRotate(root);  
    }  
    return root;  
}
```



# C-code to Implement AVL Tree

```
void preOrder(struct Node *root) {  
    if(root != NULL) {  
        printf("%d ", root->info);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

```
int main() {  
    struct node *root = NULL;  
    root = insert(root, 10);  
    root = insert(root, 20);  
    root = insert(root, 30);  
    root = insert(root, 40);  
    root = insert(root, 50);  
    root = insert(root, 25);  
    printf("Pre order traversal of the constructed AVL tree is \n");  
    preOrder(root);  
    return 0;  
}
```



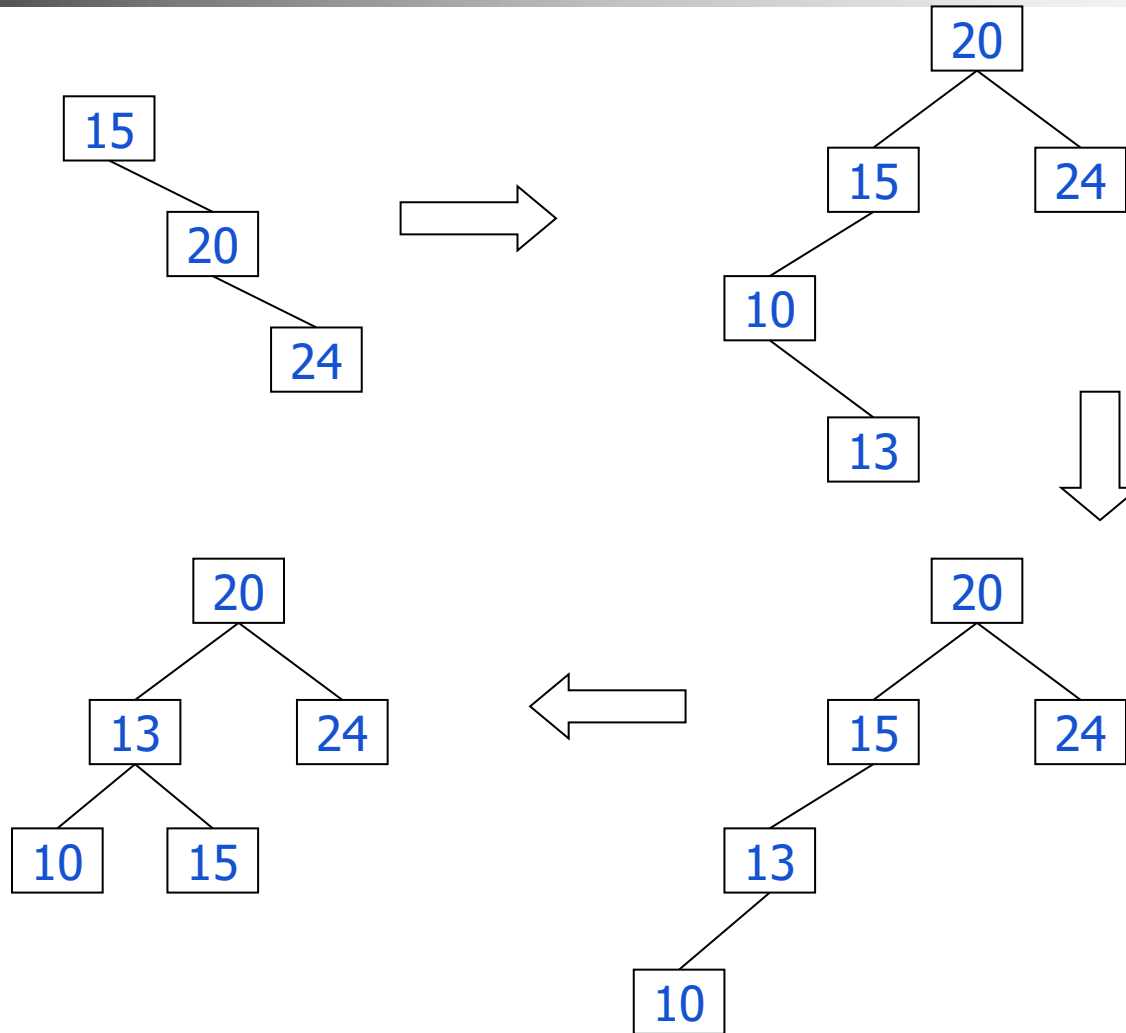
# Exercises

---

- Build an AVL tree with the following values:  
15, 20, 24, 10, 13, 7, 30, 36, 25

