




TREES



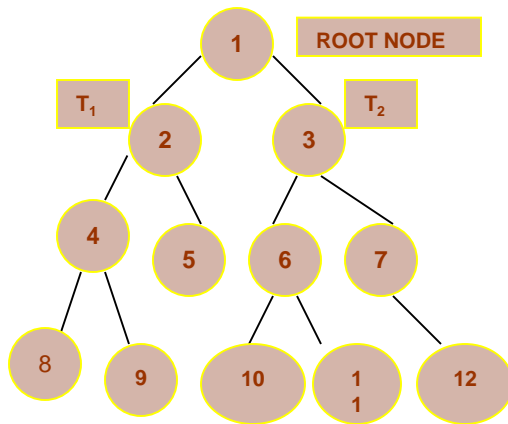
Dr. Maumita Chakraborty
University of Engineering and Management Kolkata

TREES-I



BINARY TREES

- A collection of elements called nodes. Every node contains a "left" pointer, a "right" pointer, and a data element.
- Has a root element pointed by a "root" pointer to the topmost node in the tree. If root = NULL, tree is empty.
- If the root node R is not NULL, then the two trees T_1 and T_2 are the left and right subtrees of R .
- If T_1 is non-empty, then T_1 is said to be the left successor of R . Likewise, if T_2 is non-empty then, it is called the right successor of R .



✓ In a binary tree every node has 0, 1 or at the most 2 successors.

✓ A node that has no successor is the leaf node or the terminal node.

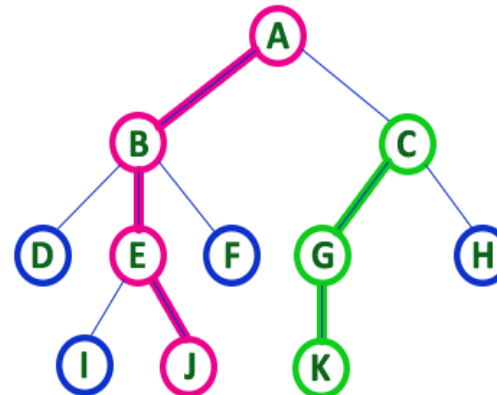
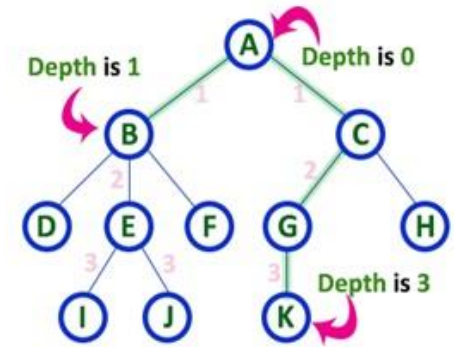
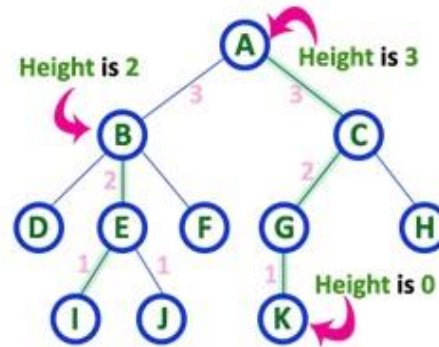
KEY TERMS

- ◉ **Sibling:** All nodes that are at the same level and share the same parent are called *siblings* (brothers).
 - **Level number:** Every node in the binary tree is assigned a *level number*. The root node is defined to be at level 0. All child nodes are defined to have level number as parent's level number + 1.
 - ◉ **Degree:** The number of children that a node has. The degree of a leaf node is zero.
 - **In-degree** of a node is the number of edges arriving at that node. The root node is the only node that has an in-degree equal to zero.
 - **Out-degree** of a node is the number of edges leaving that node.
 - **Leaf node:** A leaf node has no children.
-



KEY TERMS

- ▶ **Height:** Total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.
- ▶ **Depth:** Total number of edges from root node to a particular node is called as DEPTH of that node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. The highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.
- ▶ If there are n nodes in binary tree, maximum height of the binary tree is $n-1$ and minimum height is $\text{floor}(\log_2 n)$.
- ▶ **Path:** The sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

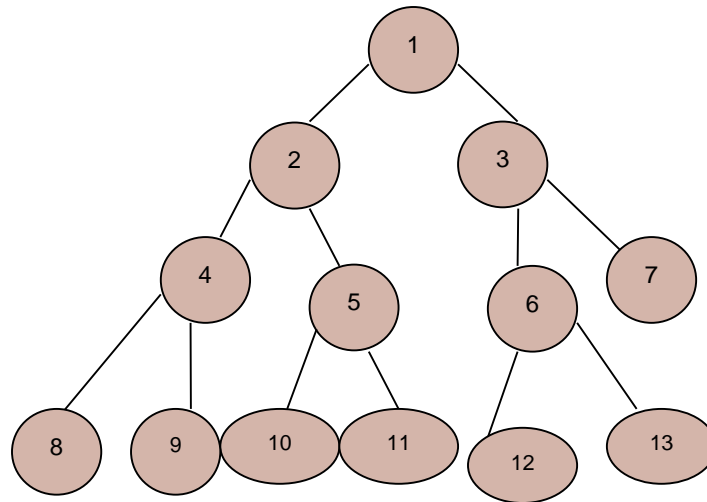
A - B - E - J

Here, 'Path' between C & K is

C - G - K

Complete Binary Trees

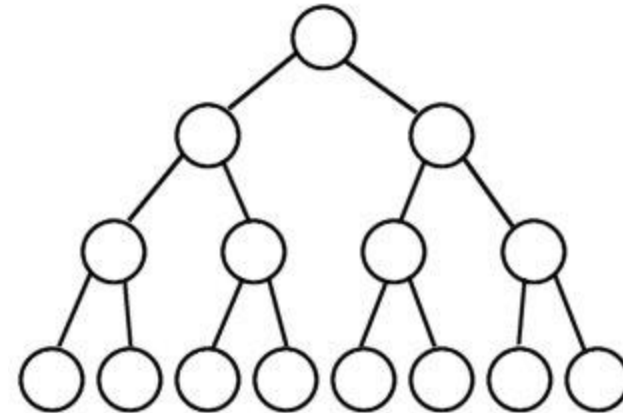
- ⦿ A *complete binary tree* is a binary tree which satisfies two properties.
 - ⦿ First, in a complete binary tree every level, except possibly the last, is completely filled.
 - ⦿ Second, all nodes appear as far left as possible.



-
- ▶ **Full Binary Tree:** If every node (other than leaves) has 2 children.

- ▶ **Strictly Binary Tree:** If every node (other than leaves) has either 0 or 2 children.

Full Binary Tree



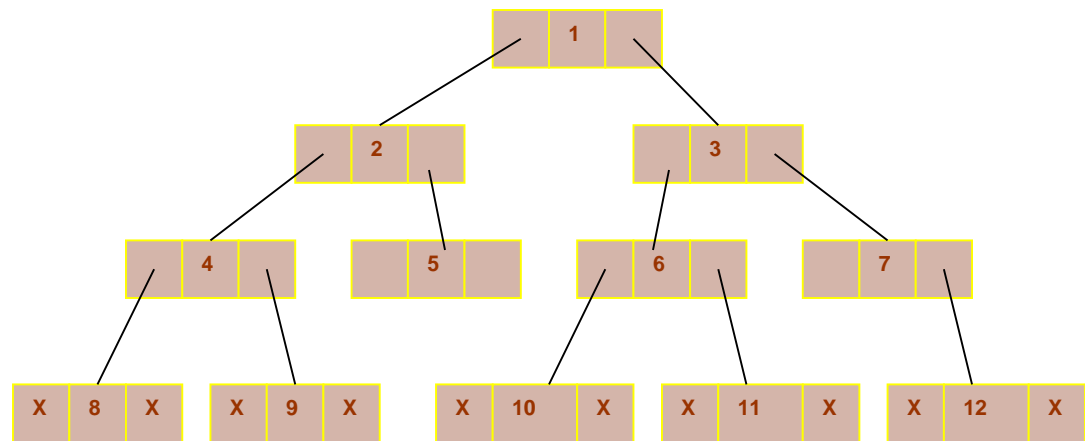
Representation of Binary Trees in Memory

- ◉ In computer's memory, a binary tree can be maintained either using a linked representation (as in case of a linked list) or using sequential representation (as in case of single arrays).

Linked Representation of Binary Trees

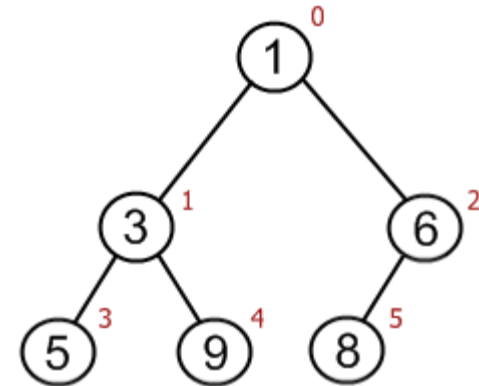
- ◉ In linked representation of binary tree, every node will have three parts: the data element, a pointer to the left node and a pointer to the right node.
- ◉ In C, the binary tree is built with a node type given as below.

