Lecture 5: Tree

Tree Data Structure

- A non-linear data structure that follows the shape of a tree (i.e. root, children, branches, leaves etc)
- Most applications require dealing with hierarchical data (eg: organizational structure)
- Trees allows us to find things efficiently
  - Navigation is O(log n) for a “balanced” tree with n nodes
- A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the root, and zero or more non-empty (sub) trees T1, T2, …, Tk each of whose roots are connected by a directed edge from r.
- A tree is a collection of N nodes, one of which is the root and N-1 edges.

Tree terminology

- The root of each subtree is said to be a **child** of r and r is said to be the **parent** of each subtree root.
- **Internal Nodes**: nodes with children
- **Siblings**: nodes with the same parent
- **Leaves**: nodes with no children (also known as external nodes)

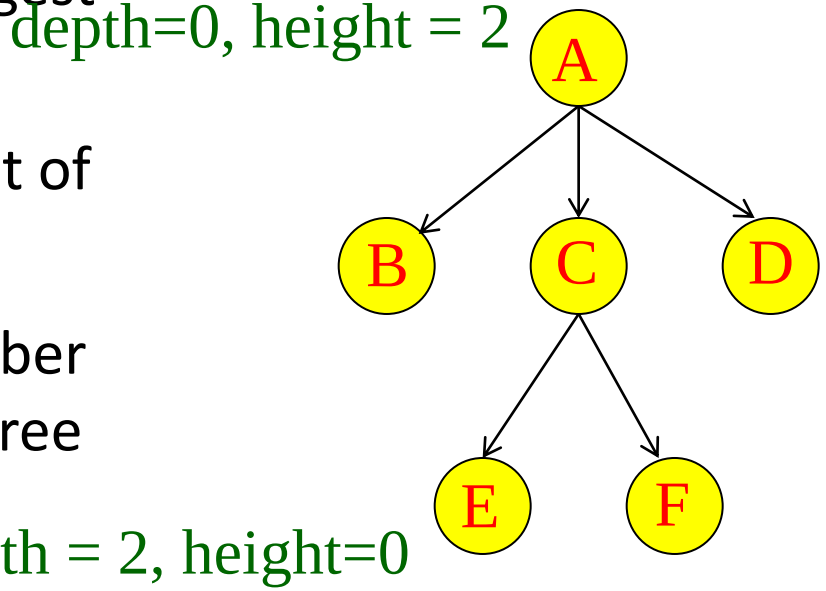
Tree terminology (continued)

- A **path** from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for $1 <= i <= k$.
- The length of this path is the number of edges on the path namely k-1.
- The length of the path from a node to itself is 0.
- There is exactly one path from from the root to each node.

Source:Mark Allen Weiss

Tree terminology (continued)

- 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
- The degree of a node is the number of subtrees of the node. A is degree 3. C is degree 2. B,D,E and F are degree 0.



Binary Tree

- A binary tree is a tree such that each node can have at most 2 children.
- Each node has an element, a reference to a left child and a reference to a right child.

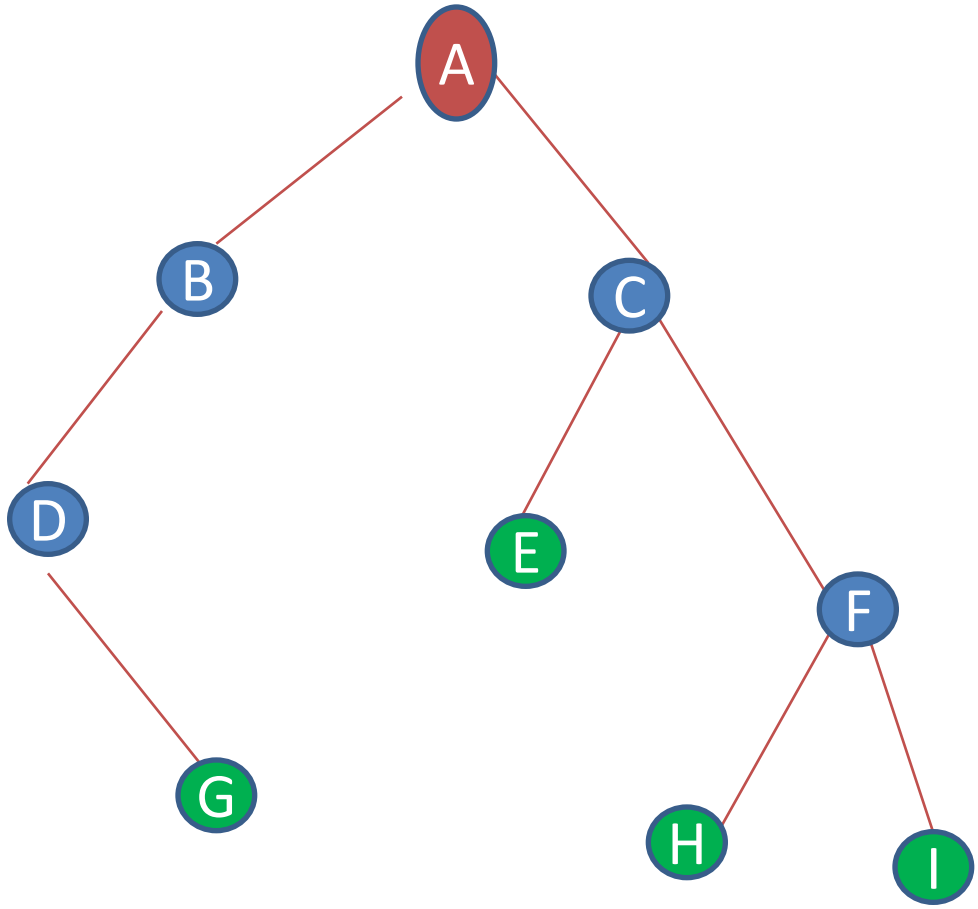
A Binary Tree of States

Each tree has a special node called its **root**, usually drawn at the top.

Each node is permitted to have two links to other nodes, called the **left child** and the **right child**.

Each node is called the **parent** of its children.

A node with no children is called a **leaf**.



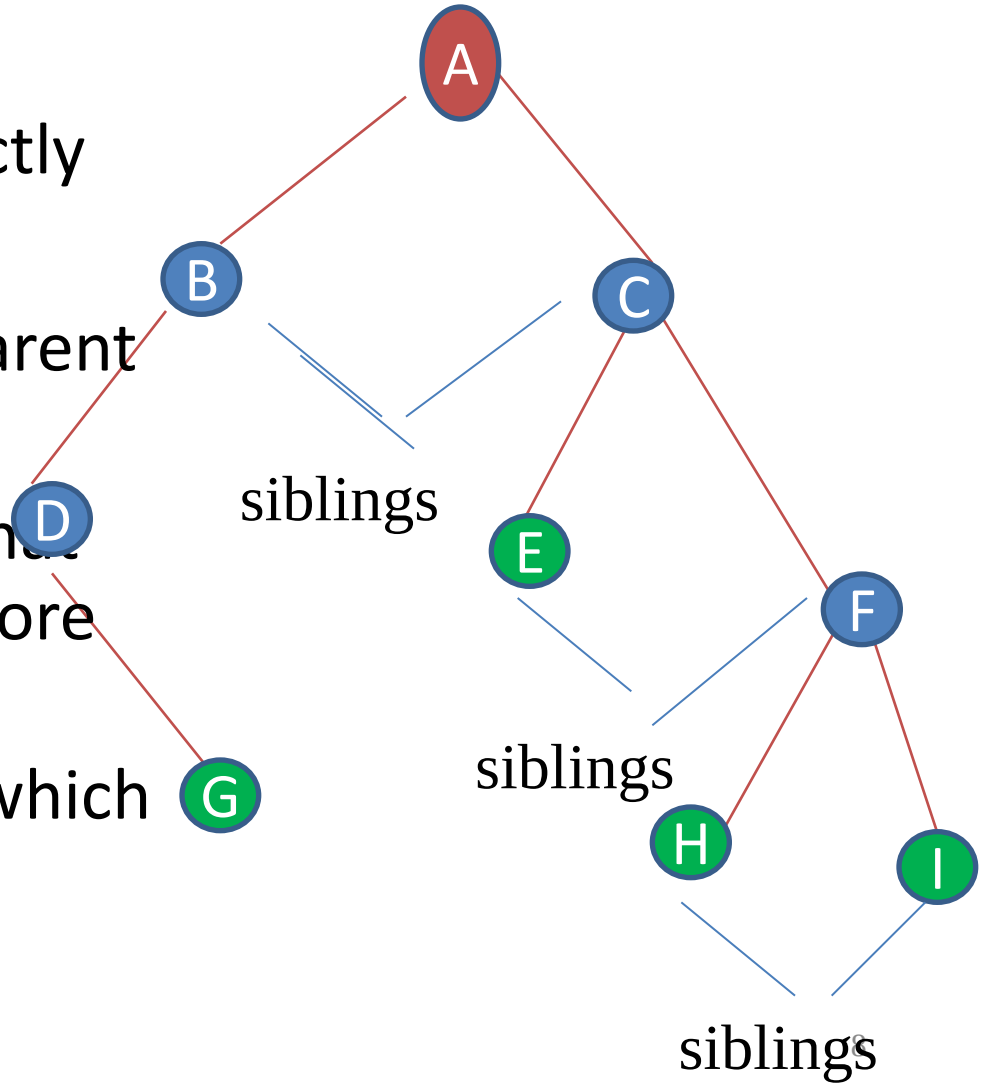
A Binary Tree of States

Two rules about parents:

- The root has no parent.
- Every other node has exactly one parent.

Two nodes with the same parent are called siblings.

- *descendants*: any nodes that can be reached via 1 or more edges from this node
- *ancestors:* any nodes for which this node is a descendant



Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all *bt*, *bt1*, *bt2* $\in$ *BinTree*, *item* $\in$ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

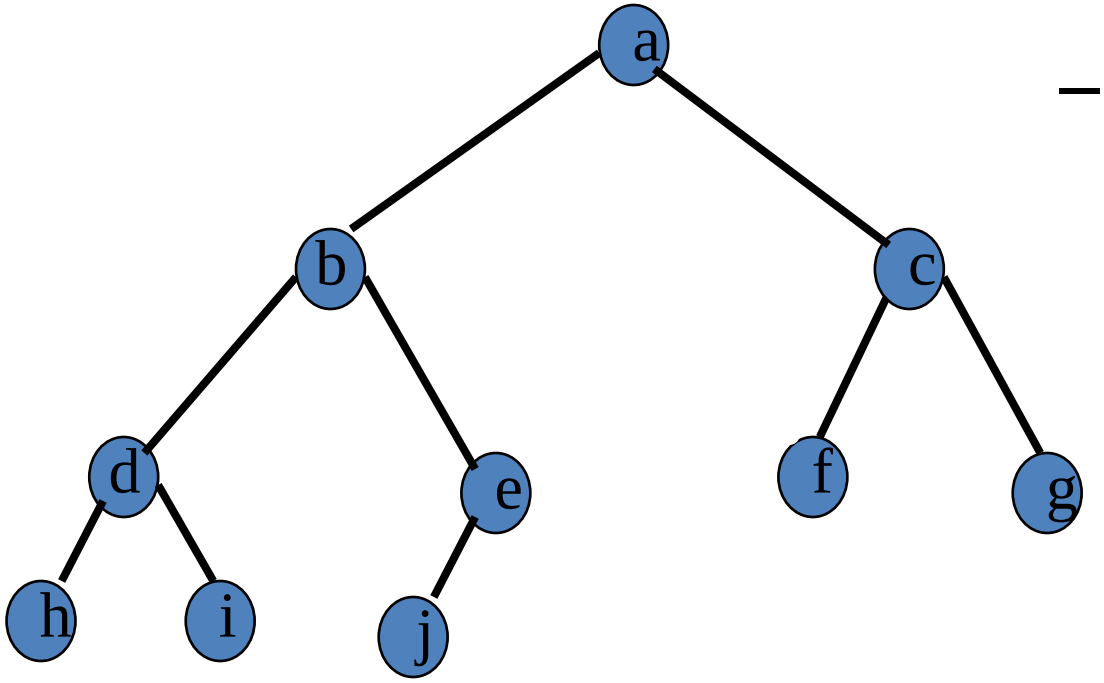
Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Binary Tree Representation

- Array representation.
- Linked representation.