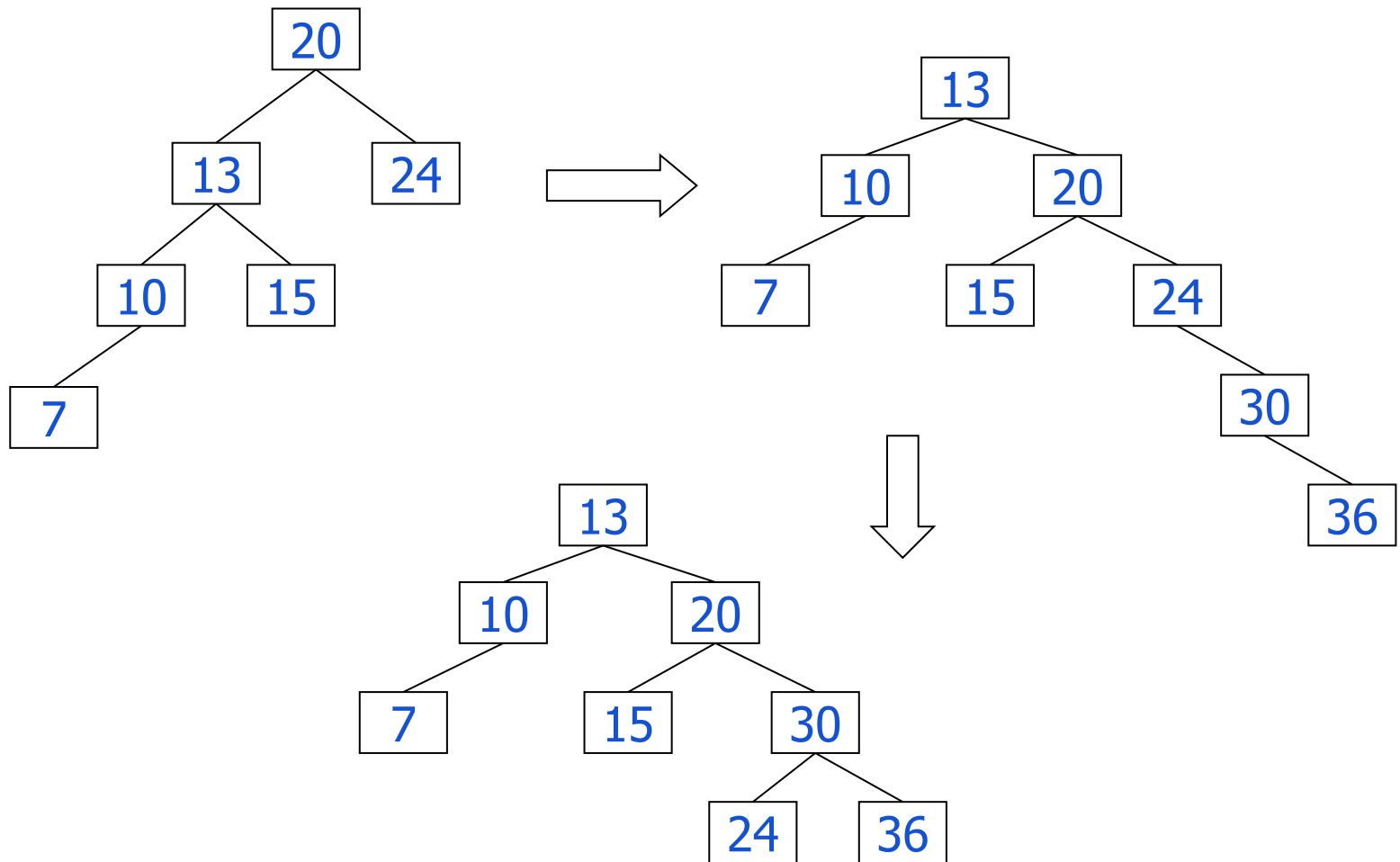


15, 20, 24, 10, 13, 7, 30, 36, 25



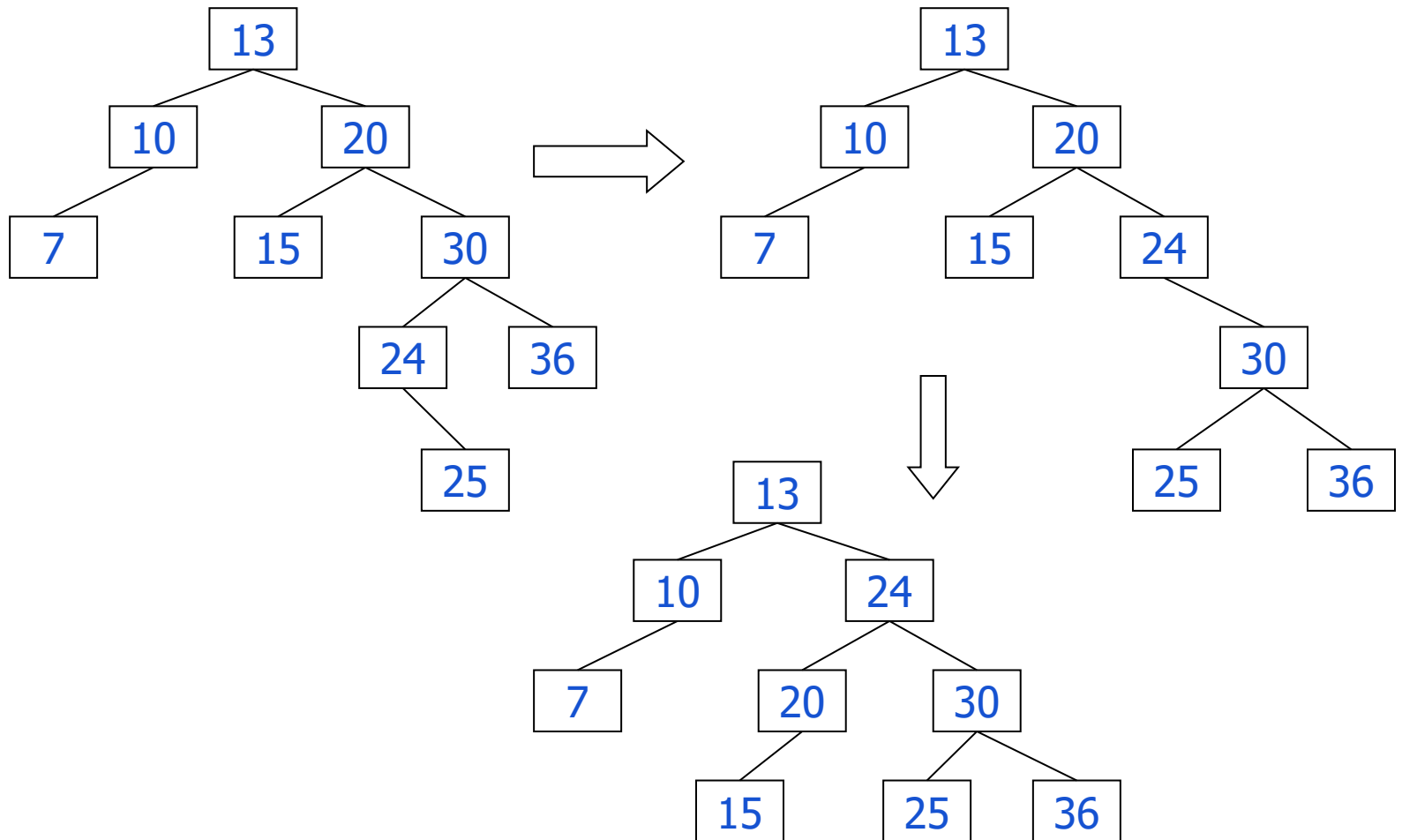


15, 20, 24, 10, 13, 7, 30, 36, 25





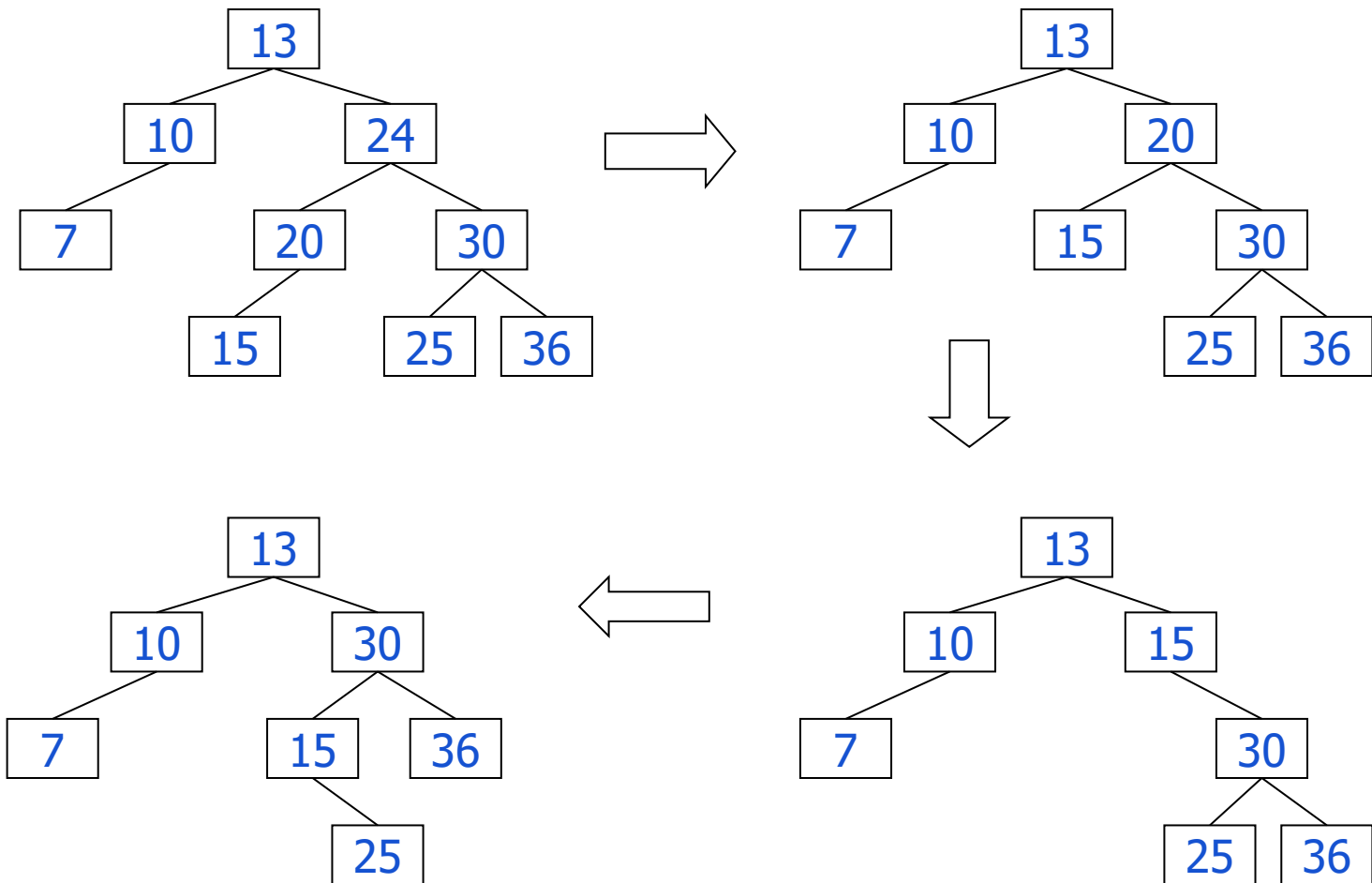
15, 20, 24, 10, 13, 7, 30, 36, 25







Remove 24 and 20 from the AVL tree.