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



Sequential Representation of Binary Trees

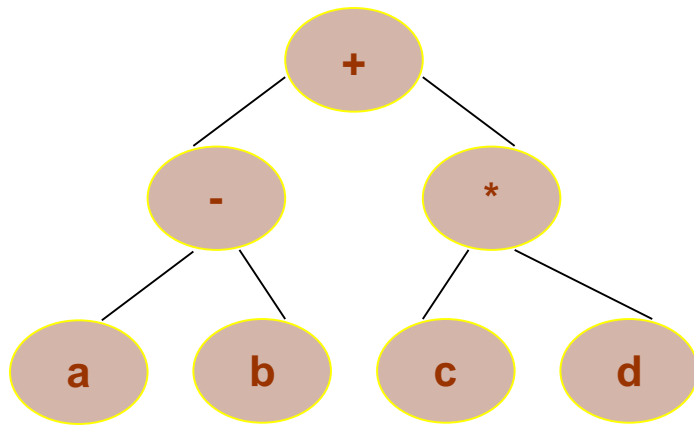
- Done using single or one dimensional array.
- Simplest technique for memory representation but very inefficient as it requires a lot of memory space. A sequential binary tree follows the rules given below:
 - ⦿ One dimensional array called TREE, will be used.
 - ⦿ The root of the tree will be stored in the first location. That is, TREE[0] will store the data of the root element.
 - ⦿ The maximum size of the array TREE is given as $(2^{d+1}-1)$, where d is the depth of the tree.
 - ⦿ An empty tree or sub-tree is specified using NULL. If TREE[0] = NULL (or default value), then the tree is empty.



1	3	6	5	9	8
0	1	2	3	4	5

EXPRESSION TREES

- Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression Exp given as,
$$\text{Exp} = (a - b) + (c * d)$$
- This expression can be represented using a binary tree as shown in figure below:



- Inorder Traversal of an Expression Tree results into an Infix expression.
- Similarly, preorder and postorder traversal of an expression tree result into prefix and postfix expression.



TRAVERSING OF A BINARY TREE

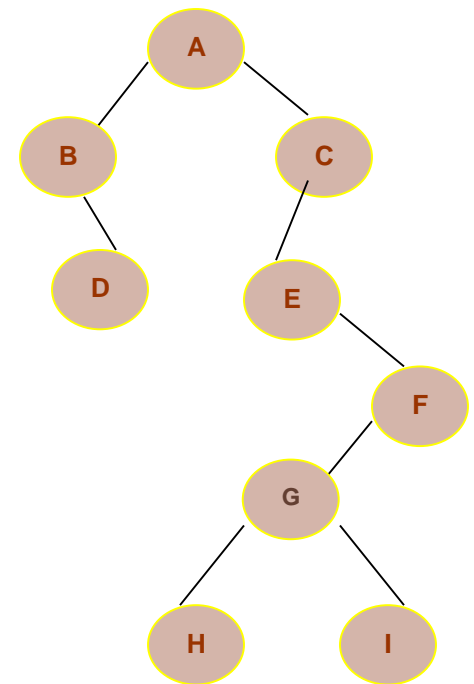
- ◉ Process of visiting each node in the tree exactly once, in a systematic way.
- ◉ Tree, being a non-linear data structure, the elements can be traversed in many different ways.
- ◉ Different algorithms for tree traversals differ in the order in which the nodes are visited.

- ◉ **Pre-order traversal**

Following operations are performed recursively at each node. The algorithm starts with the root node of the tree and continues by:

- ◉ Visiting the parent node.
- ◉ Traversing the left subtree.
- ◉ Traversing the right subtree.

A, B, D, C, E, F, G, H and I



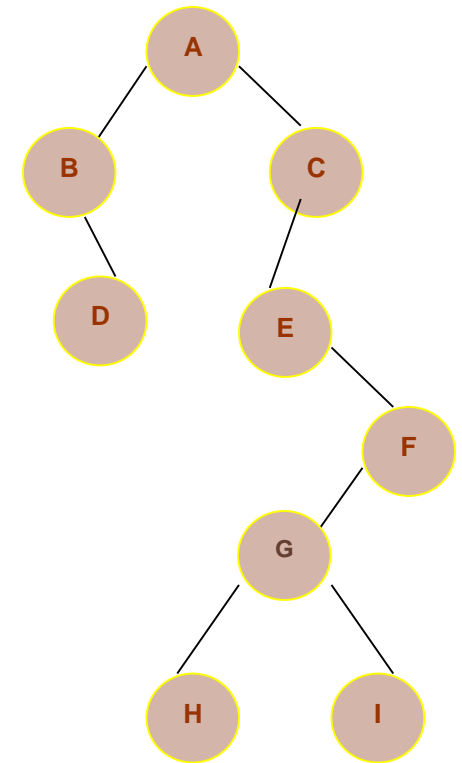
TRAVERSING OF A BINARY TREE

In-order traversal

Following operations are performed recursively at each node. The algorithm starts with the root node of the tree and continues by:

- ⦿ Traversing the left subtree.
- ⦿ Visiting the parent node.
- ⦿ Traversing the right subtree.

B, D, A, E, H, G, I, F AND C



Post-order traversal

Following operations are performed recursively at each node. The algorithm starts with the root node of the tree and continues by:

- ⦿ Traversing the left subtree.
- ⦿ Traversing the right subtree.
- ⦿ Visiting the parent node.

D, B, H, I, G, F, E, C and A



Non-recursive In-order Traversal

► nonRcrsvInorder(stack *s, bNode *T)

1. Start
 2. while(1)
 1. while(T≠NULL)
 1. push (s, T)
 2. T := T->left
 2. End while
 3. if (isEmpty(s))
 1. return
 4. End if
 5. T := pop(s)
 6. Print (T->data)
 7. T := T->right
 3. End while
-



TREES-II

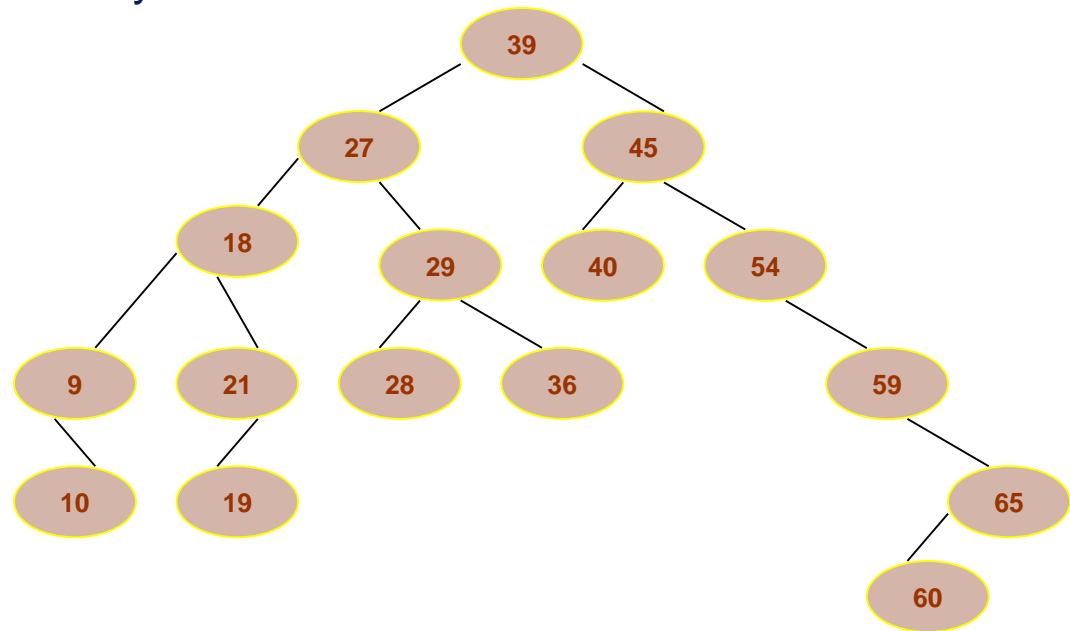


Binary Search Trees



BINARY SEARCH TREES

- ⦿ A variant of binary tree in which the nodes are arranged in order such that
 - ⦿ All the nodes in the left sub-tree have a value less than that of the root node, and all the 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.
- ⦿ At every step, we eliminate half of the sub-trees from the search process. Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.



Searching a Value in BST

- Search algorithm 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 in a BST

```
Step 1: IF ROOT->DATA = VAL
Step 2:         Return 1
Step 3: ELSE IF ROOT=NULL
Step 4:         Return 0
Step 5: ELSE IF VAL < ROOT->DATA
Step 6:         Return SEARCH(ROOT->LEFT, VAL)
Step 7: ELSE
Step 8:         Return SEARCH(ROOT->RIGHT, VAL)
Step 9: END IF
Step 10: End
```