Array Representation

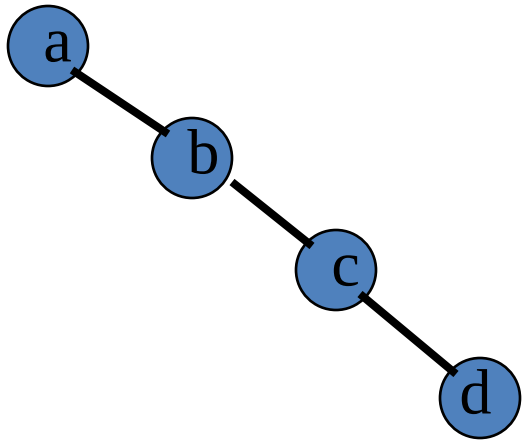
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].
  - Root index: 0
  - Given any node tree[i]
    - Left child index: 2*i+1
    - Right child index: 2*i+2
    - Parent index: $\lfloor i-1/2 \rfloor$



tree[]

a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

Right-Skewed Binary Tree



tree[]

a	-	b	-	-	-	c	-	-	-	-	-	-	-	d

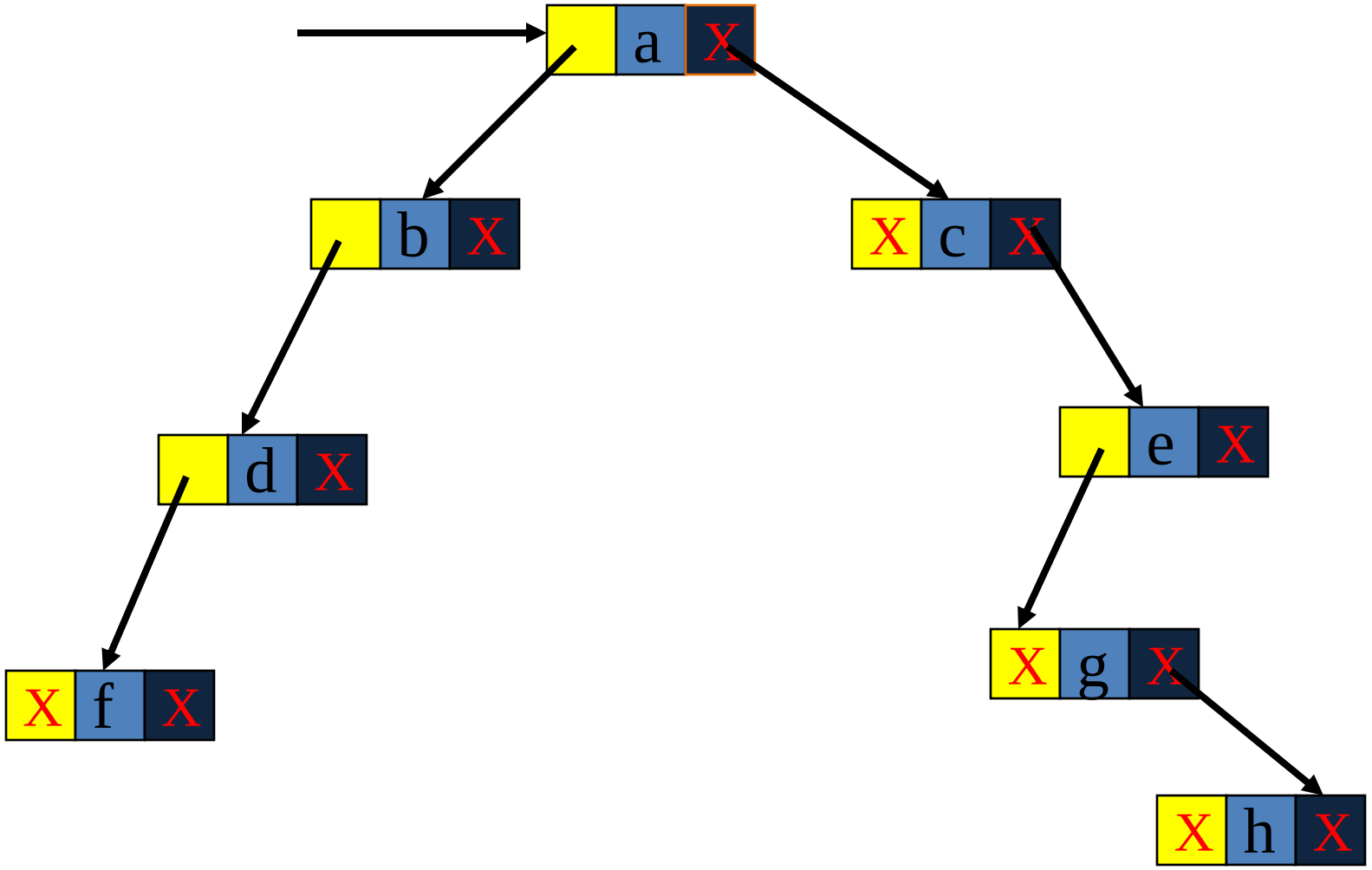
- An n node binary tree needs an array whose length is between n+1 and 2^n.

Linked Representation

- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

```
template<class elemType>
struct nodeType
{
    elemType info;
    nodeType<elemType> *llink;
    nodeType<elemType> *rlink;
};
```

Linked Representation Example



Creating Binary tree implementation Recursive

```
typedef struct node
{
int data;
struct node *left, *right;
} node;
 node *create()
{
node *p;
int x;
printf("Enter data(-1 for no data):");
scanf("%d",&x);
if(x==-1)
return NULL;
p=(node*)malloc(sizeof(node));
p->data=x;
printf("Enter left child of %d:\n",x);
p->left=create();
 printf("Enter right child of %d:\n",x);
p->right=create();
return p;
}
```

Max node and types of Binary Trees

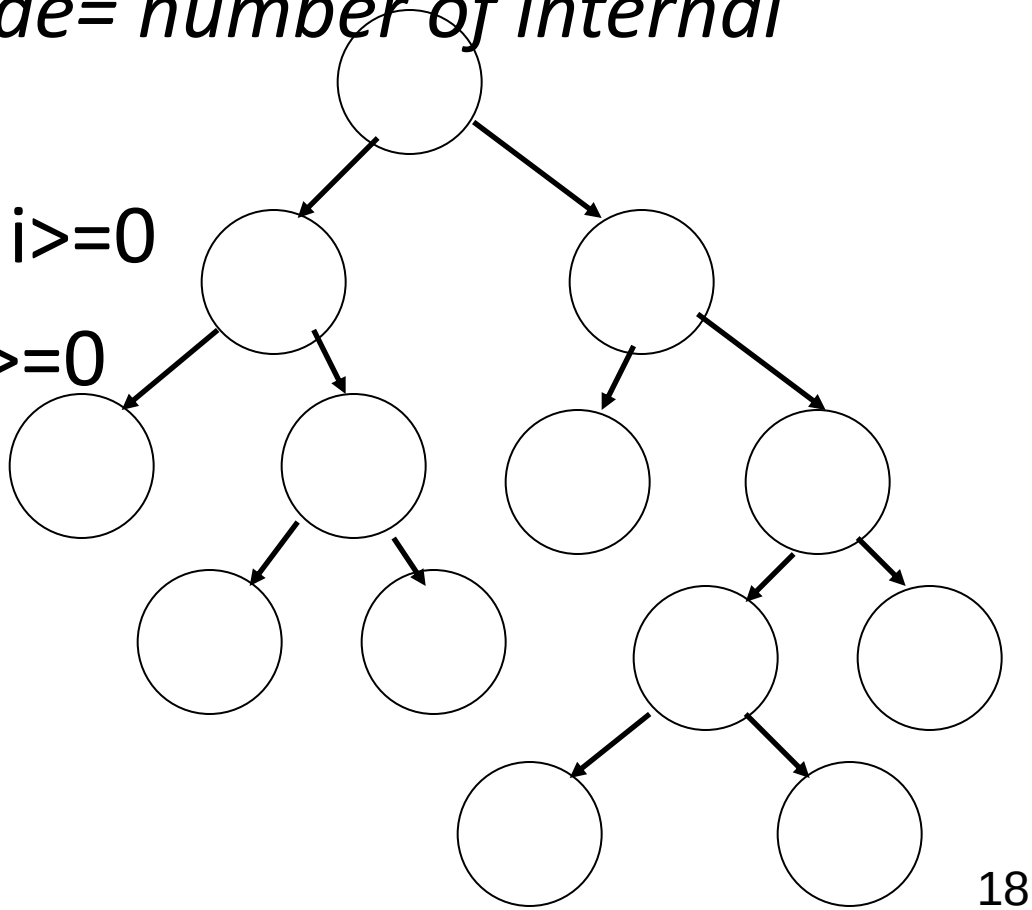
- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1 or 2^{i}, i>=0
- The maximum number of nodes in a binary tree of depth k is $2^{k}-1$, k>=1 or $2^{k+1}-1$, i>=0

Types of Binary Tree

- *Full binary tree:*
- *Complete binary tree:*
- *Perfect binary tree:*

Full Binary Tree

- *full binary tree:* a binary tree is which each node has 2 or 0 children
- *Number of leaf node= number of internal node +1*
- *Max node=* $2^{k+1}-1$, i>=0
- Min node= 2K-1, i>=0

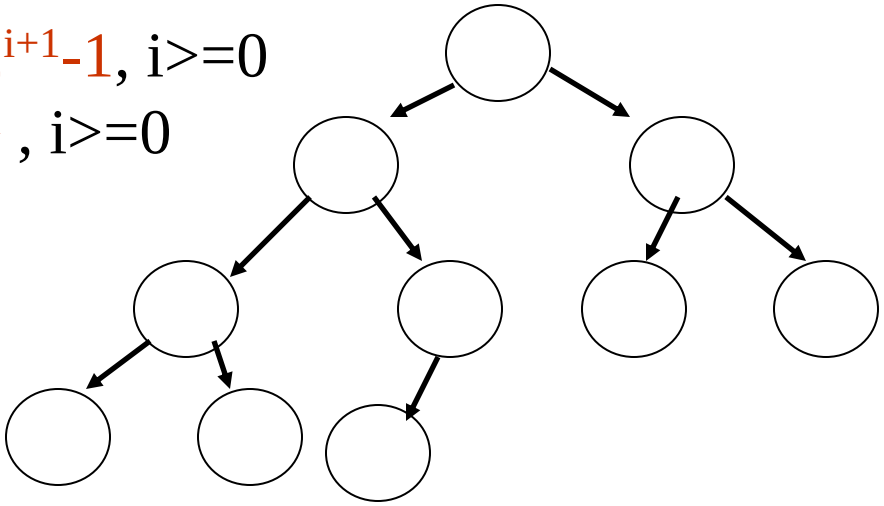


Complete Binary Tree

- *complete binary tree:* a binary tree in which every level, except possibly the deepest is completely filled. At depth n, the height of the tree, all nodes are as far left as possible

Max node= $2^{i+1}-1$, i>=0

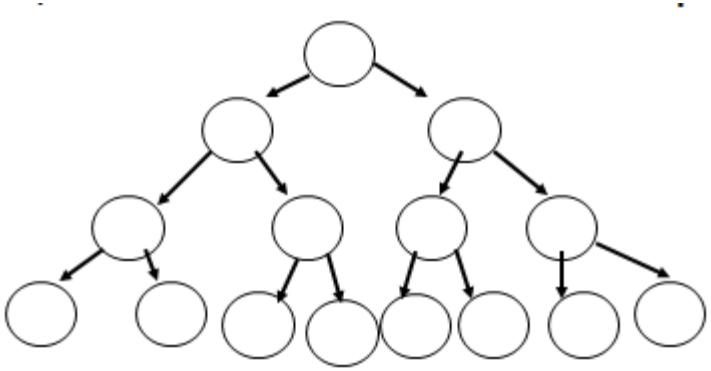
Min node= 2^{i} , i>=0



Where would the next node go to maintain a complete tree?

Perfect Binary Tree

- *perfect binary tree:* a binary tree with all leaf nodes at the same depth. All internal nodes have exactly two children.
- a perfect binary tree has the maximum number of nodes for a given height
- a perfect binary tree has $(2^{(n+1)} - 1)$ nodes where n is the height of the tree
  - height = 0 -> 1 node
  - height = 1 -> 3 nodes
  - height = 2 -> 7 nodes
  - height = 3 -> 15 nodes



Tree traversals

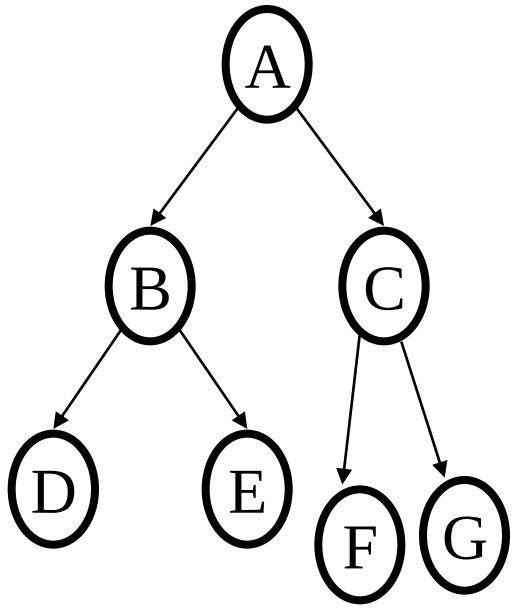
- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
  - root, left, right
  - left, root, right
  - left, right, root

Source: David Matuszek

Tree Traversals

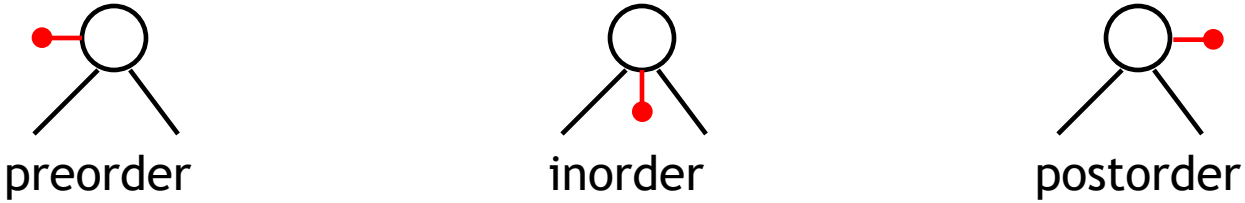
A **traversal** is an order for visiting all the nodes of a tree

- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root

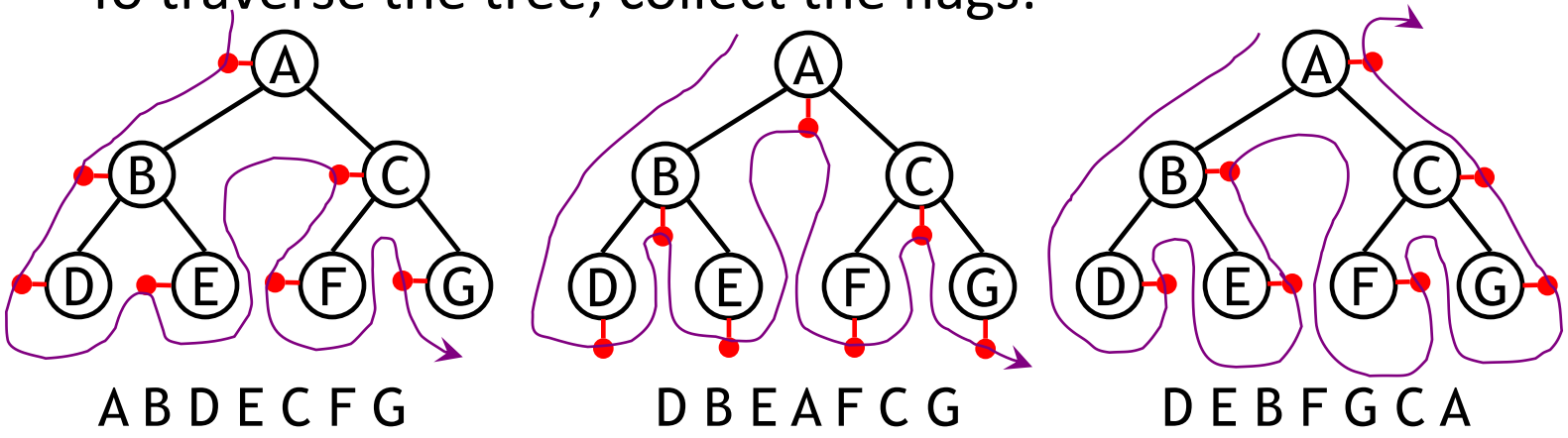


Tree traversals using “flags”

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



- To traverse the tree, collect the flags:



Source: David Matuszek

Pre-order Implementation

- Implement by recursive function

Example

```
void  print_preorder(node *root) {
    if (root) {
            printf("%d ", root->data);
            print_preorder(root->left);
            print_preorder(root->right);
    }
}
```

In-order Implementation

- Implement by recursive function

Example

```
void  print_inorder(node *root) {
    if (root) {
            print_inorder(root->left);
                        printf("%d ", root->data);
                print_inorder(root->right);
    }
}
```

post-order Implementation

- Implement by recursive function

Example

```
void  print_postorder(node *root) {
    if (root) {
            print_postorder(root->left);
                    print_postorder(root->right);
                        printf("%d ", root->data);
    }
}
```

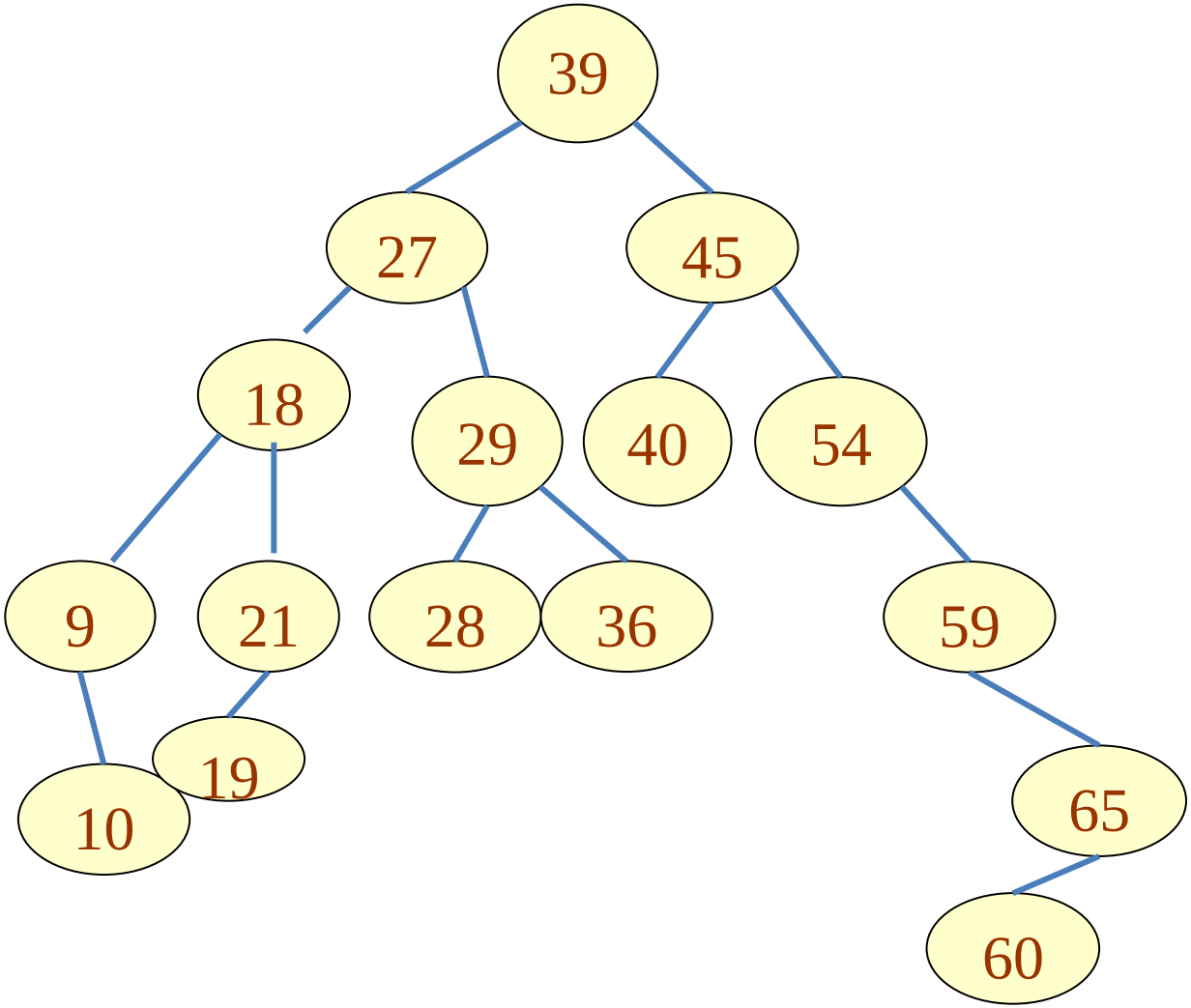
Traversal Exercise

- Construct a binary try from
  - Pre-order and In-order
  - Post-order and In-order
  - Pre-order and Post-order
- Given
- Pre-order: A B D E C F G
- In-order: D B E A F C G
- Post-order: D E B F G C A

Binary Search Trees

- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Binary Search Tree



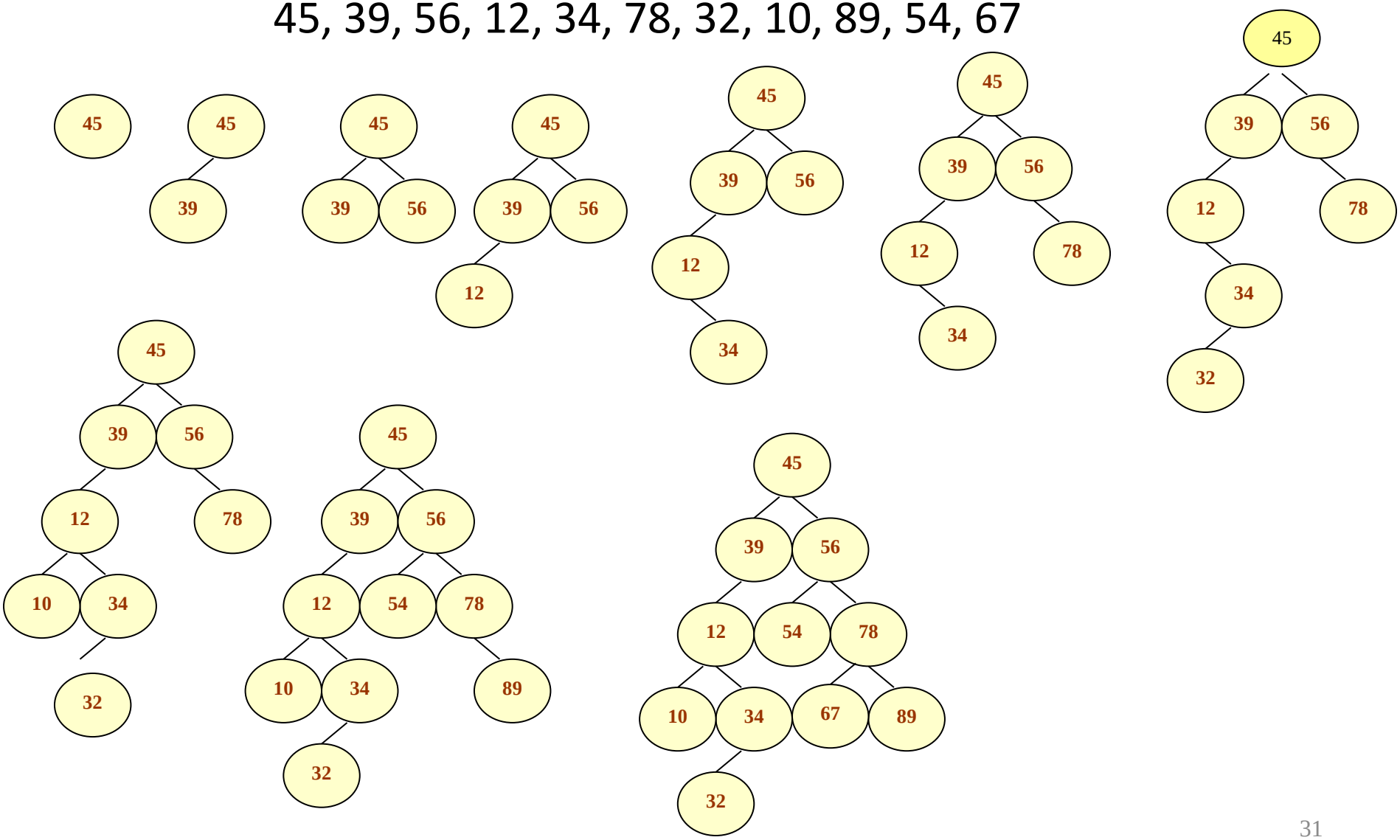
Algorithm to Insert a Value in a BST

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL, then
          Allocate memory for TREE
          SET TREE->DATA = VAL
          SET TREE->LEFT = TREE ->RIGHT = NULL
        ELSE
          IF VAL < TREE->DATA
             Insert(TREE->LEFT, VAL)
          ELSE
             Insert(TREE->RIGHT, VAL)
          [END OF IF]
        [END OF IF]
Step 2: End
```

Creating a Binary Search from Given Values

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67



Searching for a Value in a BST

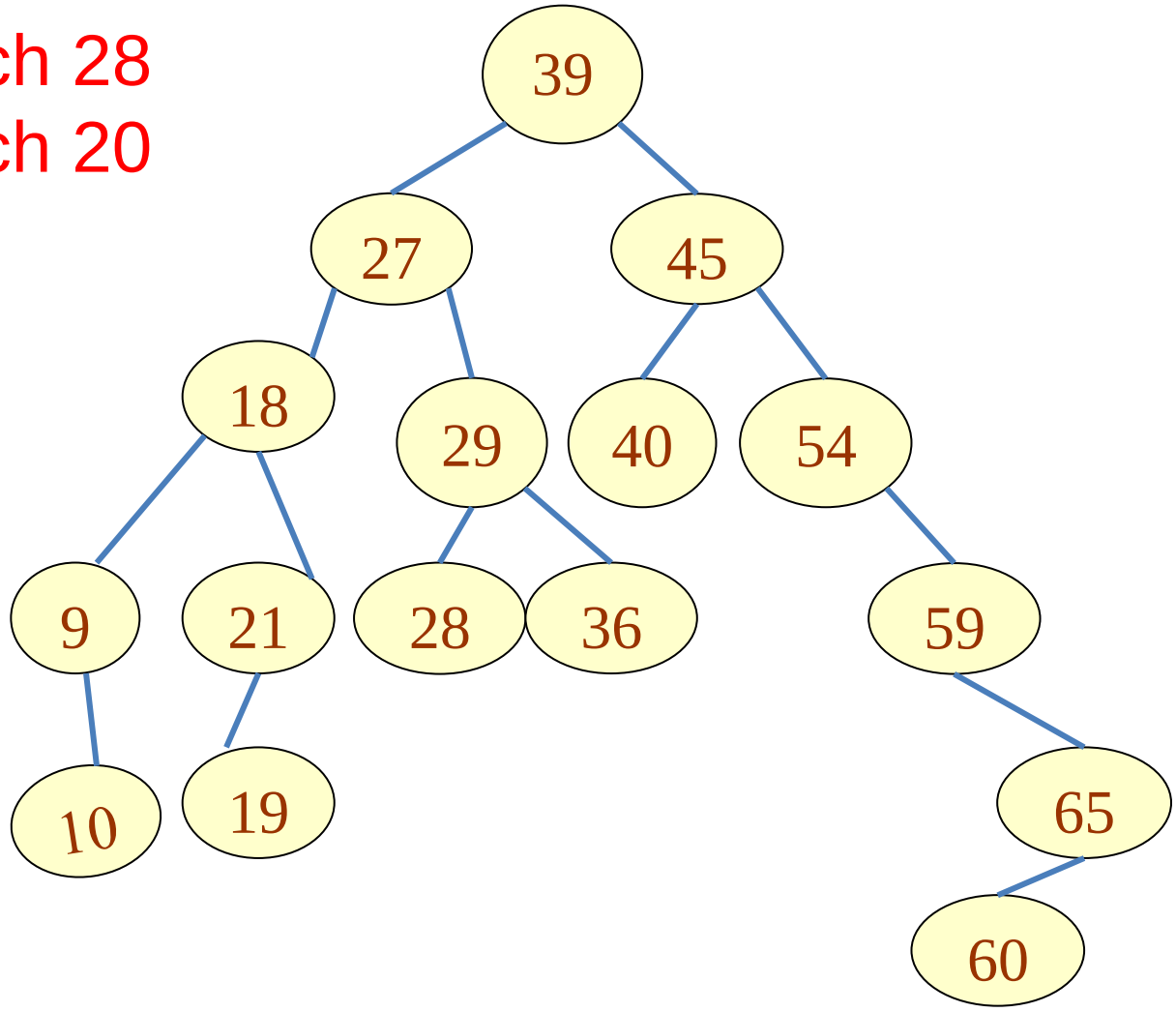
- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and the search algorithm terminates by displaying an appropriate message.
- If there are nodes in the tree then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the node, it should be recursively called on the right child node.

Algorithm to Search a Value in a BST

