Module 2:

Introduction to Trees

CO2: **Classify**, **apply** and **analyze** the concepts of trees in real life problem solving.

Subject In-charge

Ms.Pratibha Rane

Assistant Professor

Class : III SEM SEIT- B

Module 2: Introduction to Trees

CO2: **Classify**, **apply** and **analyze** the concepts trees in real life problem solving.

Introduction to Trees: Terminology, Types of Binary trees. Non recursive Preorder, in-order and post-order traversal. Creation of binary trees from the traversal of binary trees. Binary search tree: Traversal, searching, insertion and deletion in binary search tree.

Threaded Binary Tree: Finding in-order successor and predecessor of a node in threaded tree. Insertion and deletion in threaded binary tree.

AVL Tree: Searching and traversing in AVL trees. Tree Rotations: Right Rotation, Left Rotation. Insertion and Deletion in an AVL Tree.

B-tree: Searching, Insertion, Deletion from leaf node and nonleaf node. B+ Tree, Digital Search Tree, Game Tree & Decision Tree

Self-learning Topics: Implementation of AVL and B+ Tree

Chapter 9 & 10: Reema Thareja; Data Structures using C; Oxford

A tree data structure is recursively defined as a **set of one or more nodes** where one node is designated as the **root of the tree** and all the remaining nodes can be partitioned into non-empty sets each of which is a **sub-tree of the root**.

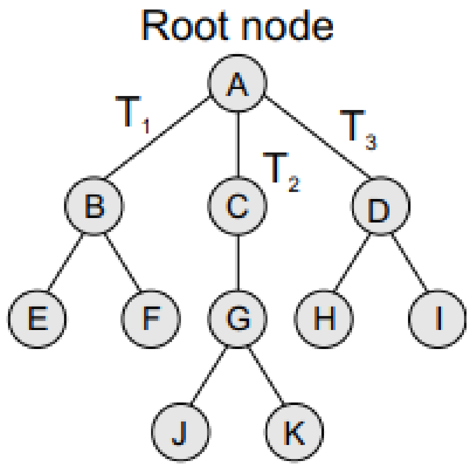


Figure 9.1 Tree

Advantage :

- To represent structural relationship, hierarchical relationships
- Flexible to add and delete data

Basic Terminology

Node: every element int the tree

Root node: topmost element (only one root)

Child Node: a node which has link from parent node, node on left link is called left child and node on right link is called right child

Leaf node: lower nodes who doesn't have child nodes

Path: A sequence of consecutive edges is called a path.

Ancestor node : predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 9.1, nodes A, C, and G are the ancestors of node K.

Descendant node

A descendant node: is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 9.1, nodes C, G, J, and K are the descendants of node A.

Subtree: every child from a node forms a subtree

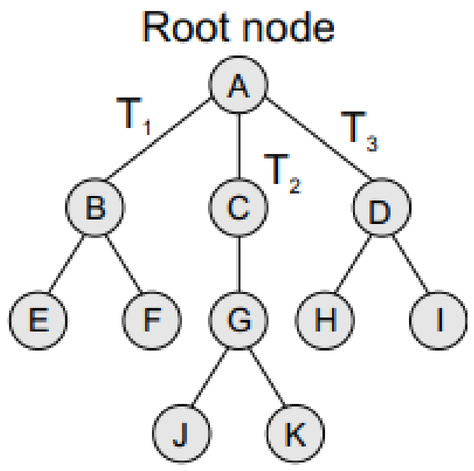


Figure 9.1 Tree

Basic Terminology

Level number Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

Degree of a node : maximum number of children that a node has. The degree of a leaf node is zero.

In-degree In-degree of a node is the number of edges arriving at that node.

Out-degree Out-degree of a node is the number of edges leaving that node.

Depth or height of a tree: maximum level of any leaf node .

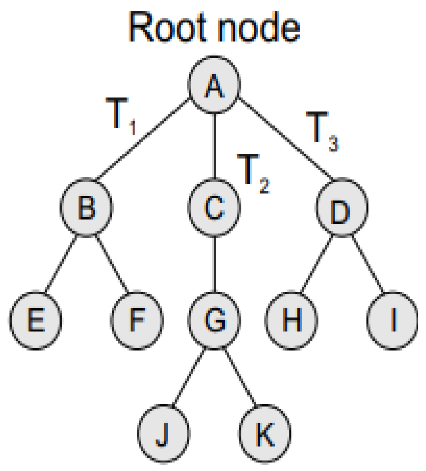


Figure 9.1 Tree

Types of tree

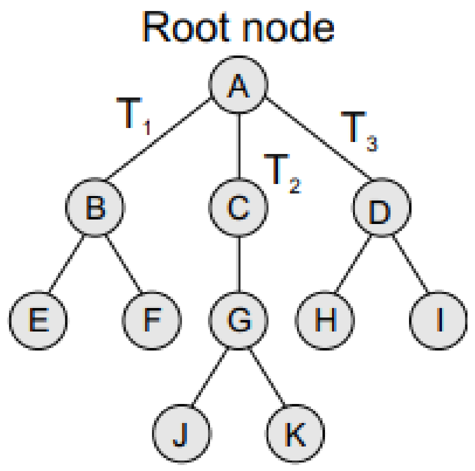


Figure 9.1 Tree

1. **General tree**: tree with multiple nodes
2. **Forests**: A forest is a disjoint union of trees
3. **Binary tree**: specialized tree data structure in which every node is allowed to have maximum of two child nodes .
4. **Binary search tree**: A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order

Binary tree

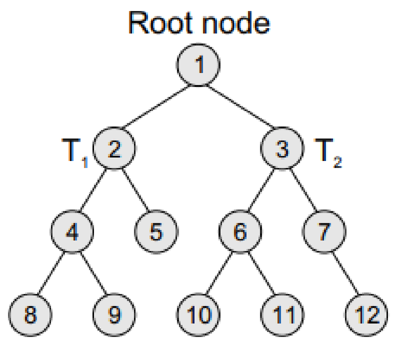


Figure 9.3 Binary tree

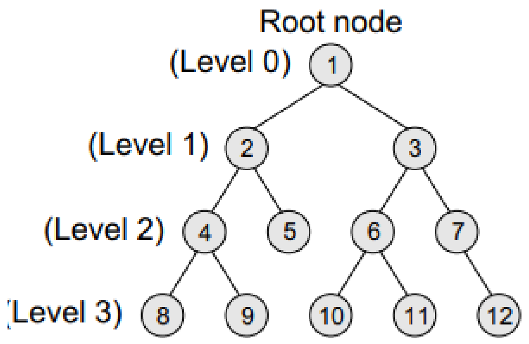


Figure 9.4 Levels in binary tree

1. A tree in which every node has atmost **two children**
2. A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes
3. So, if every level has two nodes then a tree with height h will have at the most $2^h - 1$ nodes as at level 0, there is only one element called the root
4. The height of a binary tree with n nodes is at least $\log_2 (n+1)$ and at most n.
5. Types of binary tree:

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree
- Skewed Binary tree
- Extended binary tree

Binary tree

- **Extended binary tree:**
- **A full binary tree b**
- **By Adding Dummy nodes wherever required**

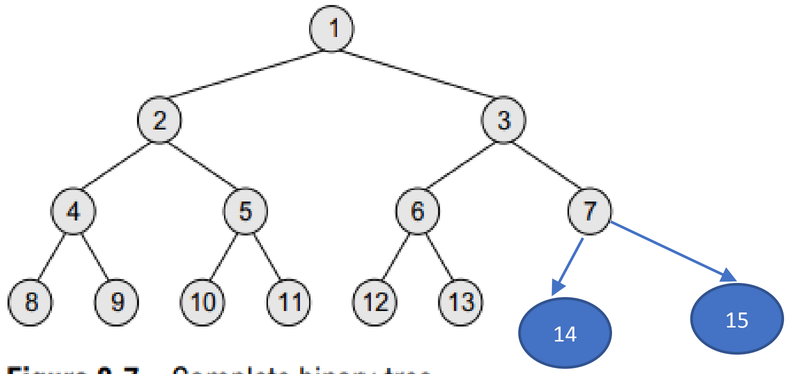


Figure 9.7 Complete binary tree

Binary tree

- **Strict binary tree:**
- **Every node is must to have exactly two child or zero nodes**
- **Also called full binary or 2-Tree**

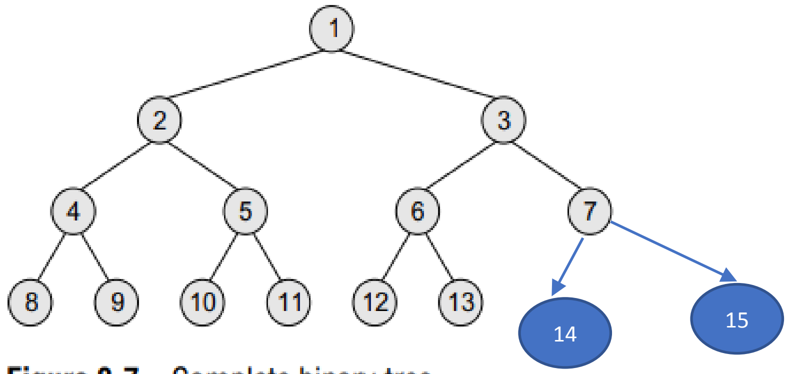
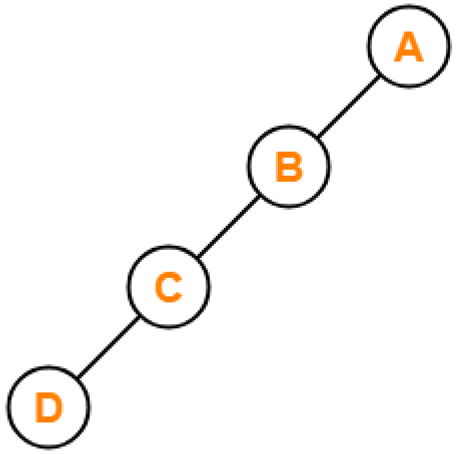


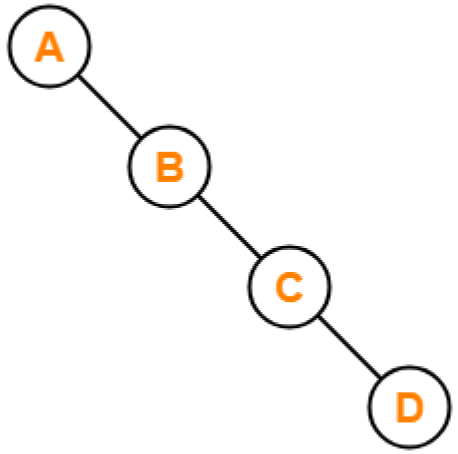
Figure 9.7 Complete binary tree

Binary tree

- **Skewed binary tree**:
- Which is either only left branches (left skewed) are present or only right branches(right skewed) are present



Left Skewed Binary Tree



Right Skewed Binary Tree

Representation of Binary tree in the memory

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

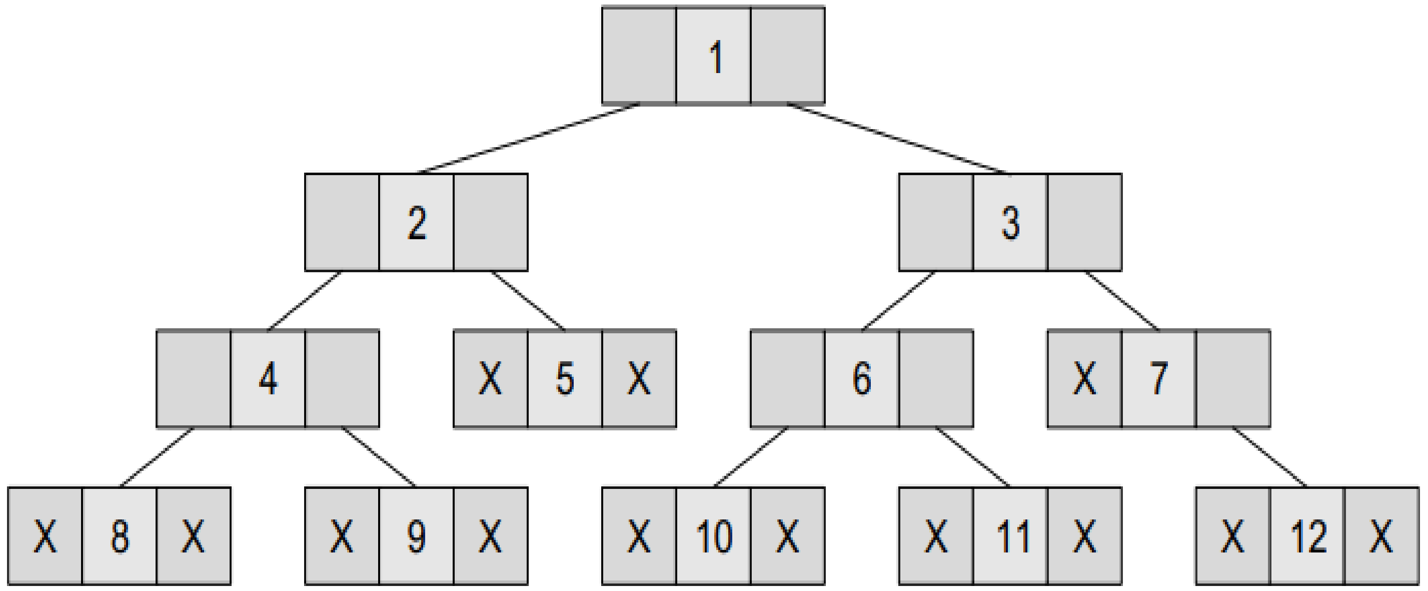


Figure 9.9 Linked representation of a binary tree

Look at the tree given in Fig. 9.10. Note how this tree is represented in the main memory using a linked list (Fig. 9.11).

Traversing a Binary tree

visiting each node in the tree exactly once

Types: (name based on order of Rootnode visit)

Pre-order traversal

In-order traversal

Post-order traversal

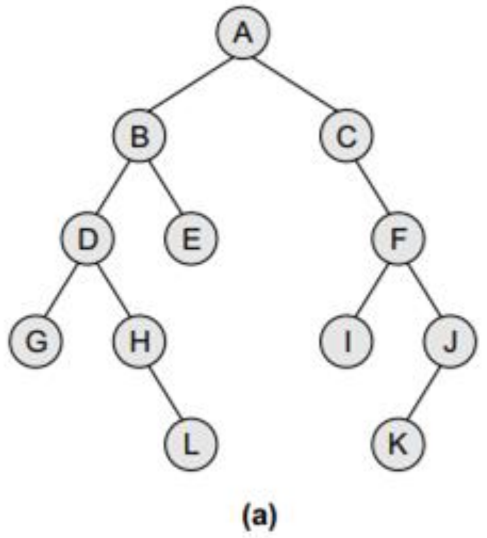
In-order Traversing a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal



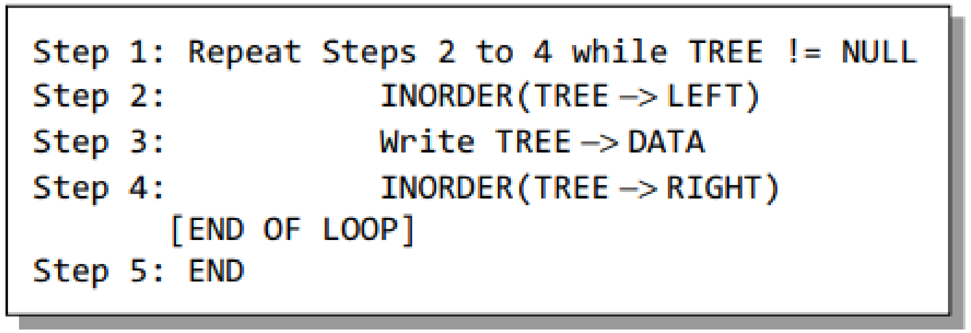
find the sequence of nodes that will be visited using in-order traversal algorithm.

TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J

In-order Traversing a Binary tree

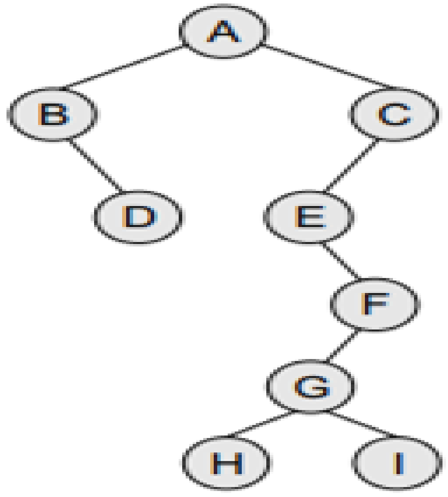
The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.



```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal



(b)

find the sequence of nodes that will be visited using in-order traversal algorithm.

TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

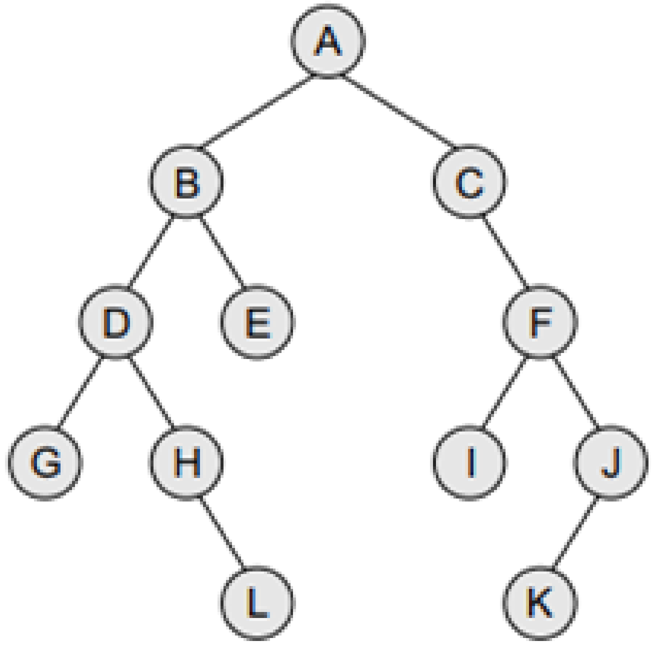
Post-order Traversal in a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree,
3. and finally Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             POSTORDER(TREE -> LEFT)
Step 3:             POSTORDER(TREE -> RIGHT)
Step 4:             Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal



For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

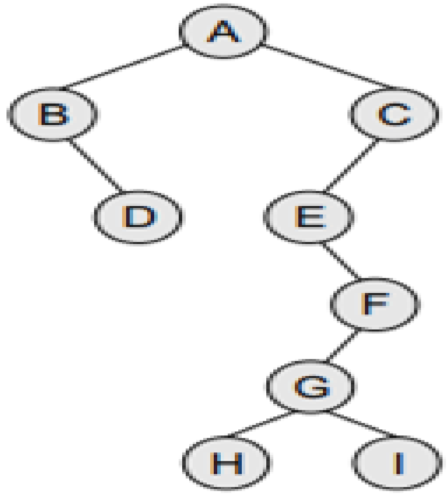
Post-order Traversal in a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree,
3. and finally Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             POSTORDER(TREE -> LEFT)
Step 3:             POSTORDER(TREE -> RIGHT)
Step 4:             Write TREE -> DATA
          [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal



(b)

For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A

Pre-order Traversal in a Binary tree

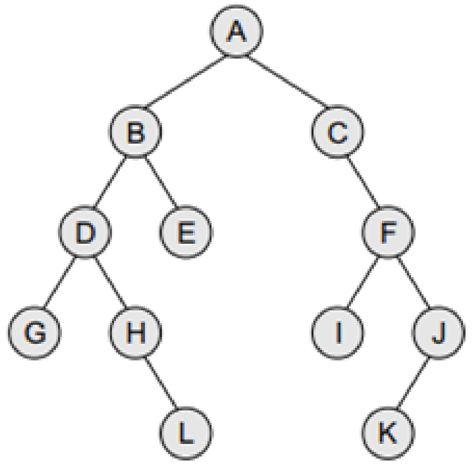
The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree,
3. and finally Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a **prefix expression.**



For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J, and K

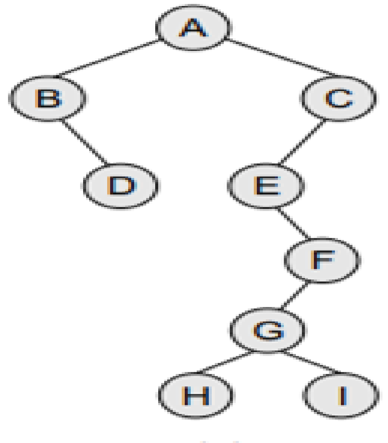
Pre-order Traversal in a Binary tree

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
            [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal

The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree,
3. and finally Traversing the right sub-tree.
4. Also called as depth-first traversal



(b)

For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: A, B, D, C, D, E, F, G, H, and I

Traversal in a Binary tree

For the trees given in Example 9.13, give the sequence of nodes that will be visited using post-order, pre-order and in-order traversal algorithm.

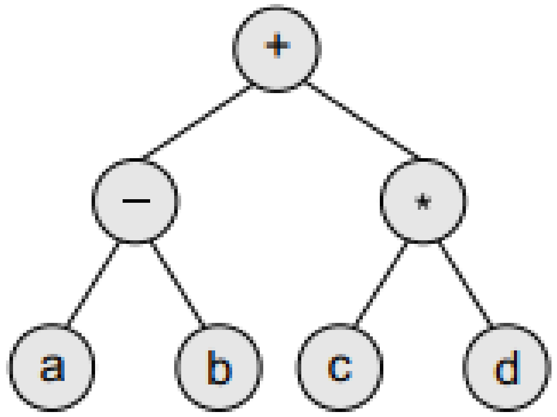


Figure 9.13 Expression tree

Pre-Order

+ – a b * c d

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a **prefix expression**.

In-order

a – b + c * d

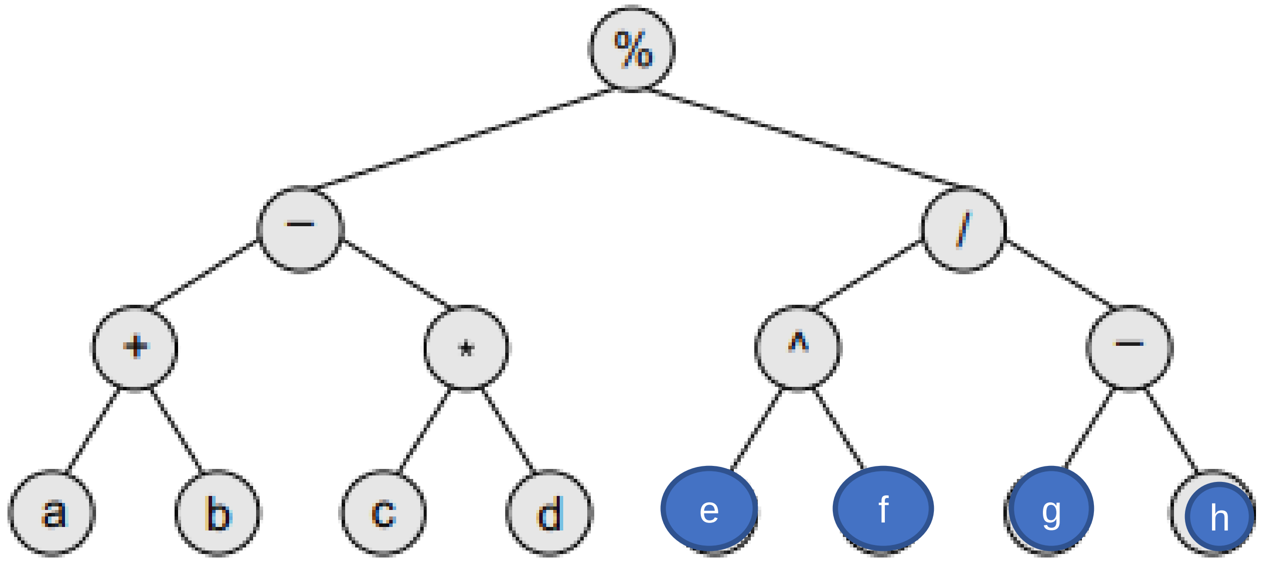
When we traverse the elements of a tree using the in-order traversal algorithm, the expression that we get is a **infix expression**.

Post-order

a b – c d * +

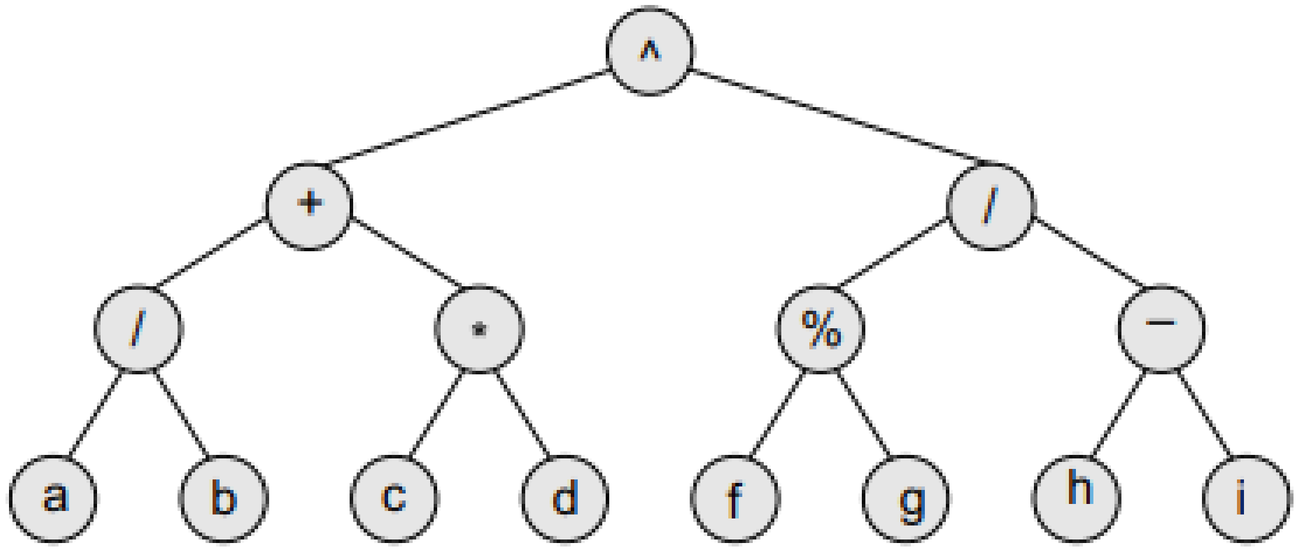
Post-order traversals are used to extract **postfix notation from an expression tree.**

Example : Given an expression,
Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)),
construct the corresponding binary tree.



Expression tree

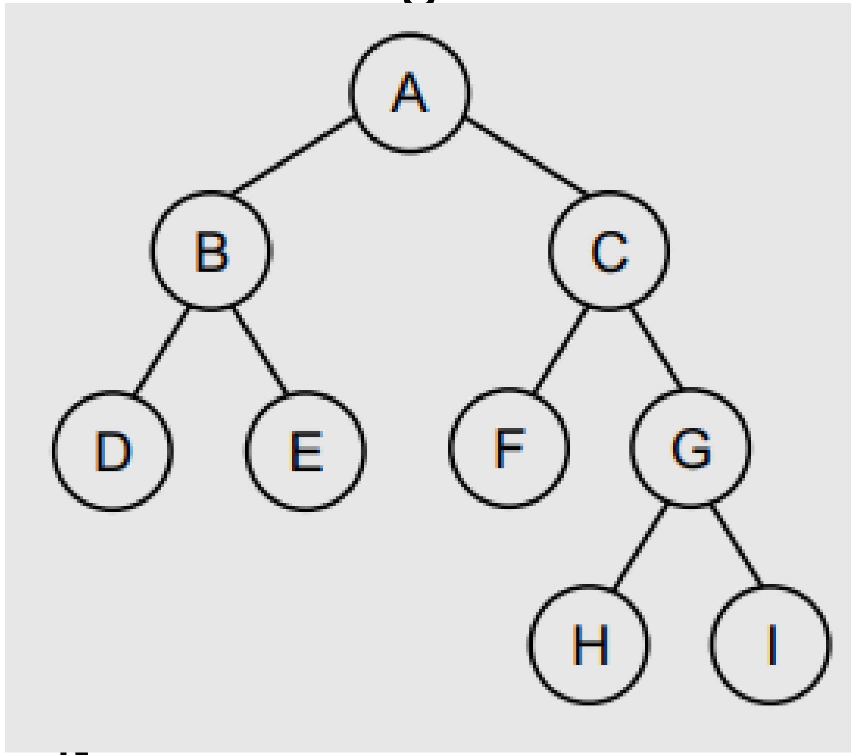
Example : Given the binary tree, write down the expression that it represents.



[{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

Example : Consider the tree given below. Now, do the following:

(a) Name the leaf nodes D, E, F, H, I
(b) Name the non-leaf nodes A,B,C,G
(c) Name the ancestors of E B, A