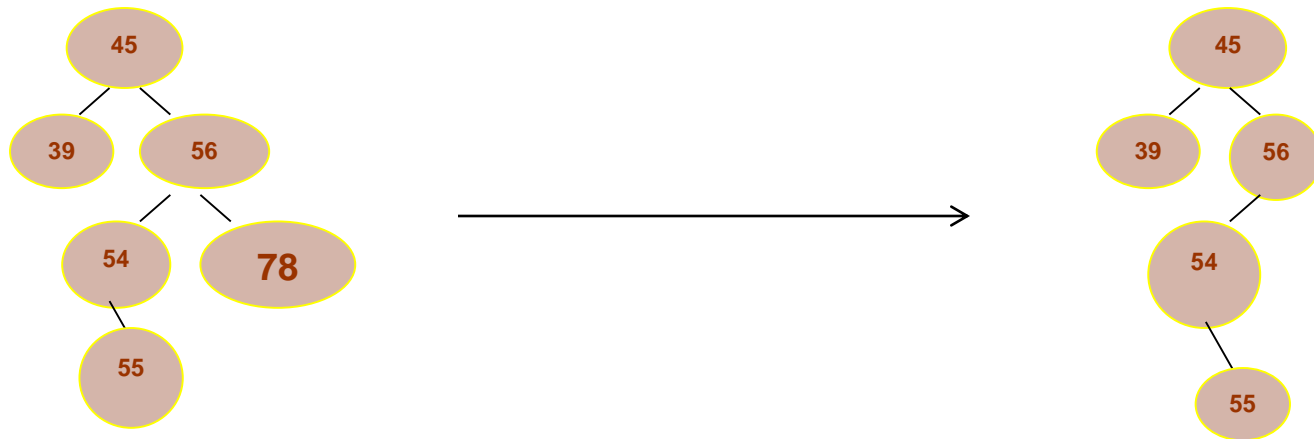
Insertion in a BST

Algorithm to insert a value in the binary search tree-

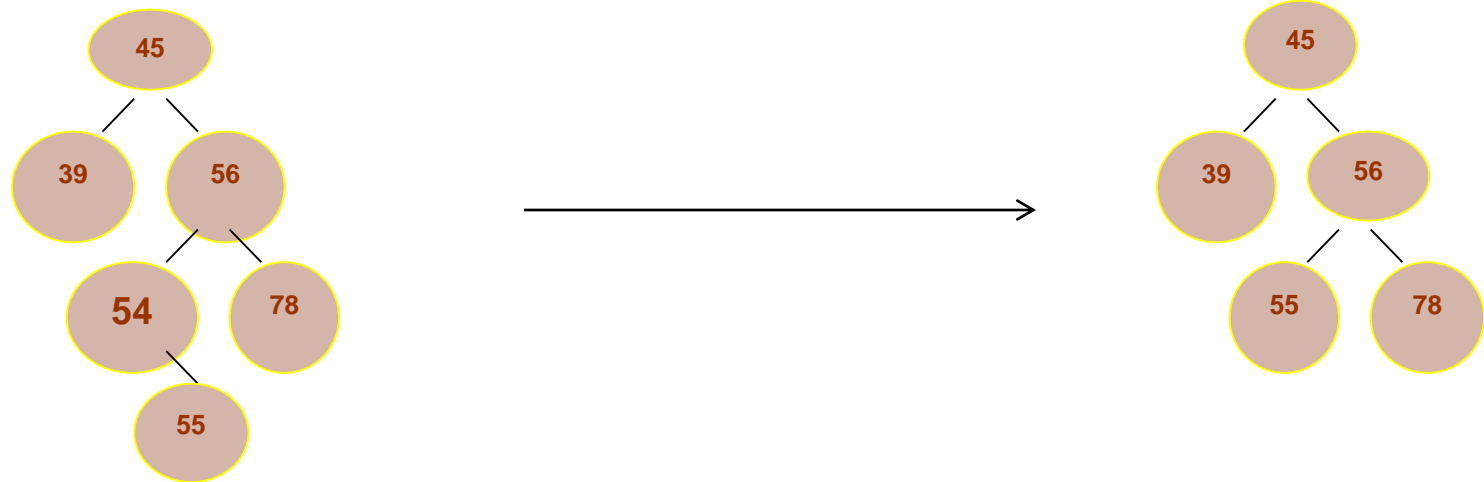
```
Step 1: IF (ROOT->DATA > NEW_NODE->DATA)
Step 2:     IF (ROOT->LEFT = NULL)
Step 3:         ROOT->LEFT = NEW_NODE
Step 4:     ELSE
Step 5:         INSERT_TREE (ROOT->LEFT, NEW_NODE)
Step 6:     END IF
Step 7: ELSE
Step 8:     IF (ROOT->RIGHT = NULL)
Step 9:         ROOT->RIGHT = NEW_NODE
Step 10:    ELSE
Step 11:        INSERT_TREE (ROOT->RIGHT, NEW_NODE)
Step 12:    END IF
Step 13: END IF
Step 14: END
```

Deletion from a BST

- During deletion, utmost care should be taken that the properties of the binary search tree does not get violated and nodes are not lost in the process. The deletion of a node can be done in any of the three cases.
- Case 1: Deleting a node that has no children.**



- ◉ **Case 2: Deleting a node with one child (either left or right).** Replace the node with its child. Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent. Similarly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.



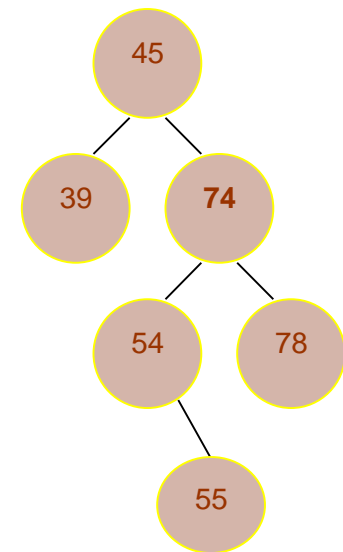
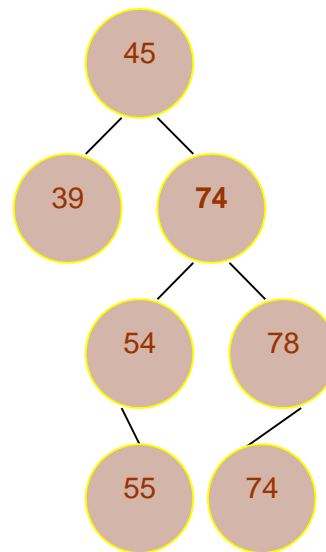
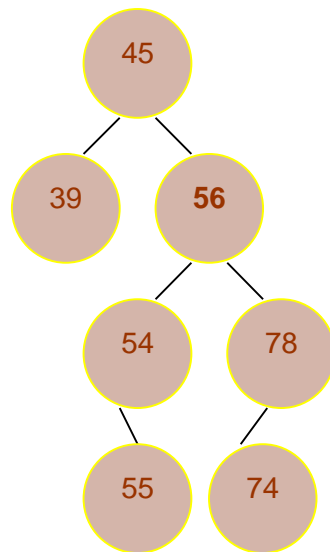
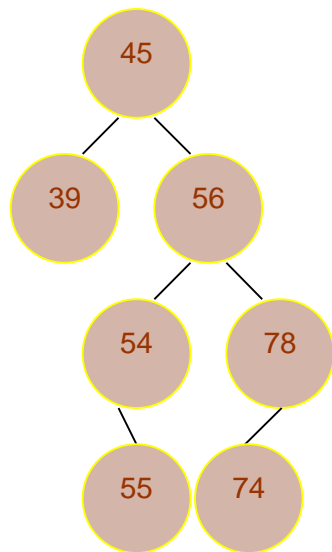
Algorithm for deletion of a node in BST (Case 1 and Case 2)

Deletion AB (root, loc, parent)

1. If ((loc->left=NULL) & (loc->right=NULL))
2. child=NULL
3. Else if (loc->left ≠ NULL)
4. child := loc->left
5. Else
6. child := loc->right
7. End if
8. If (parent ≠ NULL)
 1. If (loc = parent->left)
 2. parent->left := child
 3. Else
 4. parent->right :=child
9. Else
10. Root := child
11. End if
12. loc->right := loc->left := NULL
13. Free(loc)
14. Return Root



Case 3: Deleting a node with two children. Replace the node's value with its in-order successor (leftmost child of the right sub-tree). The in-order successor can then be deleted using any of the previous cases.



Algorithm for deletion of a node in BST (Case 3)

Deletion_C (root, loc, parent)

1. in_suc := loc->right
2. par_in_suc := loc
3. While (in_suc->left ≠ NULL)
4. par_in_suc := in_suc
5. in_suc := in_suc->left
6. End while
7. loc->data := in_suc->data
8. Deletion_AB(root, in_suc, par_in_suc)
9. Return root