```
searchElement(TREE, VAL)
Step 1:
      IF TREE->DATA = VAL OR TREE = NULL, then
         Return TREE
      ELSE
         IF VAL < TREE->DATA
            Return searchElement(TREE->LEFT, VAL)
         ELSE
            Return searchElement(TREE->RIGHT, VAL)
         [END OF IF]
       [END OF IF]
Step 2: End
```

Searching for a Value in a BST

Search 28
Search 20

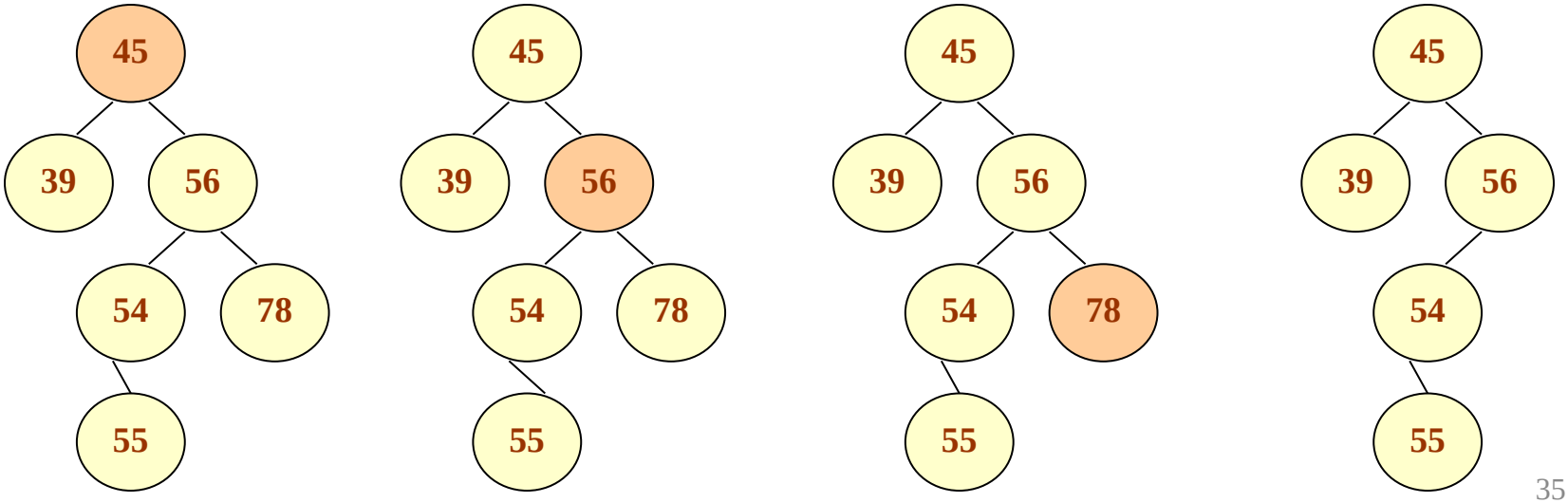


Deleting a Value from a BST

- The delete function deletes a node from the binary search tree.
- Deletion operation needs to keep the property of BSTs.
- The deletion of a node involves any of the three cases.

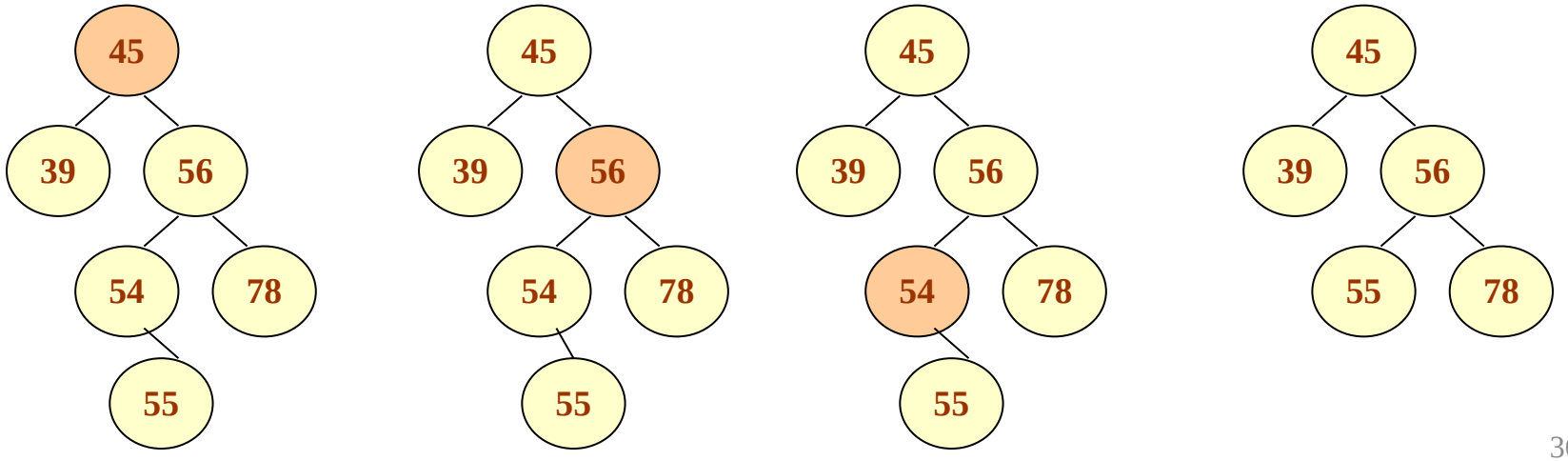
Case 1: Deleting a node that has no children.

For example, deleting node 78 in the tree below.



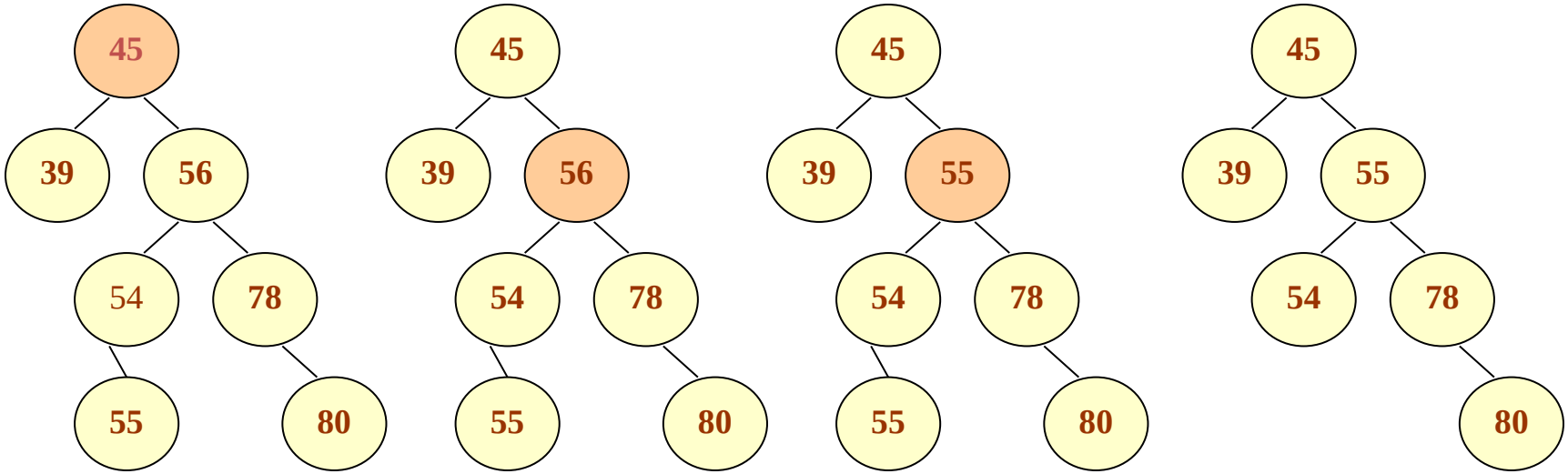
Deleting a Value from a BST

- *Case 2:* Deleting a node with one child (either left or right).
- To handle the deletion, the node's child is set to be the child of the node's parent.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.



Deleting a Value from a BST

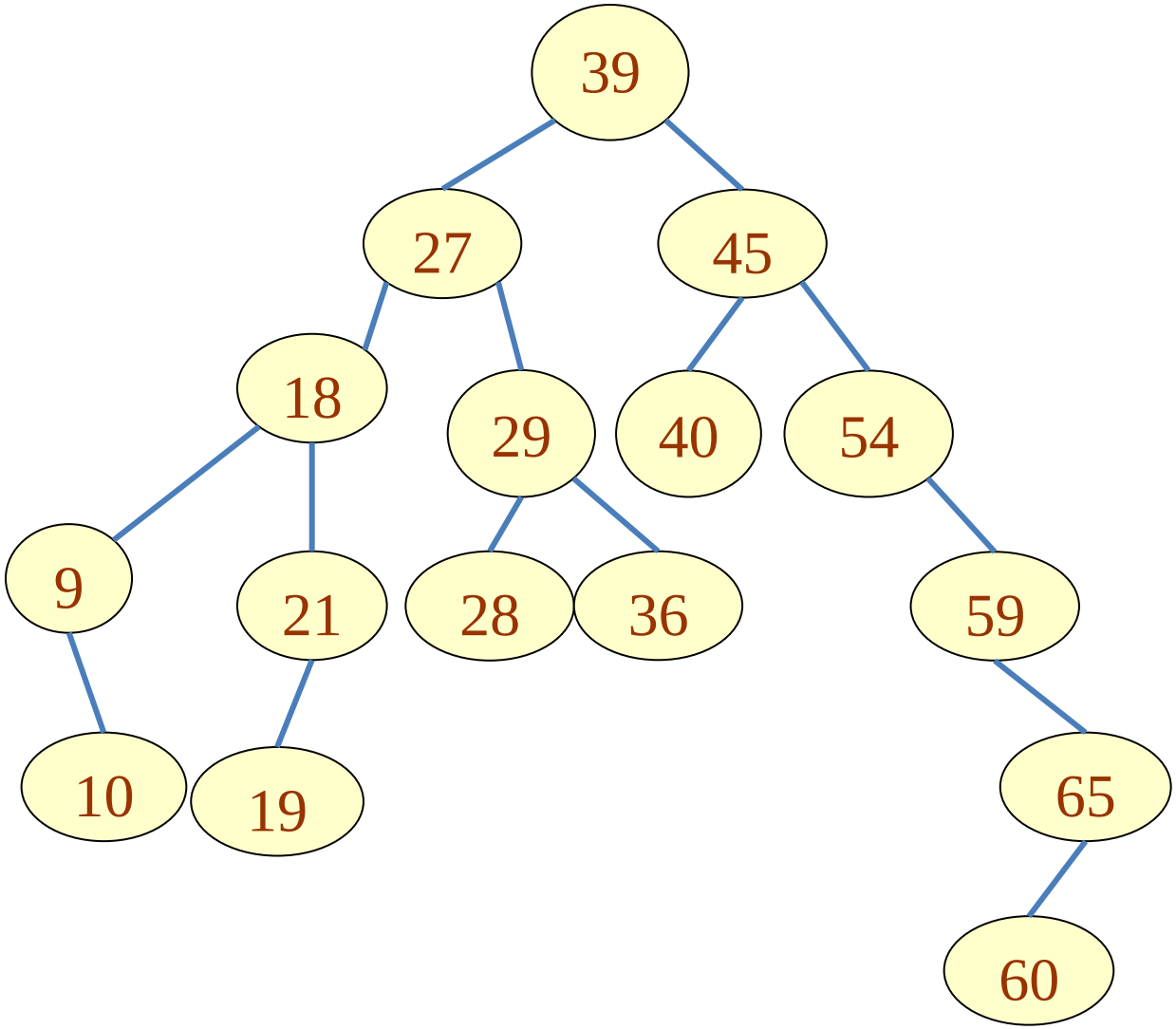
- *Case 3:* Deleting a node with two children.
- To handle this case of deletion, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
- The in-order predecessor or the successor can then be deleted using any of the above cases.



Algorithm to Delete from a BST

```
Delete (TREE, VAL)
Step 1: IF TREE = NULL, then
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE->DATA
           Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE->DATA
          Delete(TREE->RIGHT, VAL)
        ELSE IF TREE->LEFT AND TREE->RIGHT
            SET TEMP = findLargestNode(TREE->LEFT)
            SET TREE->DATA = TEMP->DATA
            Delete(TREE->LEFT, TEMP->DATA)
        ELSE
           SET TEMP = TREE
           IF TREE->LEFT = NULL AND TREE ->RIGHT = NULL
               SET TREE = NULL
           ELSE IF TREE->LEFT != NULL
               SET TREE = TREE->LEFT
           ELSE
               SET TREE = TREE->RIGHT
           [END OF IF]
             FREE TEMP
         [END OF IF]
Step 2: End
```

Find the Largest Value in a BST



Finding the Largest Node in a BST

- The basic property of a BST states that the larger value will occur in the right sub-tree.
- If the right sub-tree is NULL, then the value of root node will be largest as compared with nodes in the left sub-tree.
- So, to find the node with the largest value, we will find the value of the rightmost node of the right sub-tree.
- If the right sub-tree is empty then we will find the value of the root node.

