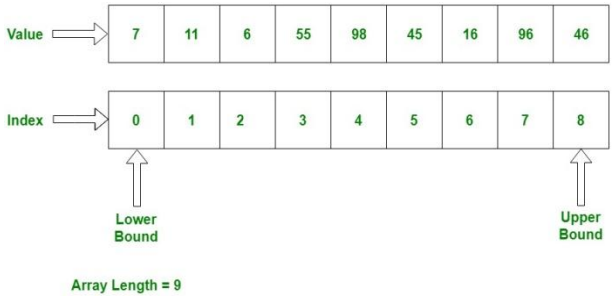
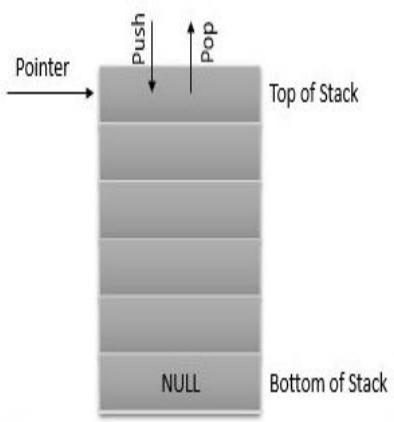
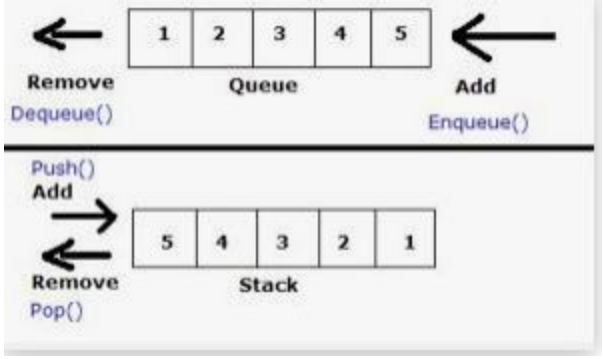
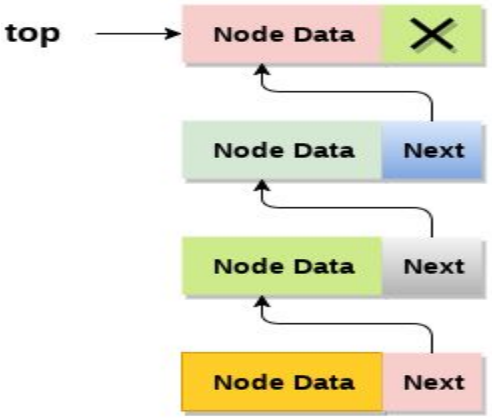
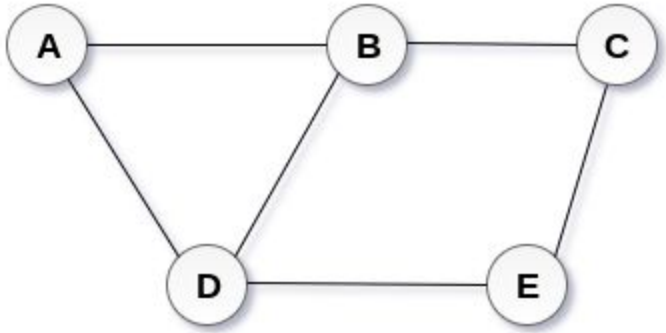


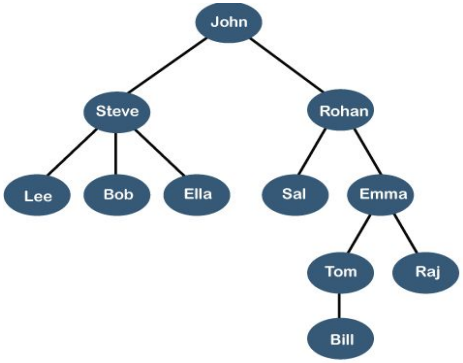




Non Linear Data Structure Part 1[Tree]