Algorithm to delete a value in the binary search tree

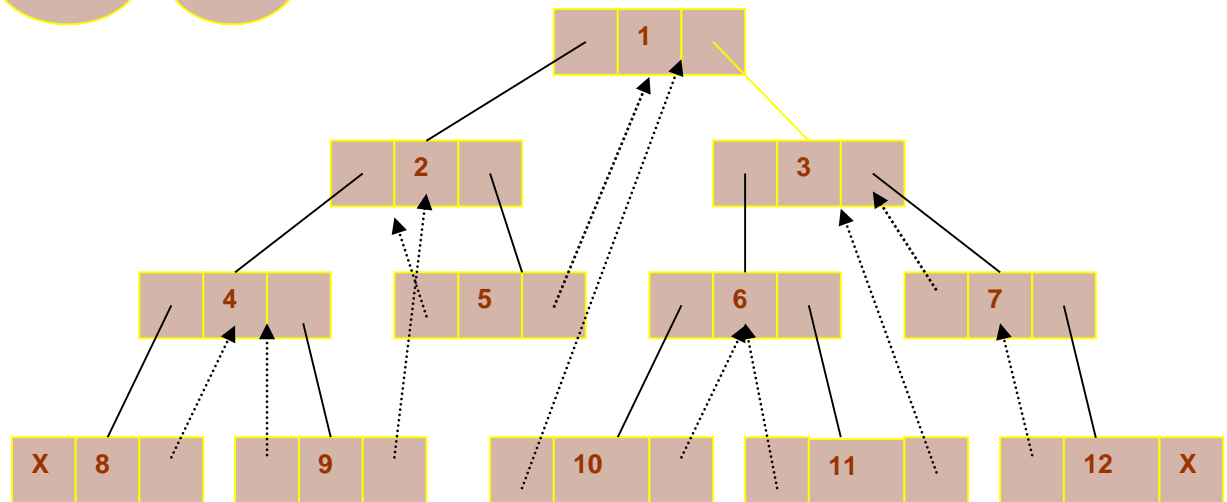
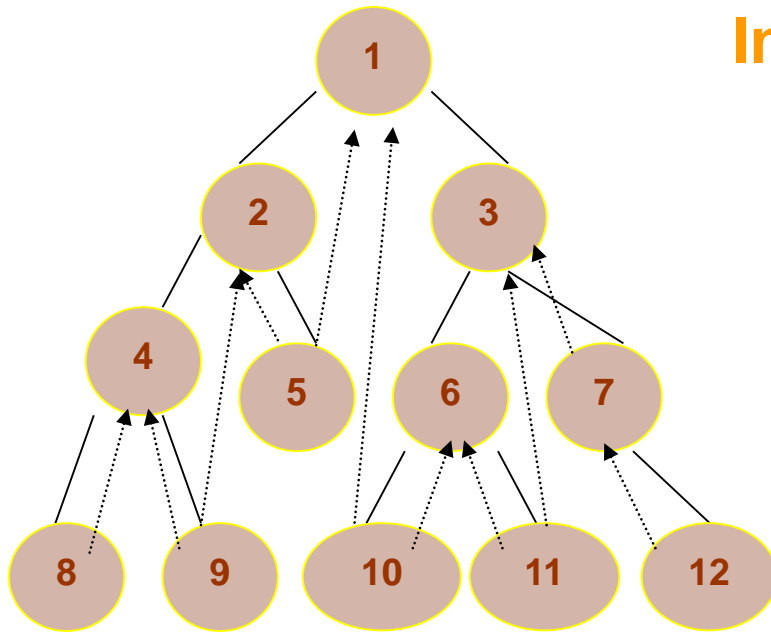
```
Step 1: IF ROOT = NULL, then
Step 2:     RETURN NULL
Step 3: ELSE IF VAL < ROOT->DATA
Step 4:     ROOT->LEFT = Delete(ROOT->LEFT, VAL)
Step 5: ELSE IF VAL > ROOT->DATA
Step 6:     ROOT->RIGHT = Delete(ROOT->RIGHT, VAL)
Step 7: ELSE
Step 8:     IF ROOT->LEFT = NULL, then
Step 9:         SET TEMP = ROOT->RIGHT
Step 10:        FREE (ROOT)
Step 11:        RETURN TEMP
Step 12:    ELSE IF ROOT->RIGHT = NULL
Step 13:        TEMP = ROOT->LEFT
Step 14:        FREE (ROOT)
Step 15:        RETURN TEMP
Step 16:    END IF
Step 17:    TEMP = Inorder_Successor(ROOT->RIGHT)
Step 18:    ROOT->DATA = TEMP->DATA
Step 19:    ROOT->RIGHT = Delete(ROOT->RIGHT, TEMP->DATA)
Step 20: END IF
Step 21: RETURN ROOT
Step 22: End
```


THREADED BINARY TREE

- ◉ Space that is wasted in a binary tree in storing a NULL pointer in the linked representation of trees can be efficiently used to store some other useful piece of information, like the in-order predecessor, or the in-order successor of the node.
- ◉ These special pointers are called threads and binary trees containing threads are called threaded trees. In the linked representation of a threaded binary tree, threads will be denoted using dotted lines.
- ◉ Presence of a thread indicates absence of that corresponding child.
- ◉ If a node has no left child, then the thread appears in the left field, and the left field will be made to point to the in-order predecessor of the node.
- ◉ On the contrary, if a node has no right child, then the thread appears in the right field, and it will point to the in-order successor of the node.
- ◉ If a node has no left and right child, then threads appear in both left and right field pointing to the in-order predecessor and successor of the node respectively.
- ◉ Similarly, we can have pre-order as well as post-order threaded tree.



In-order Threaded Binary tree



Structure of a node and its In-order Successor in a Binary Threaded Tree

- ▶ typedef struct
- ▶ {
 - ▶ int rightFlag, leftFlag;
 - ▶ char data;
 - ▶ struct bThreadNode *left, *right;
- ▶ }

Left	Left Flag	Data	Right Flag	Right
------	-----------	------	------------	-------

inorderSuccessor

(bThreadNode *root)

1. Start
2. if (root->rightFlag=1)
 1. return (root->right)
3. End if
4. root=root->right
5. while (root->leftFlag=0)
 1. root := root->left
6. End while
7. return root
8. Stop



Generation of In-order Threaded Tree

inorderThreadedTree(bThreadNode *root)

1. Start
2. $x := \text{root}$
3. while($x \rightarrow \text{leftFlag} = 0$)
 1. $x := x \rightarrow \text{left}$
4. End while
5. while($x \neq \text{NULL}$)
 1. print ($x \rightarrow \text{data}$)
 2. $x := \text{inorderSuccessor}(x)$
6. End while
7. Stop



Generation of Pre-order Threaded Tree

preorderThreadedTree(bThreadNode *root)

1. Start
 2. $X := \text{root}$
 3. while($x \neq \text{NULL}$)
 1. Print ($x \rightarrow \text{data}$)
 2. if ($\text{leftFlag} = 0$)
 1. $x := x \rightarrow \text{left}$
 3. Else if ($x \rightarrow \text{rightFlag} = 0$)
 1. $x := x \rightarrow \text{right}$
 4. Else
 1. while (($x \neq \text{NULL}$) and ($x \rightarrow \text{rightFlag} = 1$))
 1. $x := x \rightarrow \text{right}$
 2. End while
 3. if ($x \neq \text{NULL}$)
 1. $x := x \rightarrow \text{right}$
 4. End if
 5. End if
 4. Endwhile
 5. Stop
-



TREES-III



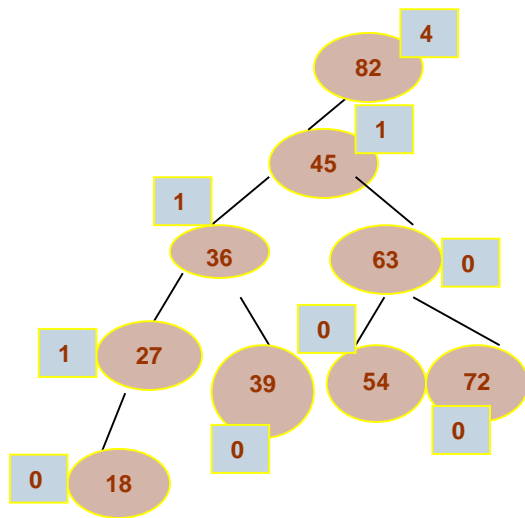
AVL TREES

- ◉ Self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one.
- ◉ Also known as a height-balanced tree.
- ◉ Advantage: It takes $O(\log n)$ time to perform search, insert and delete 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 **BalanceFactor**.

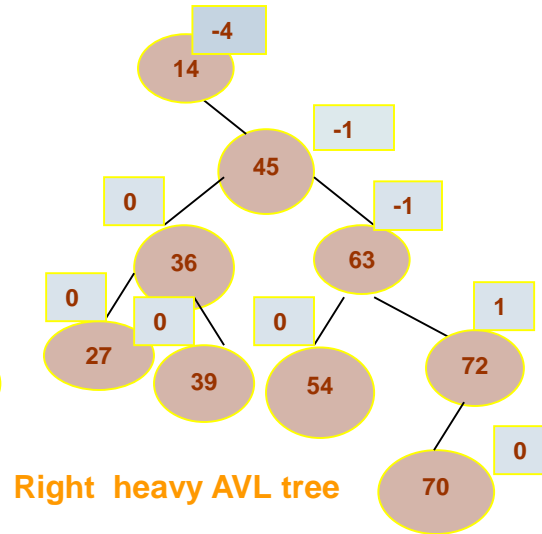


- ◎ ***Balance factor = Height (left sub-tree) – Height (right sub-tree)***
- ◎ A BST where every node has a balance factor of -1, 0 or 1. A node with any other balance factor is considered to be unbalanced and requires rebalancing the tree.
- ◎ If the balance factor of a node is more than 1, then the 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 (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- ◎ If the balance factor of a node is less than -1, then the tree is called ***Right-heavy tree***.

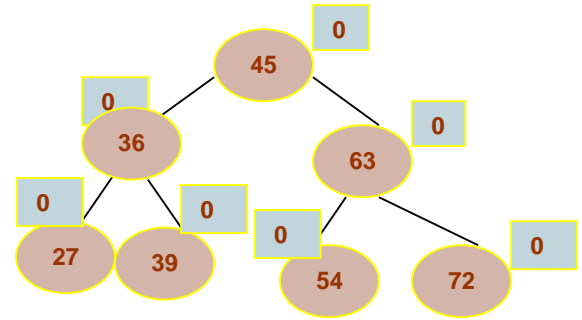




Left heavy AVL tree



Right heavy AVL tree



Balanced AVL tree

Search Operation

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.