# C-code to Implement AVL Tree

```
struct Node* deleteNode(struct Node* root, int key) {
    int bf;
    struct Node* temp;
    if (root == NULL) return root;
    if (key < root->info)
        root->left=deleteNode(root->left, key);
    else if (key > root->info)
        root->right=deleteNode(root->right, key);
    else {
        if((root->left==NULL)||(root->right==NULL)) {
            temp=root->left?root->left:root->right;
            if(temp == NULL) {
                temp = root;
                root = NULL;
            }
            else root->info=temp->info; (or *root = *temp)//Copy the content of temp to root
            free(temp);
        }
        else {
            temp=maxLSTree(root->left);
            root->info=temp->info; // Copy the inorder predecessor's data to this node
            root->left=deleteNode(root->left, temp->info); // Delete the inorder predecessor
        }
    }
}
```



# C-code to Implement AVL Tree

---

```
if (root == NULL) return root;
root->ht=max(height(root->left), height(root->right))+1;
bf=getBalance(root);
if(bf > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);
if(bf > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
}
if(bf < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);
if(bf < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
}
return root;
}

struct Node* maxLSTree(struct Node* root) {
    struct Node* curr = root;
    while (curr->right != NULL) curr = curr->right;
    return curr;
}
```



# C-code to Implement AVL Tree

---

```
int getBalance(struct Node *root) {
    if (root == NULL) return 0;
    return height(root->left)-height(root->right);
}
```

```
int height(struct Node *root) {
    if (root == NULL) return 0;
    return root->ht;
}
```

```
void preOrder(struct Node *root) {
    if(root != NULL) {
        printf("%d ", root->info);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
int main() {
    struct Node *root = NULL;
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);
    printf("\nPre order traversal after
deletion of 10 \n");
    preOrder(root);
    return 0;
}
```



# Pros and Cons of AVL Trees

---

## Arguments for AVL trees:

1. Search is  $O(\log n)$  since AVL trees are always balanced.
2. Insertion and deletions are also  $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

## Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. [B-trees](#)).



# Threaded Binary Tree

---



# Threaded Binary Tree

---

- Inorder traversal of a Binary tree can either be done using **recursion** or with the use of a **auxiliary stack**.
- The **idea** of threaded binary trees is to **make inorder traversal faster** and do it **without stack** and **without recursion**.
- A binary tree is made threaded by making all right child pointers that would normally be NULL **point to the inorder successor** of the node (if it **exists**).



# Threaded Binary Tree

---

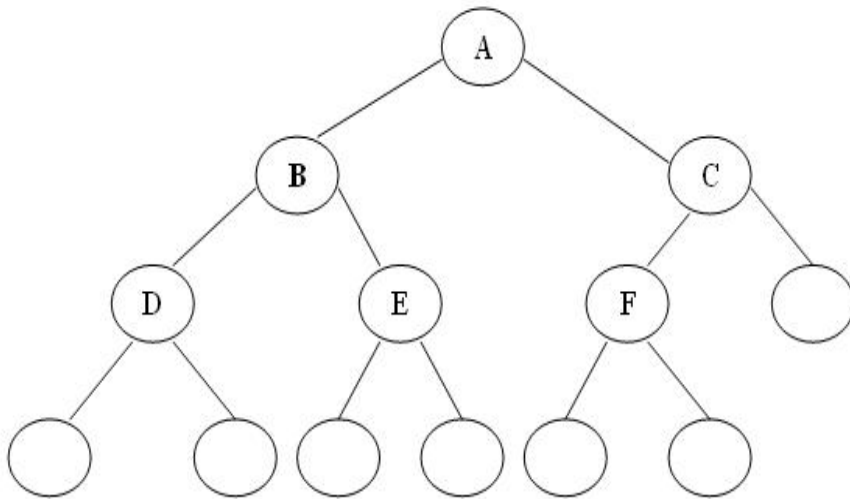
There are two types of threaded binary trees.

- **Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)
- **Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.
  - The predecessor threads are useful for reverse inorder traversal and postorder traversal.
- The threads are also **useful for fast accessing ancestors** of a node.



# Threaded Binary Tree

- In a linked representation of a binary tree, # **NULL** pointers are more than non-**NULL** pointers.
- Consider the following binary tree:

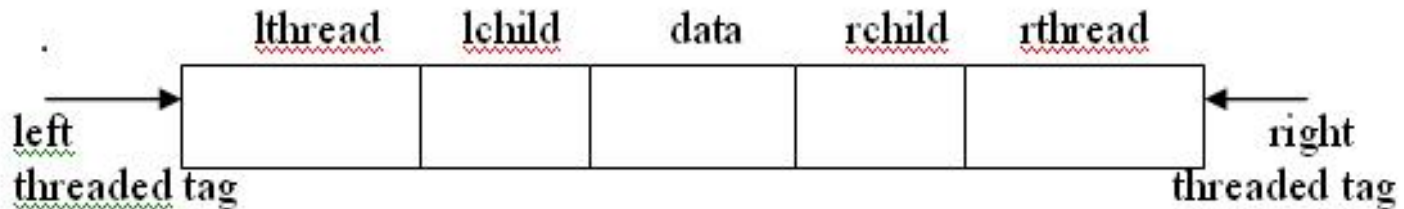


A Binary tree with the null pointers

- In this binary tree, out of 12 pointers there are 7 NULL pointers & 5 non-NULL pointers.
- In a  $n$  node binary tree,  $(n+1)$  NULL pointers out of  $2 \times n$  pointers.
- The threaded binary tree makes effective use of these NULL pointers.
- A. J. perils & C. Thornton jointly proposed idea to make effective use of these NULL pointers.
- All the NULL pointers are replaced by the appropriate pointer values called **threads**.

# Threaded Binary Tree

- And binary tree with such pointers are called **threaded binary tree**.
- In the threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.
- The node representation for a threaded binary tree contains five fields which is shown below:



For any node  $p$ , in a threaded binary tree.

lthread( $p$ )=1 indicates lchild ( $p$ ) is a thread pointer

lthread( $p$ )=0 indicates lchild ( $p$ ) is a normal

rthread( $p$ )=1 indicates rchild ( $p$ ) is a thread

rthread( $p$ )=0 indicates rchild ( $p$ ) is a normal pointer



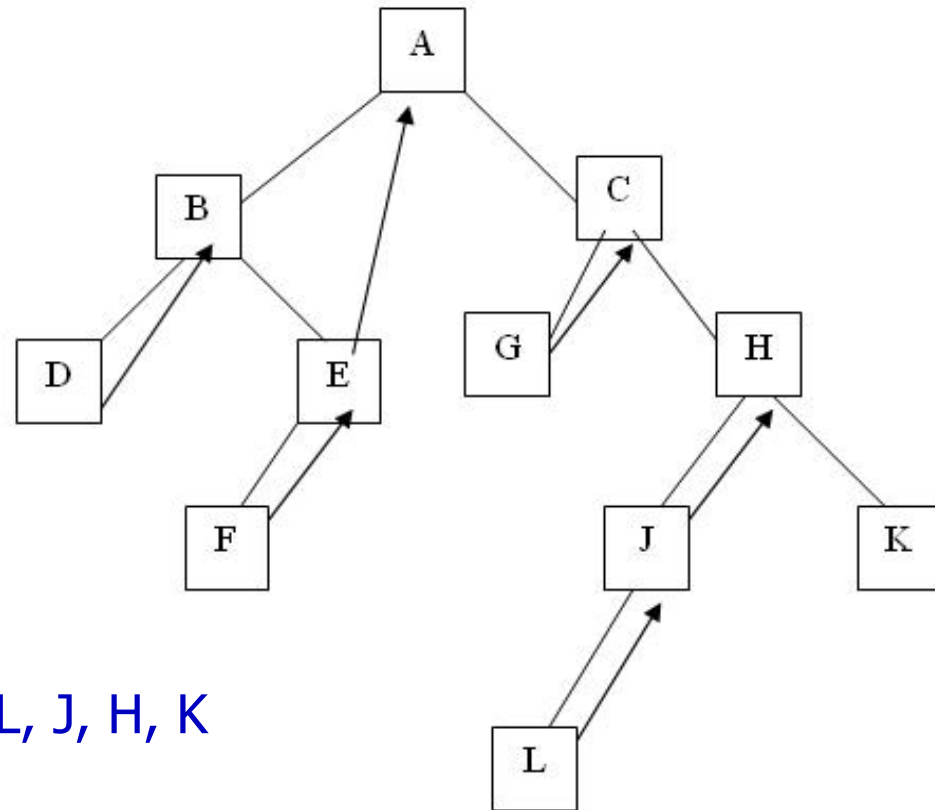
# Threaded Binary Tree

---

- One may choose a **one-way** threading or **single theading** (left-threaded binary tree, right-threaded binary tree), a **two-way threading** or **double threading**.
- Here, the threading will correspond to the inorder traversal of T.