Outline

- Definition of Tree
- Representation of Tree
- Basic Terminology of Tree
- Types of Tree [General Trees, Forests, Binary Trees, Binary Search Trees, Expression Trees, Tournament Trees].
- Binary Tree
- Types of Binary Tree
- Conversion of General Tree to a Binary Tree
- Understanding and Implementing Binary Tree Traversal [In-order, Pre-Order, Post-Order]
- Constructing a Binary Tree from Traversal Results
- Storage Representation and Manipulation of Binary Tree
- Applications of Tree.
- Binary Search Trees

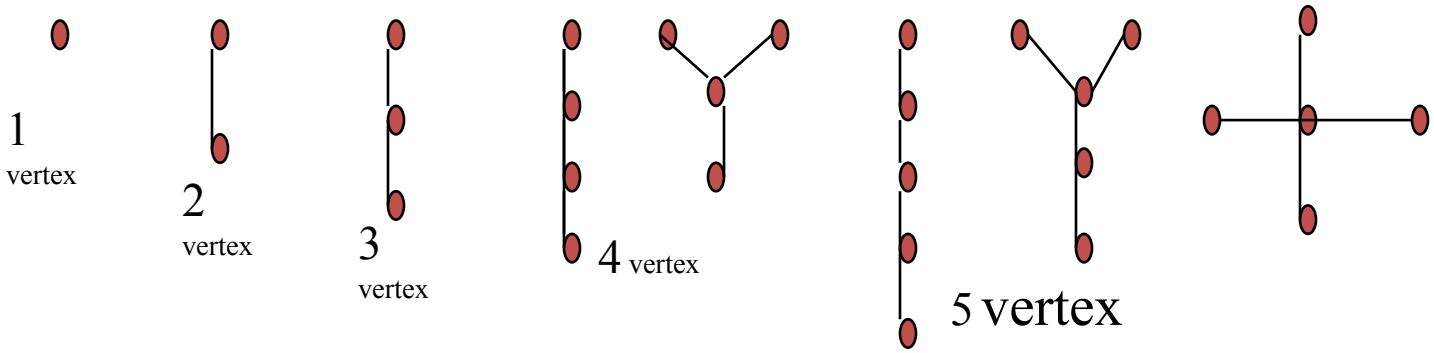
Outline

- Operations on Binary Search Trees
  - Searching for a Node in a Binary Search Tree
  - Inserting a New Node in a Binary Search Tree
  - Deleting a Node from a Binary Search Tree
  - Determining the Height of a Binary Search Tree
  - Determining the Number of Nodes
  - Finding the Smallest Node in a Binary Search Tree
  - Finding the Largest Node in a Binary Search Tree
  - Traversal and Search in Binary Search Tree
  - Implementation of Basic Operation in Binary Search Tree
- Representation of AVL Tree [Height Balanced Binary Tree]
- Understanding Insertion and Deletion in AVL Tree
- Determining Height in an AVL Tree.

Tree Terminology

- **Tree** : A tree is a connected graph without any circuit. OR A graph G is called a tree if it is a connected acyclic graph.
- **Acyclic** : A graph G is called acyclic if it contains no cycles.