Insertion in an AVL Tree

- ◉ As in binary search tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
 - ◉ Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.
 - ◉ During insertion, the new node is inserted as the leaf node, so it will always have balance factor equal to zero. The only nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.
-

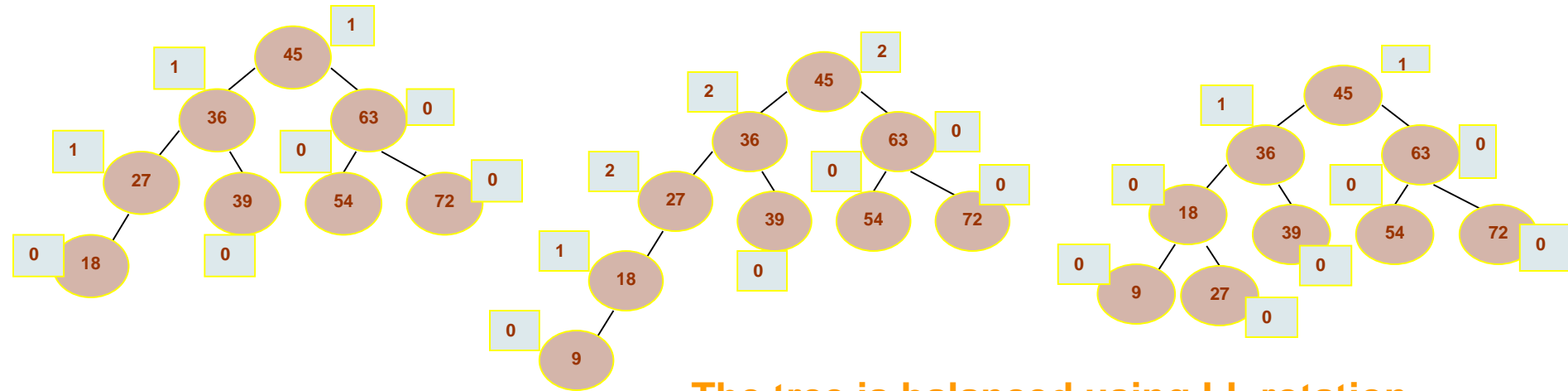


Rotations to Balance the AVL Tree After Insertion of a New Node

- ◉ To perform rotation, our first work is to find the *critical node*. Critical node is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- ◉ The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node.:
- ◉ **LL rotation:** if the left child of the critical node is left-heavy; then, a right rotation is applied to the critical node.
- ◉ **RR rotation:** if the right child of the critical node is right-heavy; then, a left rotation is applied to the critical node.
- ◉ **LR rotation:** if the left child of the critical node is right-heavy; then, a left rotation to the left sub-tree followed by a right rotation to the critical node is applied.
- ◉ **RL rotation:** if the right child of the critical node is left-heavy; then, a right rotation to the right sub-tree followed by a left rotation to the critical node is applied.

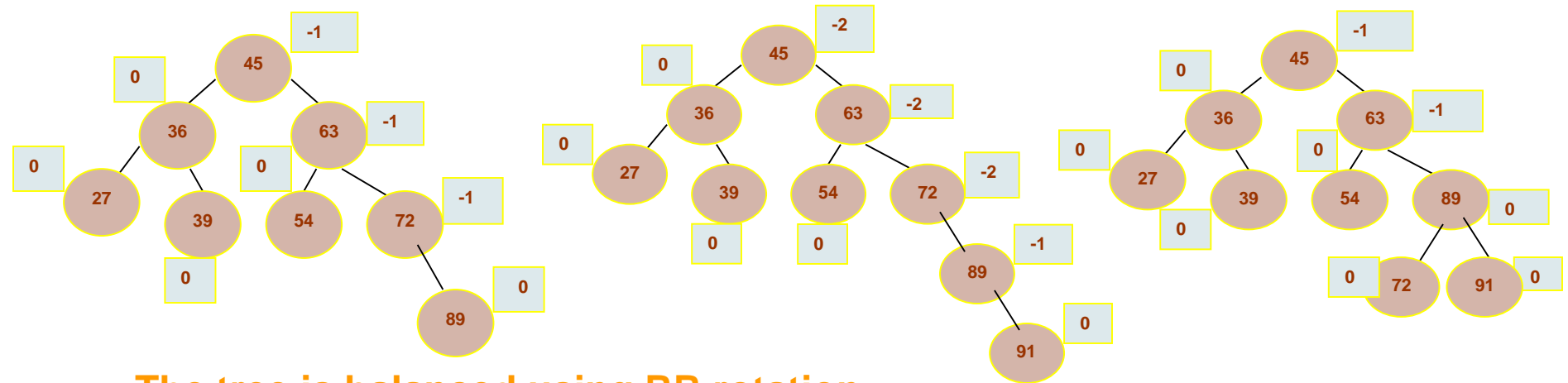


Example: Consider the AVL tree given below and insert 9 into it.



The tree is balanced using LL rotation

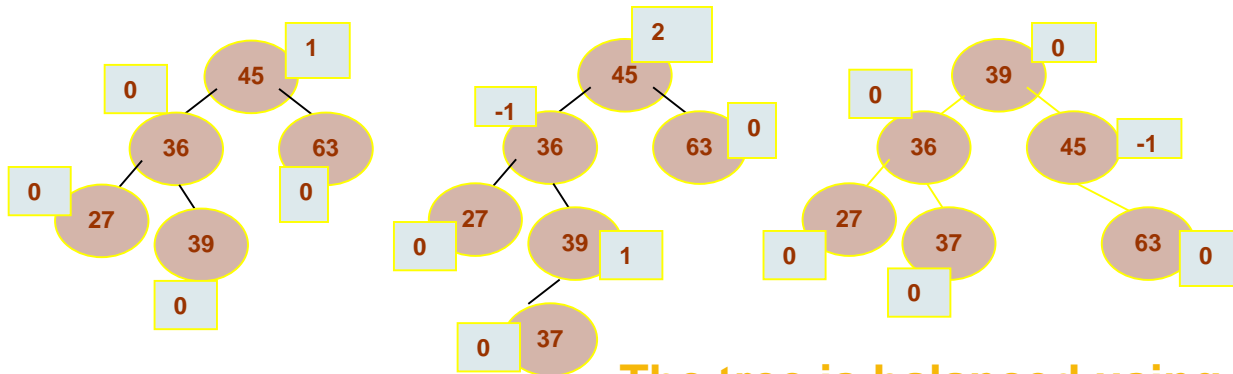
Consider the AVL tree given below and insert 91 into it.



The tree is balanced using RR rotation

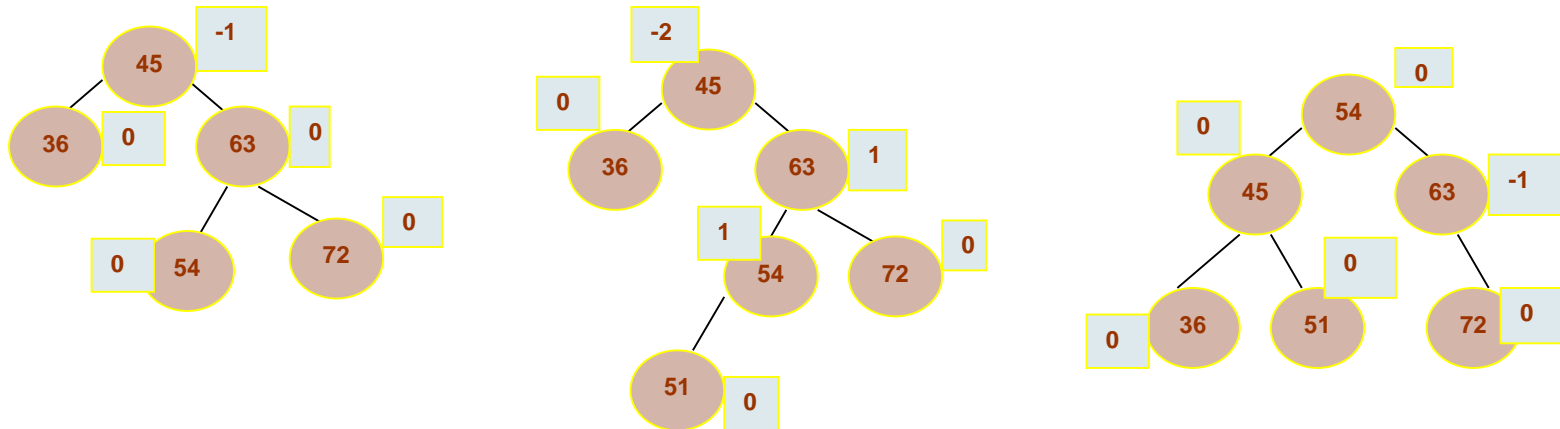


Consider the AVL tree given below and insert 37 into it.



The tree is balanced using LR rotation

Consider the AVL tree given below and insert 51 into it.



The tree is balanced using RL rotation

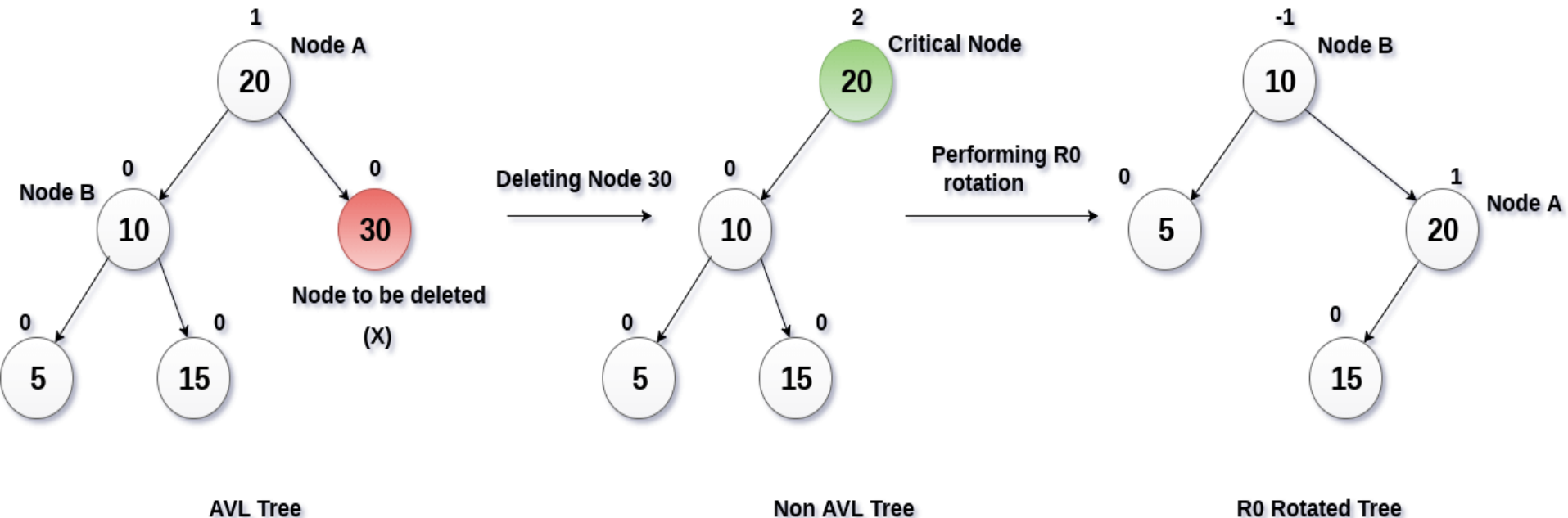


Deletion of a node from an AVL Tree

- Deletion of a node in the AVL tree is similar to that of BST. But it may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations. There are two classes of rotation that can be performed on an AVL tree after deleting a given node. These rotations are- R rotation and L rotation.
- If the node to be deleted is present in the left sub-tree of the critical node, then L rotation is applied. Else, if it is on the right sub-tree, R rotation is performed.
- Further, there are three categories of L and R rotations. The variations of L rotation are: L-1, L0 and L1 rotation. Correspondingly for R rotation, there are R0, R-1 and R1 rotations.