# Threaded Binary Tree: One-Way

- In the one way threading of T, a thread will appear in the right field of a node and will point to the next node in the in-order traversal of T.
- Example of one-way in-order threading.



Inorder of the tree: D, B, F, E, A, G, C, L, J, H, K

One-way inorder threading



# Threaded Binary Tree: Two-Way

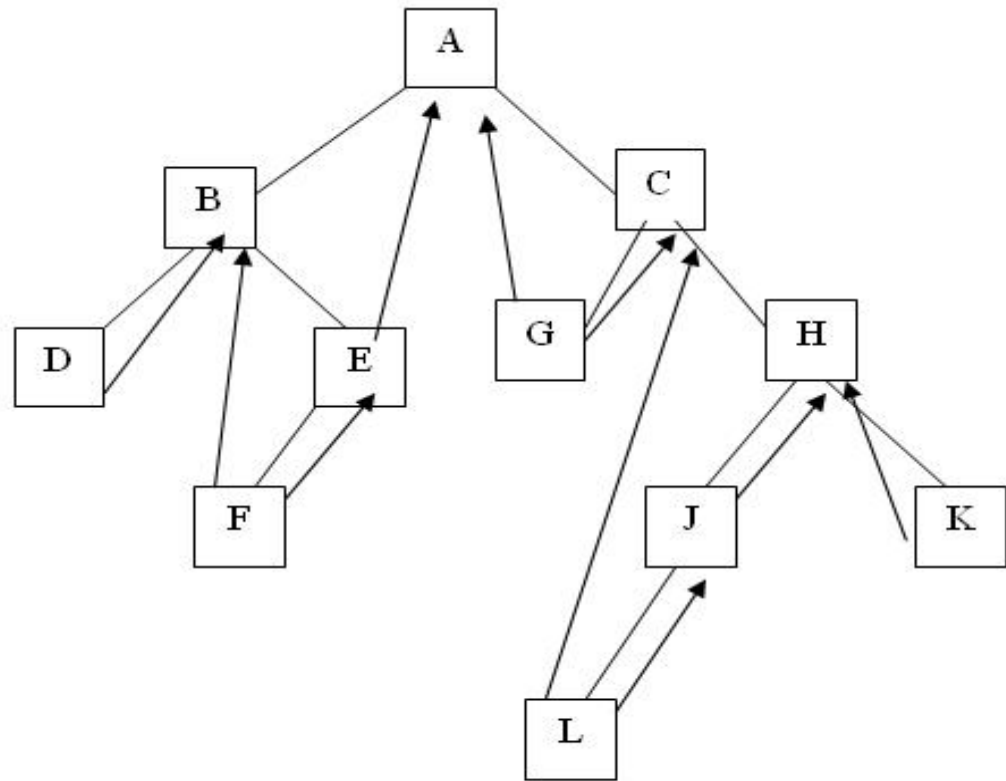
---

In the two-way threading of T.

- A thread will also appear in the left field of a node and will point to the preceding node in the **in-order** traversal of tree T.
- Furthermore, the left pointer of the first node and the right pointer of the last node (in the **in-order** traversal of T) will contain the NULL value.

# Threaded Binary Tree

- The figure shows **two-way in-order** threading.
- Here, right pointer=next node of **in-order** traversal and left pointer=previous node of **in-order** traversal
- Inorder traversal tree: D, B, F, E, A, G, C, J, L, H, K

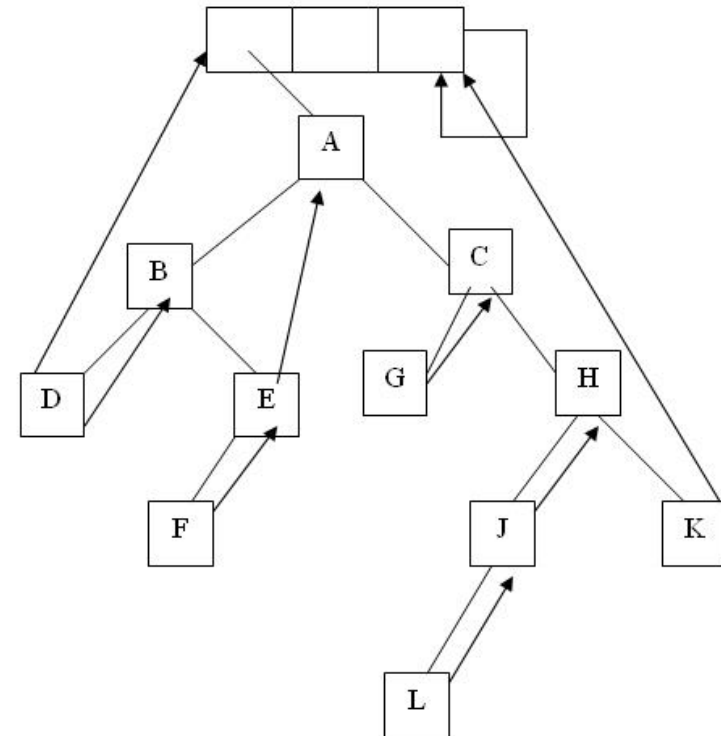


Two-way inorder threading

# Threaded Binary Tree

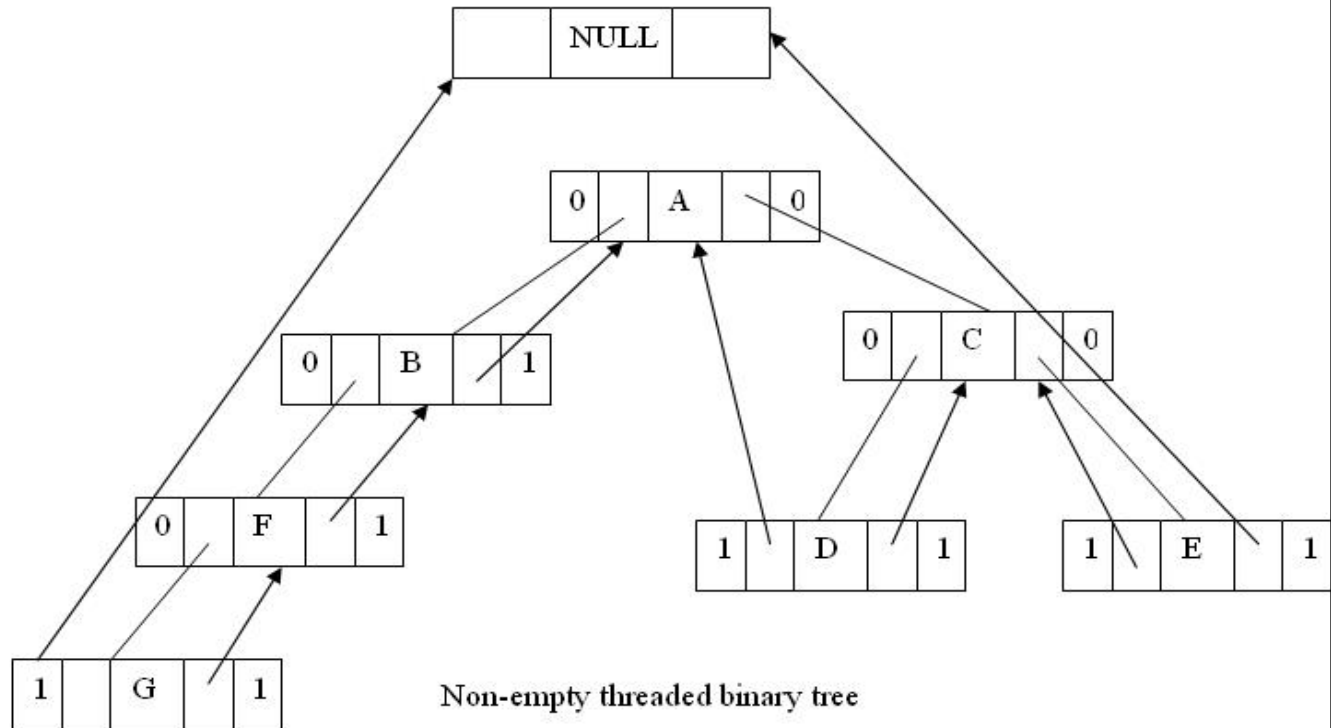
## Two-way Threading with Header node

- Again two-way threading has left pointer of the first node and right pointer of the last node (in the inorder traversal of T) will contain the NULL value when T will point to the header nodes is called two-way threading with **header node threaded binary tree**.
- This figure explains **two-way threading with header node**.