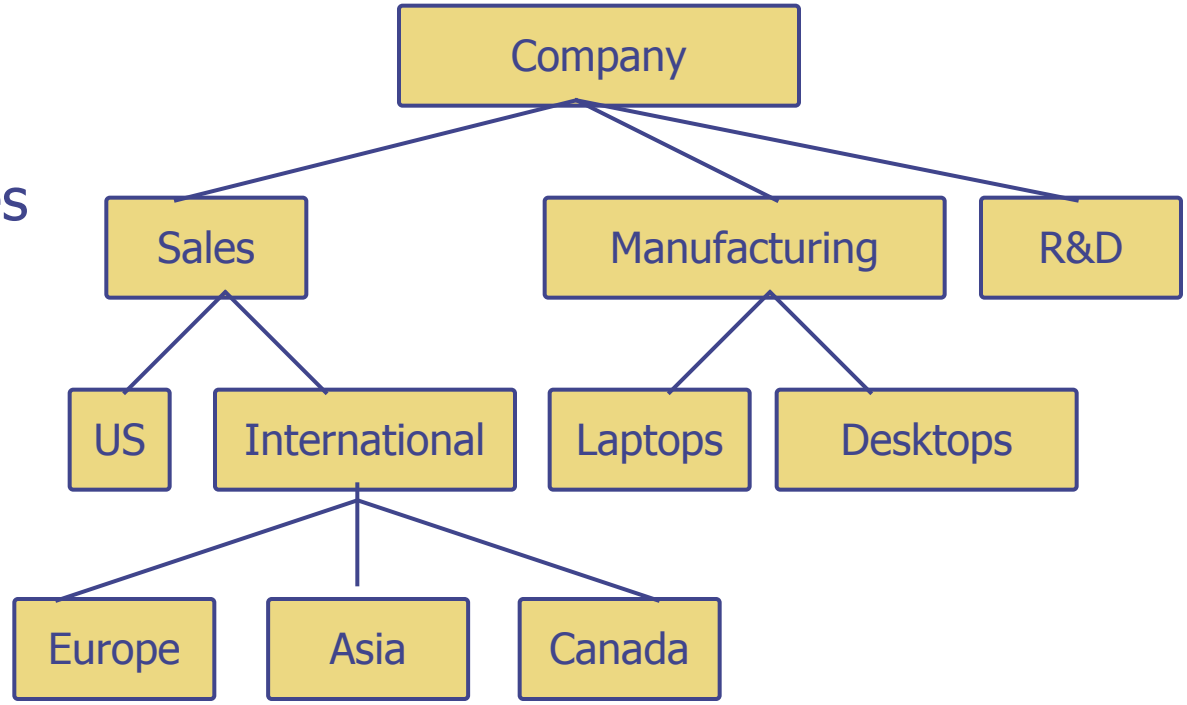
The tree with at most 5 vertices.



- Trees:
  - Definition of tree
  - Representation of tree
  - Types of tree
  - Binary tree traversal
  - Storage representation and manipulation of binary tree
  - Conversion of general tree to a binary tree
  - Other representation of tree, application to tree.
- Graphs :
  - Representation of graphs
  - Graph traversal and spanning forest.
  - Finding the shortest path (Warshall's Algorithm, Warshall's modified algorithm, Dijkstra's Technique)
  - Graph traversal (depth first search, breadth first search)

Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - File systems
  - Programming environments



Tree: Recursive Definition

1. An empty structure is an empty tree
2. If $t_1, t_2, \ldots t_K$ are disjoint trees, then the structure whose root has, as its children, the roots of $t_1, t_2, \ldots t_K$ is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

Terminology for a Tree

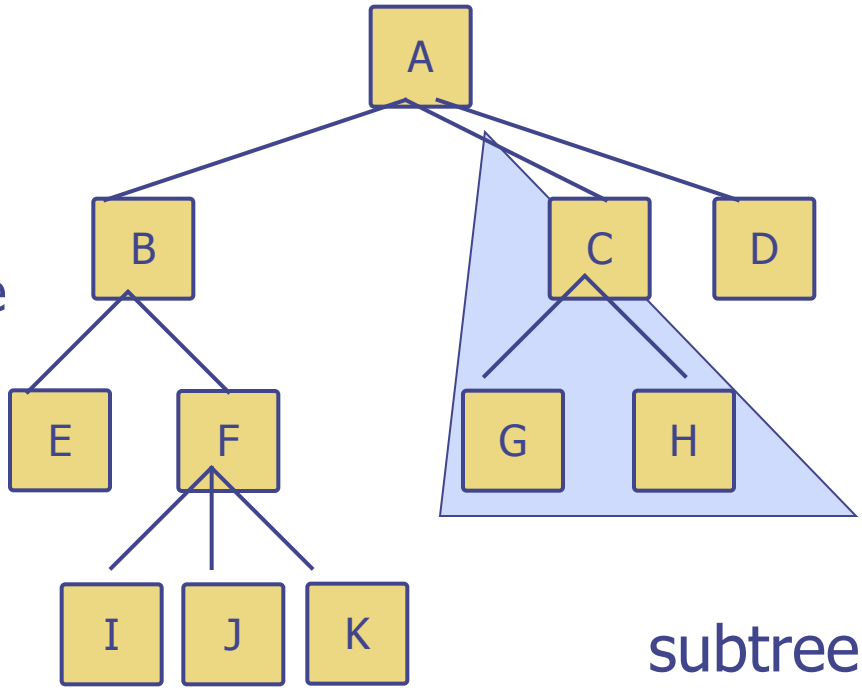
- **Tree:** A collection of data whose entries have a hierarchical organization
- **Node:** An entry in a tree
- **Root node:** The node at the top
- **Terminal** or **leaf node:** A node at the bottom
- **Parent:** The node immediately above a specified node
- **Child:** A node immediately below a specified node
- **Ancestor:** Parent, parent of parent, etc.
- **Descendent:** Child, child of child, etc.
- **Siblings:** Nodes sharing a common parent