(d) Name the descendants of A B,C,D,E,F,G,H,I
(e) Name the siblings of C B
(f) Find the height of the tree 4
(g) Find the height of sub-tree rooted at E 1
(h) Find the level of node E 2
(i) Find the in-order, pre-order, post-order traversal]



Inorder - D B E A F C H G I

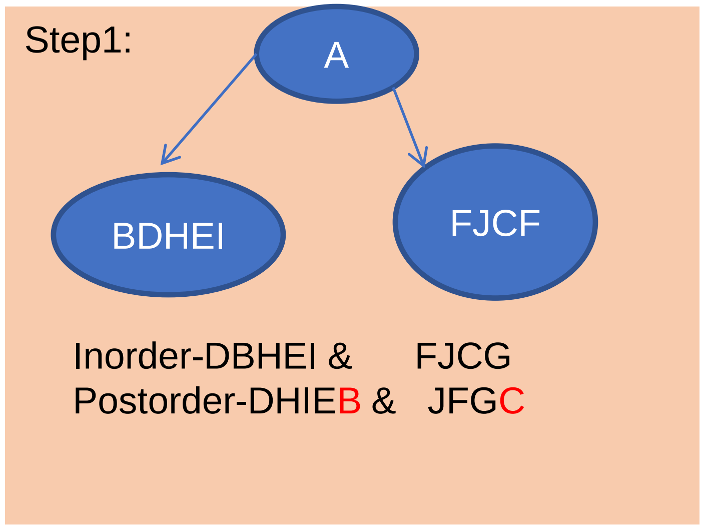
Preorder - A B D E C F G H I

Postorder - D E B F H I G C A

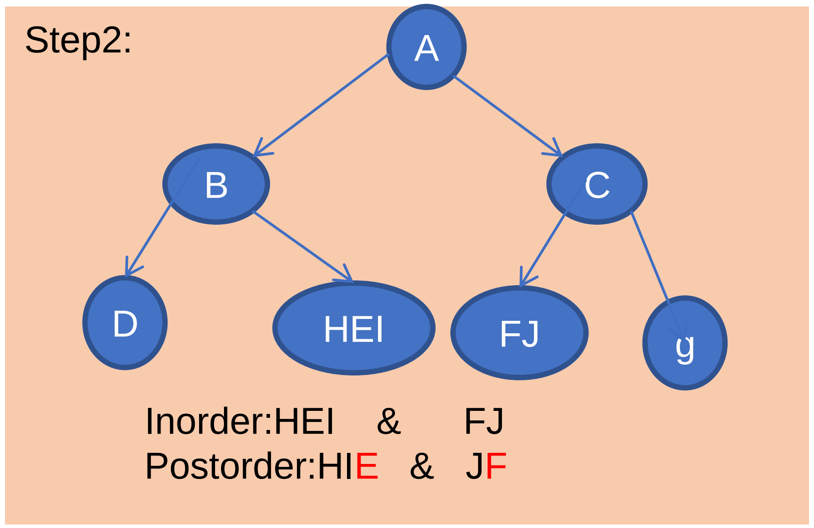
Construct a binary tree for:

In-order : DBHEIAFJCG

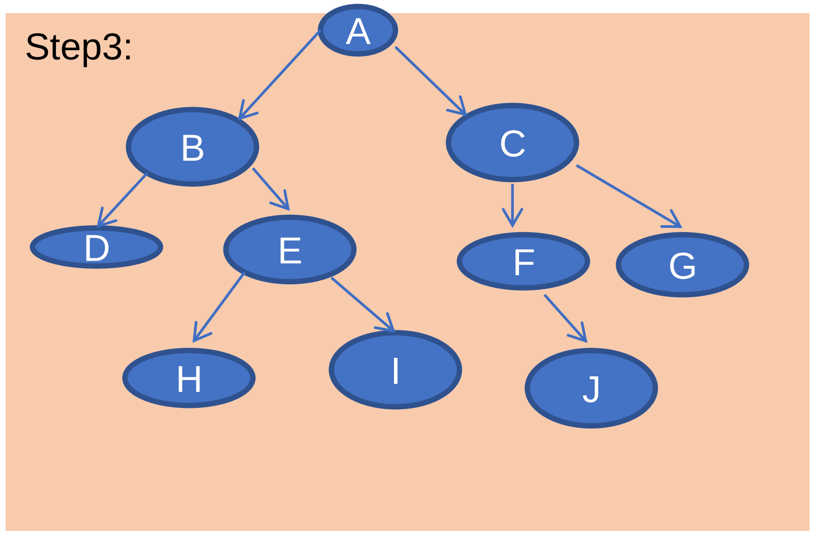
Post-order: DHIEBJFGCA



Inorder-DBHEI & FJCG
Postorder-DHIEB & JFGC



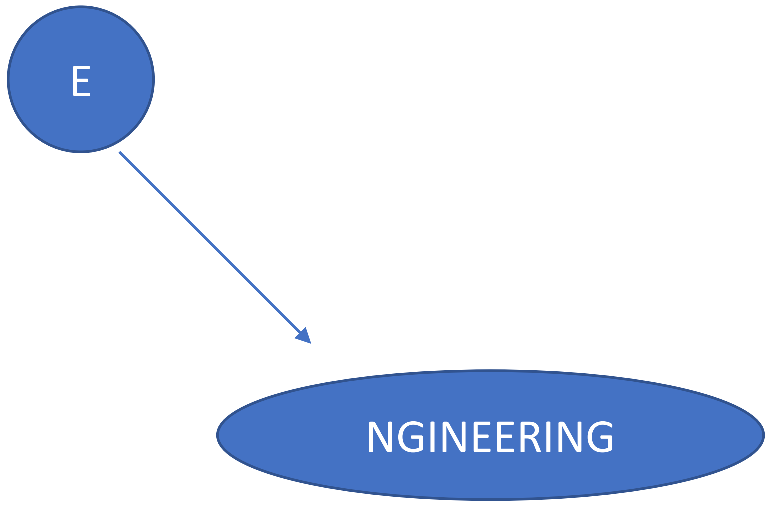
Inorder:HEI & FJ
Postorder:HIE & JF



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

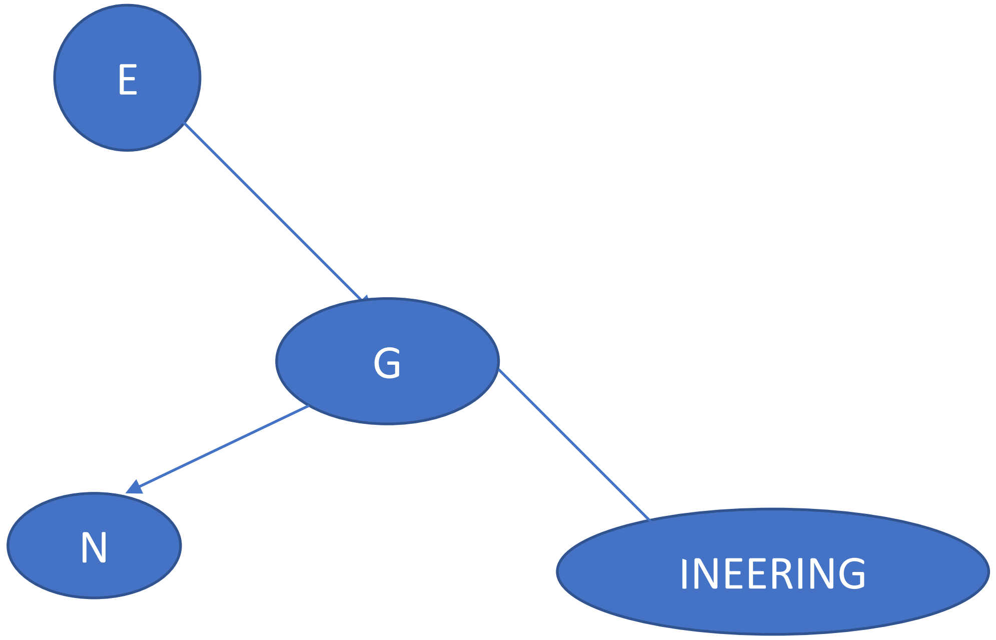
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

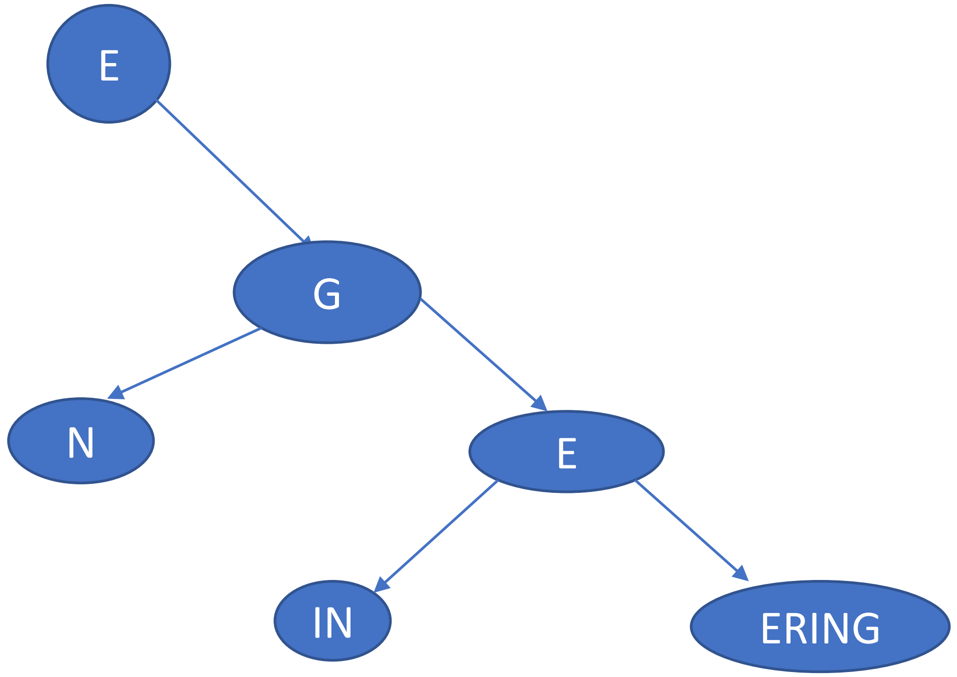
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

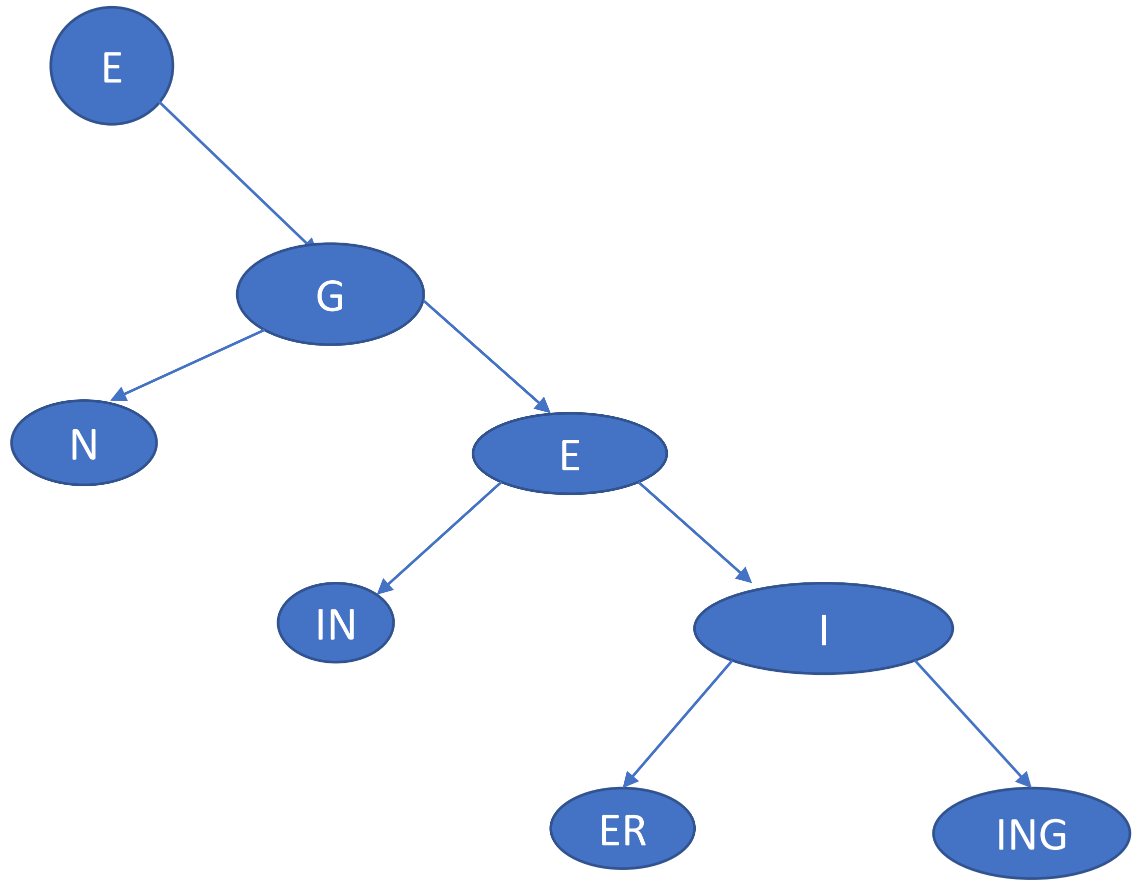
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

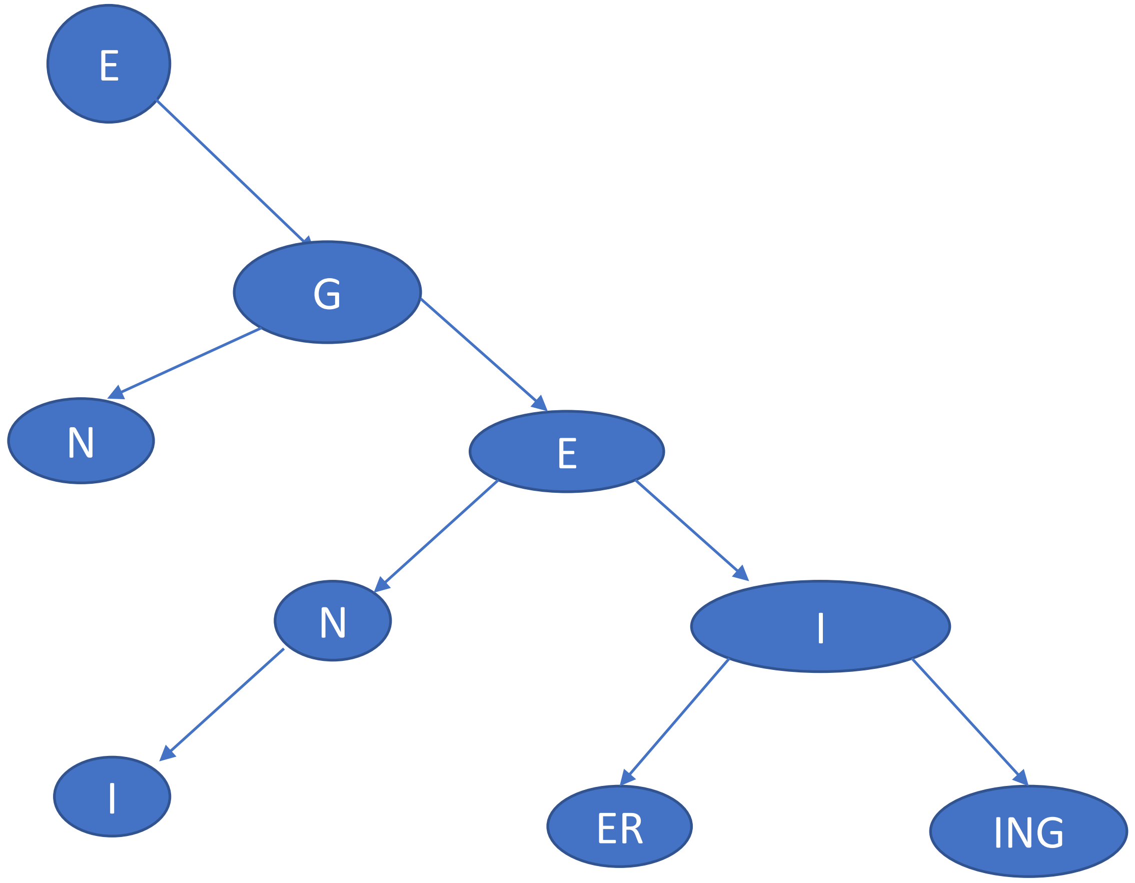
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

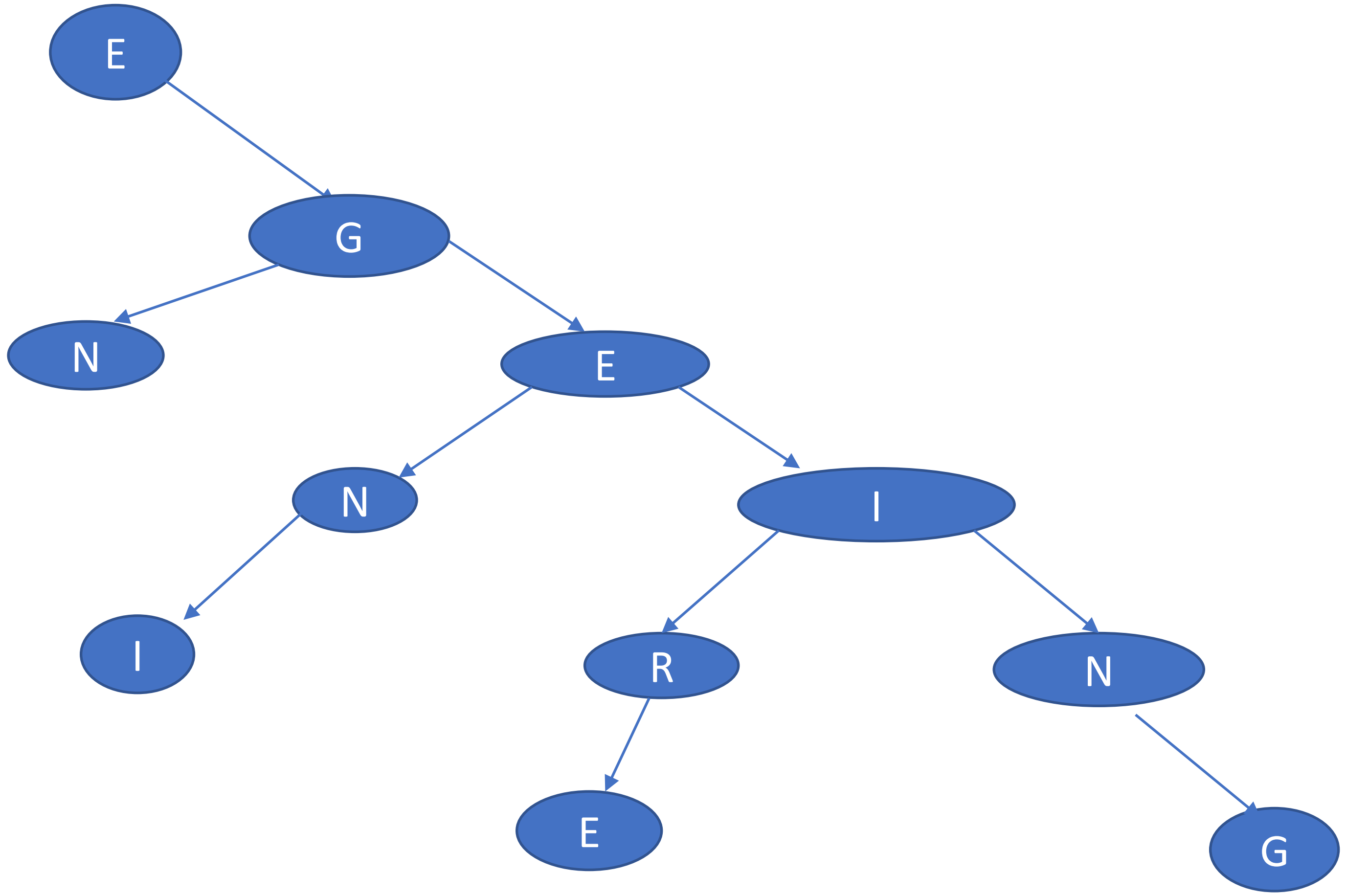
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

Pre-Order : EGNENIIRENG



Binary search tree

- A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order.

a binary search tree is a binary tree with the following properties:

- The left sub-tree of a node N contains values that are less than N's value.
- The right sub-tree of a node N contains values that are greater than N's value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

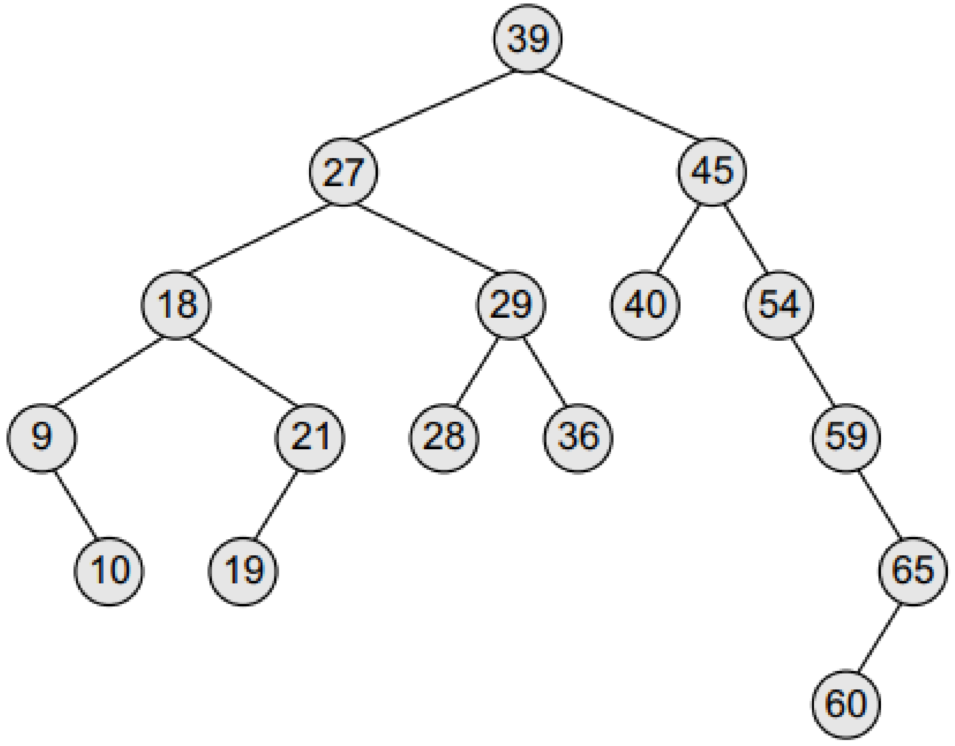


Figure 10.1 Binary search tree

Binary search tree

- A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order.

Advantage:

- time needed to search an element in the tree is greatly reduced.
- speeds up the insertion and deletion operations
- Efficient data structures than arrays and linked list

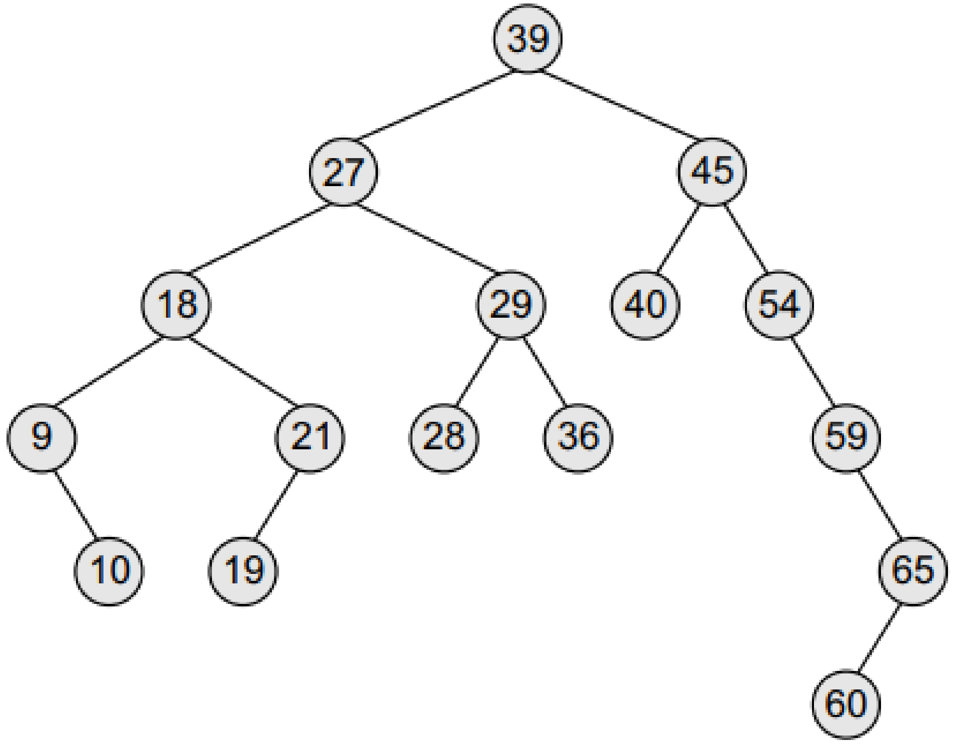


Figure 10.1 Binary search tree

Binary search tree

State whether the binary trees in Fig. 10.3 are binary search trees or not.

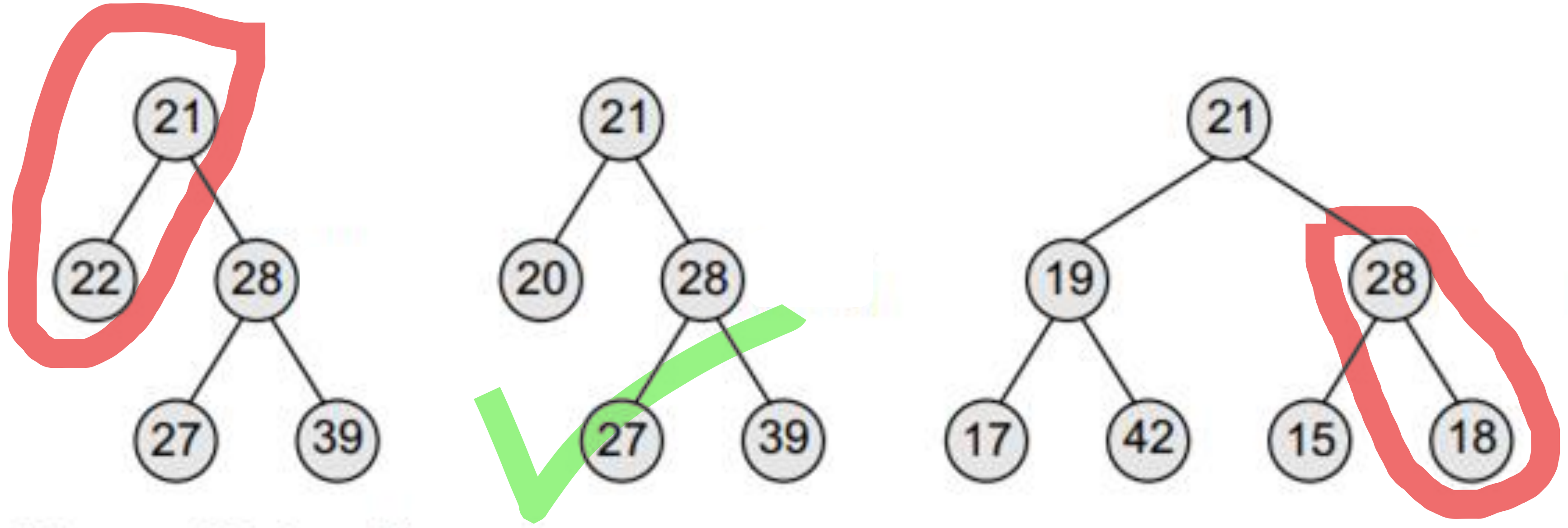


Figure 10.3 Binary trees

Binary search tree

State whether the binary trees in Fig. 10.3 are binary search trees or not.

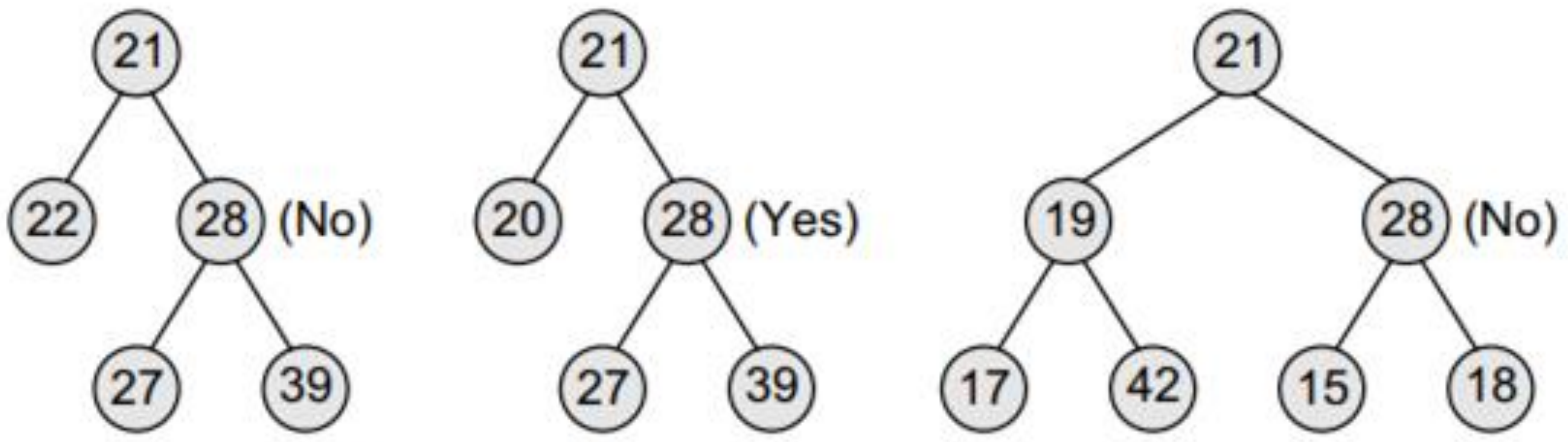
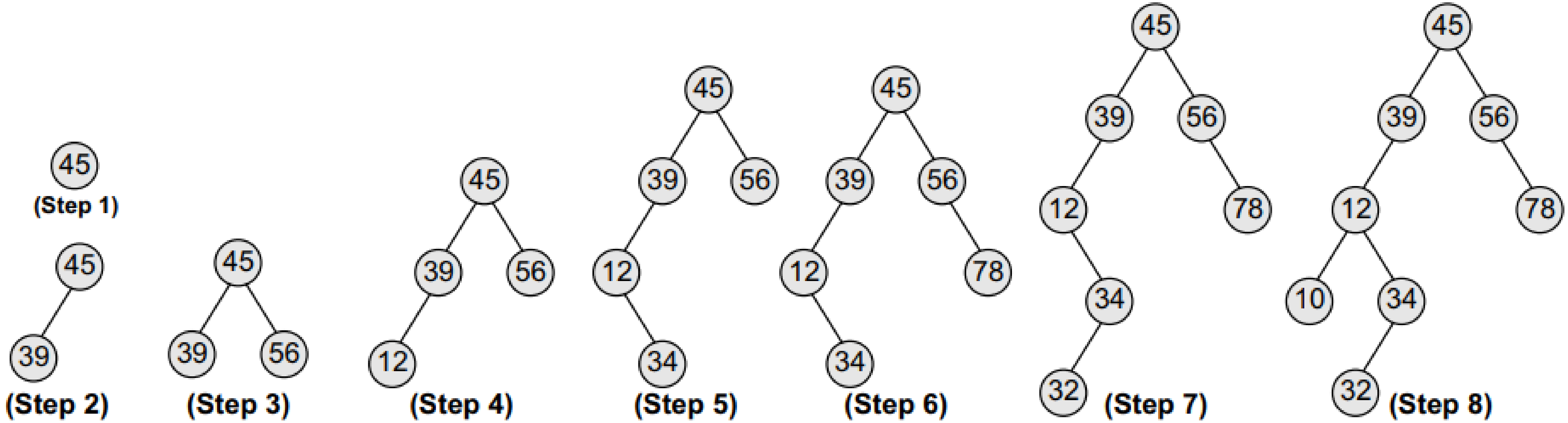


Figure 10.3 Binary trees

Binary search tree

Create a binary search tree using the following data elements:
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



Binary search tree

Create a binary search tree using the following data elements:
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

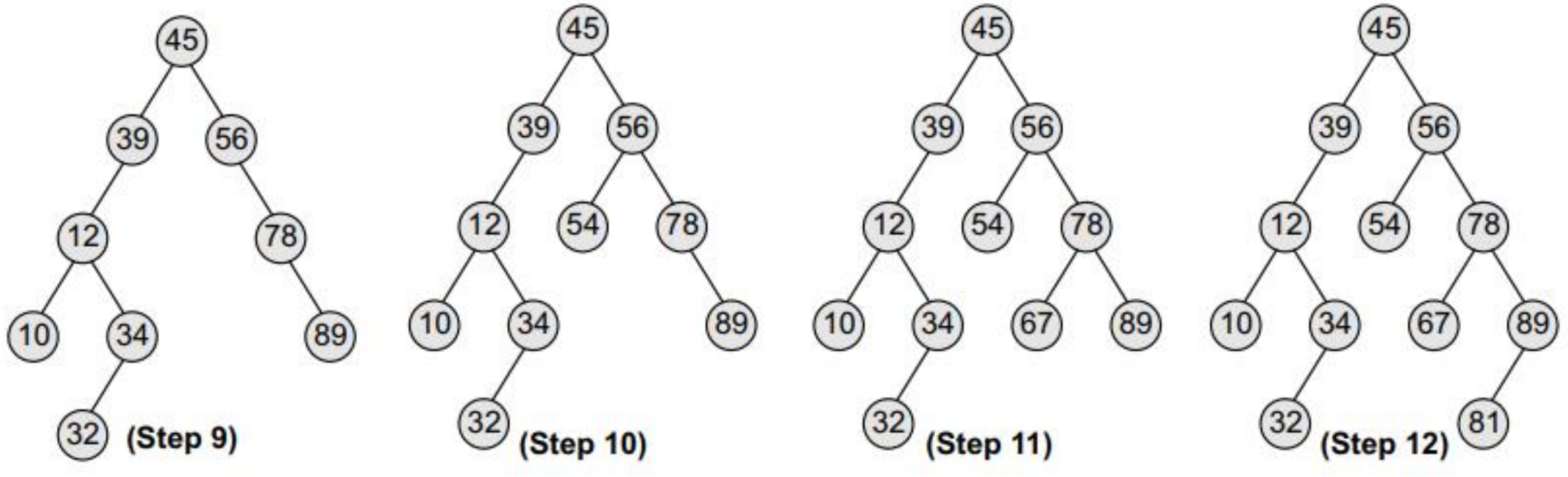


Figure 10.4 Binary search tree

Binary search tree

Create a binary search tree using the following data elements:
J,R,D,G,T,E,M,H,P,A,F,Q

Binary search tree

Create a binary search tree using the following data elements:
50, 33, 44, 22, 77, 35, 60, 40

OPERATIONS ON BINARY SEARCH TREES

- Traversal
- Searching
- Insertions
- deletions

Searching operation on binary search trees

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          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
```