# Threaded Binary Tree

- Example of link representation of threading binary tree.
- **In-order** traversal :  
G, F, B, A, D, C, E







# Threaded Binary Tree

---

## Advantages of threaded binary tree

- The traversal operation is more faster than that of its unthreaded version
  - non-recursive implementation of threaded binary tree can run faster and
  - does not require the botheration of stack management
- Determining the predecessor and successor nodes starting from any node is efficient.
  - In case of unthreaded binary tree, this task is more time consuming and difficult because a stack is required to provide upward pointing information
  - whereas in a threaded binary tree, without using the stack the same can be carried out with the threads.



# Threaded Binary Tree

---

## Advantages of threaded binary tree

- Any node can be accessible from any other node.
  - **Threads** are usually **move to upward** whereas **links** are **downward**.
  - Thus in a threaded tree, one can move in their direction and nodes are in fact circularly linked.
  - But in unthreaded tree one can move only in downward direction starting from root.
- Insertion into and deletions from in a threaded tree are although time consuming operations but these are very **easy to implement**.



# Threaded Binary Tree

---

## Disadvantages of threaded binary tree

- Insertion and deletion from a threaded tree are very **time consuming** operation compare to non-threaded binary tree.
- This tree **require additional bit** to identify the threaded link.



# Threaded Binary Tree

---

## C Structure

```
struct node {  
    struct node *left;  
    boolean lthread;  
    int info;  
    boolean rthread;  
    struct node *right;  
};
```



# Threaded Tree Traversal Code

---

```
struct node *insert(struct node *root, int ikey) {
    struct node *tmp,*par,*ptr;
    int found=0;
    ptr = root;
    par = NULL;
    while(ptr!=NULL) {
        if(ikey == ptr->info) {
            found =1;
            break;
        }
        par = ptr;
        if(ikey < ptr->info) {
            if(ptr->lthread == false)
                ptr = ptr->left;
            else
                break;
        }
        else {
```

```
            if(ptr->rthread == false)
                ptr = ptr->right;
            else
                break;
        }
    }
    if(found)
        printf("\nDuplicate key");
    else {
        tmp=(struct node *)malloc(sizeof(struct
node));
        tmp->info=ikey;
        tmp->lthread = true;
        tmp->rthread = true;
        if(par==NULL) {
            root=tmp;
            tmp->left=NULL;
            tmp->right=NULL;
        }
    }
}
```



# Threaded Tree Traversal Code

---

```
    else if(ikey < par->info) {  
        tmp->left=par->left;  
        tmp->right=par;  
        par->lthread=false;  
        par->left=tmp;  
    }  
    else {  
        tmp->left=par;  
        tmp->right=par->right;  
        par->rthread=false;  
        par->right=tmp;  
    }  
}  
return root;  
}
```



# Threaded Tree Traversal Code

```
struct node *del(struct node *root, int dkey) {
    struct node *par,*ptr;
    int found=0;
    ptr = root;
    par = NULL;
    while(ptr!=NULL) {
        if(dkey == ptr->info) {
            found = 1;
            break;
        }
        par = ptr;
        if(dkey < ptr->info) {
            if(ptr->lthread == false)
                ptr = ptr->left;
            else
                break;
        }
        else {
```

```
            if(ptr->rthread == false)
                ptr = ptr->right;
            else
                break;
        }
    }
    if(found==0)
        printf("\ndkey not present in tree");
    else if(ptr->lthread==false && ptr->rthread==false) /*2 children*/
        root = case_c(root, par, ptr);
    else if(ptr->lthread==false) /*only left child*/
        root = case_b(root, par, ptr);
    else if(ptr->rthread==false) /*only right child*/
        root = case_b(root, par, ptr);
    else /*no child*/
        root = case_a(root,par,ptr);
    return root;
}
```



# Threaded Tree Traversal Code

---

```
struct node *case_a(struct node *root, struct node *par, struct node *ptr) {  
    if(par==NULL)    /*root node to be deleted*/  
        root=NULL;  
    else if(ptr==par->left) {  
        par->lthread=true;  
        par->left=ptr->left;  
    }  
    else {  
        par->rthread=true;  
        par->right=ptr->right;  
    }  
    free(ptr);  
    return root;  
}
```





# Threaded Tree Traversal Code

```
struct node *case_b(struct node *root, struct node *par, struct node *ptr) {
    struct node *child,*s,*p;
    /*Initialize child*/
    if(ptr->lthread==false)          /*node to be deleted has left child */
        child=ptr->left;
    else                            /*node to be deleted has right child */
        child=ptr->right;
    if(par==NULL)                  /*node to be deleted is root node*/
        root=child;
    else if(ptr==par->left)         /*node is left child of its parent*/
        par->left=child;
    else                           /*node is right child of its parent*/
        par->right=child;
    s=in_succ(ptr);
    p=in_pred(ptr);
    if(ptr->lthread==false)         /*if ptr has left subtree */
        p->right=s;
    else {
        if(ptr->rthread==false) /*if ptr has right subtree*/
            s->left=p;
        }
    free(ptr);
    return root;
}
```



# Threaded Tree Traversal Code

---

```
struct node *case_c(struct node *root, struct node *par, struct node *ptr) {
    struct node *succ,*parsucc;
    /*Find inorder successor and its parent*/
    parsucc = ptr;
    succ = ptr->right;
    while(succ->left!=NULL) {
        parsucc = succ;
        succ = succ->left;
    }
    ptr->info = succ->info;
    if(succ->lthread==true && succ->rthread==true)
        root = case_a(root, parsucc,succ);
    else
        root = case_b(root, parsucc,succ);
    return root;
}
```



# Threaded Tree Traversal Code

```
struct node *in_succ(struct node *ptr) {  
    if(ptr->rthread==true)  
        return ptr->right;  
    else {  
        ptr=ptr->right;  
        while(ptr->lthread==false)  
            ptr=ptr->left;  
        return ptr;  
    }  
}
```

```
struct node *in_pred(struct node *ptr) {  
    if(ptr->lthread==true)  
        return ptr->left;  
    else {  
        ptr=ptr->left;  
        while(ptr->rthread==false)  
            ptr=ptr->right;  
        return ptr;  
    }  
}
```