Terminology for a Tree

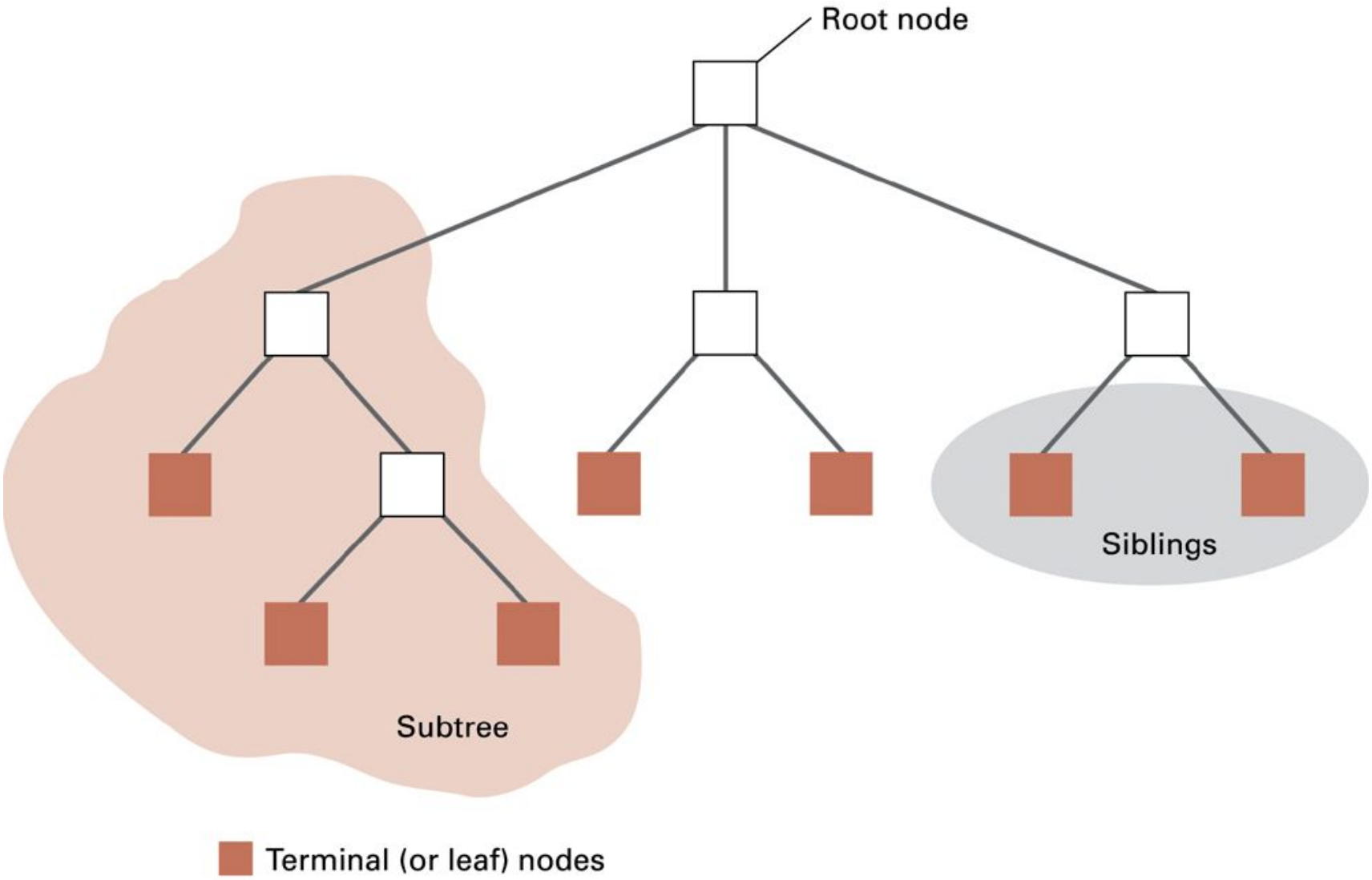
- **Binary tree:** A tree in which every node has at most two children
- **Depth:** The number of nodes in longest path from root to leaf
- **Depth of a node**: number of ancestors
- **Height of a tree**: maximum depth of any node
- **Subtree**: tree consisting of a node and its descendants