Figure 10.8 Algorithm to search for a given value in a binary search tree

insertion operation on binary search trees

add a new node with a given value at the correct position in the binary search tree

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            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
```

Figure 10.9 Algorithm to insert a given value in a binary search tree

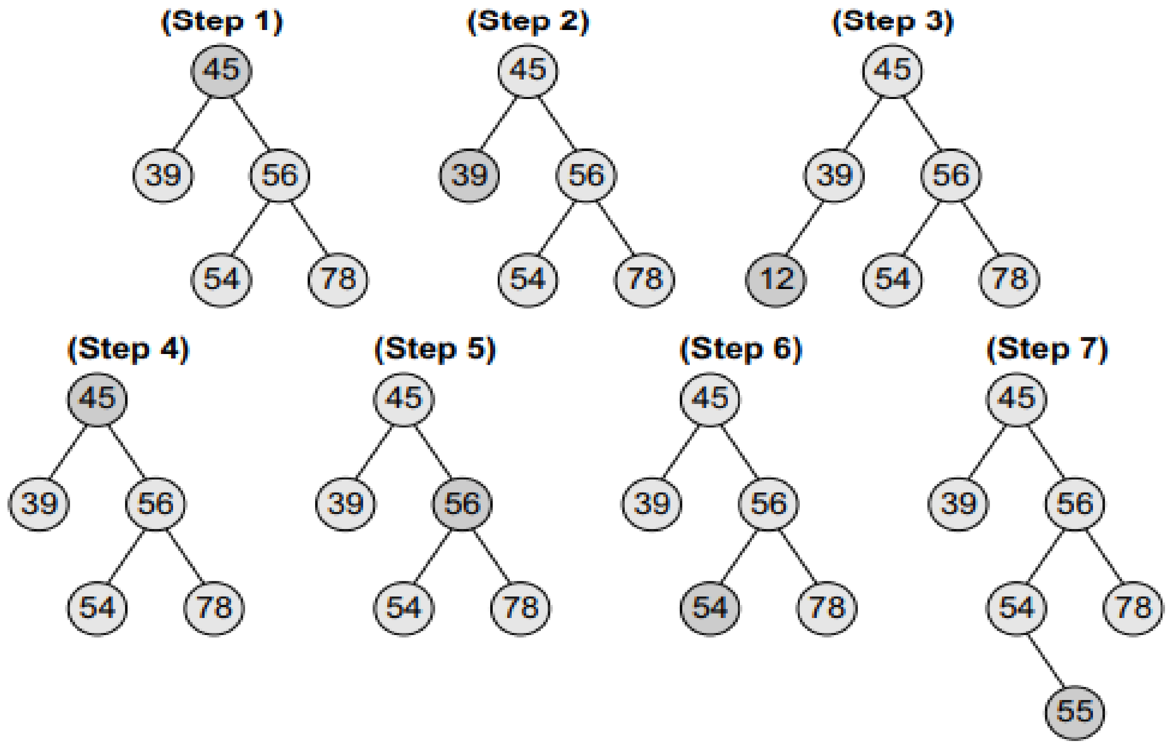
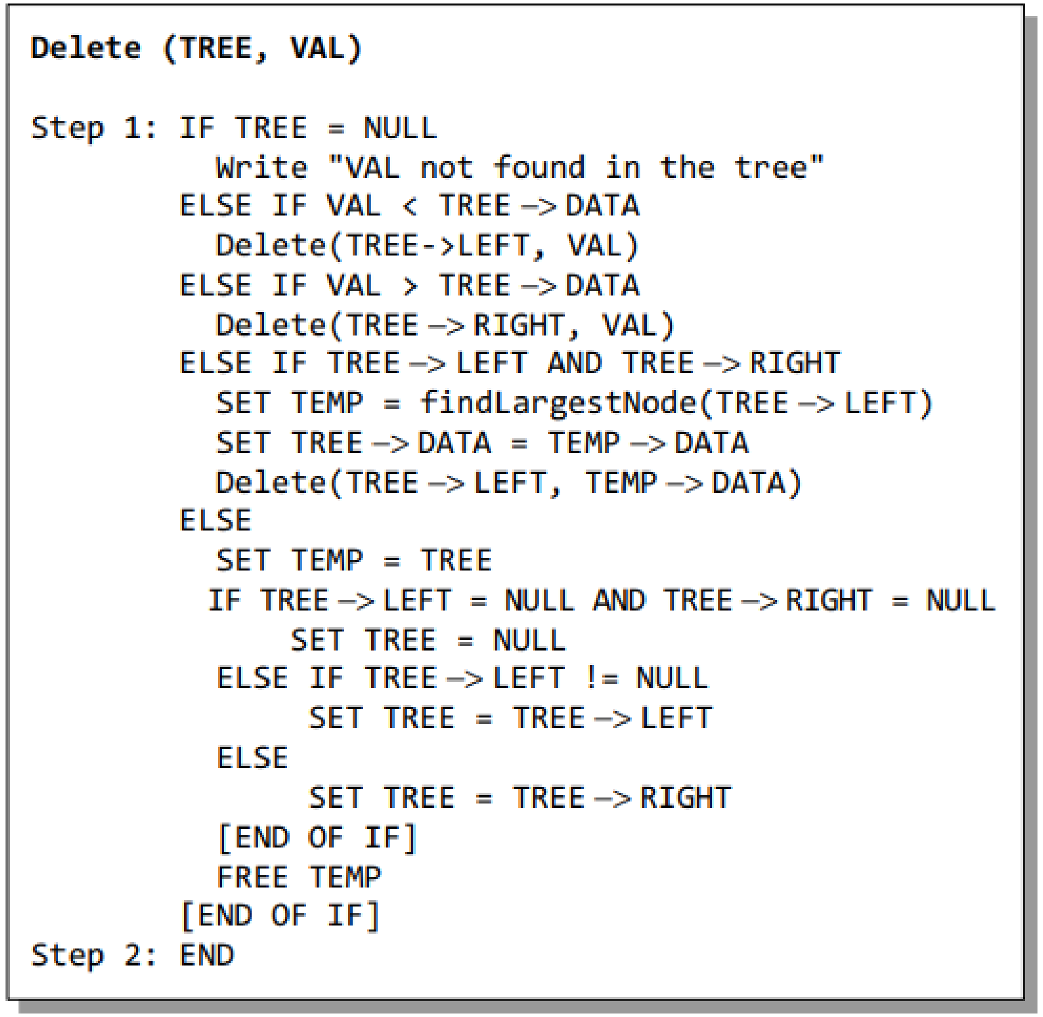


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

Deletion operation on binary search trees



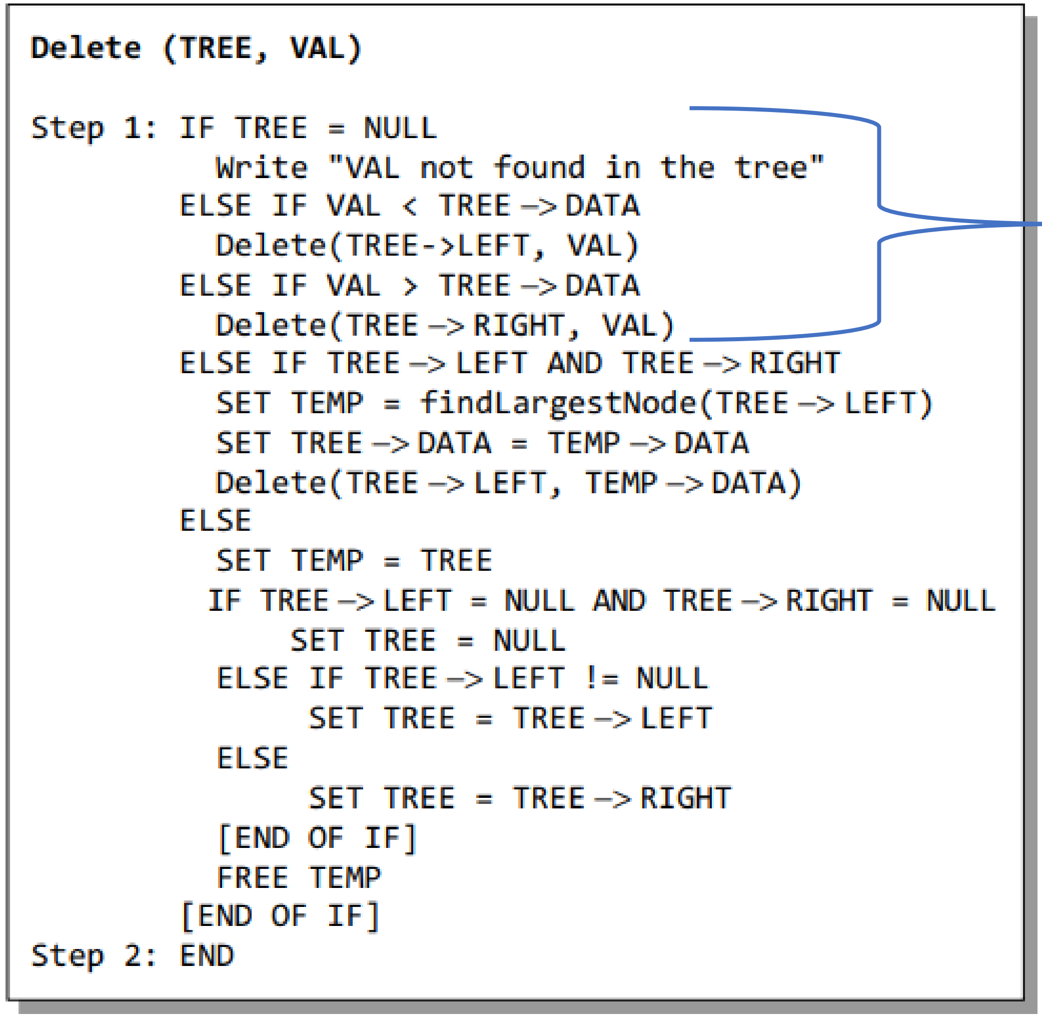
```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
            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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

1. Deleting a leaf node
2. Delete node with one child
3. Delete node with two children

Deletion operation on binary search trees



```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
            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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

1. Deleting a leaf node

- Find a node to be deleted using search operation
- Delete the node using free function

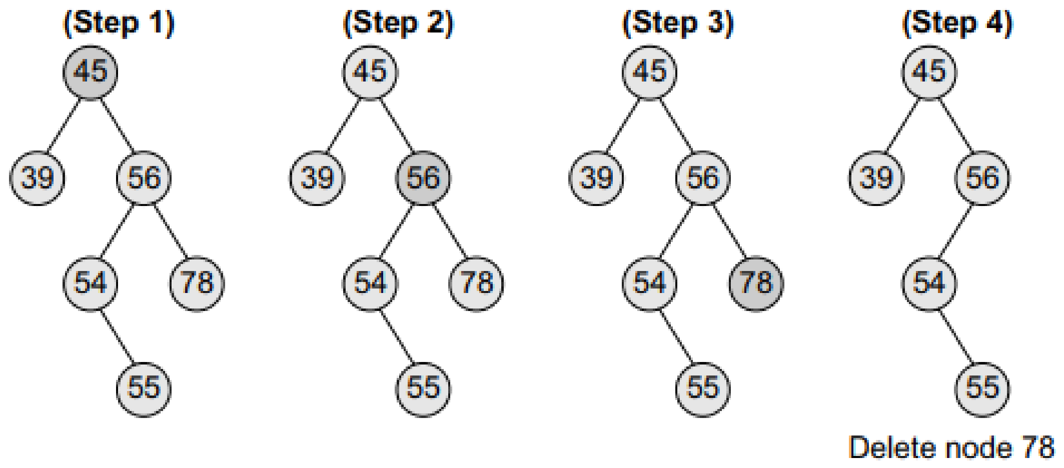
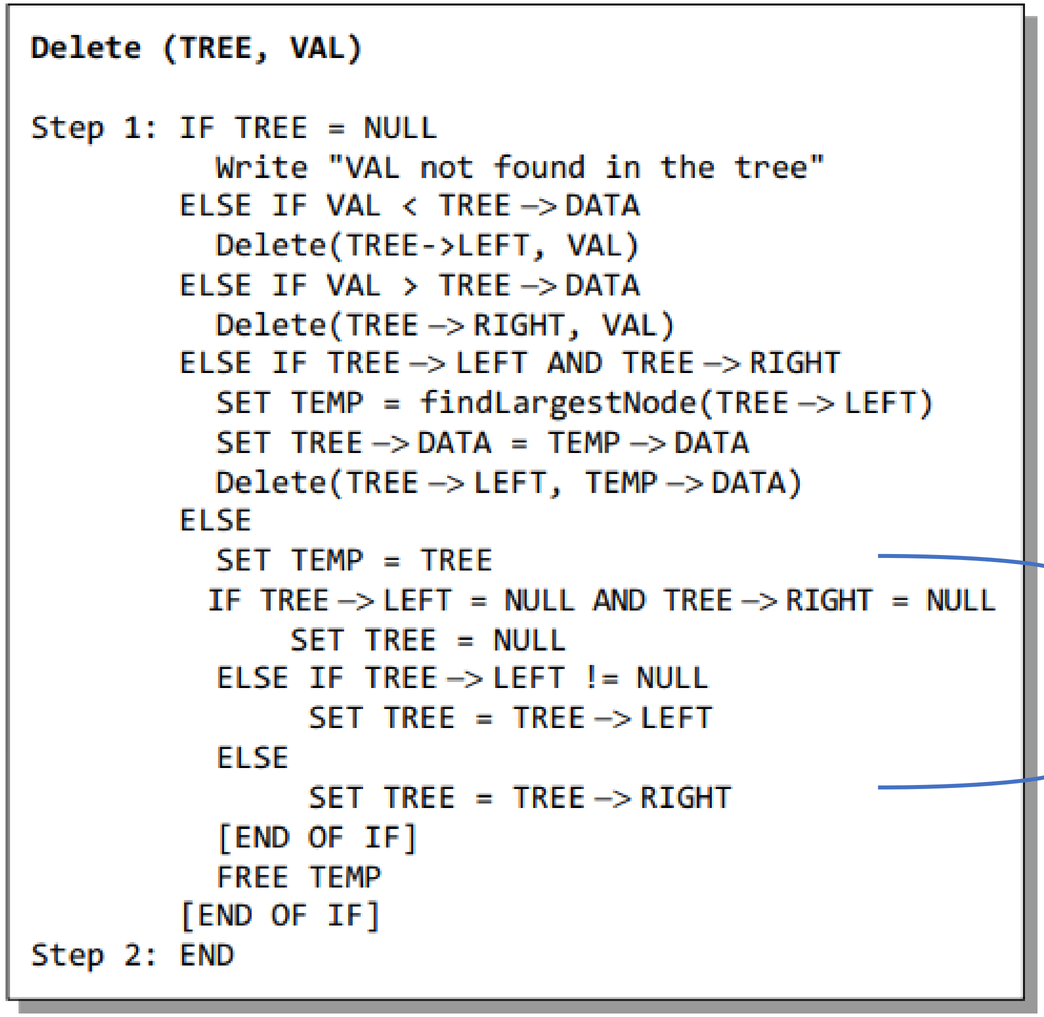


Figure 10.11 Deleting node 78 from the given binary search tree

Deletion operation on binary search trees



```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

2. **Delete node with one child**

- Find a node to be deleted using search operation
- If it has one child, then create link between its parent and child nodes
- Delete the node using free function

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Delete node with one child

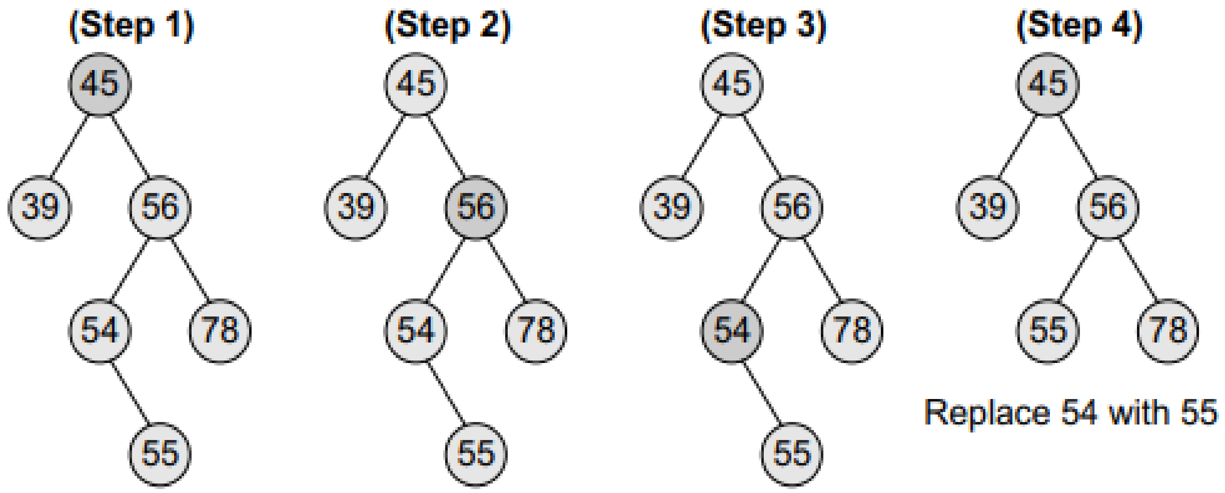
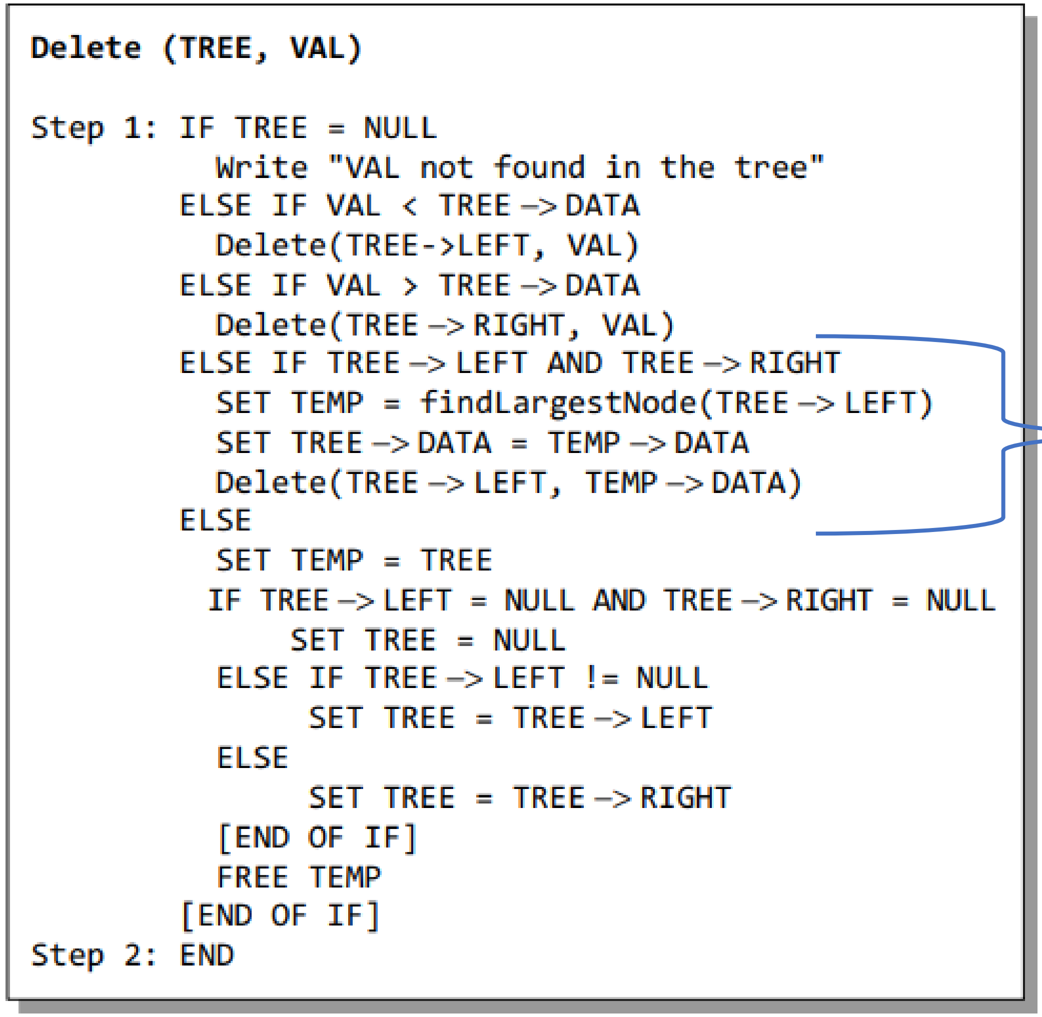