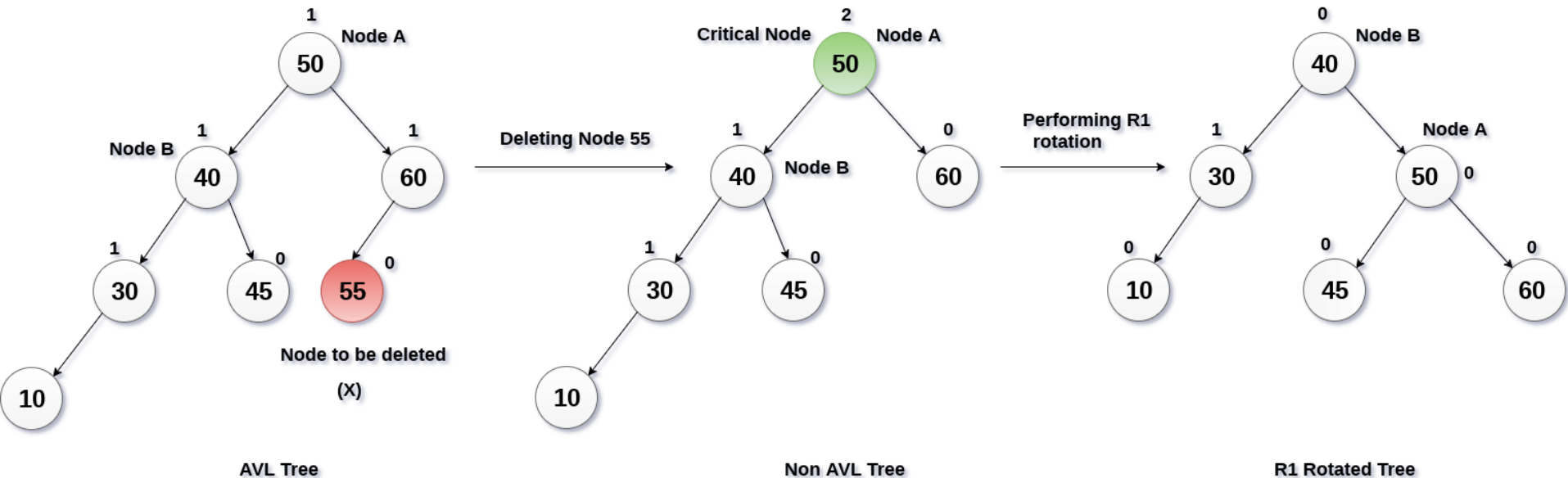
R0 Rotation

- Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0. Consider the AVL tree given below and delete 30 from it.



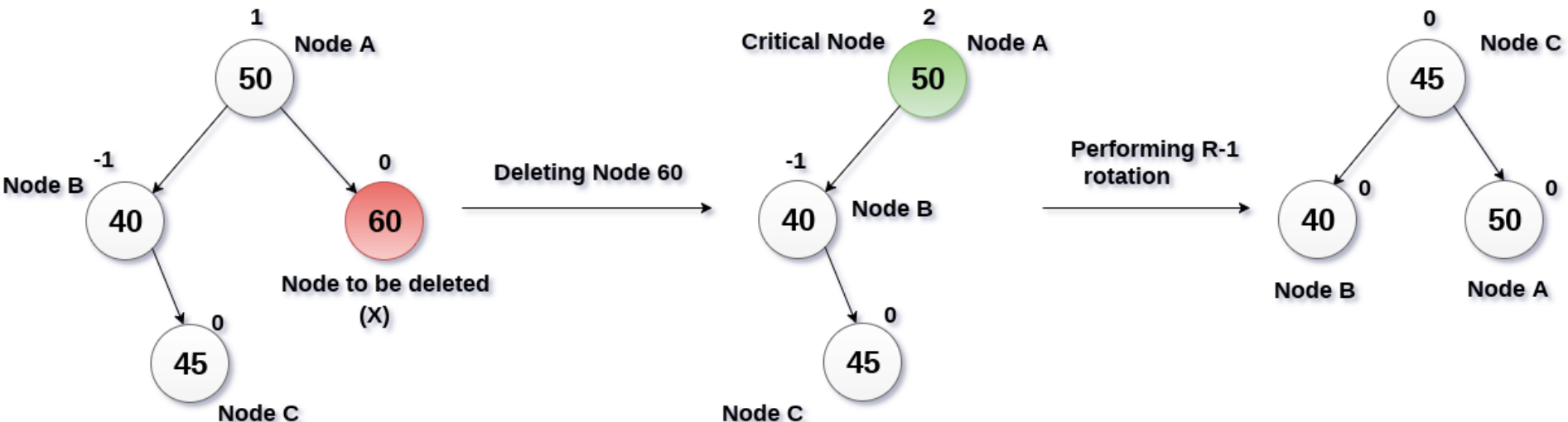
R1 Rotation

Let B be the root of the left or right sub-tree of the critical node. R1 rotation is applied if the balance factor of B is 1. Consider the AVL tree given below and delete 55 from it.



R-1 Rotation

Let B be the root of the left or right sub-tree of the (critical node. R-1 rotation is applied if the balance factor of B is -1. Consider the AVL tree given below and delete 60 from it.



M-WAY SEARCH TREES

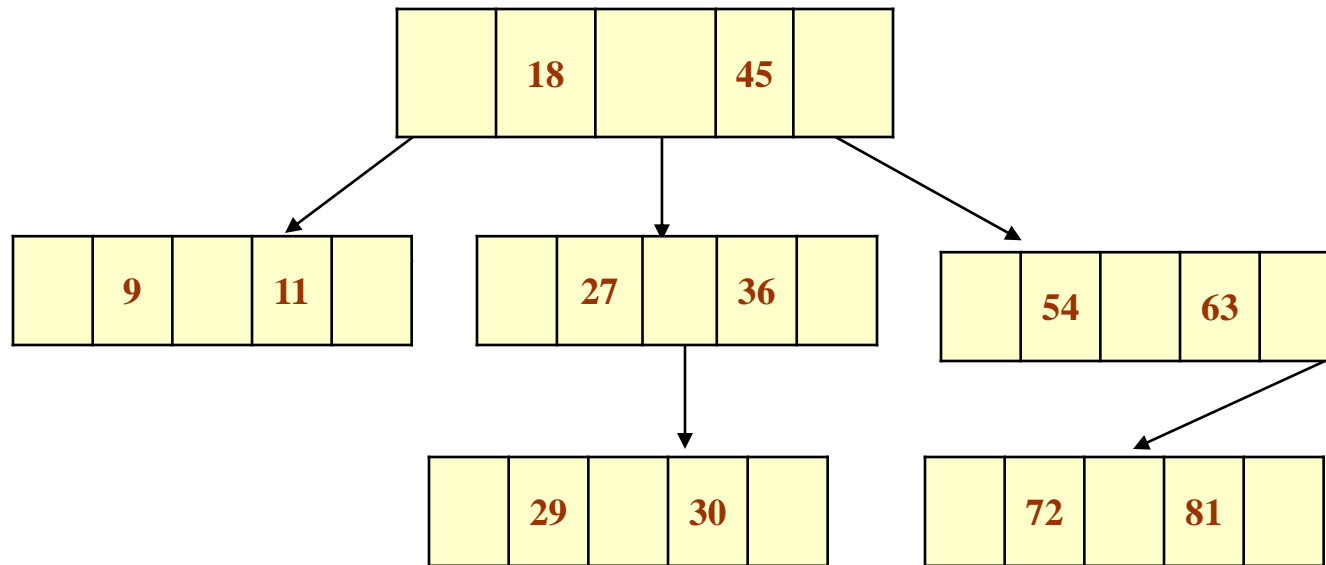
In such a tree M is called the order of the tree. Note in a binary search tree $M = 2$, so it has one value and 2 sub-trees. In other words, every internal node of an M -way search tree consists of pointers to M sub-trees and contains $M - 1$ keys, where $M > 2$.

P_0	K_0	P_1	K_1	P_2	K_2	\cdots \dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-------	-------	-------	---------------------	-----------	-----------	-------

In the structure, $P_0, P_1, P_2, \dots, P_n$ are pointers to the node's sub-trees and $K_0, K_1, K_2, \dots, K_{n-1}$ are the key values of the node. All the key values are stored in ascending order. That is, $K_i < K_{i+1}$ for $0 \leq i \leq n-2$.

In an M -way search tree, it is not compulsory that every node has exactly $(M-1)$ values and have exactly M sub-trees. Rather, the node can have anywhere from 1 to $(M-1)$ values, and the number of sub-trees may vary from 0 (for a leaf node) to $1 + i$, where i is the number of key values in the node. M is thus a *fixed upper limit* that defines how much key values can be stored in the node.