Tree Terminology

- Internal node: node with at least one child (A, B, C, F)
- External node (leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Subtree: tree consisting of a node and its descendants

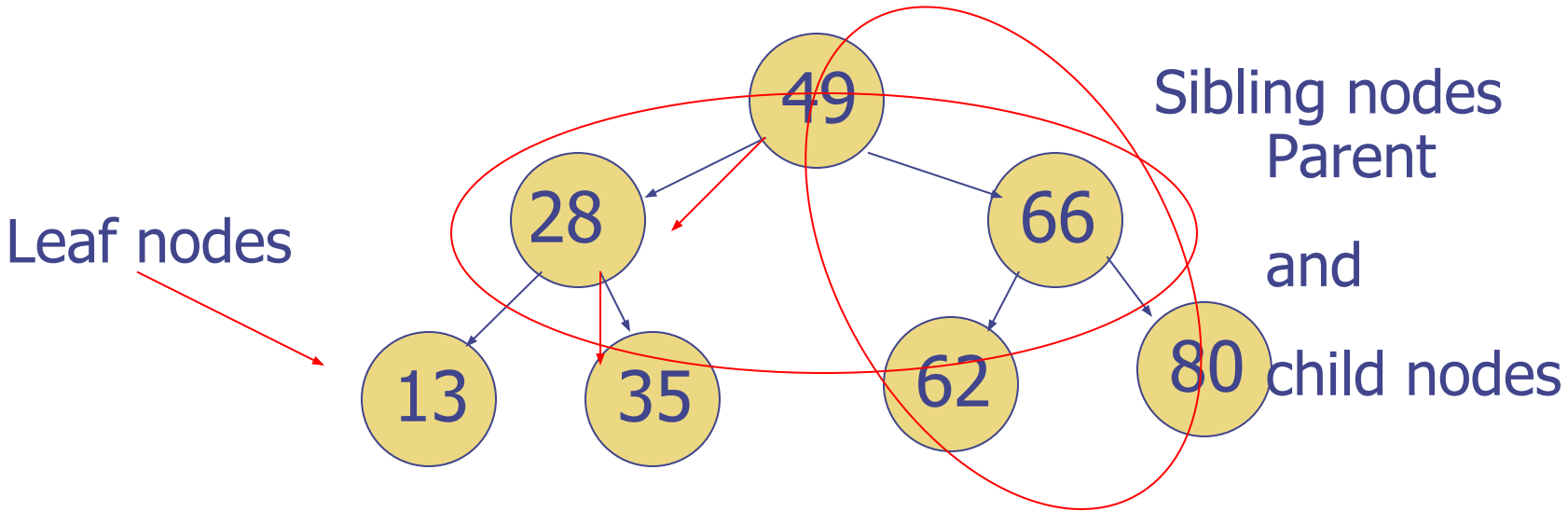


Tree terminology



Tree Terminology

- A tree consists of:
  - finite collection of nodes
  - non empty tree has a root node
  - root node has no incoming links
  - every other node in the tree can be reached from the root by unique sequence of links