Figure 10.12 Deleting node 54 from the given binary search tree

Deletion operation on binary search trees



```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

i. Three Cases of deletion

3. **Delete node with two children**

i. Find a node to be deleted using search operation
ii. If it has two children, 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)
iii. Swap both deleting node and node which is found in above step
iv. Check whether deleting node came to case 1 or 2 else goto step 2
v. If it comes to case 1, delete the node using case 1
vi. If it comes to case 2, delete using case 2 logic
vii. Repeat the same process until node is deleted from the tree

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling **findLargestNode(TREE -> LEFT)** and replace the current node's value with that of its in-order predecessor. Then, we call Delete(TREE -> LEFT, TEMP -> DATA) to delete the initial node of the in-order predecessor. Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

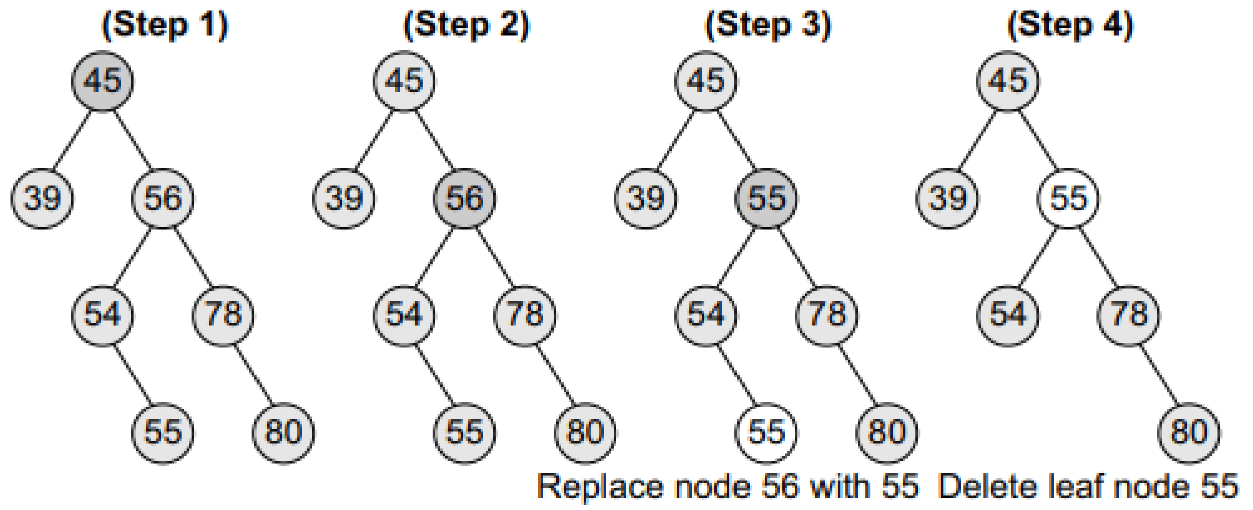


Figure 10.13 Deleting node 56 from the given binary search tree

Threaded Binary Tree:

A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.

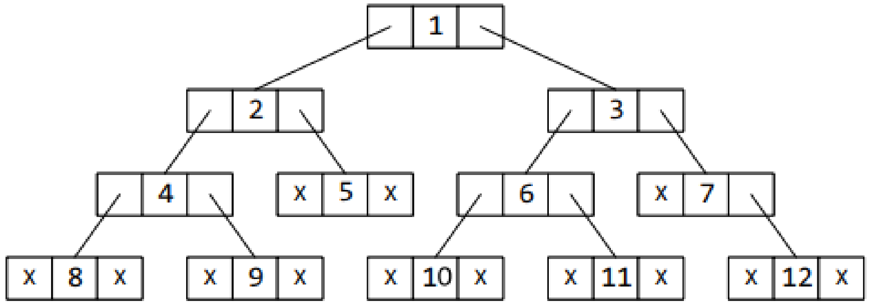


Figure 10.29 (b) Linked representation of the binary tree (without threading)

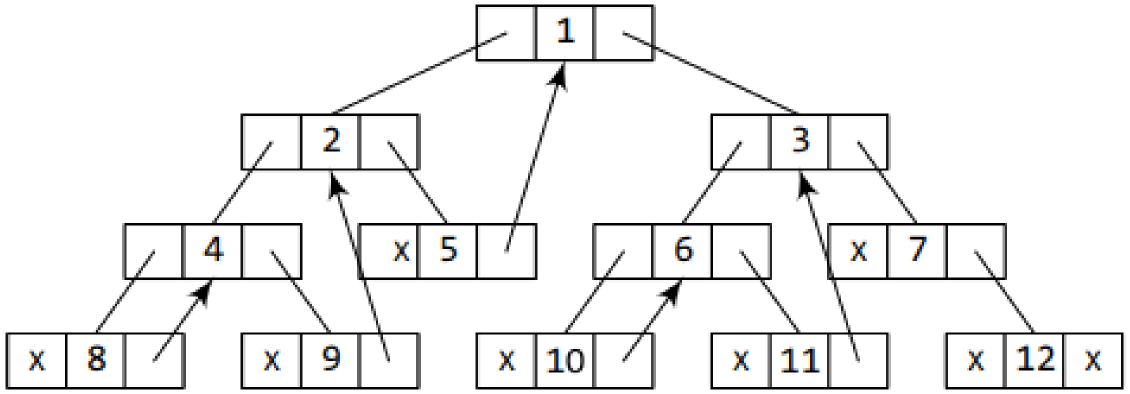


Figure 10.30 (a) Linked representation of the binary tree with one-way threading

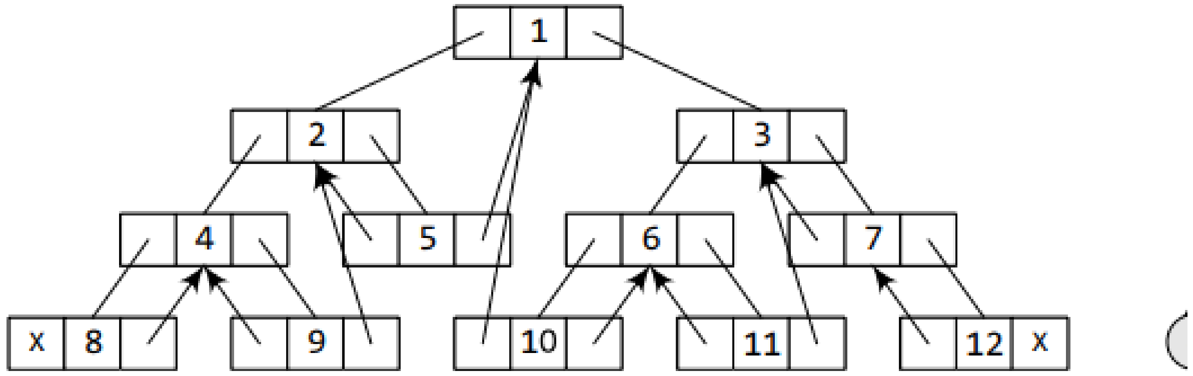


Figure 10.31 (a) Linked representation of the binary tree with threading,

B-Tree :

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

B-Tree of Order m has the following properties...

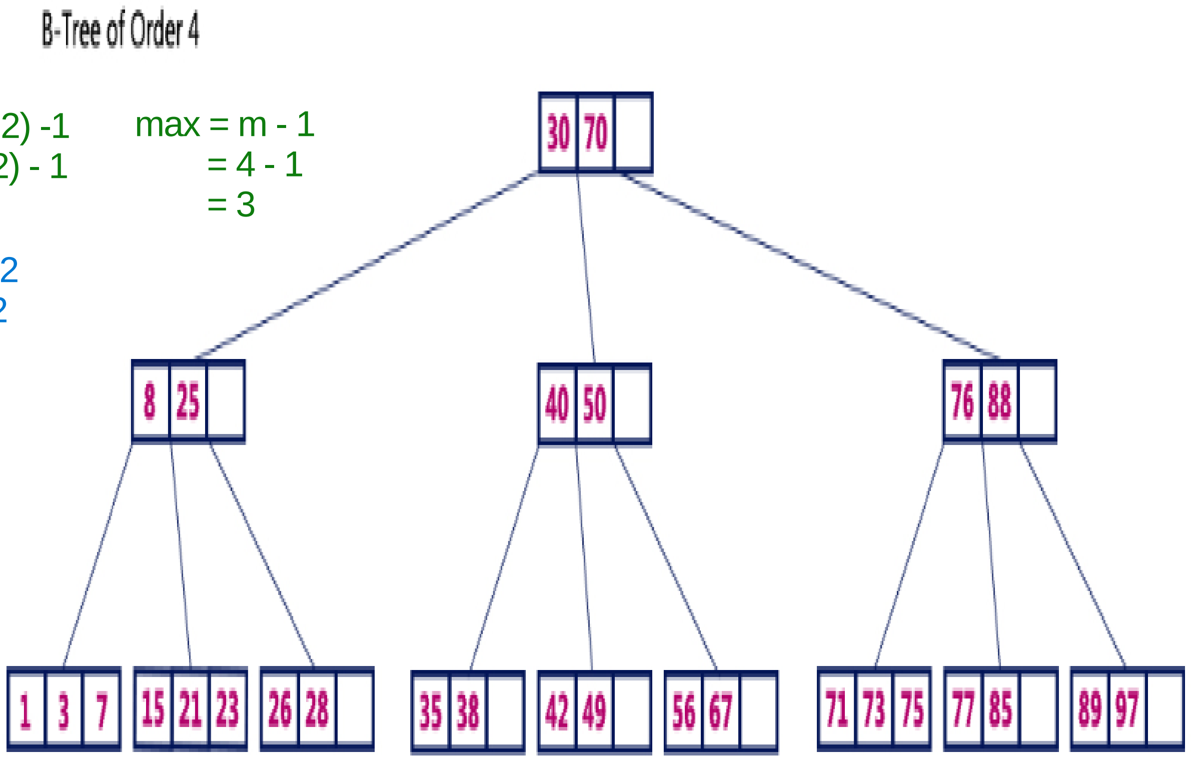
Property #1 - All **leaf nodes** must be **at same level**.
Property #2 - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
Property #4 - If the root node is a non leaf node, then it must have **atleast 2** children.
Property #5 - A non leaf node with **n-1** keys must have **n** number of children.
Property #6 - All the **key values in a node** must be in **Ascending Order**.

m = 4

min = (m / 2) -1
= (4 / 2) - 1
= 1

max = m - 1
= 4 - 1
= 3

min = m / 2
= 4 / 2
= 2



Need: need for lesser time in accessing physical storage media like a hard disk.

Operations on a B-Tree:

Search:

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with first key value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Operations on a B-Tree:

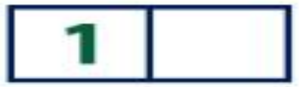
Insertion-In a B-Tree, a new element must be added only at the leaf node.

Step 1 - Check whether tree is Empty.
Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Insertion Operation on a B-Tree: Example-Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

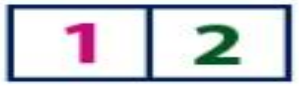
insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



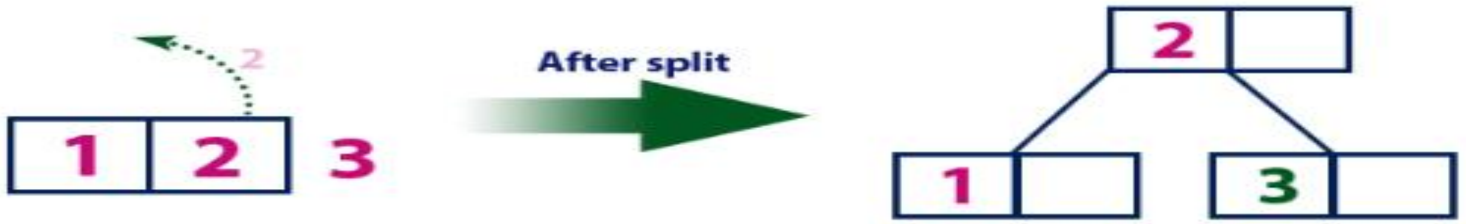
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



insert(3)

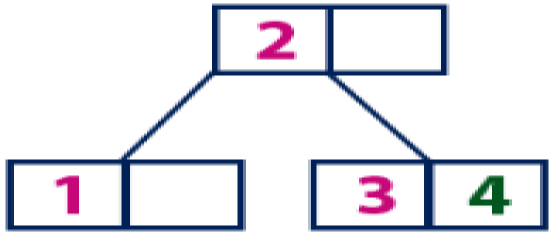
Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



Insertion Operation on a B-Tree: Example-Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

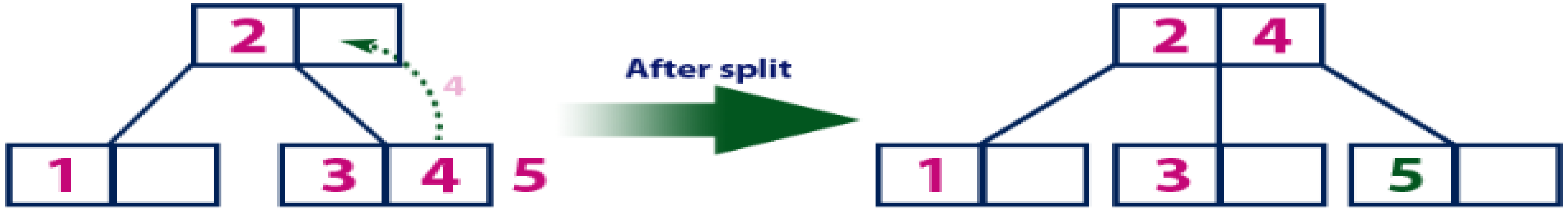
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



insert(5)

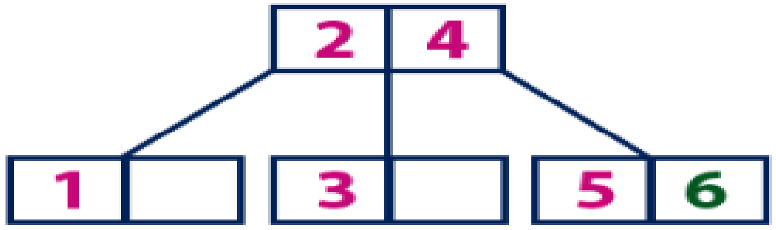
Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



Insertion Operation on a B-Tree: Example-Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

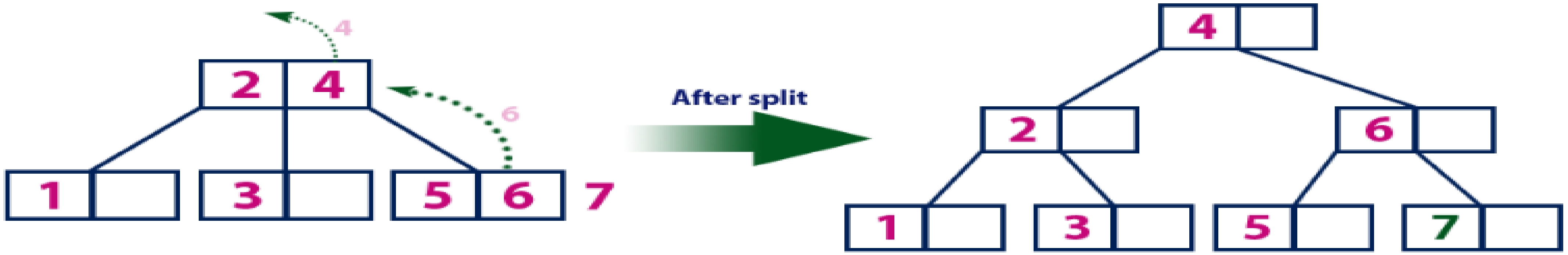
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



insert(7)

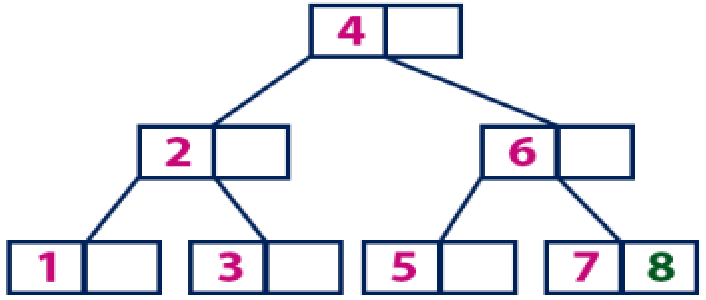
Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



Insertion Operation on a B-Tree: Example-Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

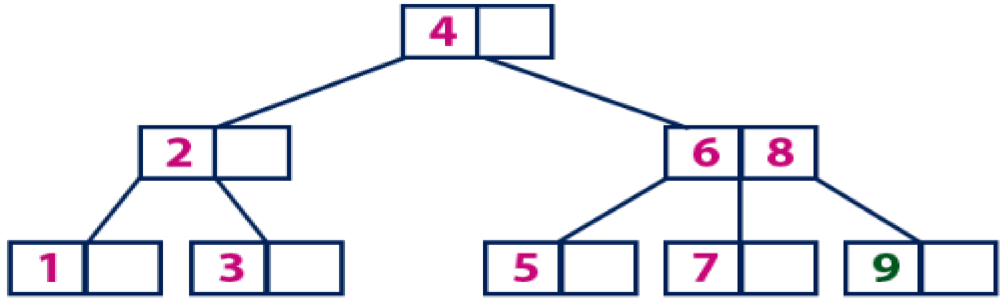
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



insert(9)

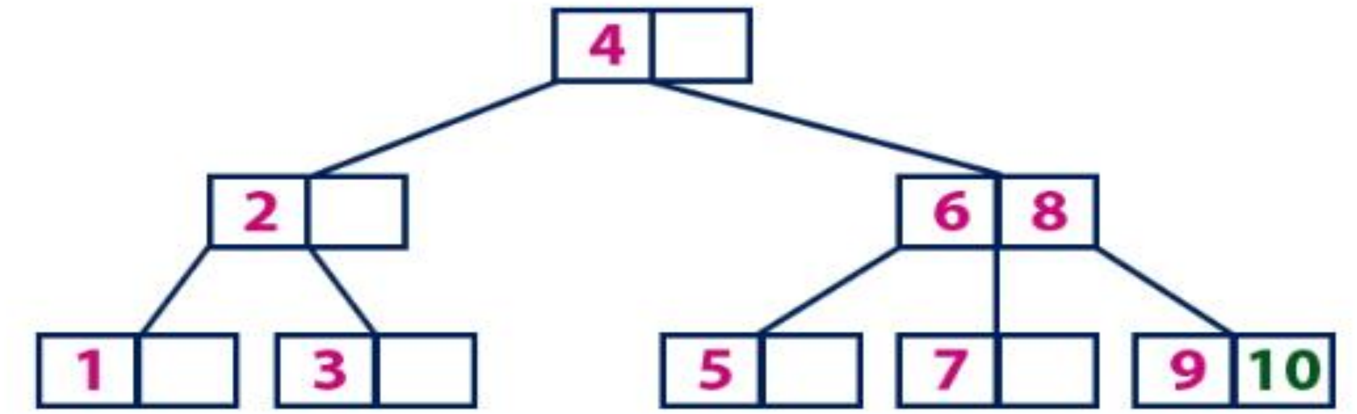
Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



Insertion Operation on a B-Tree: Example-Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

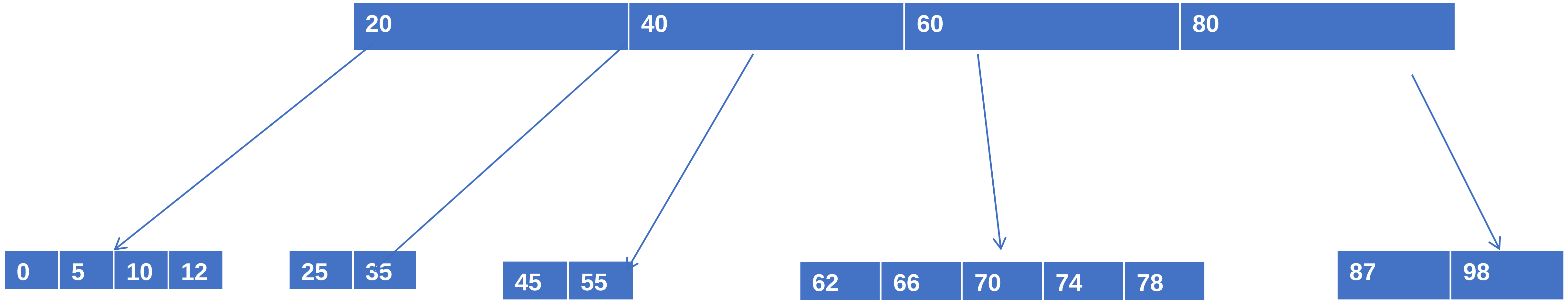
insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '.'10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



http://www.btechsmartclass.com/data_structures/b-trees.html