```
findLargestElement (TREE)
Step 1: IF TREE = NULL OR TREE->RIGHT =
NULL, then
            Return TREE
         ELSE
            Return
findLargestElement(TREE->RIGHT)
      [END OF IF]
```

Finding the Smallest Node in a BST

- The basic property of a BST states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of root node will be smallest as compared with nodes in the right sub-tree.
- So, to find the node with the smallest value, we will find the value of the leftmost node of the left sub-tree.
- However, if the left sub-tree is empty then we will find the value of the root node.

```
findSmallestElement (TREE)
Step 1: IF TREE = NULL OR TREE->LEFT =
NULL, then
            Return TREE
      ELSE
            Return
findSmallestElement(TREE->LEFT)
      [END OF IF]
```

Balanced BST

Balanced BST, i.e. height of left and right subtrees are equal or not much differences at any node

Example: a full binary tree of n nodes

The search in can be done in log(n) time, O(log n).

Depth of recursion is O(log n)

Time complexity O(log n)

Space complexity O(log n)

A BST is not balanced in general !

Balance factor

- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

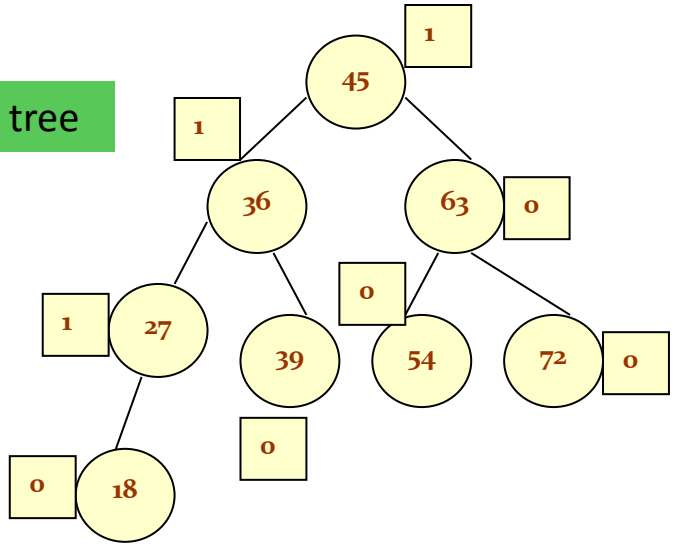
  Balance factor = Height (left sub-tree) – Height (right sub-tree)

Balance factor

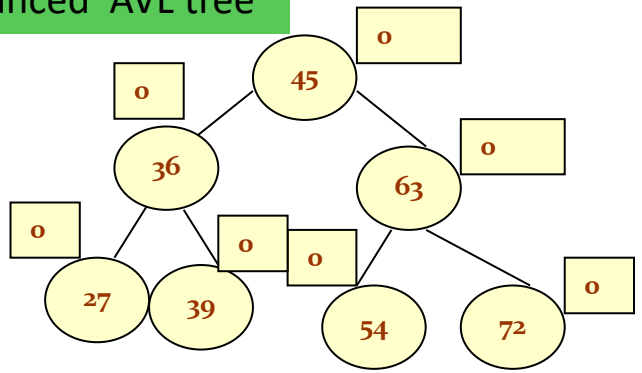
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called *Left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called Right-*heavy tree*.
- A node is unbalanced if its balance factor is not -1, 0, or 1.

Examples of Balance Factor

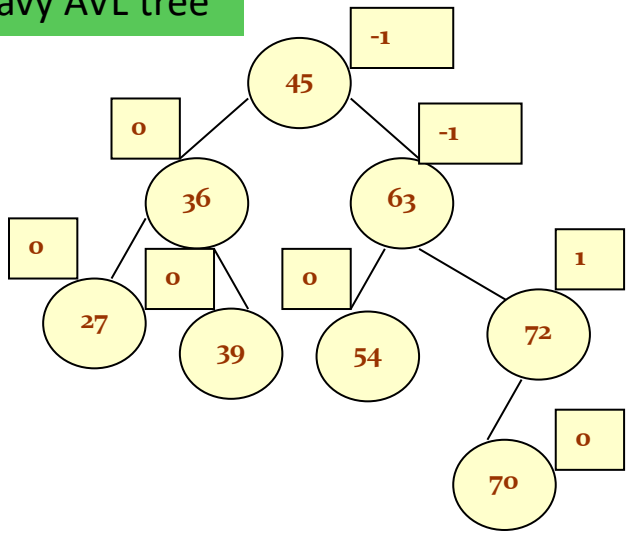
Left heavy AVL tree



Balanced AVL tree



Right heavy AVL tree



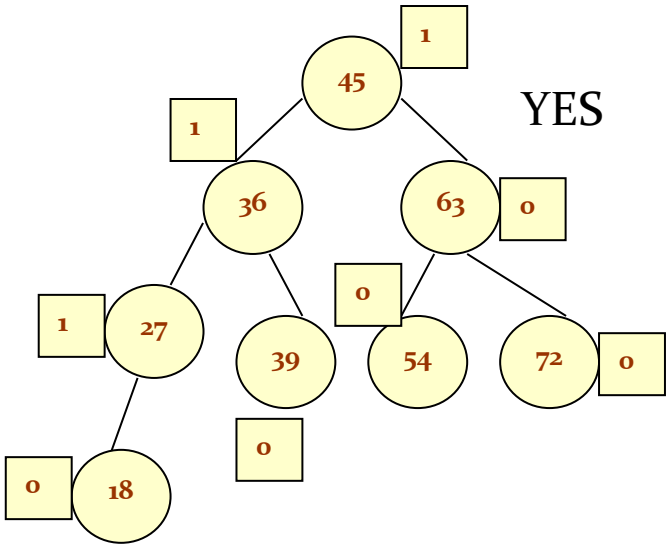
Height Balanced Tree

- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be height balanced.

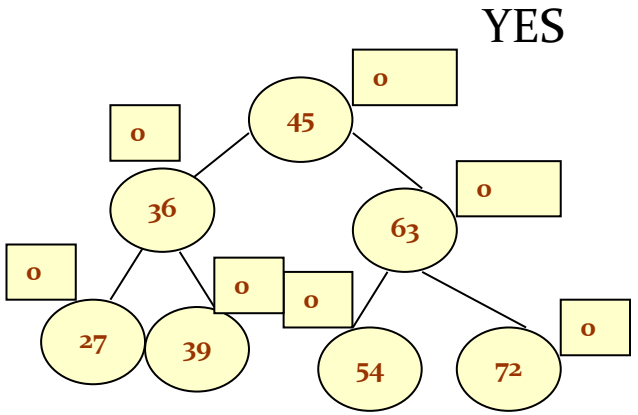
In another word: at any node, the difference of heights of two sub-trees is at most 1

- *Property: height of balanced tree of n nodes is O(log n).*

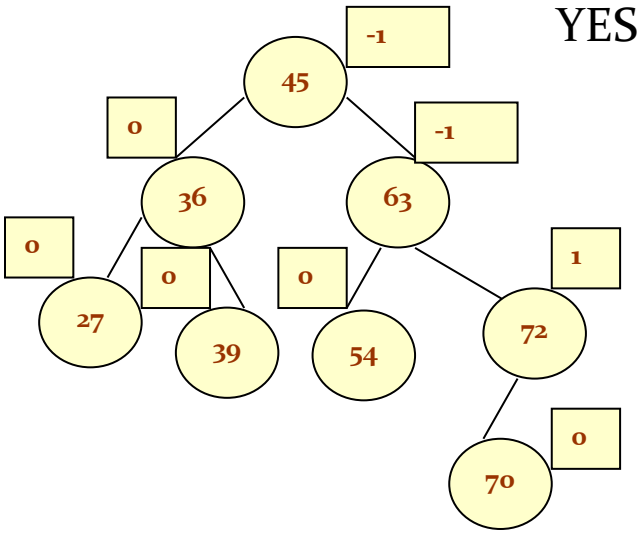
Height Balanced Tree



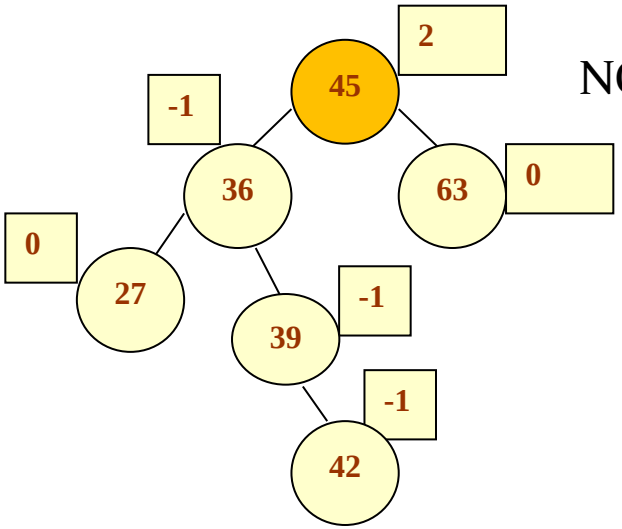
YES



YES



NO



AVL Trees

- AVL tree is a self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one.
- AVL tree is a height-balanced tree
- Self-balancing : re-balance after BST inserting or deleting
- AVL is named by its inventor **Adelson-Velskii** and **Landis**