Applications of Trees

- Genealogical tree
  - pictures a person's descendants and ancestors
- Game trees
  - shows configurations possible in a game such as the Towers of Hanoi problem
- Parse trees
  - used by compiler to check syntax and meaning of expressions such as 2 * (3 + 4)

Tree Terminology

- **Degree**: *The degree of a node is the number of subtrees of the node OR*
  Degree of a node is equal to the number of children that a node has.
- **Leaf** :The node with an outdegree 0 is a leaf or terminal node that is, a node with no successors..
- **Children** : It is the immediate successor of the node. Child, which is at the left side is called left child and the child which is at the right is called right OR A node with a predecessor is called a child.
- **Siblings** : Children of the same parent are siblings.
- **Internal node** : A node that is not a root or a leaf is known as an internal node.
- **Parent** : A node is a parent if it has successor nodes; that is, if it has outdegree greater than zero.

Applications of Trees

- Genealogical tree
  - pictures a person's descendants and ancestors
- Game trees
  - shows configurations possible in a game such as the Towers of Hanoi problem
- Parse trees
  - used by compiler to check syntax and meaning of expressions such as 2 * (3 + 4)

General Tree

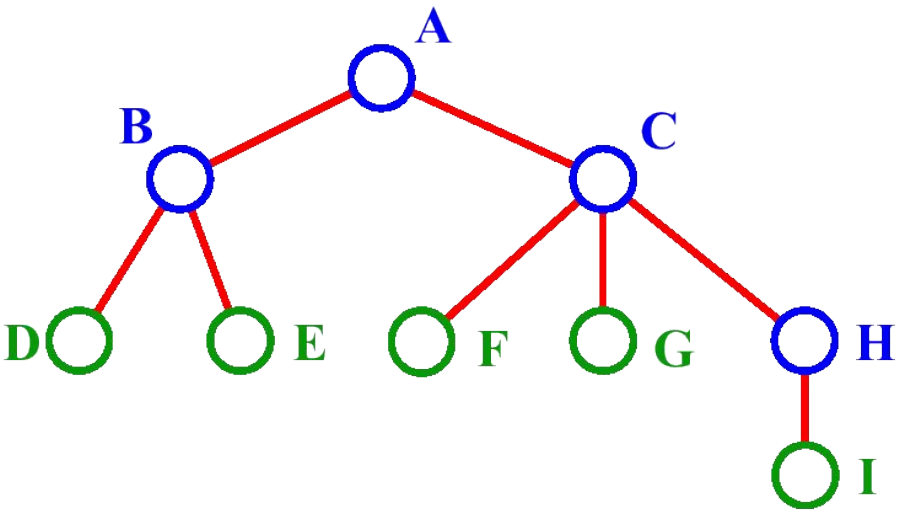
- Defnied as non empty finite set of elements called nodes such that
  - Tree contains the root element
  - The remaining elements of the tree form an ordered collection of zero or more disjoint trees, t_1, t_2,..... t_m

t_1, t_2,..... t_m subtrees of root.

The roots of t_1, t_2,..... t_m are called successors of the root.

Trees

- *A, B, C, H* are *internal nodes*
- The *depth (level)* of *E* is *2*
- The *height* of the tree is *3*
- The *degree* of node *B* is *2*