B-Tree of order 6: Example.



1.Insert 3

2.Insert 38

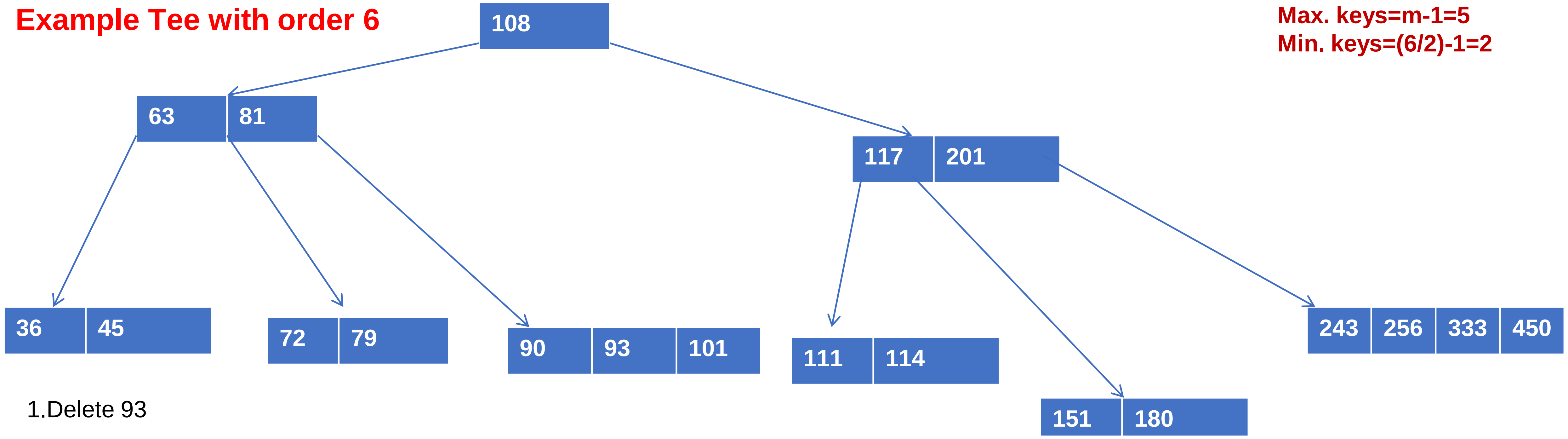
3.Insert 61- *

4.Insert 4

B Tree: Deletion- Any node should not have less than m/2 keys

Example Tee with order 6

Max. keys=m-1=5
Min. keys=(6/2)-1=2



1.Delete 93

2.Delete 201

3.Delete 180

4.Delete 72

1,https://www.programiz.com/dsa/deletion-from-a-b-tree
2. https://www.youtube.com/watch?v=GKa_t7fF8o0

Homework: Create B-Tree of Order 5 by inserting elements- 3,14,7,1,8,5,11,17,13,6,23,12,20,26,4,16,18,24,25,19

Threaded Binary Tree:

- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both
- space wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information
- the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node
- special pointers are called threads and binary trees containing threads are called threaded trees
- Allows fast traversal
- Types – Single threaded, Double threaded

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).

Single Threaded Binary Tree:

- Each node is threaded towards **either** in-order predecessor or successor

Double Threaded Binary Tree:

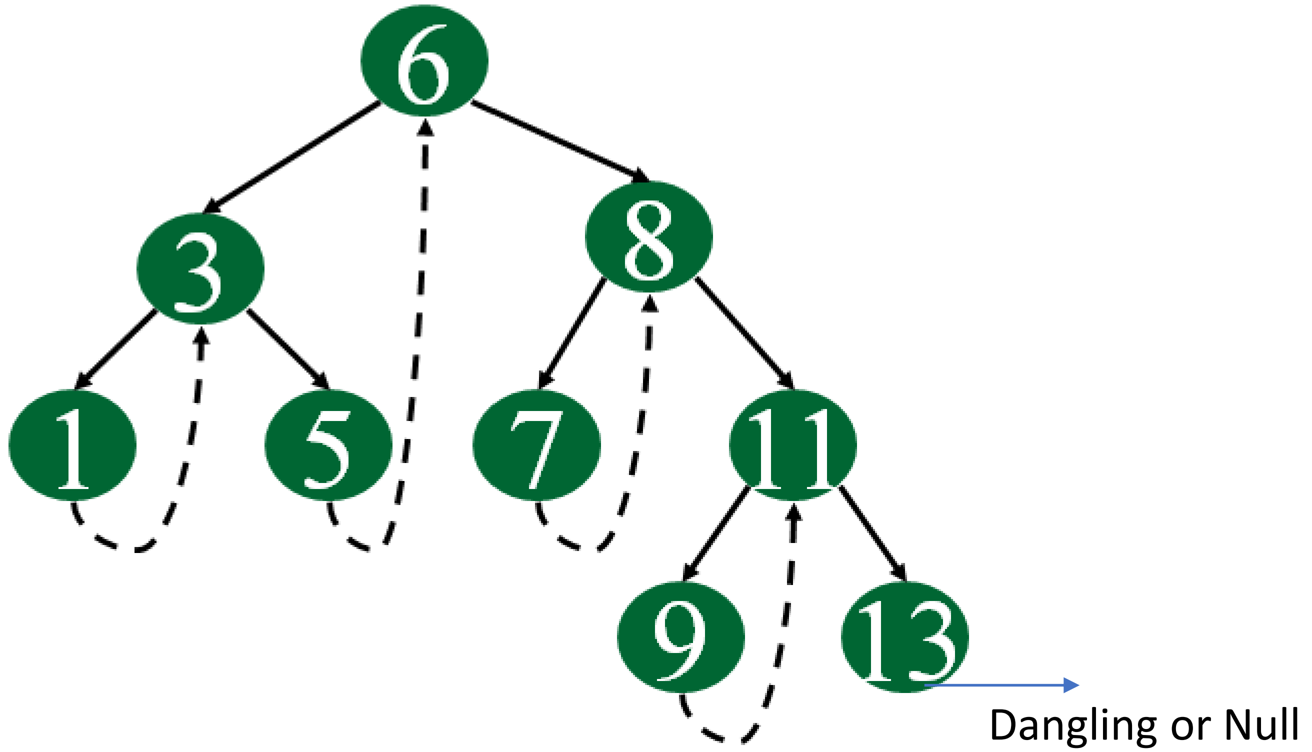
- Each node is threaded **both** in-order predecessor and successor (left-right)

Single Threaded Binary Tree:

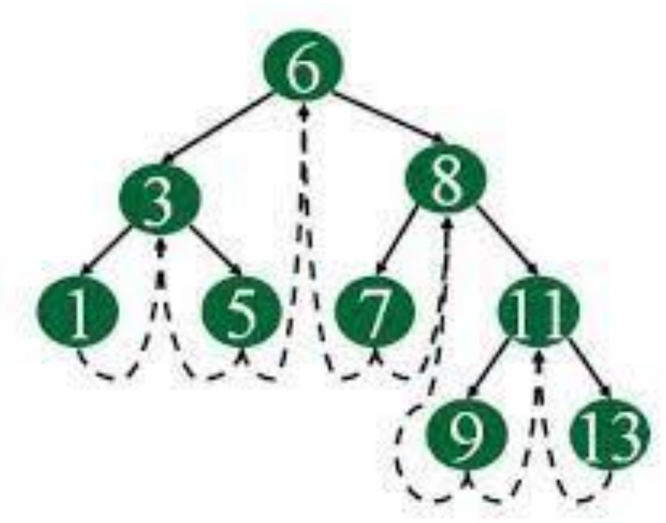
- Each node is threaded towards **either** in-order predecessor or successor

Double Threaded Binary Tree:

- Each node is threaded **both** in-order predecessor and successor (left-right)



Single threaded in-order predecessor (right in-threaded binary tree)



Double Threaded Binary Tree

AVL TREES:

- Self balanced binary search tree
- Difference between the heights of left and right subtrees of every node in the tree is in the range of -1, 0 or +1
- Means height of children differ by atmost 1.
- Every node maintains extra node called balance factor

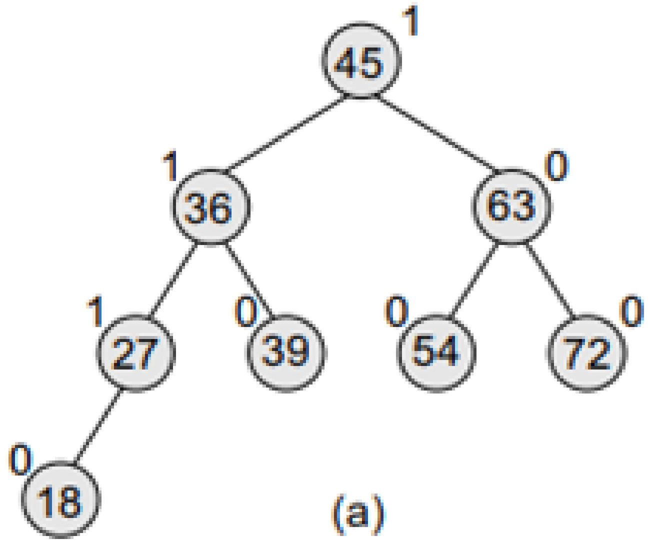
AVL TREES:

Balance factor = Height (left sub-tree) – Height (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 higher than that of the right sub-tree. Such a tree is therefore called as a 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 –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 therefore called as a right-heavy tree.

AVL TREES:

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



(a)

The nodes 18, 39, 54, and 72 have no children, so their balance factor = 0.

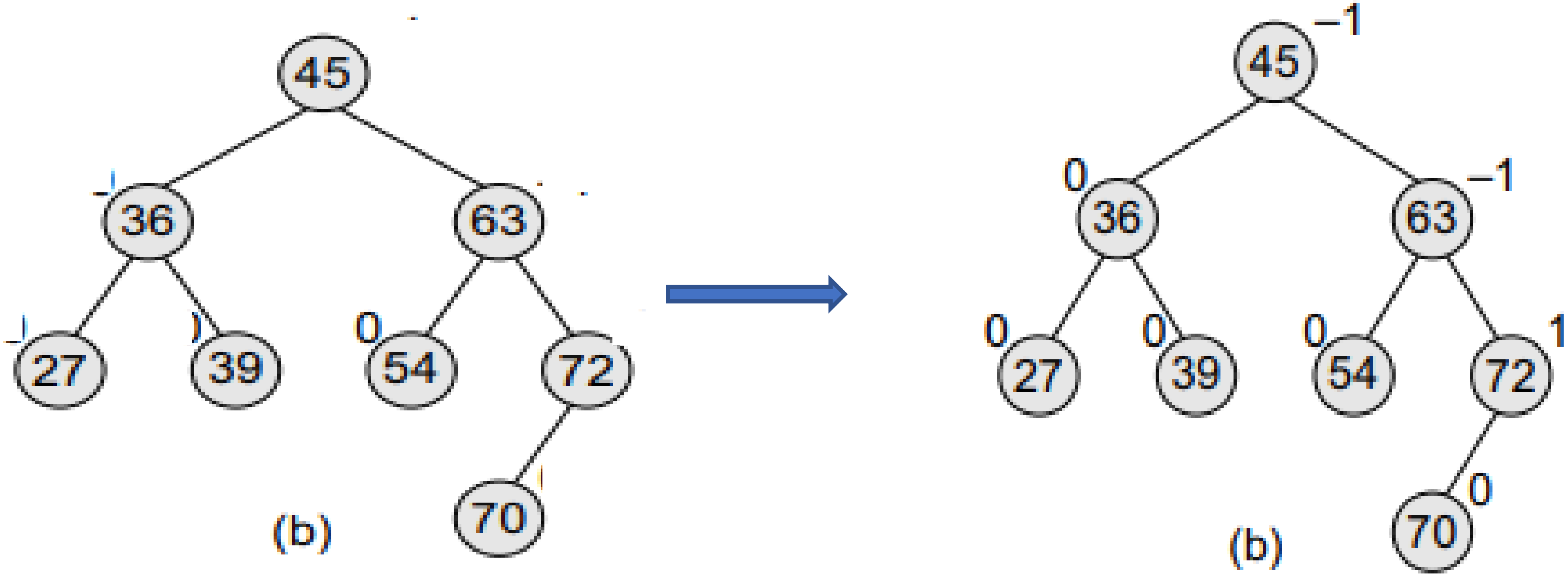
Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height of right sub-tree = 0. Thus, its balance factor = 1.

Look at node 36, it has a left sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = 2 – 1 = 1.

Similarly, the balance factor of node 45 = 3 – 2 =1; and node 63 has a balance factor of 0 (1 – 1)

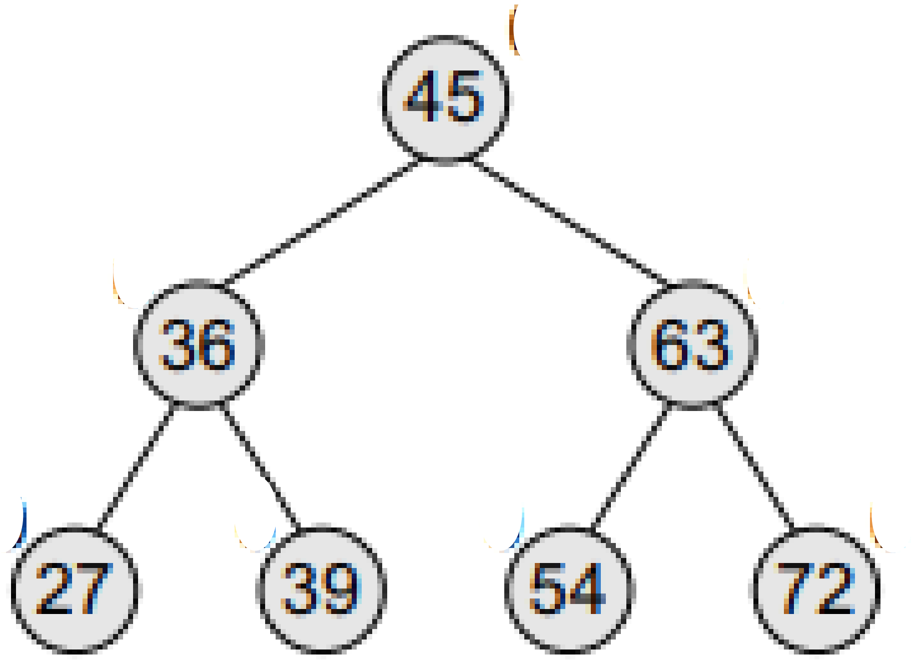
AVL TREES:

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



AVL TREES:

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



(c)

AVL TREES:

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

insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, **rebalancing of the tree** may have to be done.

by performing rotation at the critical node.

AVL TREES:

There are four types of rotations:

1. LL rotation,
2. RR rotation,
3. LR rotation, and
4. RL rotation.

AVL TREES:

During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node.

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, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes 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.

AVL TREES:

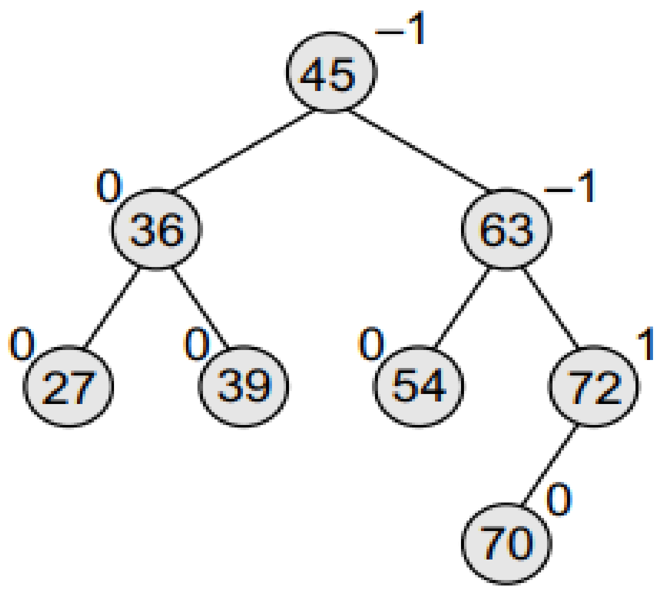


Figure 10.36 AVL tree

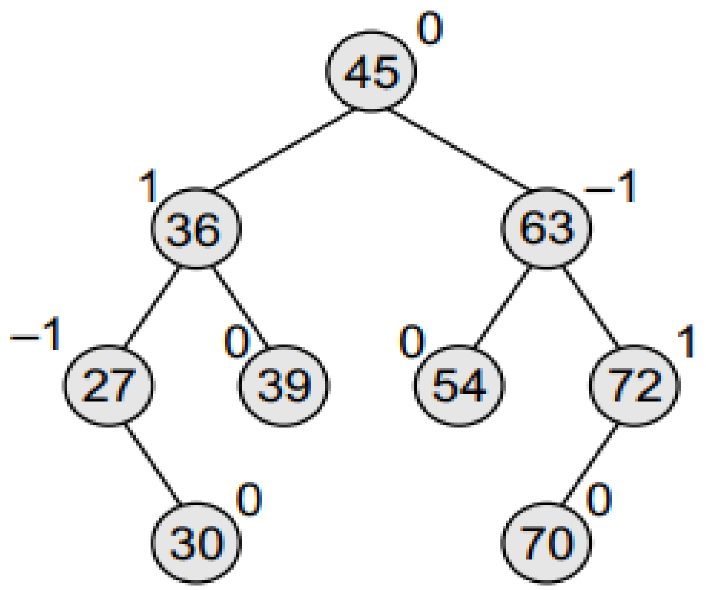


Figure 10.37 AVL tree after inserting a node with the value 30

The tree is still balanced after adding node 30, so no rotation is required

AVL TREES:

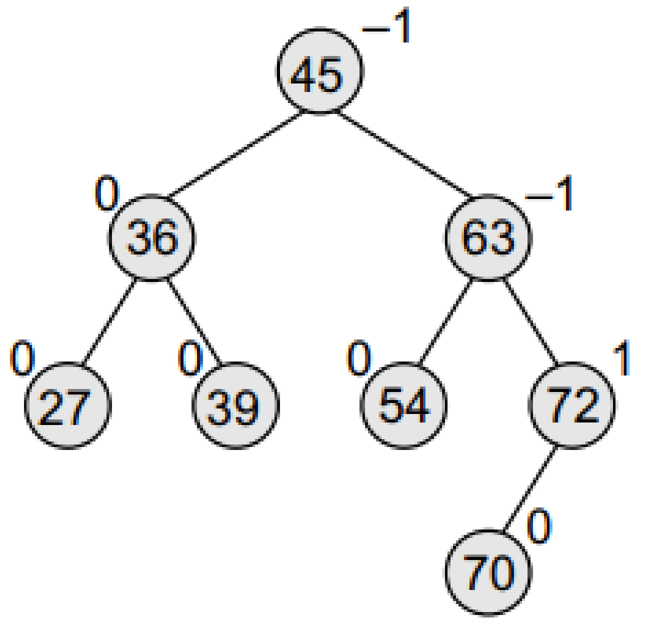


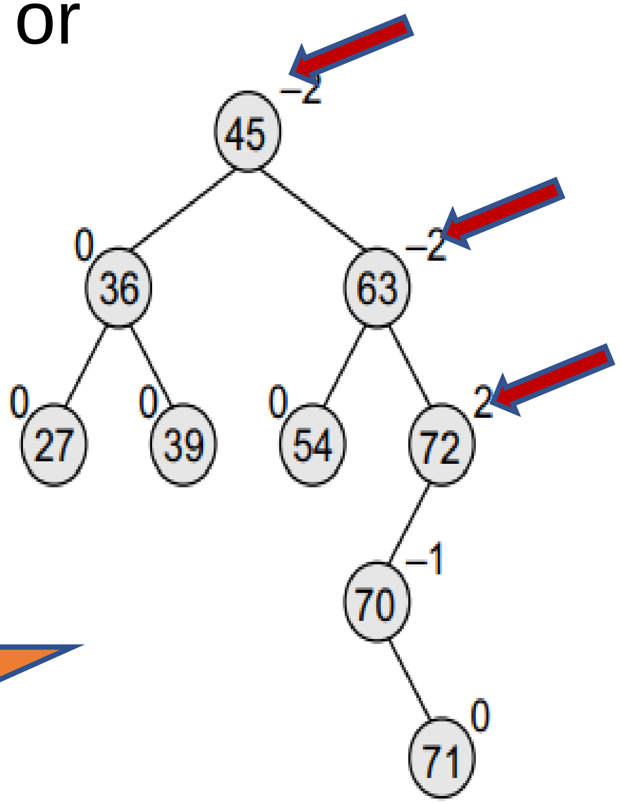
Figure 10.38 AVL tree

Add Node 71 to this node and check if the tree is balanced or not?

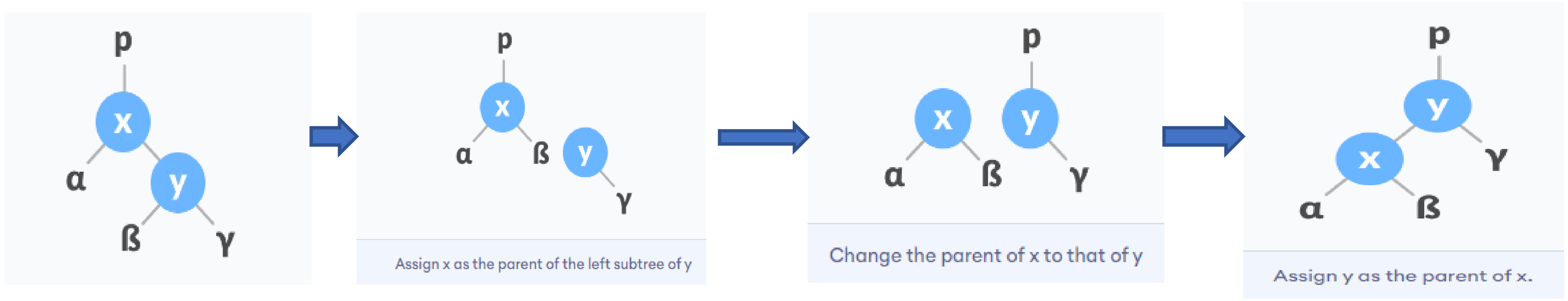
Ans:

Unbalanced tree

Rebalance

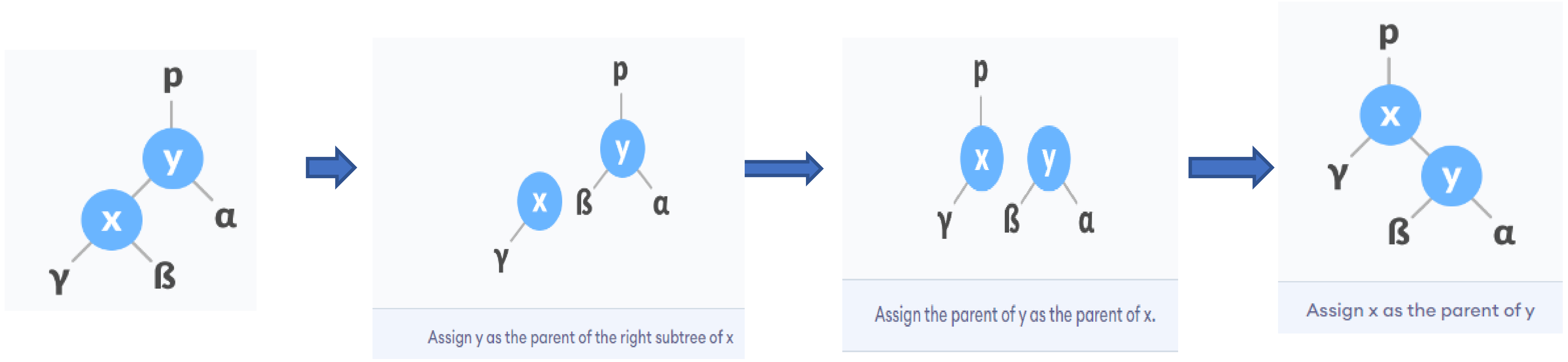


AVL TREES:Left Rotate


p
x
y
α
ß
γ
Assign x as the parent of the left subtree of y
Change the parent of x to that of y
Assign y as the parent of x.

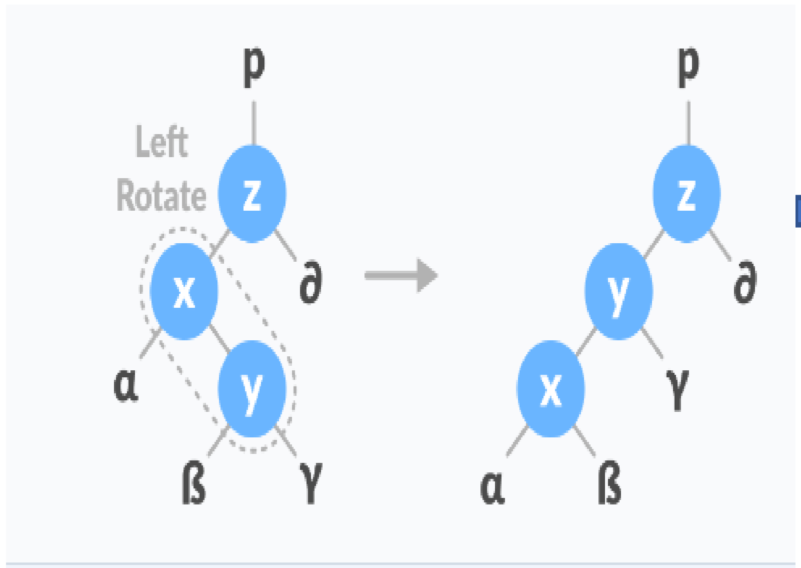