  - **provided the height balance property and insertion and deletion operations that maintain the property in time complexity of O(log n)**

Features of AVL Trees

- The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to O(log n).

- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the *Balance Factor*.

Searching for a Node in an AVL

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes O(log *n*) time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Inserting a Node in an AVL Tree

- Algorithm:

  Step 1. insert new node as BST

  Step 2: if not AVL tree, do re-balancing to derive an AVL tree

- A new node is inserted as the leaf node, so it will always have balance factor equal to zero.

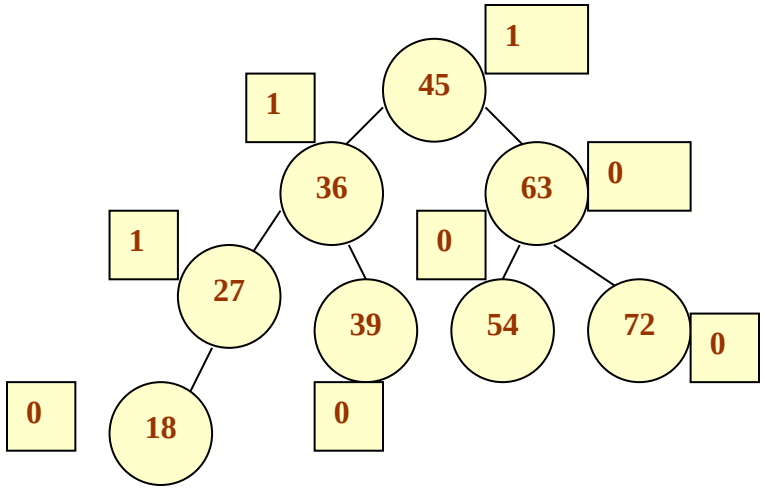
- The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.

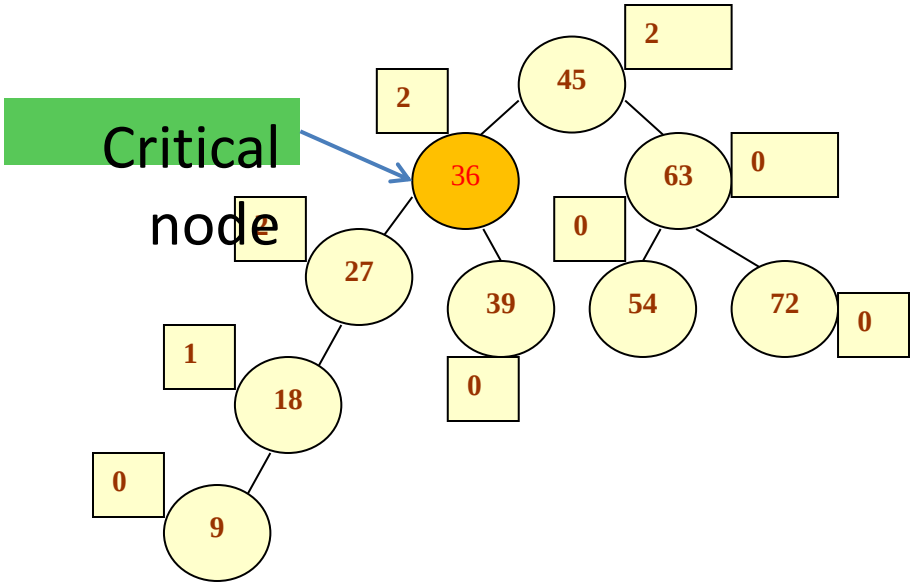
Inserting a Node in an AVL Tree

- The possible changes which may take place in any node on the path are as follows:
  - Initially the node was either left or right heavy and after insertion has become balanced.
  - Initially the node was balanced and after insertion has become either left or right heavy.
  - Initially the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree thereby creating an unbalanced sub-tree. Such a node is said to be a critical node.
- The critical node is the unbalanced node of lowest level

Example of critical node

Example: Consider the AVL tree given below and insert 9 into it.





Cases of critical nodes

- Height of sub-tree where a new node is inserted can increase at most 1.
- The balance factor can crease by 1 or decrease by 1, so the balance factor of critical node is either 2 or -2.
  Let x denote the critical node.

There are four possible cases:

Case 1: balance-factor(x) = 2, balance-factor(x->left) >=0,

Case 2: balance-factor(x) = 2, balance-factor(x->left) < 0,

Case 3: balance-factor(x) = -2, balance-factor(x->right) < =0,

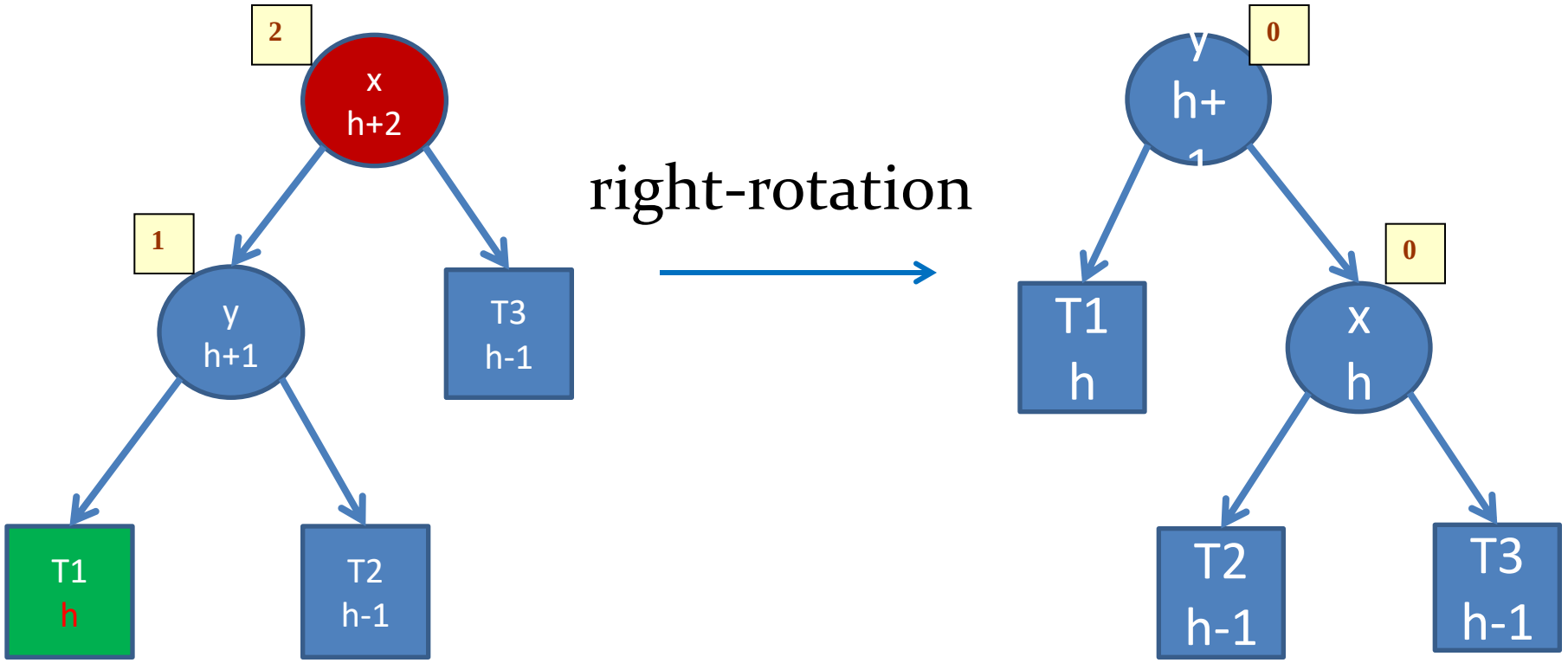
Case 4: balance-factor(x) = -2, balance-factor(x->right) > 0

Each case will different rotation operation for re-balancing

Case 1 : right-rotation to re-balance

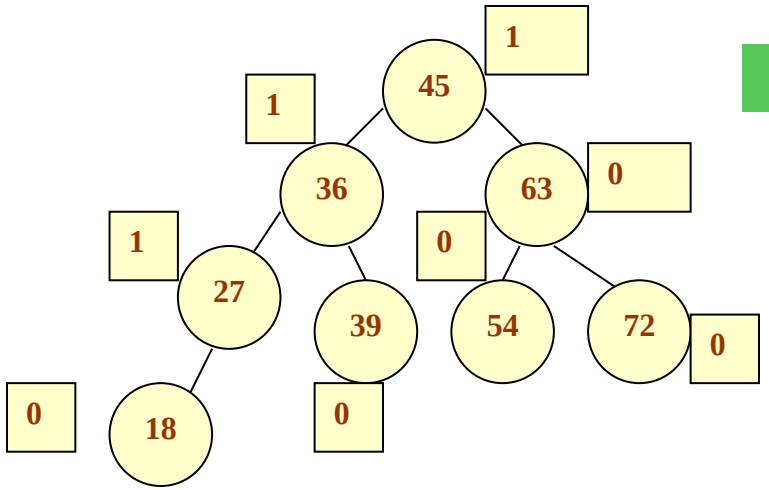
Case 1: balance-factor(x) = 2, balance-factor(x->left) >= 0

right_rotation(x) it return y as new top node

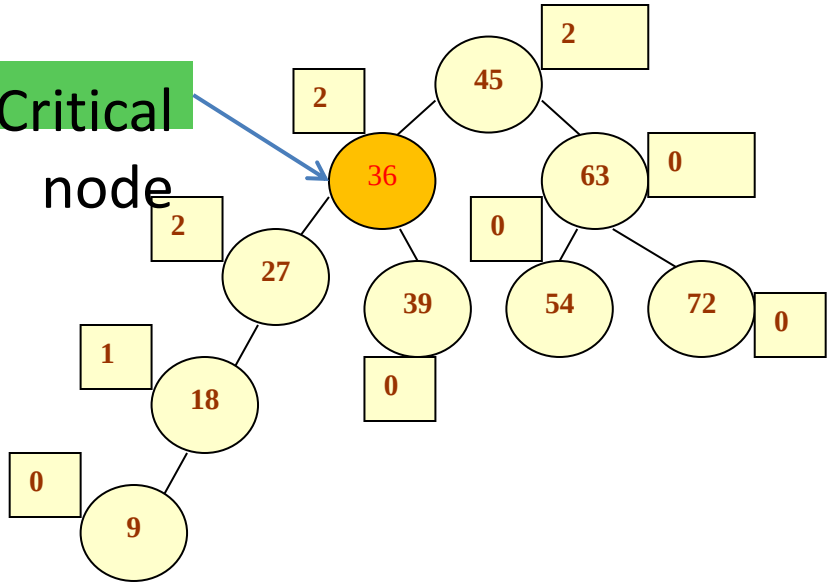


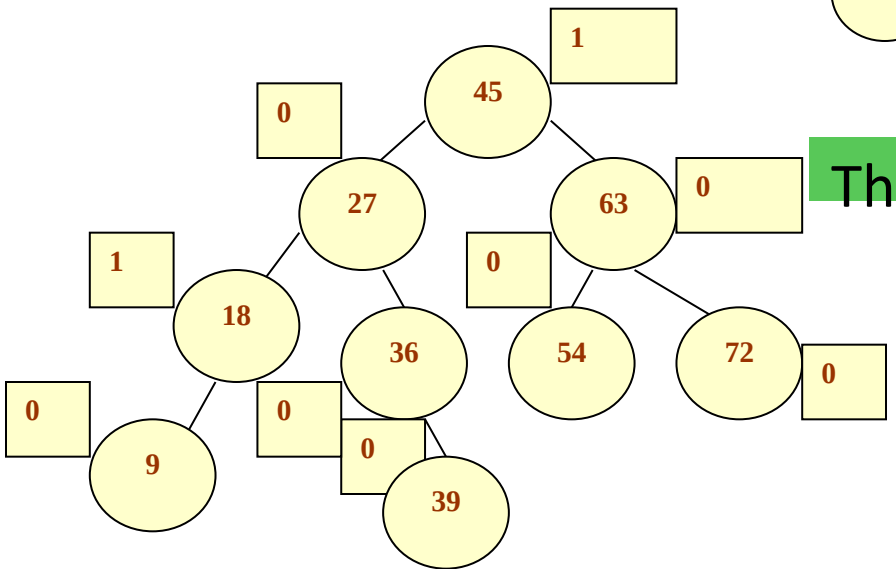