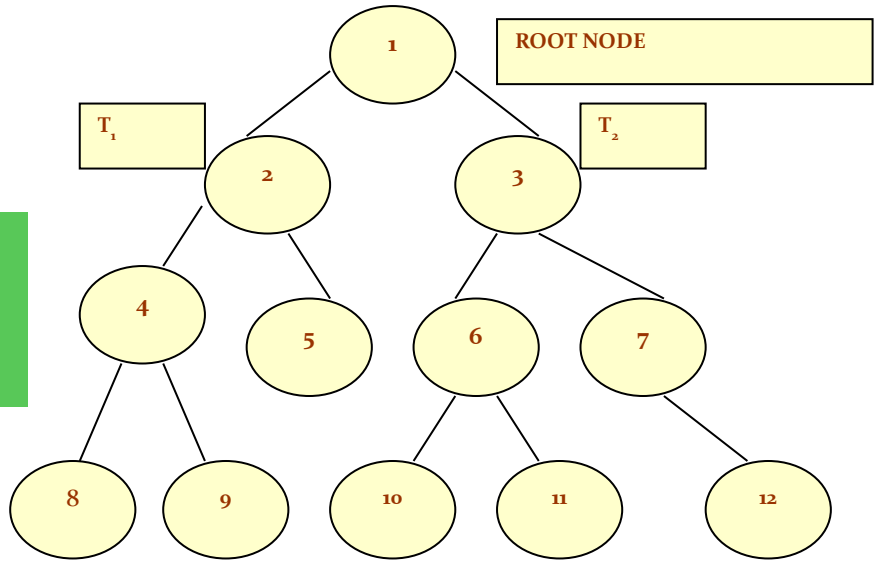
Types of Tree

- **Binary tree** : A binary tree is a data structure which is defined as a collection of elements called nodes.
  - In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
  - Every node contains a data element, a "left" pointer which points to the left child, and a "right" pointer which points to the right child.
  - The root element is pointed by a "root" pointer.
  - If root = NULL, then it means the tree is empty.
- **Level number**: Every node in the binary tree is assigned a level number. The root node is defined to be at level 0. The left and right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents.

R – Root node (node 1)

T_1- left sub-tree (nodes 2, 4, 5, 8, 9)

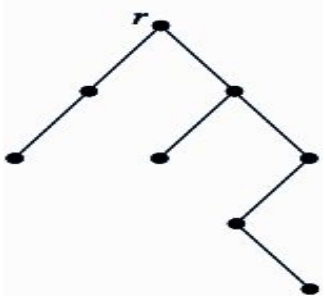
T_2- right sub-tree (nodes 3, 6, 7, 10, 11, 12)



Types of Tree

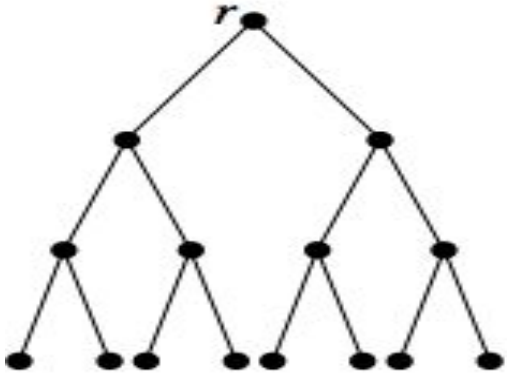
- **Binary Tree** : A binary tree is defined as a tree in which there is exactly one vertex of degree two, and each of the remaining vertices is of degree one or three.

  Since the vertex of degree two is distinct from all other vertices, this serves as a root. Thus every binary tree is a rooted tree.

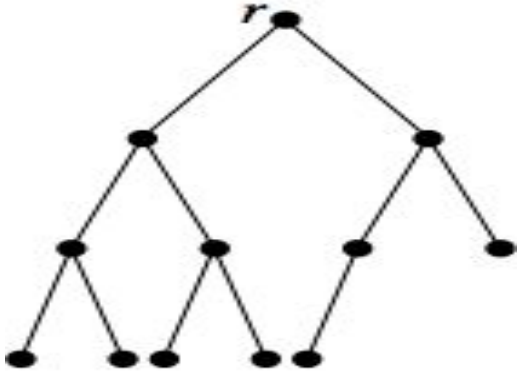


- **Full Binary Tree** : A full binary tree (as seen in the middle figure below) occurs when all internal nodes have two children and all leaves are at the same depth.

  A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



- **Complete Binary Tree** : A complete binary tree (as seen in the right figure below) is an almost-full binary tree; the bottom level of the tree is filling from left to right but may not have its full complement of leaves.