- **If the parent of x is NULL, make y as the root of the tree.**
- **Else if x is the left child of p, make y as the left child of p.**
- **Else assign y as the right child of p**

AVL TREES:Right Rotate

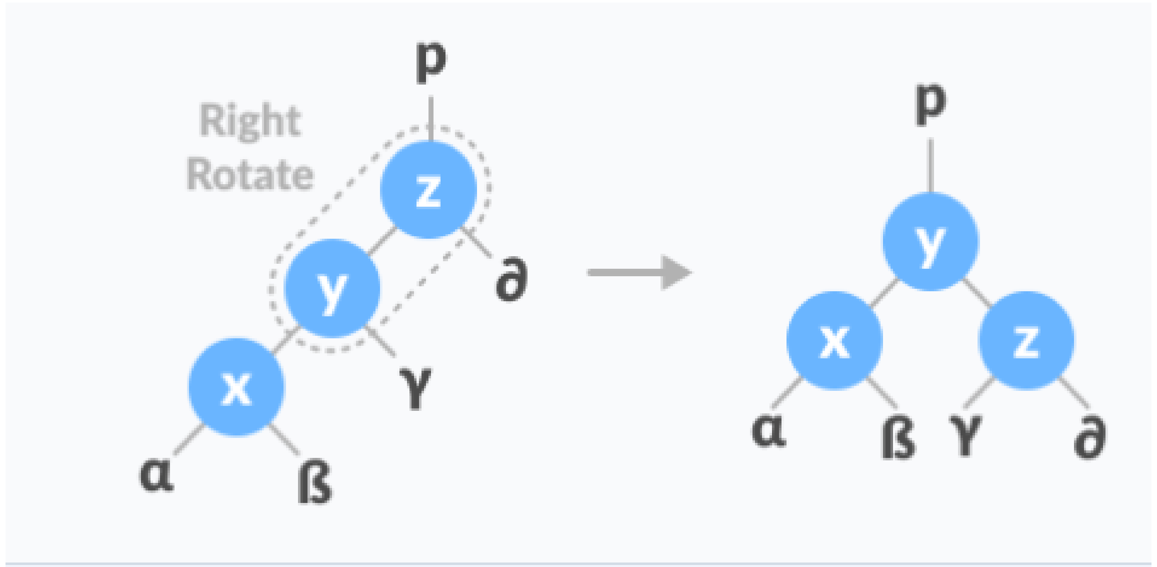

p
y
x
α
γ
ß
Assign y as the parent of the right subtree of x
Assign the parent of y as the parent of x.
Assign x as the parent of y

- **If the parent of y is NULL, make x as the root of the tree.**
- **Else if y is the right child of its parent p, make x as the right child of p.**
- **Else assign x as the left child of p.**

AVL TREES:Left-Right

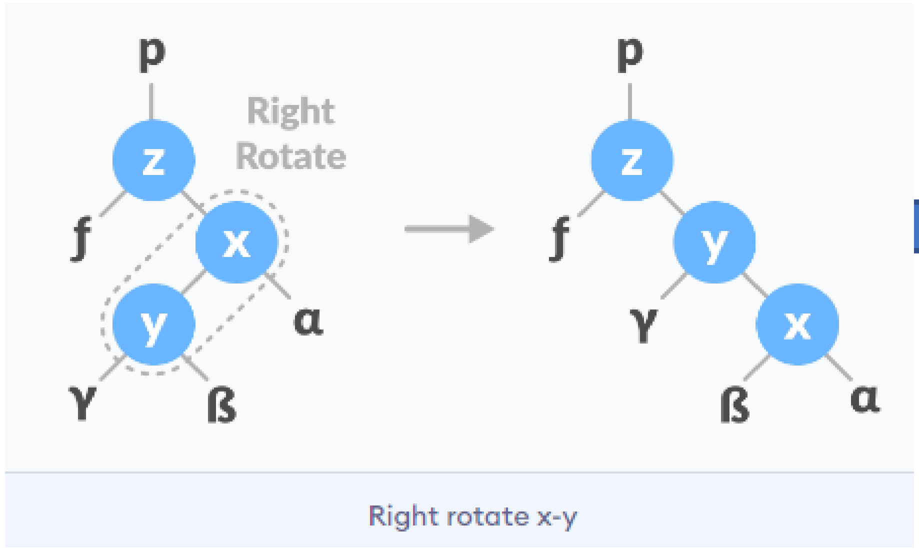


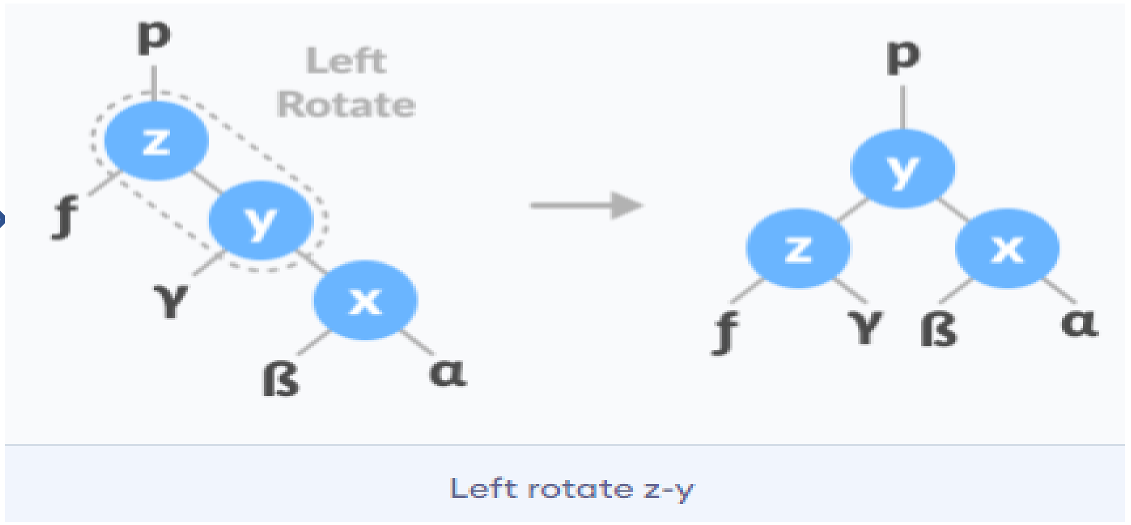
Left rotate x-y



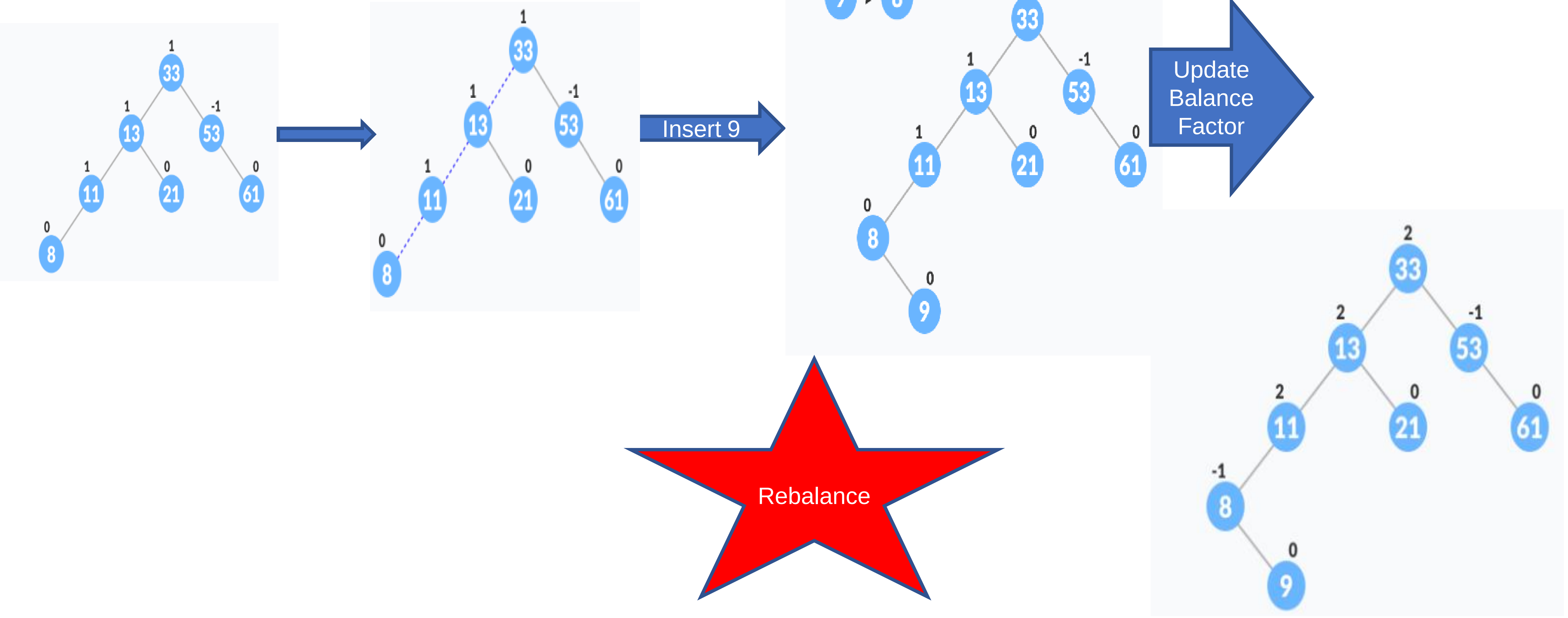
Right rotate z-y

AVL TREES:Right-Left Rotation

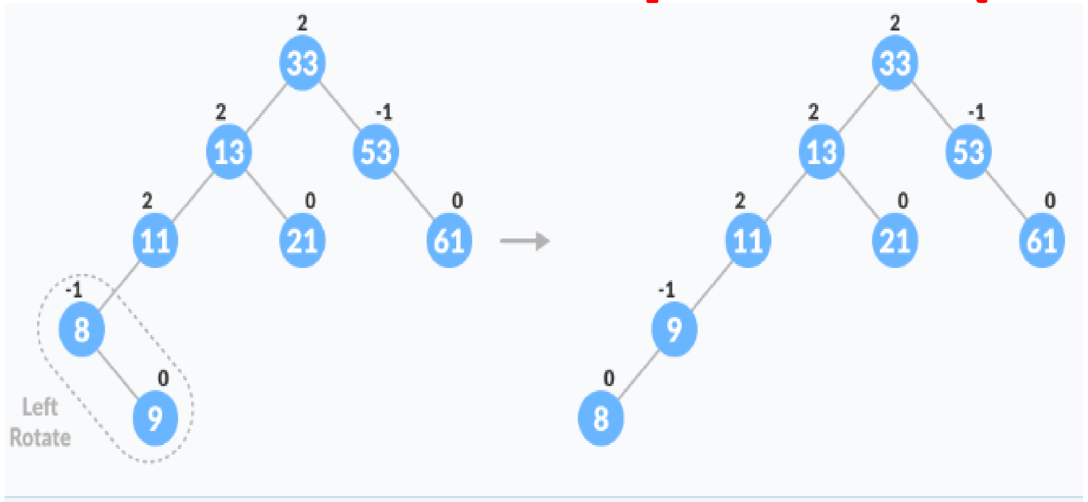




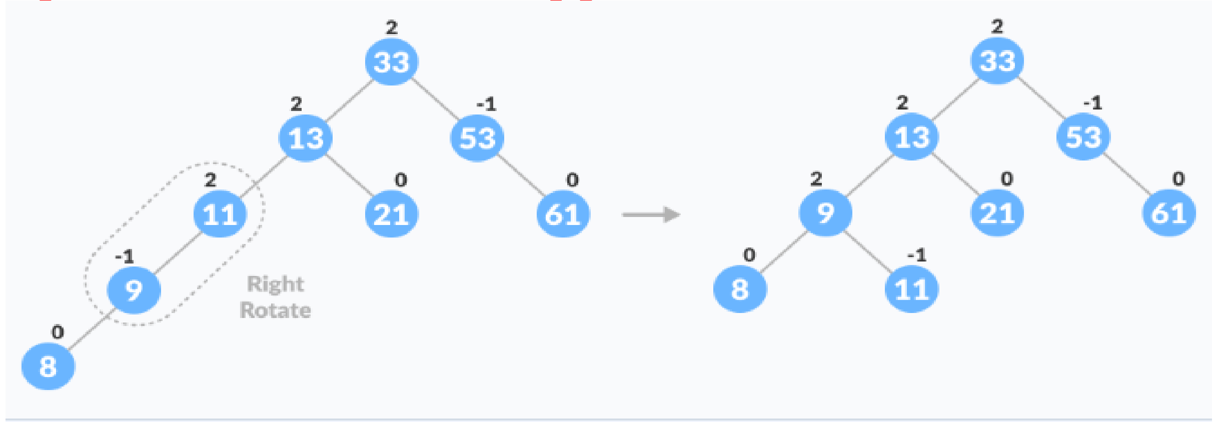
AVL TREES:


1 33
1 13
-1 53
1 11
0 21
0 61
0 8
Insert 9
9 > 8
0 9
Update Balance Factor
2 33
2 13
2 11
-1 8
Rebalance

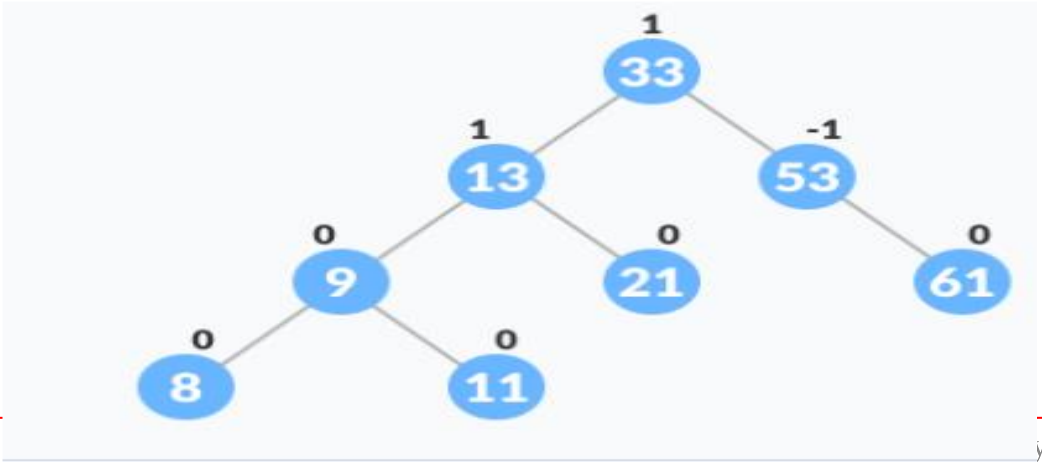
AVL TREES:Example cntd(insert 9)…rebalancing



Balancing the tree with rotation



Balancing the tree with rotation



Final balanced tree

1. **If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation**
   a. **If newNodeKey < leftChildKey do right rotation.**
   b. **Else, do left-right rotation**

AVL TREES:Example cntd…rebalancing

1. If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

a.If newNodeKey < leftChildKey do right rotation.
b.Else, do left-right rotation

1.If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do left rotation or right-left rotation

a.If newNodeKey > rightChildKey do left rotation.
b.Else, do right-left rotation

AVL TREES:Example : insert 23,12,25,01,45,63,27,29

AVL TREES:Deletion

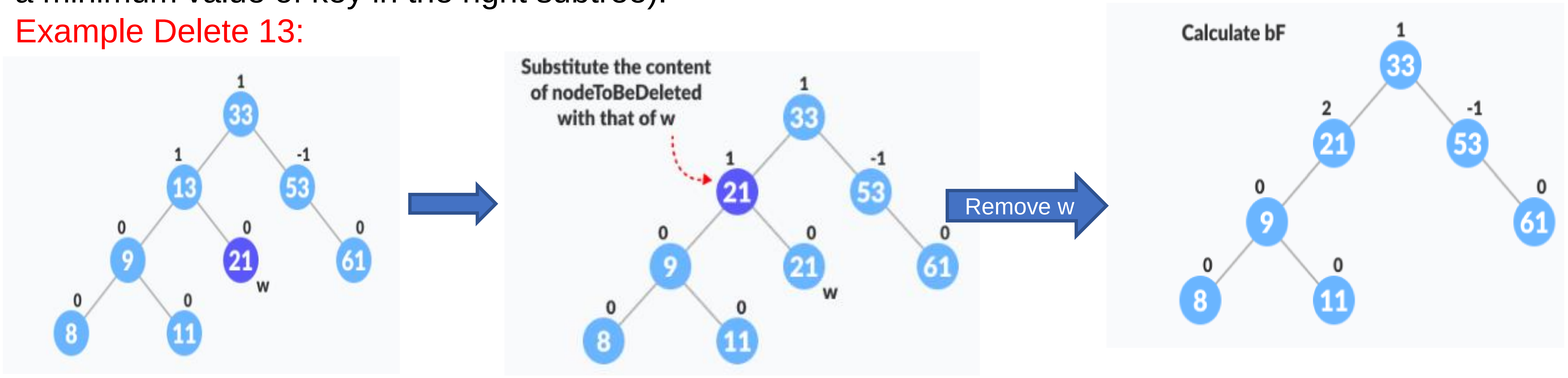
There are three cases for deleting a node:

1.If nodeToBeDeleted is the leaf node (ie. does not have any child), then remove nodeToBeDeleted .

2.If nodeToBeDeleted has one child, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.

3.If nodeToBeDeleted has two children, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).

Example Delete 13:

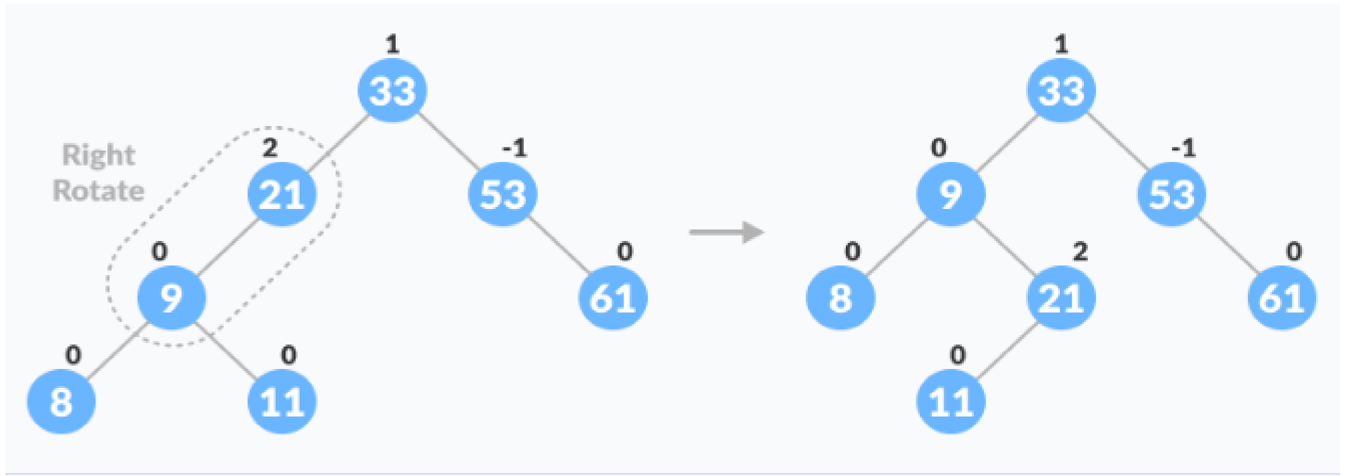


AVL TREES:Deletion-Example cntd...Example Delete 13:

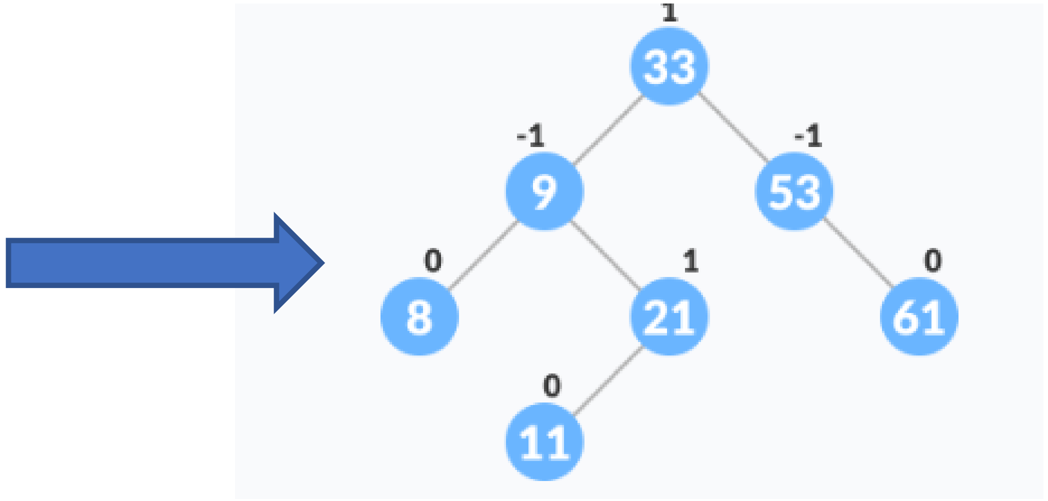
Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1

1.If balanceFactor of currentNode > 1,
If balanceFactor of leftChild >= 0, do right rotation.
Else do left-right rotation.

2.If balanceFactor of currentNode < -1,
If balanceFactor of rightChild <= 0, do left rotation.
Else do right-left rotation.



Right-rotate for balancing the tree



AVL-Question-What is AVL Tree ? Construct AVL Tree for following dataset: 22,27,31,10,5,15,39,19,16,11,3,4,8, Mention rotation at each step

Complexity

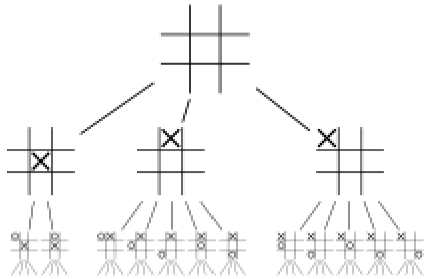
Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

AVL Tree Applications

For indexing large records in databases

For searching in large databases

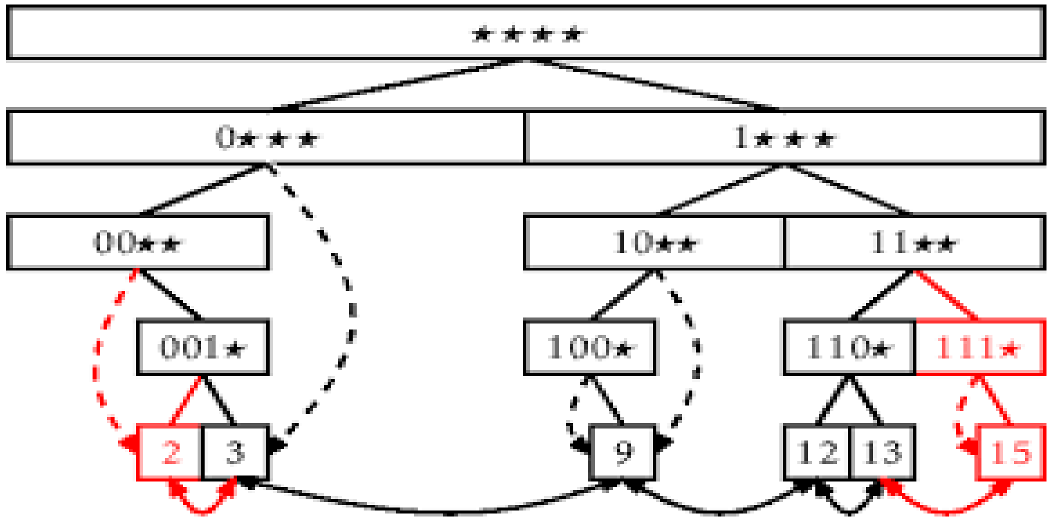
Game Tree:



- A **game tree** is a type of recursive search function that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move.
- They are very useful for Artificial Intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play. The most commonly-cited example is chess, but they are applicable to many situations such as tic-tac-toe.
- Game trees are generally used in board games to determine the best possible move. For the purpose of this article, Tic-Tac-Toe will be used as an example.
- The idea is to start at the current board position, and check all the possible moves the computer can make. Then, from each of those possible moves, to look at what moves the opponent may make. Then to look back at the computer's. Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion. It will do this for every possible outcome (see image below), effectively playing thousands (often more) of games. From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.
- This can be likened to how people think during a board game - for example "if I make this move, they could make this one, and I could counter with this" etc. Game trees do just that. They examine every possible move and every opponent move, and then try to see if they are winning after as many moves as they can think through.