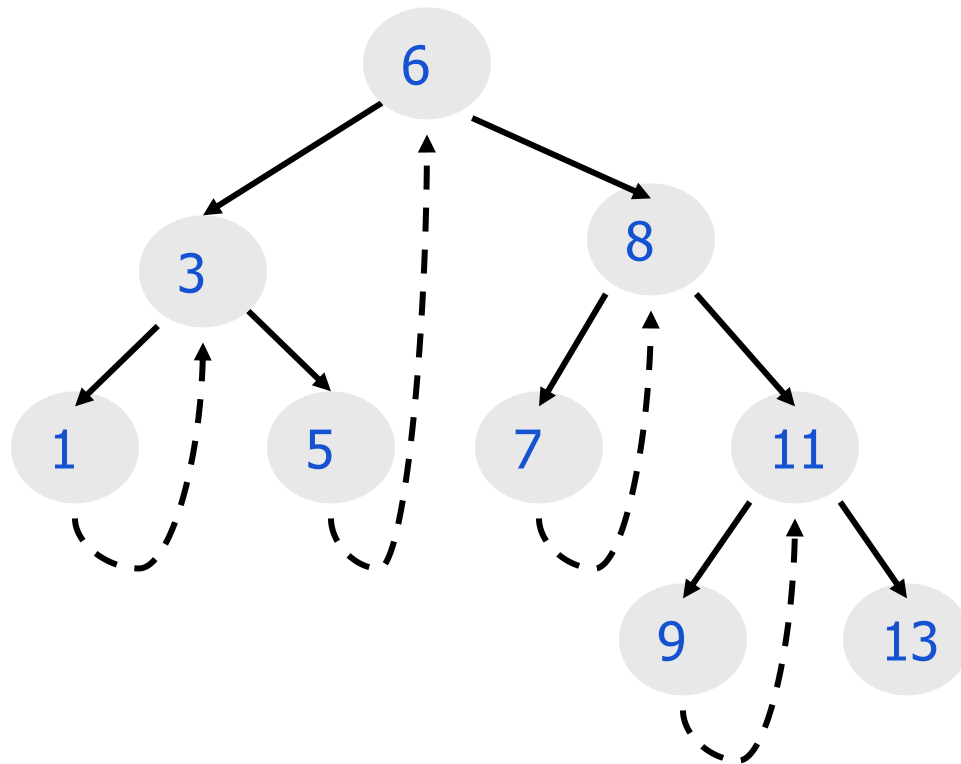


# Threaded Tree Traversal Code

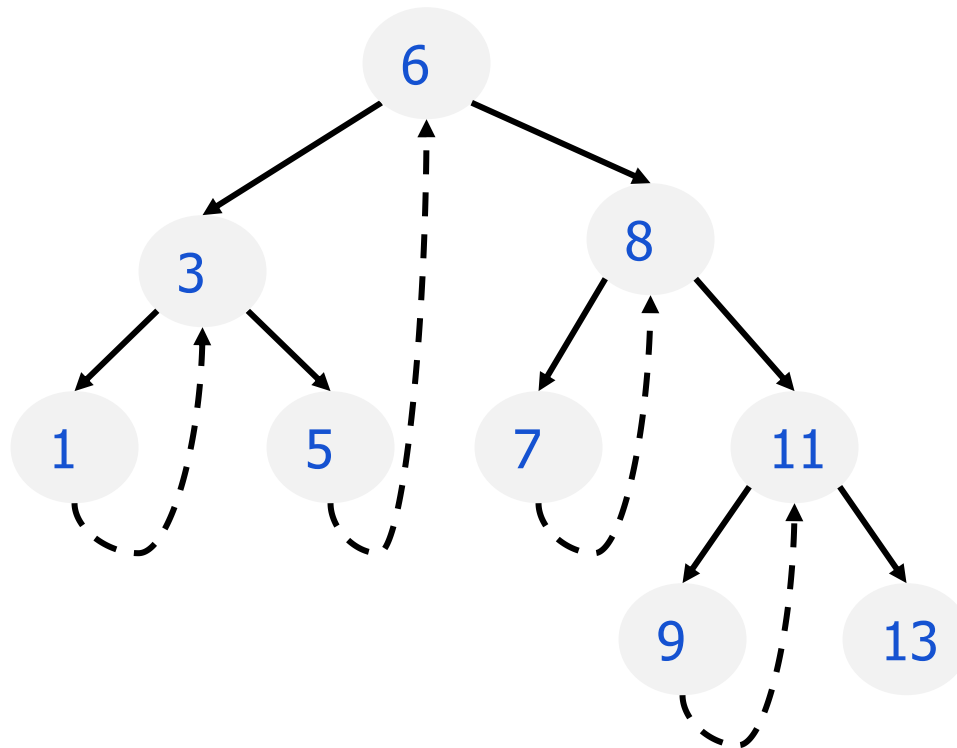
```
void inorder( struct node *root) {
    struct node *ptr;
    if(root == NULL ) {
        printf("Tree is empty");
        return;
    }
    ptr=root;
    /*Find the leftmost node */
    while(ptr->lthread==false)
        ptr=ptr->left;
    while(ptr!=NULL) {
        printf("%d ",ptr->info);
        ptr=in_succ(ptr);
    }
}/*End of inorder( )*/
```

```
void preorder(struct node *root ) {
    struct node *ptr;
    if(root==NULL) {
        printf("Tree is empty");
        return;
    }
    ptr=root;
    while(ptr!=NULL) {
        printf("%d ",ptr->info);
        if(ptr->lthread==false)
            ptr=ptr->left;
        else if(ptr->rthread==false)
            ptr=ptr->right;
        else {
            while(ptr!=NULL && ptr->rthread==true)
                ptr=ptr->right;
            if(ptr!=NULL)
                ptr=ptr->right;
        }
    }
}/*End of preorder( )*/
```