Using the above 3-way search tree, let us devise some basic properties of an M-way search tree.

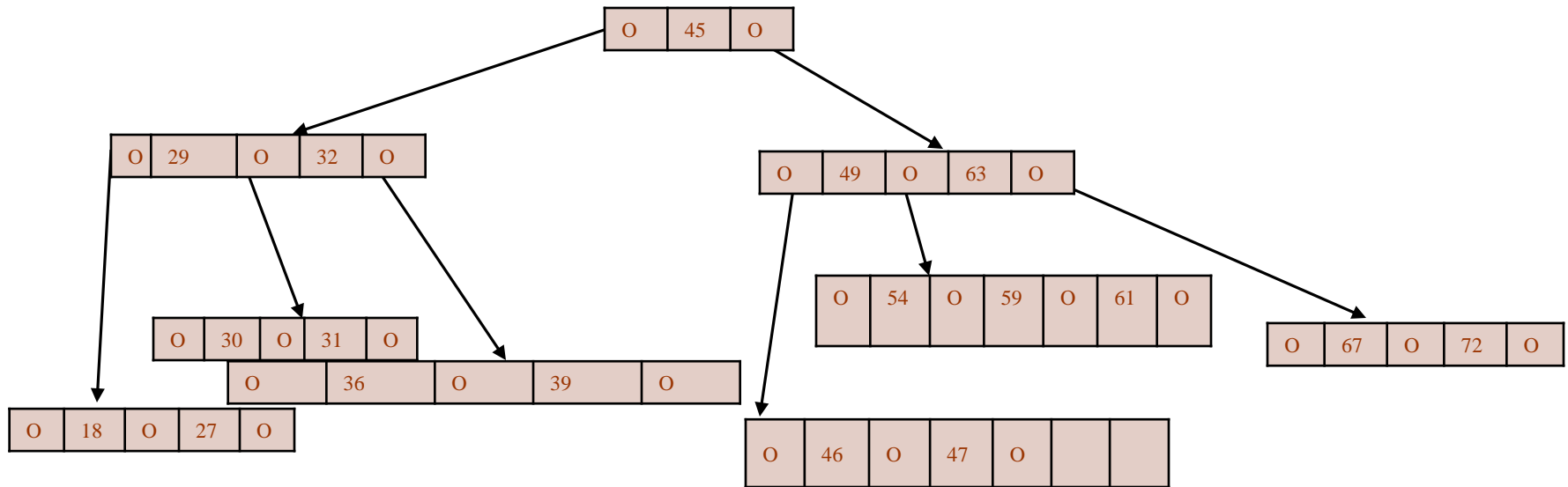
- All key values in the sub-tree pointed by P_i are less than K_i , where $0 \leq i \leq n-1$.
- All key values in the sub-tree pointed by P_i are greater than K_{i-1} , where $0 \leq i \leq n-1$.
- In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.



B TREES

- ◉ A specialized m -way tree. A B tree of order m can have maximum $m-1$ keys and m pointers to its sub-trees.
 - ◉ A B-tree is designed to store sorted data and allows search, insert, and delete operations to be performed in logarithmic time. A B-tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an m -way search tree and in addition has the following properties:
 - ◉ Every node in the B-tree has at most (maximum) m children.
 - ◉ Every node in the B-tree except the root node and leaf nodes have at least (minimum) $m/2$ children.
 - ◉ The root node has at least two children if it is not a terminal (leaf) node.
 - ◉ All leaf nodes are at the same level.
 - ◉ An internal node in the B tree can have n number of children, where $0 \leq n \leq m$. it is not necessary that every node has the same number of children, but the only restriction is that the node should have at least $m/2$ children.
-



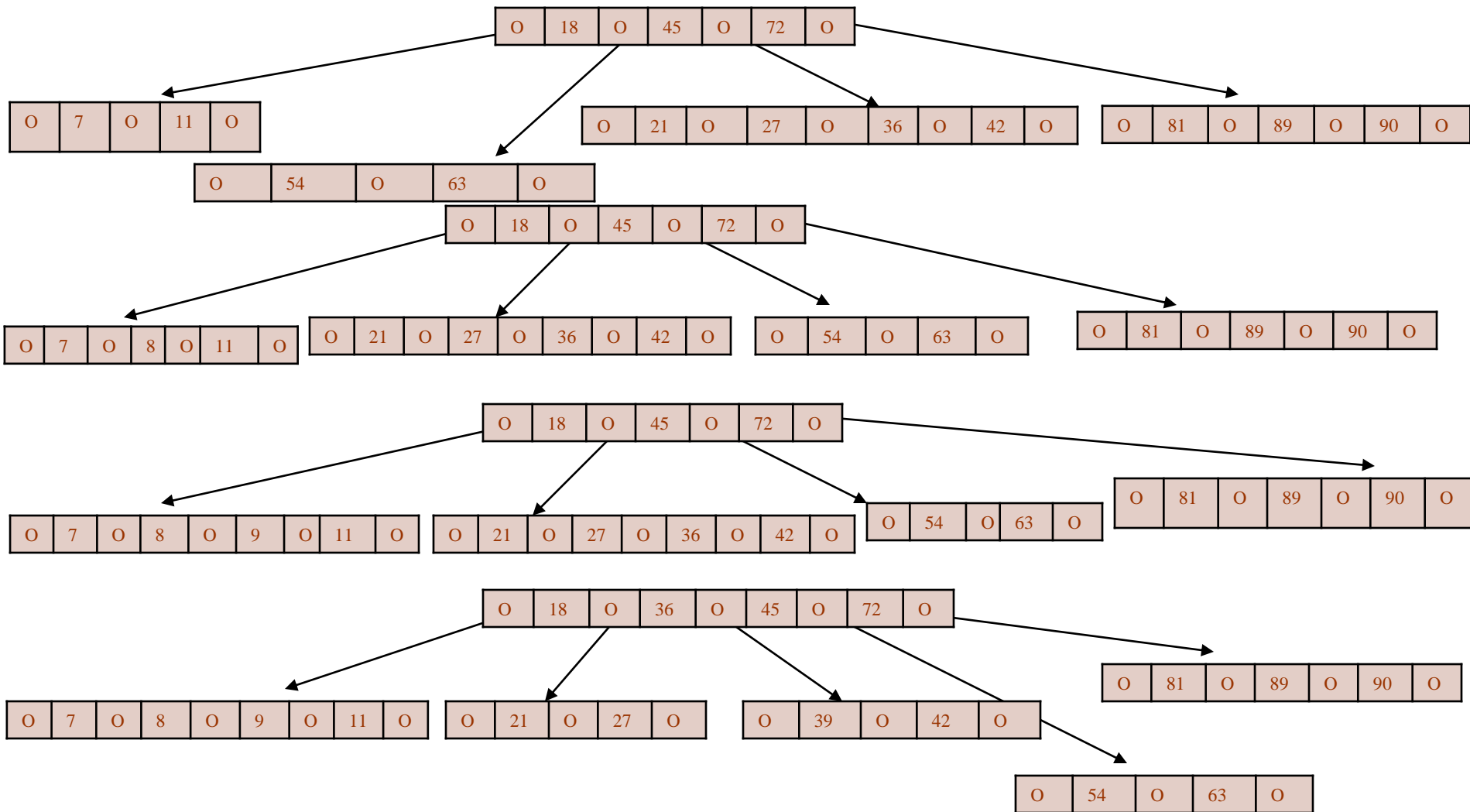


Insert Operation

In a B tree all insertions are done at the leaf node level.

1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is it contains less than $m-1$ key values, then insert the new element in the node, keeping the node's elements ordered.
3. if the leaf node is full, that is the leaf node already contain $m-1$ key values, then insert the new value in order into the existing set of keys.
4. Split the node at its median into two nodes. Note that the split nodes are half full.
5. Push the median element up to its parent's node.
6. If the parent's node is already full, then split the parent node by following the same steps.

Example: Look at the B tree of order 5 given below and insert 8, 9, 39 into it.