Case 1: R-Rotations to Balance AVL Tree

Example: Consider the AVL tree given below and insert 9 into it. LL case



Critical node



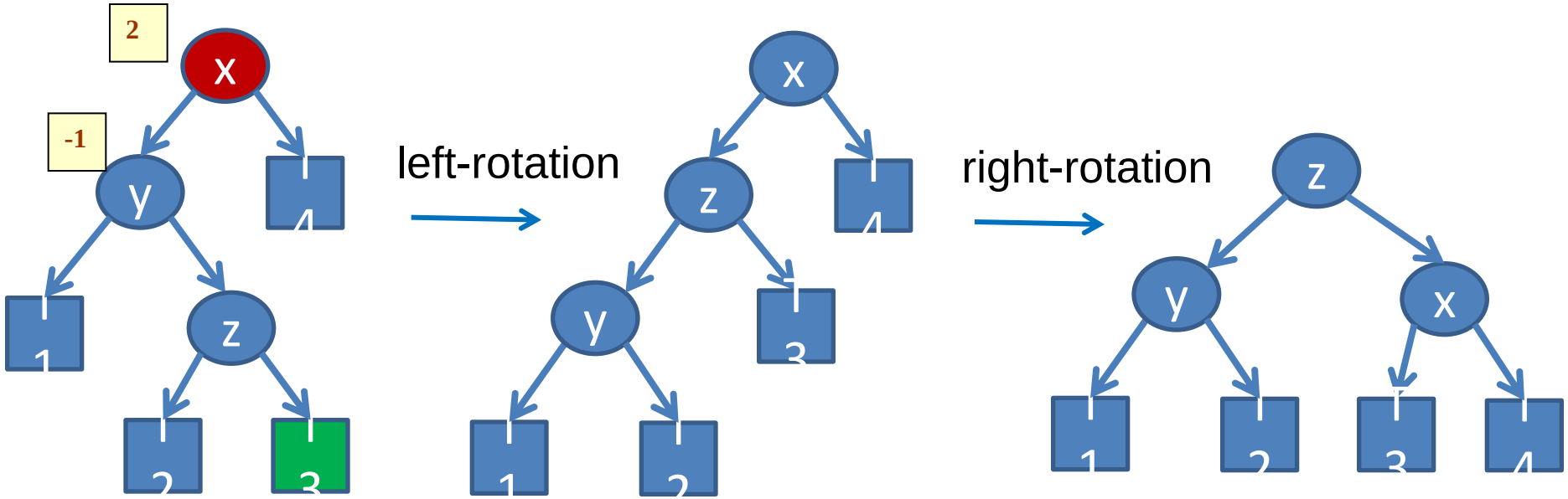


The tree is balanced using R-rotation

Case2: L-R-Rotation for Re-balancing

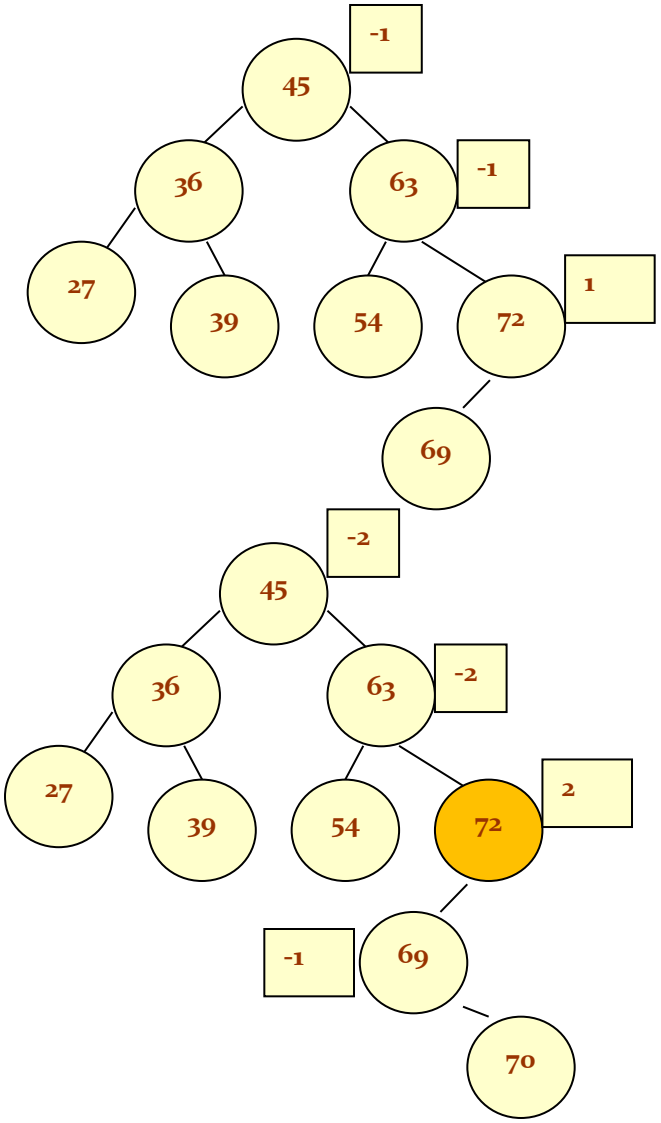
Case 2 balance-factor(x) = 2, balance-factor(x->left) < 0

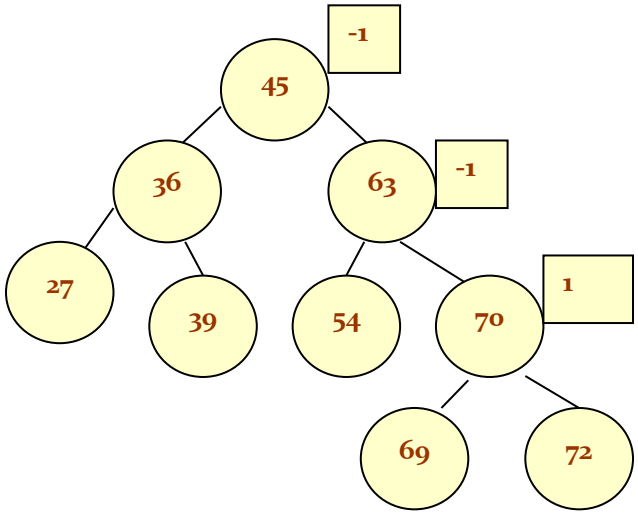
x->left = left_rotate(x->left); right_rotation(x);



Case 2: L-R-Rotation

Example: Consider the AVL tree given below and insert 70 into it.



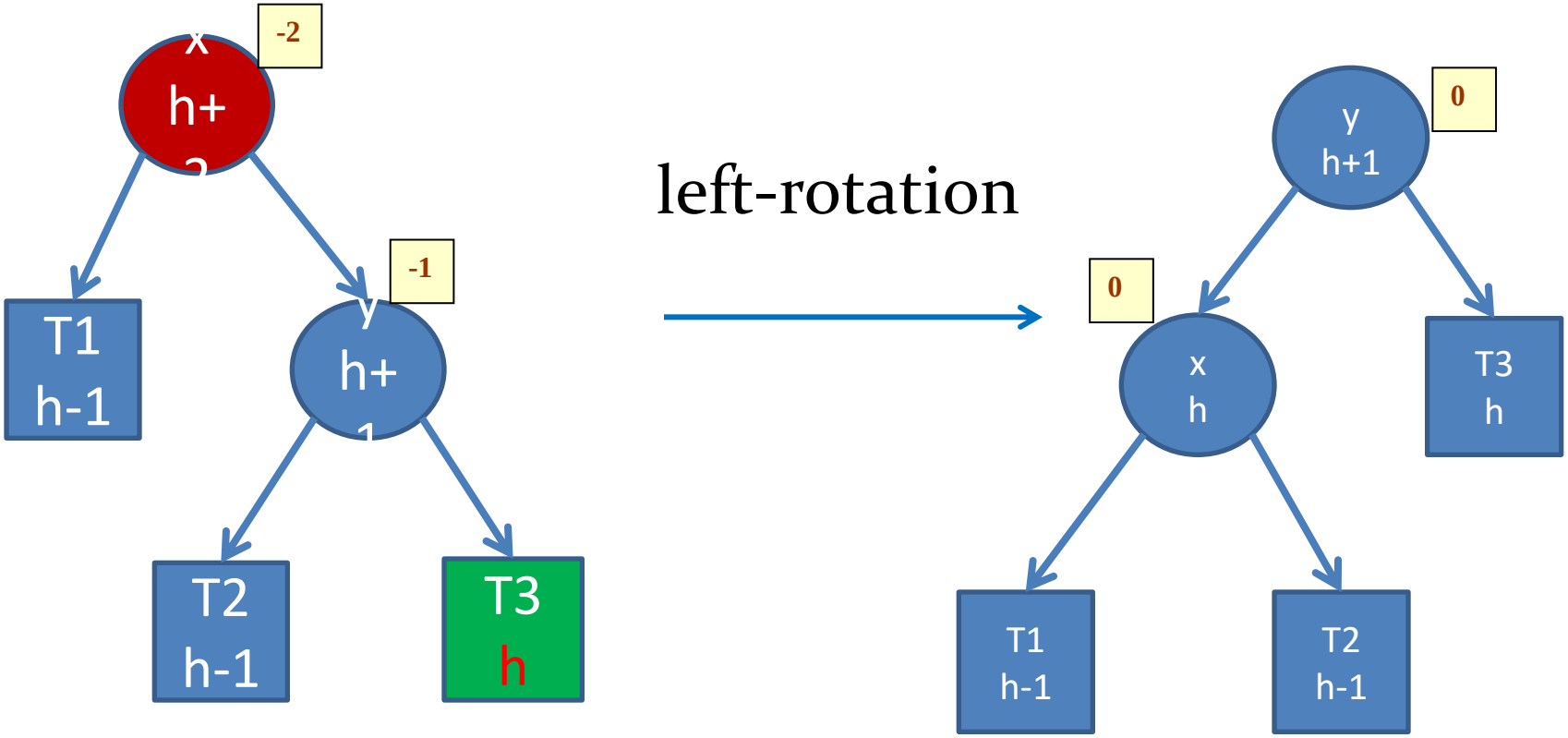


The tree is balanced using L-R-rotation

Case3:left-rotation to re-balance

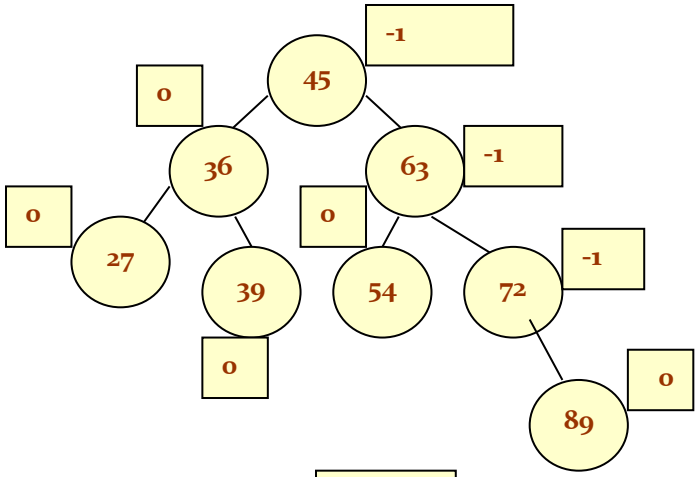
Case 3: balance-factor(x) = -2, balance-factor(x->right) <= 0

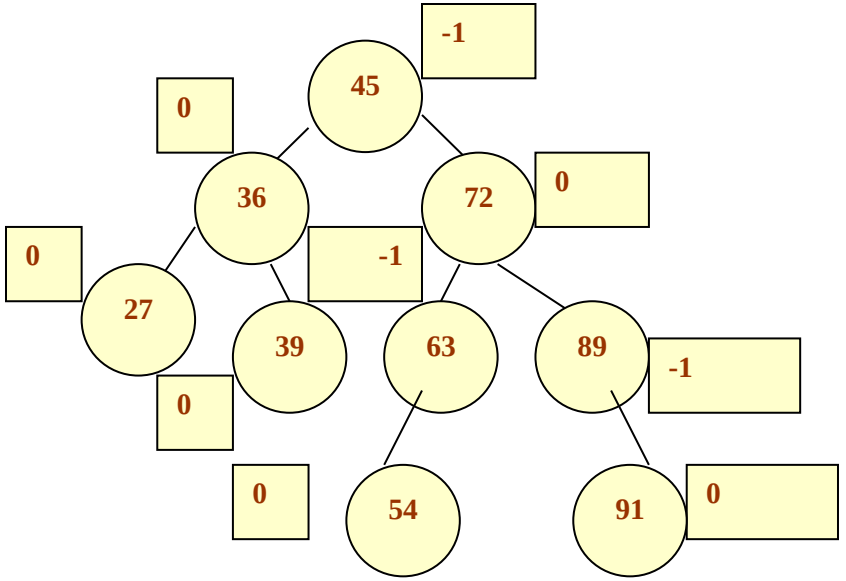
left_rotation(x);

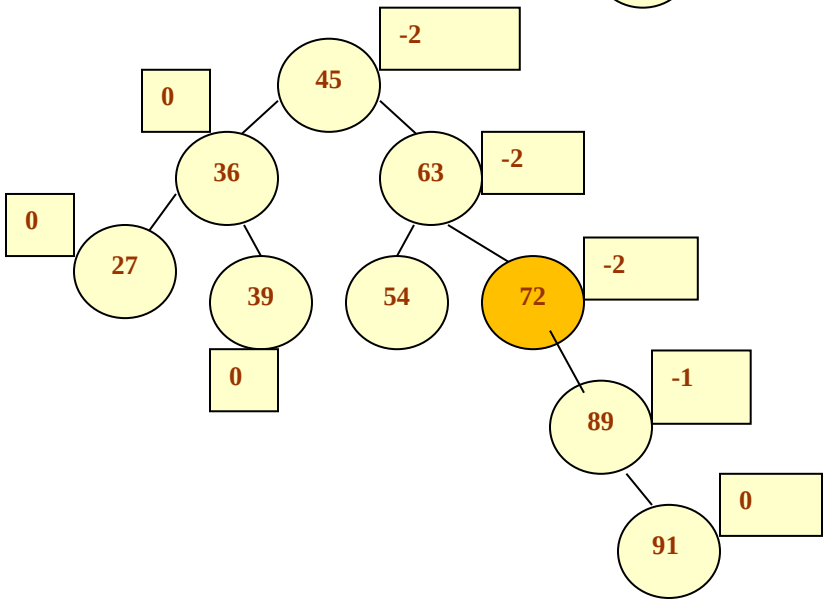


Case 3. L-Rotation to Balance

Example: Consider the AVL tree given below and insert 91 into it. RR-case





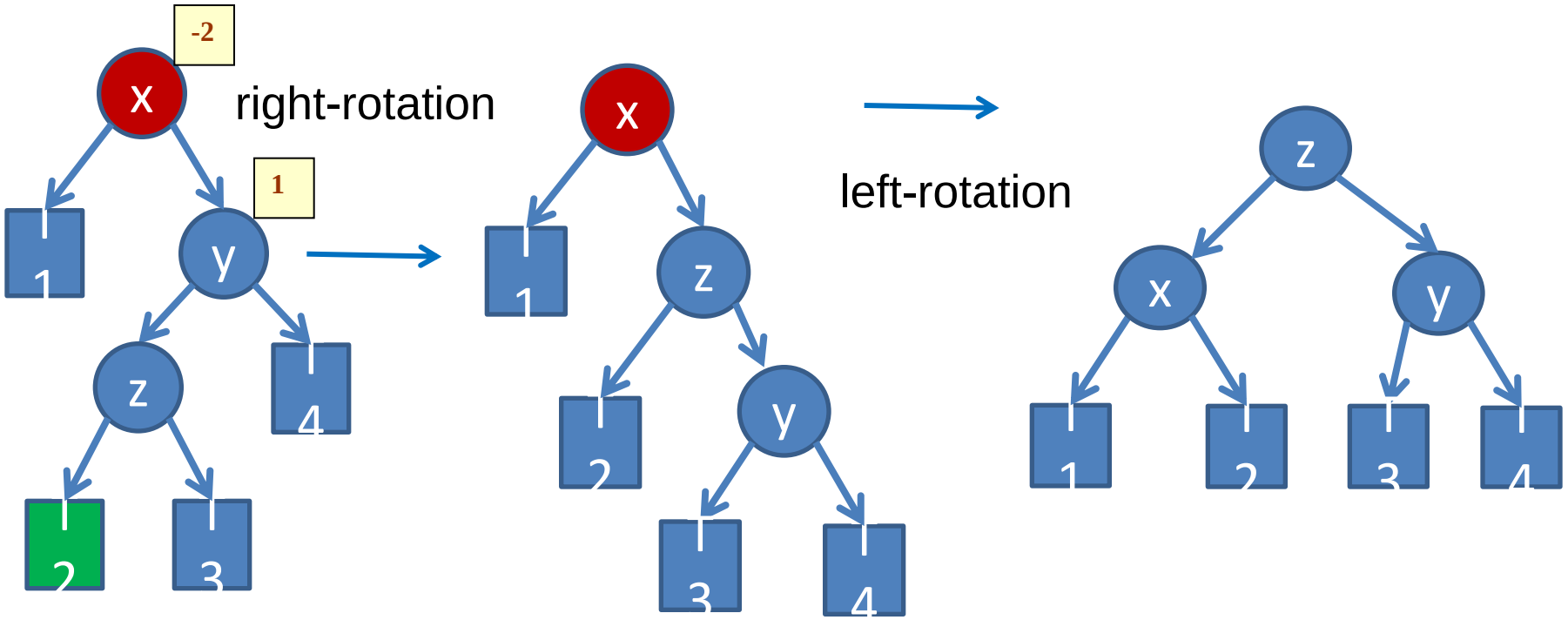


The tree is balanced using L-rotation

Case 4: R-L-Rotation re-balancing

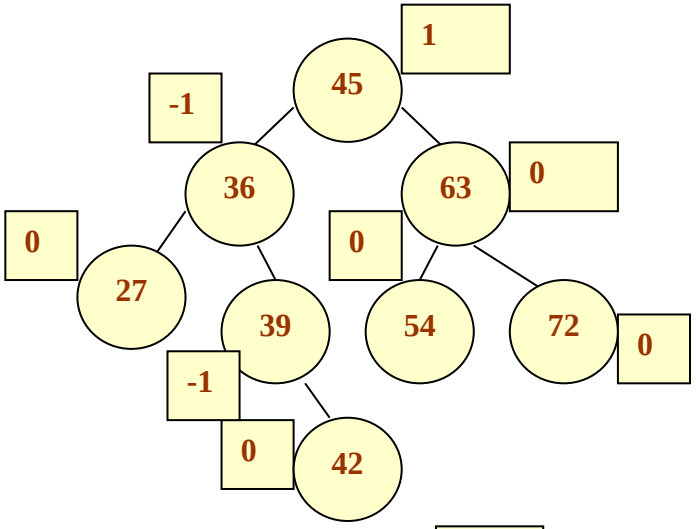
Case 4. balance-factor(x) = -2, balance-factor(x->right) > 0

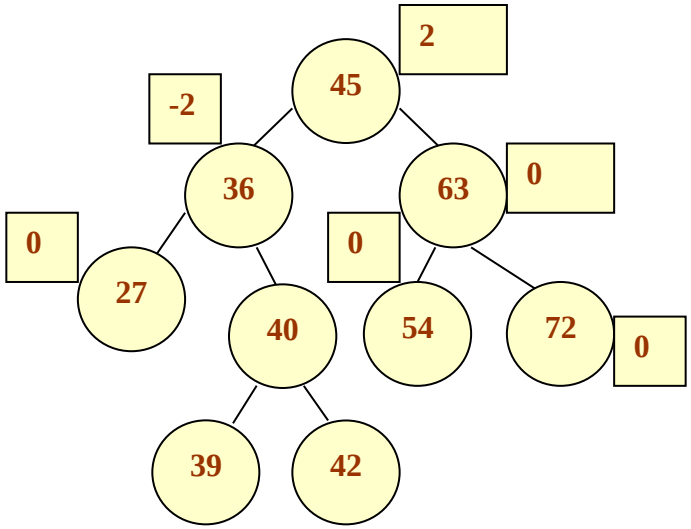
x->right = right_rotate(x->right); left_rotation(x);

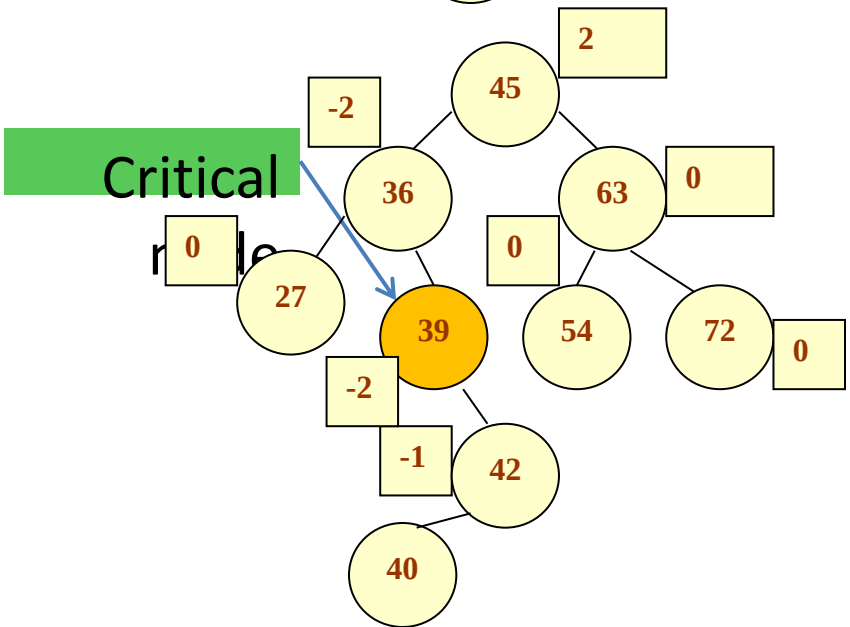


Case 4. R-L-Rotation to Balance

Example: Consider the AVL tree given below and insert 40 into it.







Critical node

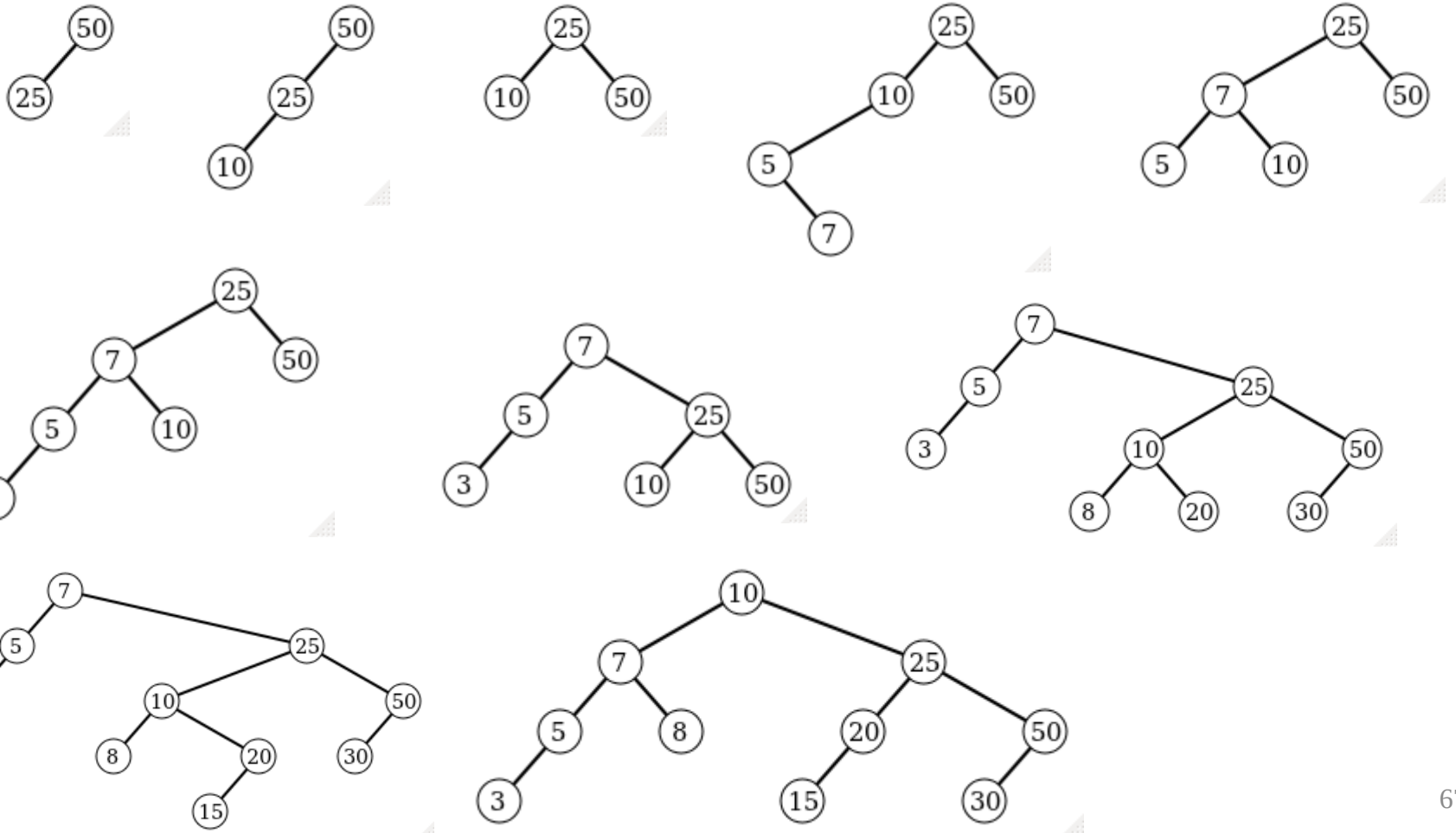
The tree is balanced using R-L rotation

Example of AVL tree insert

Insert 50, 25, 10, 5, 7, 3, 30, 20, 8, 15 into AVL tree

Example of AVL tree insert

Insert 50, 25, 10, 5, 7, 3, 30, 20, 8, 15 into AVL tree



Deleting a Node from an AVL Tree

- Deleting algorithm

  Step 1. delete node as BST

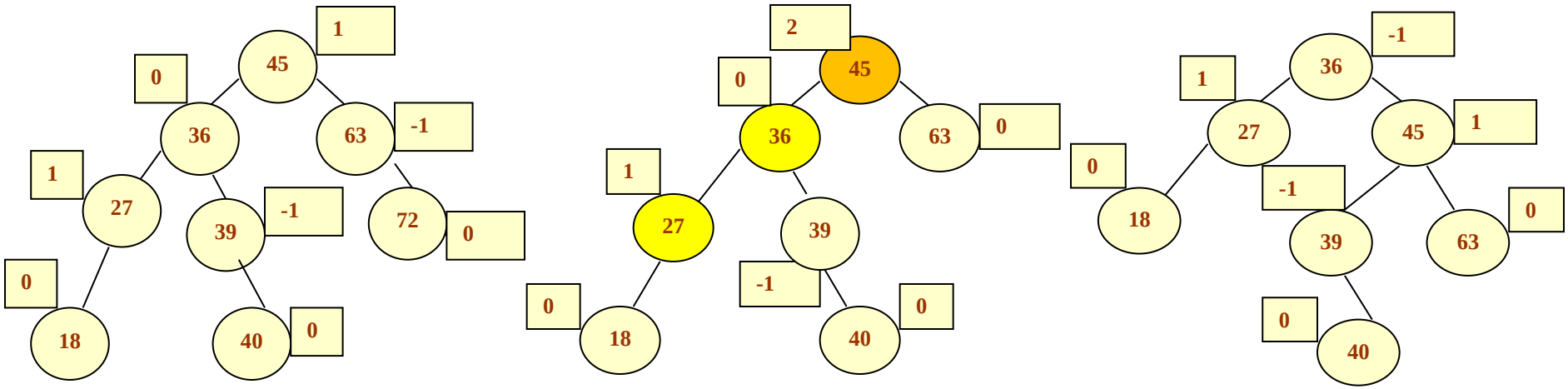
  Step 2. if not height balanced, rebalance by rotation

If the resulting BST of step 1 is not height-balanced, find the critical node. There are four possible cases similar to the inserting. Then do corresponding rotation by the case

Deleting a Node from an AVL Tree

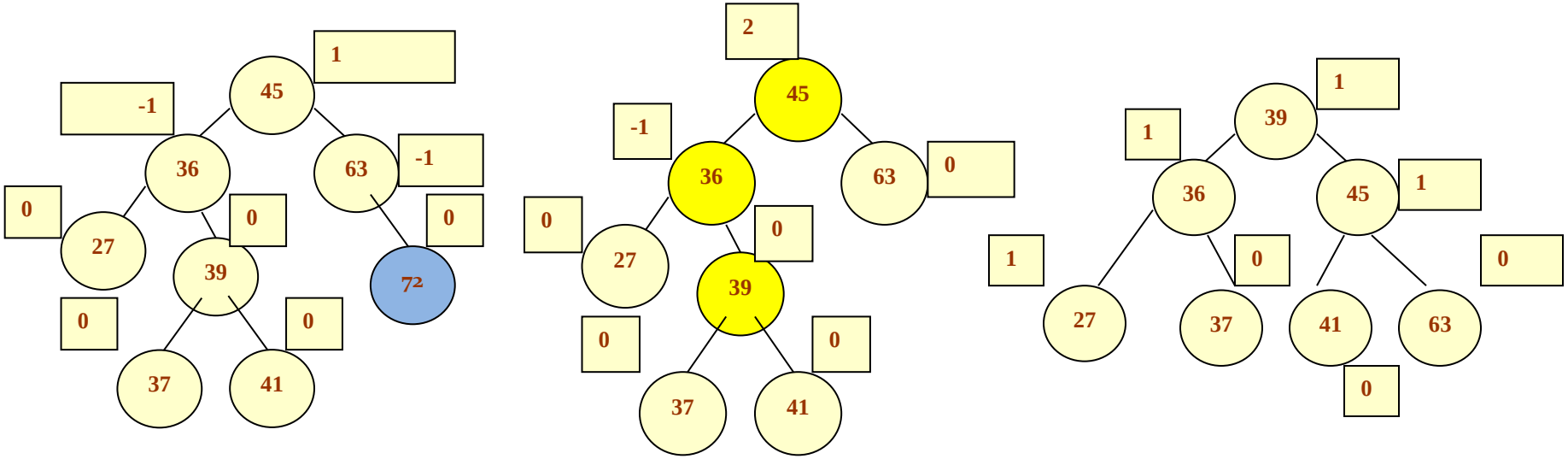
Example: Consider the AVL tree given below and delete 72 from it.

Case 1. R-rotation



Deleting a Node from an AVL Tree

- Consider the AVL tree given below and delete 72 from it.
- Case 2. L-R-rotation



AVL implementation node design