# Threaded Tree Traversal



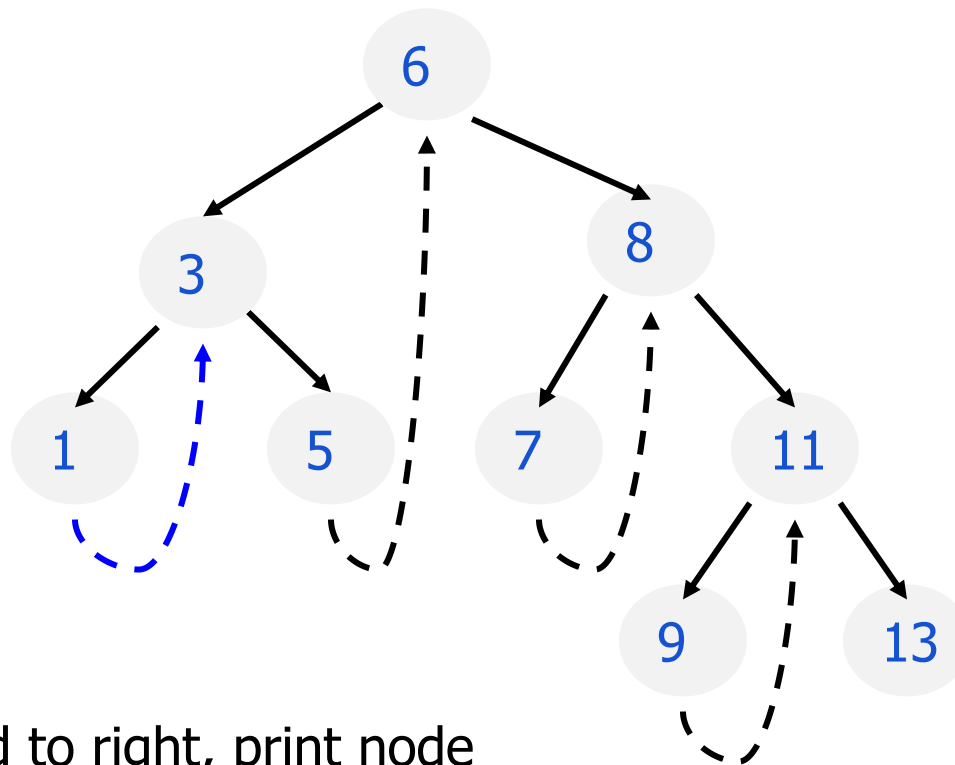
# Threaded Tree Traversal



Output  
1

Start at leftmost node, print it

# Threaded Tree Traversal



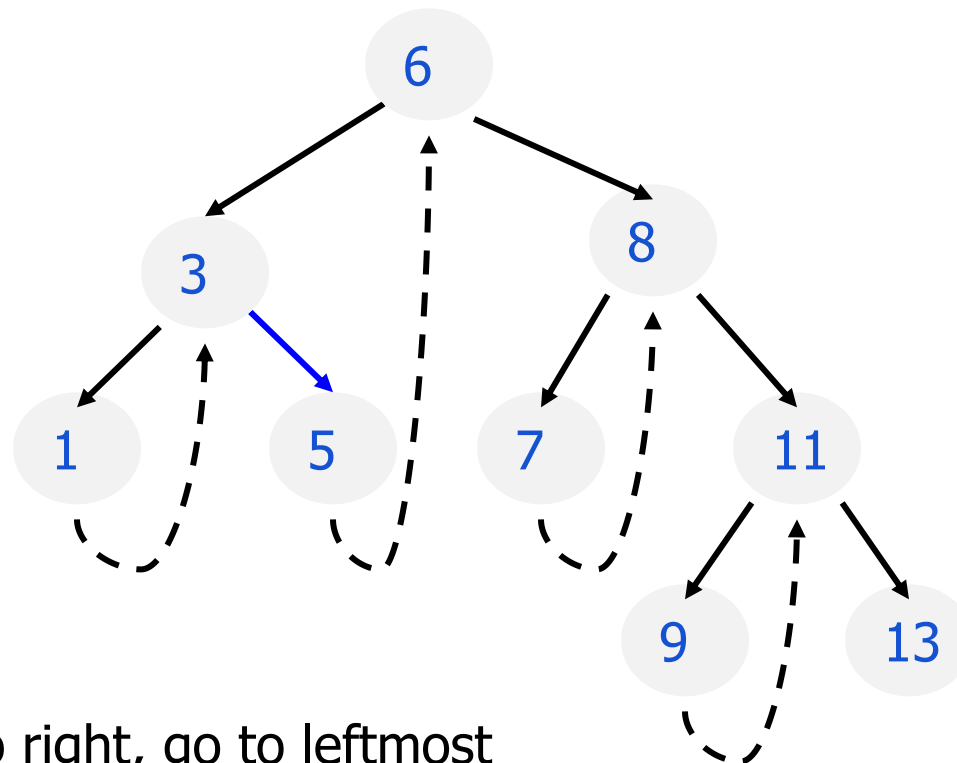
Output

1

3

Follow thread to right, print node

# Threaded Tree Traversal



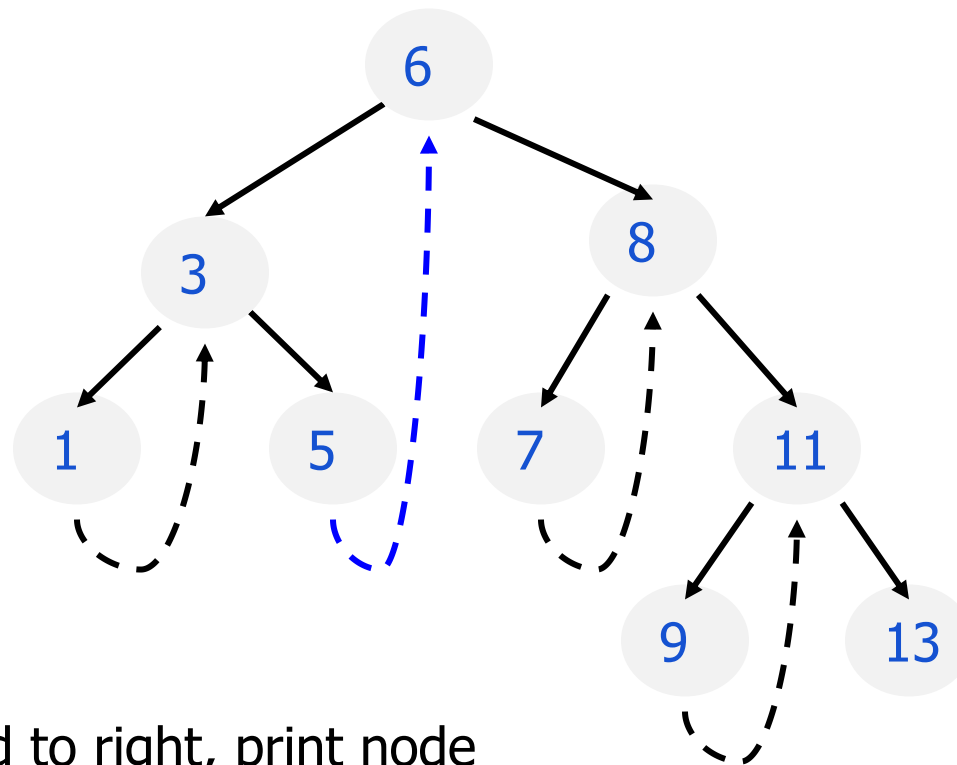
Output

1  
3  
5

Follow link to right, go to leftmost  
node and print



# Threaded Tree Traversal

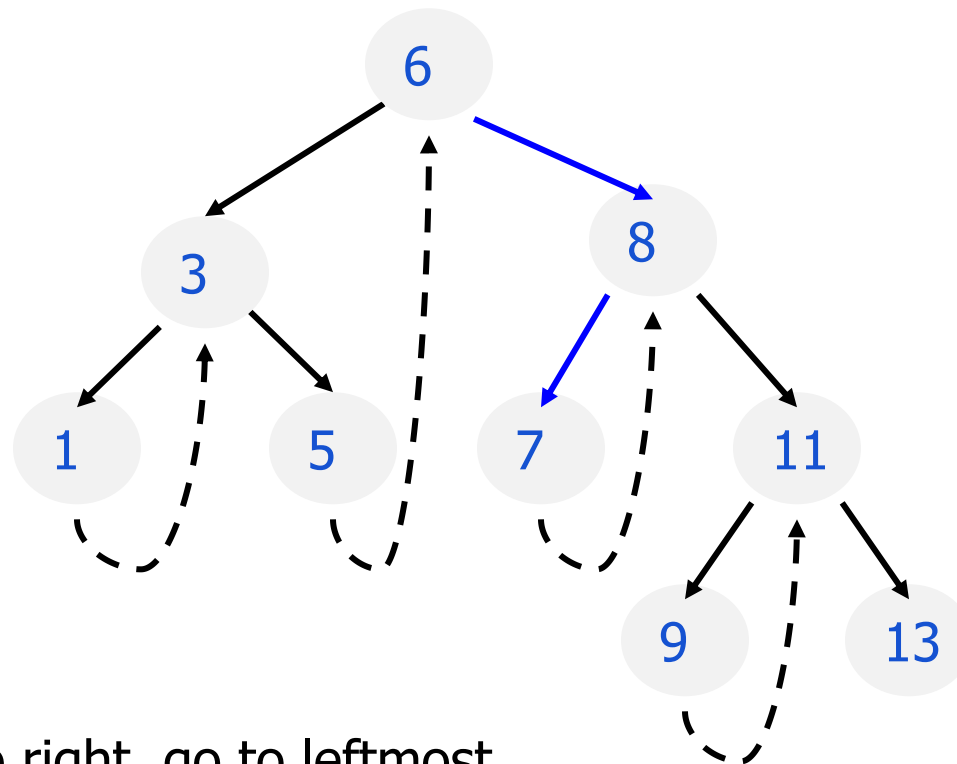


Follow thread to right, print node

Output

1  
3  
5  
6

# Threaded Tree Traversal

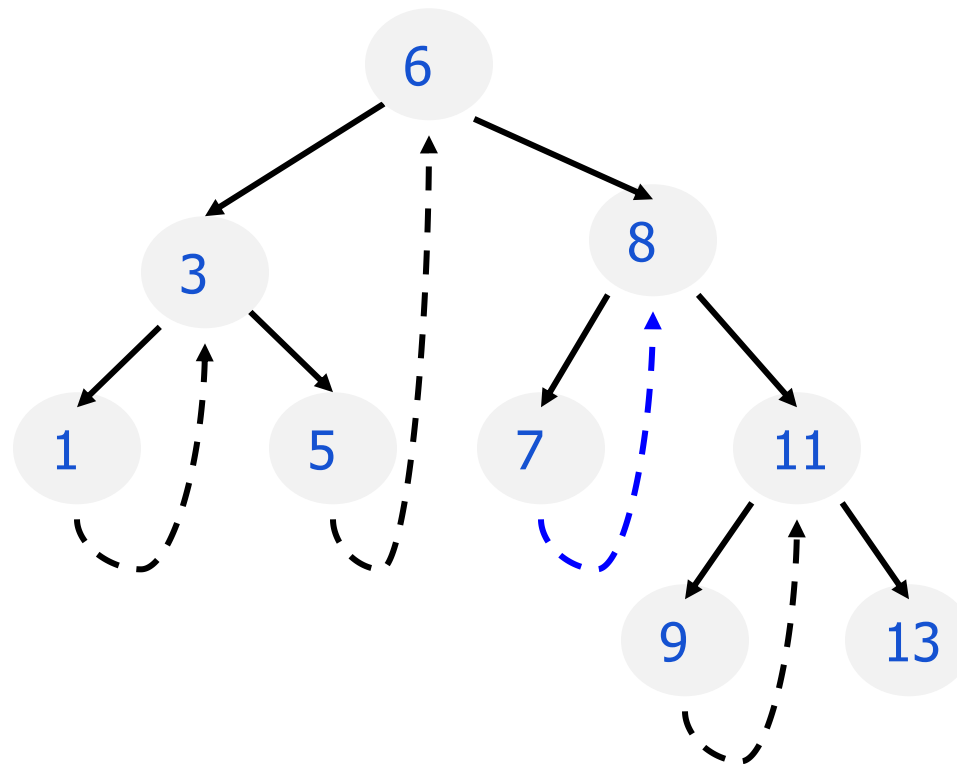


Output

1  
3  
5  