Deletion

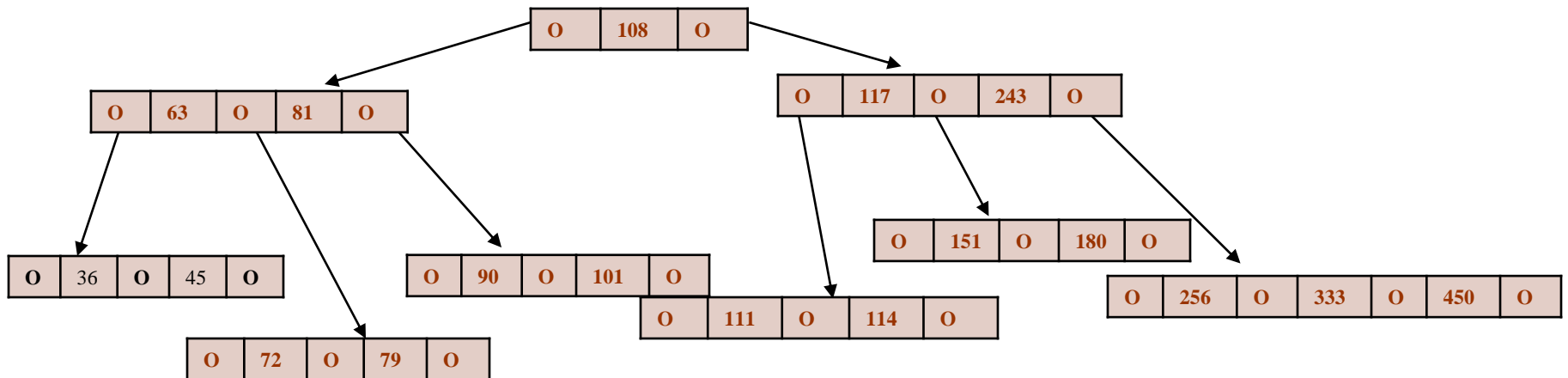
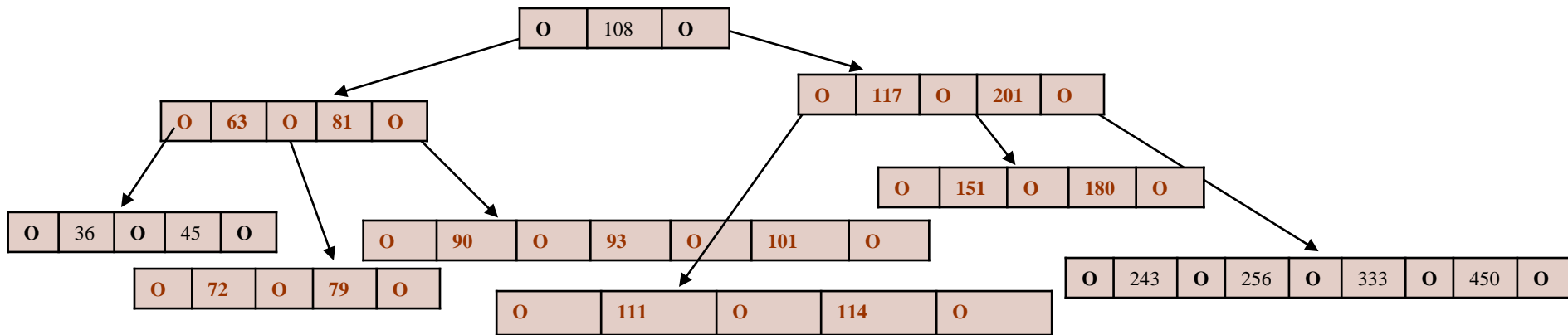
- Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. First, a leaf node has to be deleted. Second, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted
2. If the leaf node contains more than minimum number of key values (that is, $m/2$ elements), then delete the value.
3. Else, if the leaf node does not contain even $m/2$ elements, then fill the node by taking an element either from the left or from the right sibling
4. Else, if both left and right siblings contain only minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements do not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent node leaves it with less than minimum number of keys in the node, then propagate the process upwards thereby reducing the height of the B tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So further processing will be done as if a value from the leaf node has been deleted.

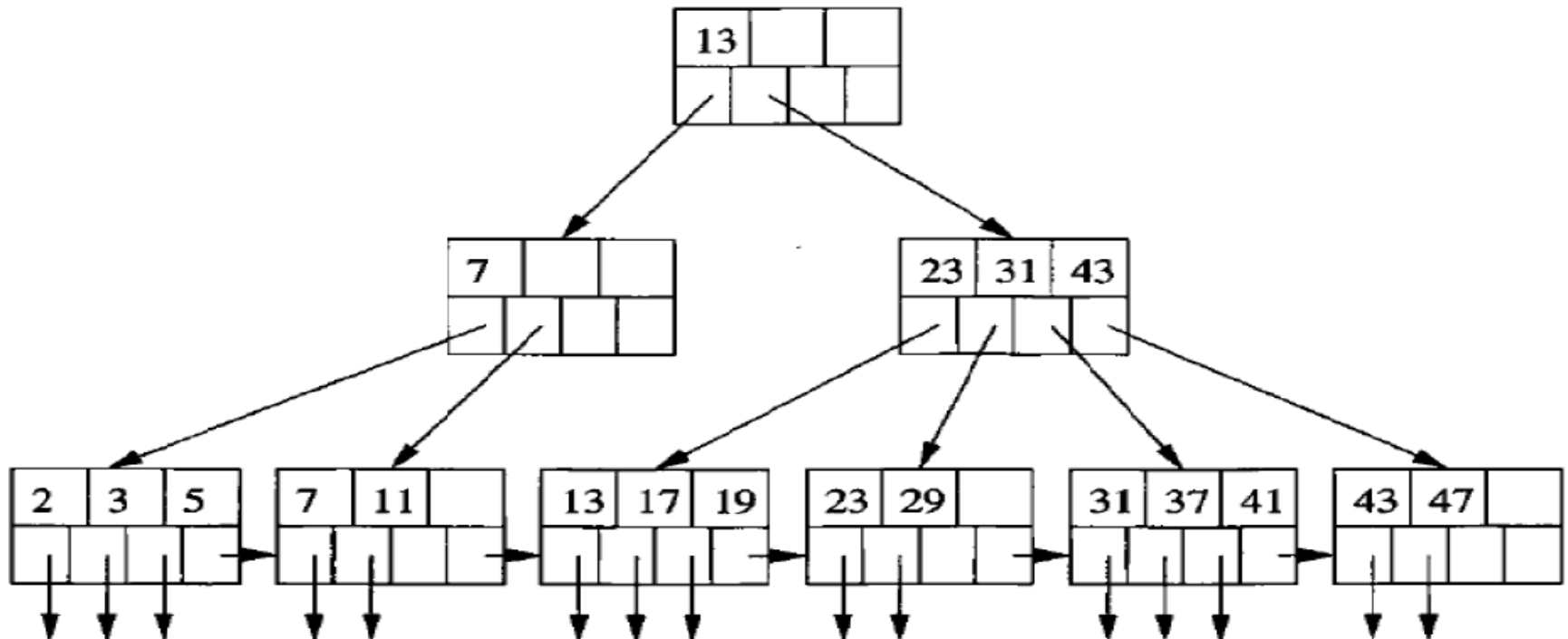


Consider the B tree of order 5 given below and delete the values - 93, 201 from it.



B+ TREES

- A variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all records at the leaf level of the tree; only keys are stored in interior nodes.
- The leaf nodes of the B+ tree are often linked to one another in a linked list.
- B+-tree stores data only in the leaf nodes. All other nodes (internal nodes) are called index nodes or i-nodes and store index values which allow us to traverse the tree from the root down to the leaf node that stores the desired data item.

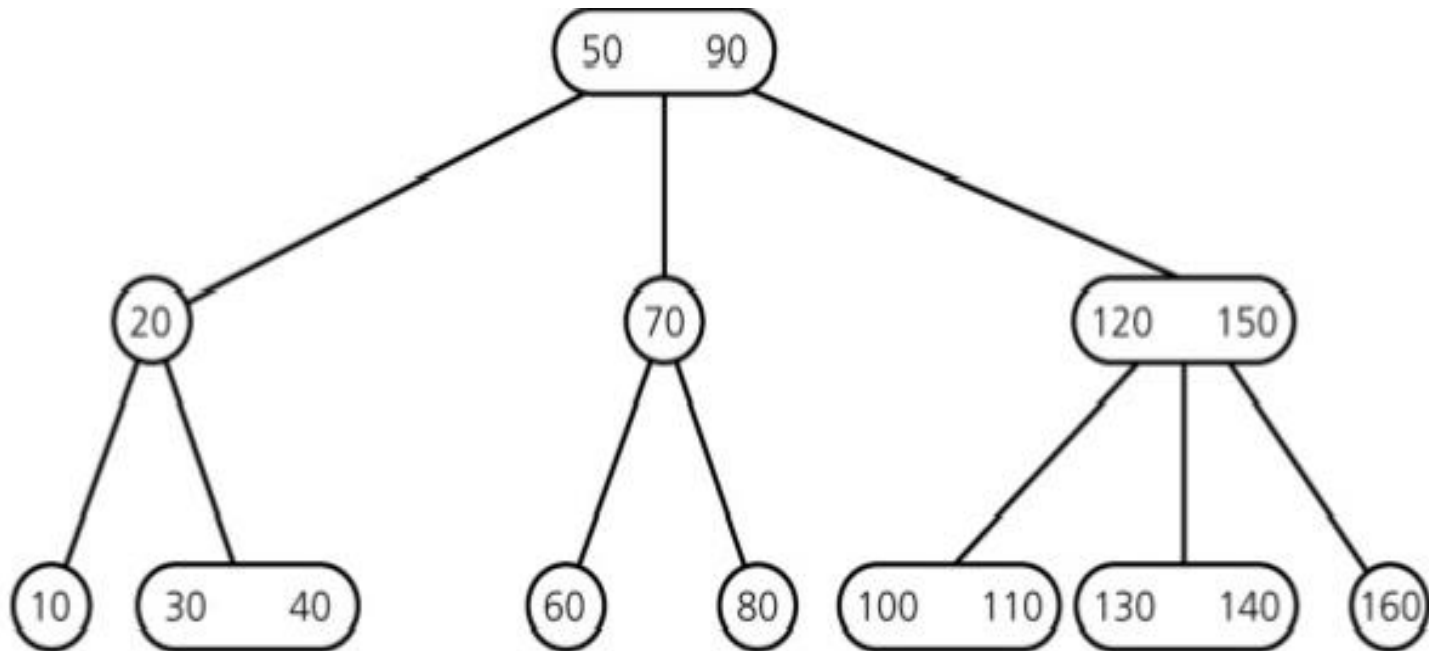


2-3 Trees

- ▶ Every internal node (non-leaf node) has either **one data element and two children** or **two data elements and three children**.
- ▶ Height of 2-3 trees is always $O(\log n)$.
- ▶ In a 2-3 tree of height h
 - ▶ If all non-leaf nodes have 3 children, then total number of elements in the tree is $3^{h+1}-1$.
 - ▶ If all non-leaf nodes have 2 children, then total number of elements in the tree is $2^{h+1}-1$.
- ▶ In a 2-3 tree with n elements having height h ,
 - ▶ $2^{h+1}-1 \leq n \leq 3^{h+1}-1$ and
 - ▶ $\log_2 (n+1) \leq h \leq \log_3 (n+1)$.



Example of 2-3 Tree



THANK YOU