```
struct Node
{
    int key;
    int height;   // or using balance factor
    int data;     // application data associated with key
    struct Node *left, *right;
};
```

AVL right-rotate

```
node *right_rotate(node *y)  {
    node *x = y->left;
    node *T2 = x->right;
    // perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
     return x;
}
```

http://www.geeksforgeeks.org/avl-tree-set-2-deletion/

AVL left-rotate

```
node *left_rotate(node *x)
{
    node *y = x->right;
    node *T2 = y->left;
   //  perform rotation
    y->left = x;
    x->right = T2;
    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    return y;
}
```

AVL insert

```
node* insert(node* root, int key) {
   if (root == NULL)  return(new_node(key));
   if (key < root->key)
      root->left  = insert(root->left, key);
   else if (key > root->key)
      root->right = insert(root->right, key);
   else return root;

   root->height = 1 + max(height(root->left), height(root->right));
    int balance = balance_factor(root);

   // do rotation by cases
```

AVL implementation insert

```
if (balance > 1 &&  balance_factor(root->left) >=0 )
   return right_rotate(root);
if (balance < -1 && balance_factor(root->left) <=0 )
   return left_rotate(root);
if (balance > 1 && balance_factor(root->left) < 0)  {
   root->left =  left_rotate(root->left);
   return right_rotate(root);
}
if (balance < -1 && balance_factor(root->left) > 0)  {
   root->right = right_rotate(root->right);
   return left_rotate(root);
}
return root;     }
```