6  
7

Follow link to right, go to leftmost  
node and print

# Threaded Tree Traversal

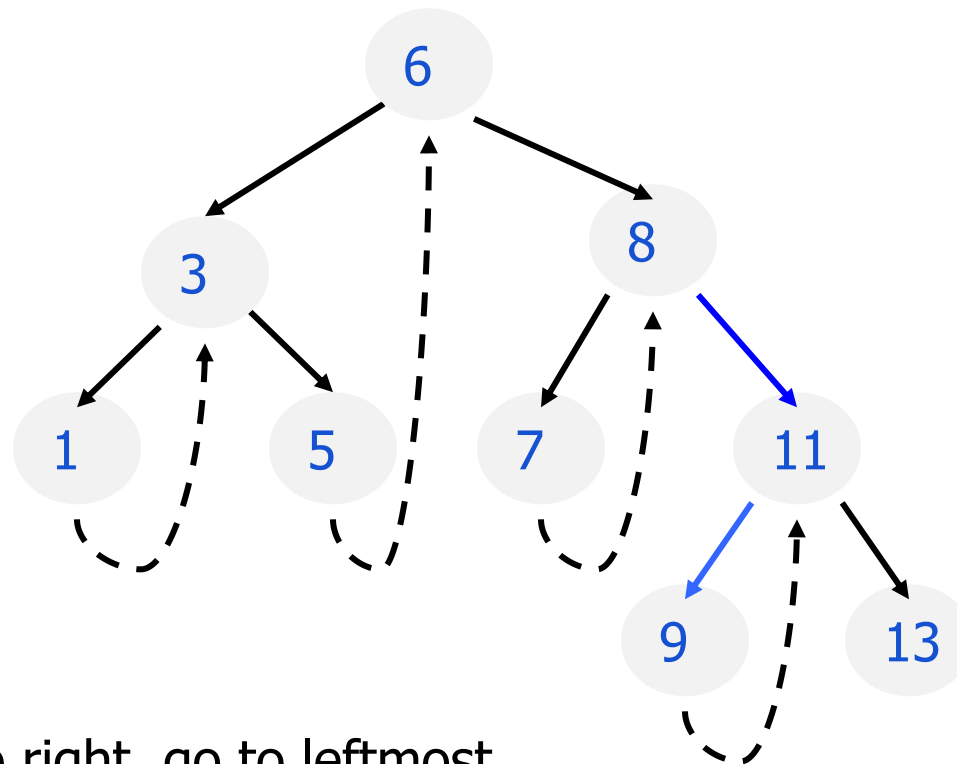


Output

1  
3  
5  
6  
7  
8

Follow thread to right, print node

# Threaded Tree Traversal

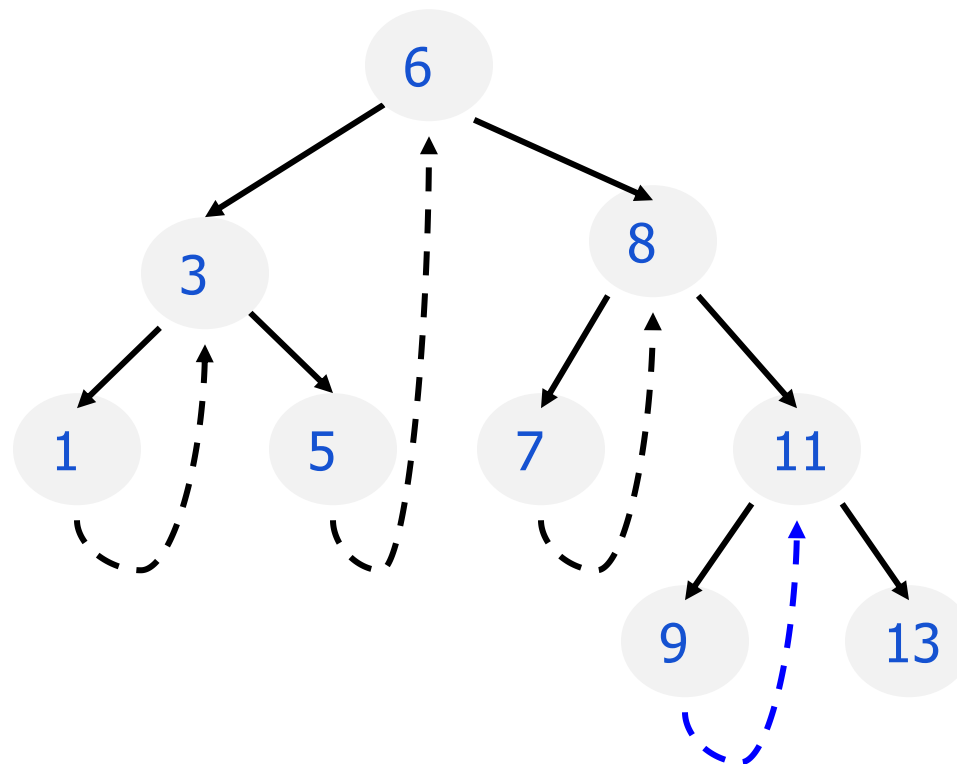


Output

1  
3  
5  
6  
7  
8  
9

Follow link to right, go to leftmost  
node and print

# Threaded Tree Traversal

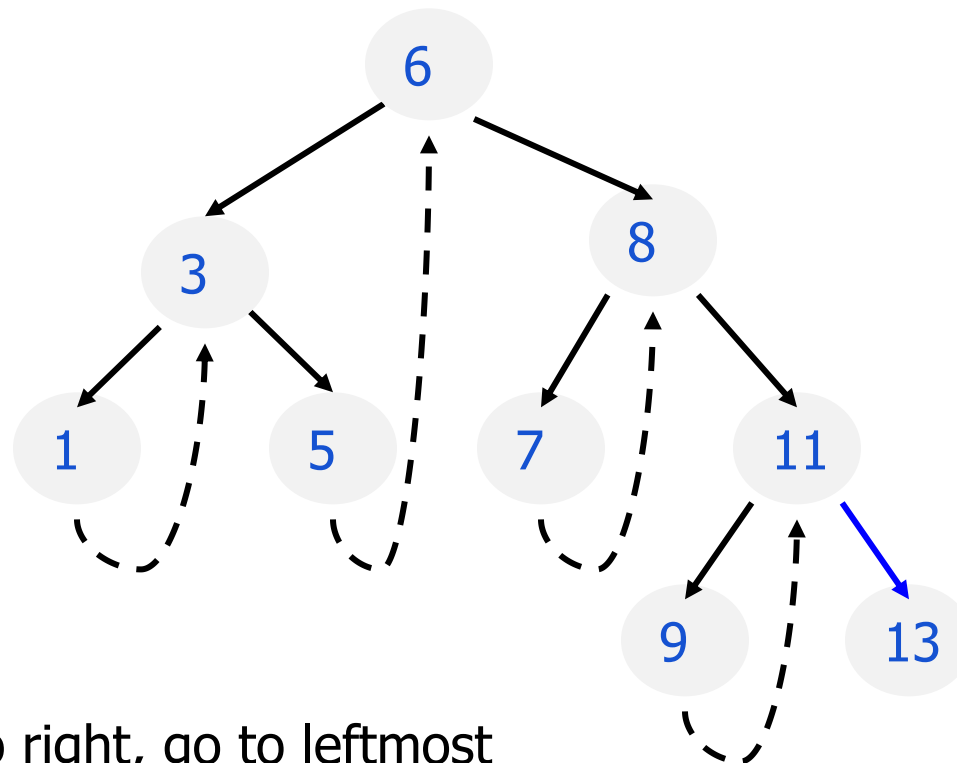


Output

1  
3  
5  
6  
7  
8  
9  
11

Follow thread to right, print node

# Threaded Tree Traversal



Output

1  
3  
5  
6  
7  
8  
9  
11  
13

Follow link to right, go to leftmost  
node and print