.

Digital search tree :



- A Digital search tree is a binary tree that helps us access and manages all the data safely.
- The tree tends to hold only binary data and helps us. There seem to be a catch implying that if the search tree started with a zero (0), then it is present in the left subtree of the search tree, and if the tree started with a one (1), then it is present in the right subtree of the search tree.
- Both sides of the tree worked exclusively to make the work of the tree efficient.

Digital search tree :

Advantages of the digital search tree:

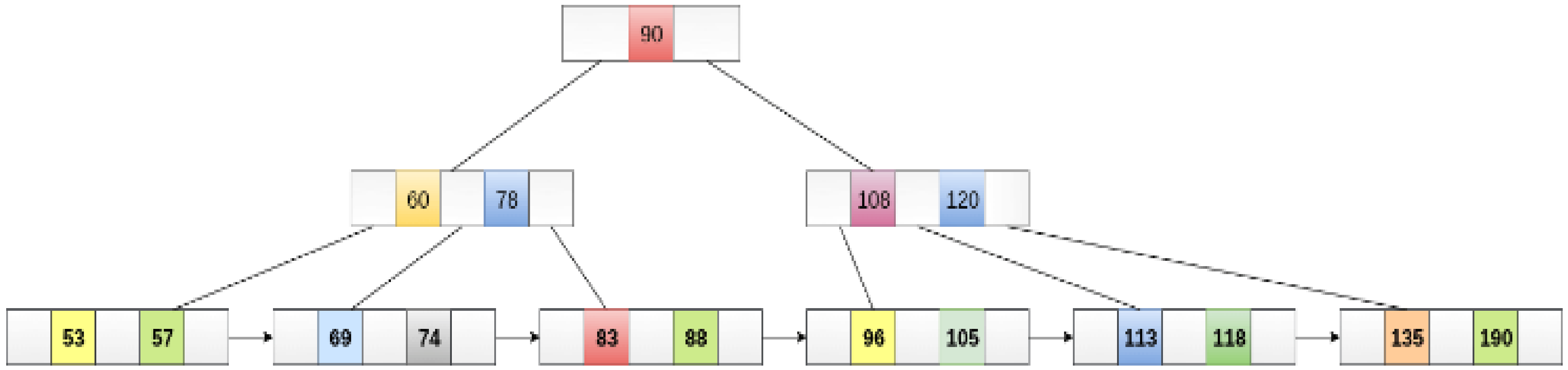
- The main benefit of the digital search tree is that it occupies and consumes less space and memory as compared to the rest of the other trees, says AVL tree or binary tree.
- We don't have to compile any extra data or information to maintain the hierarchy of the tree because the depth of the tree is usually maintained by the length of the key. This key maintains the posture by keeping a count on the number of bits present.
- Multiple operations can take place very efficiently.
- Various operations such as deletion, insertion, or search can take place in DST much more easily as compared to the other trees such as binary search, AVL, or simply binary tree.

Disadvantages of the digital search tree

- If we are given a key or node with long data, and the comparison takes a lot of time, this can lead to a costly procedure.
- We already know that the operations we perform in these trees are conducted via bits, which can result in difficulties when handling bitwise.
- The data or records that we encounter are generally not organized.
- There might be a case where we encounter keys that have similar errors, and they have different lengths. https://www.tutorialandexample.com/digital-search-tree-in-data-structures

B+ tree :

- B+ tree in data structures is nothing but just an extended version of the B tree. It allows the smooth working of all the operations such as insertion, deletion, traversing, and several others.
- They are generally known for their excellent application in storing and managing a large amount of data in the storage space rather than fitting them all into the main memory.



B+ tree : Properties

- The very first property is that all the nodes that are present in the tree are at the same magnitude or say level.
- Next, another trait of the B+ tree is that the root must have at least two children or nodes.
- In order to count the keys that are present in this kind of tree, we use the following formulas. To find out the maximum key present within a node we use m -1 and to find the least number of keys present within a node we use m / 2 – 1.
- Another trait of the B+ tree is that each and every node present in the tree can either acquire a maximum number of nodes that will be calculated by m children or a minimum number of nodes that will be calculated by m / 2 – 1.

B+ Tree Applications:

Multilevel Indexing

Faster operations on the tree (insertion, deletion, search)

Database indexing

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

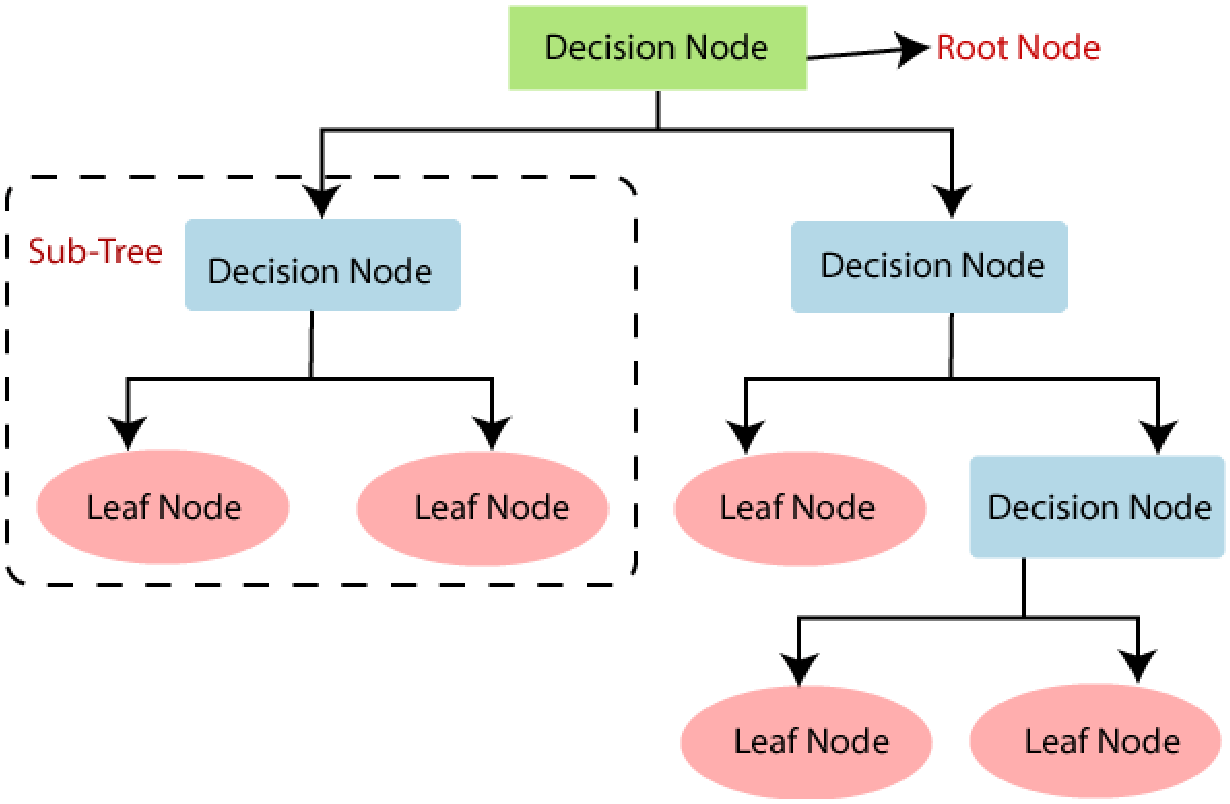
Decision tree :

- Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**
- In a Decision tree, there are two nodes, which are the **Decision Node** and **Leaf Node.** Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.

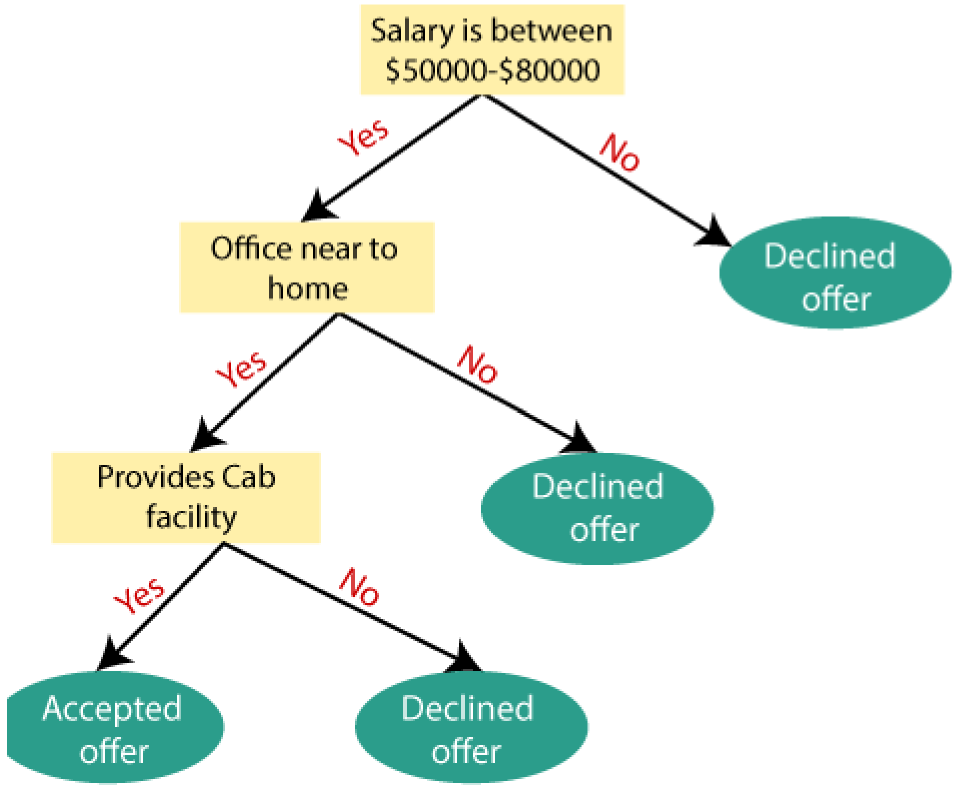
It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.

- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.

Decision tree :



Example:



Decision tree :

Advantages of the Decision Tree:

1.It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
2.It can be very useful for solving decision-related problems.
3.It helps to think about all the possible outcomes for a problem.
4.There is less requirement of data cleaning compared to other algorithms.

Disadvantages of the Decision Tree:

1.The decision tree contains lots of layers, which makes it complex.
2.It may have an overfitting issue, which can be resolved using the **Random Forest algorithm.**
3.For more class labels, the computational complexity of the decision tree may increase.

Thank You

https://www.javatpoint.com/b-plus-tree
https://www.programiz.com/dsa/insertion-on-a-b-plus-tree