Types of Tree

- **Full Binary Tree** : Fig a
- **Perfect Binary Tree :** A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level. Fig b

Fig a

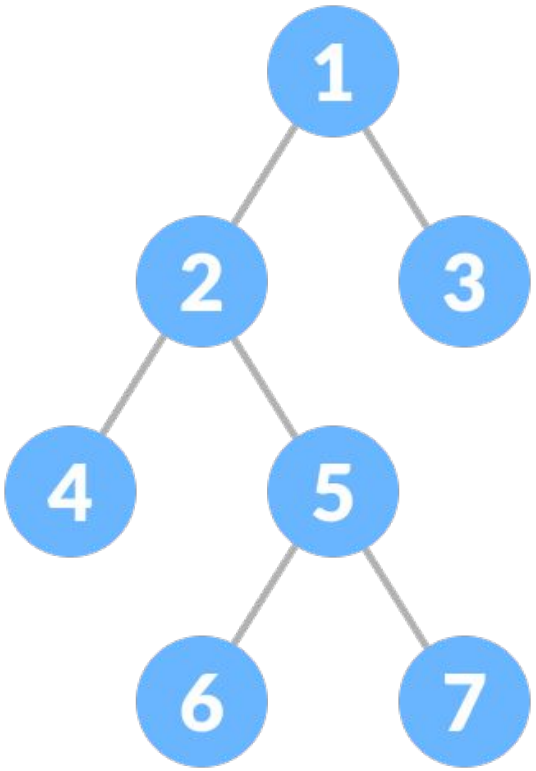
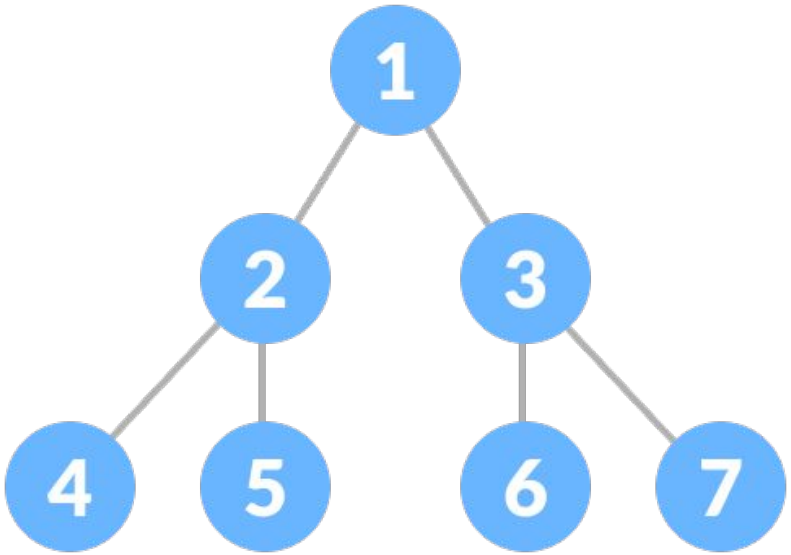
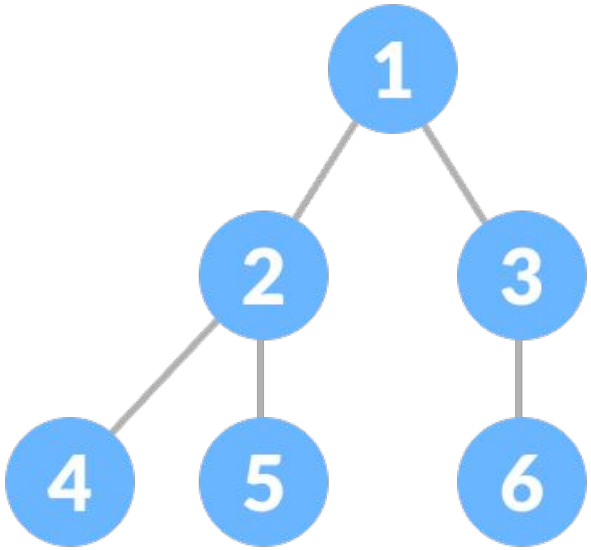


Fig b



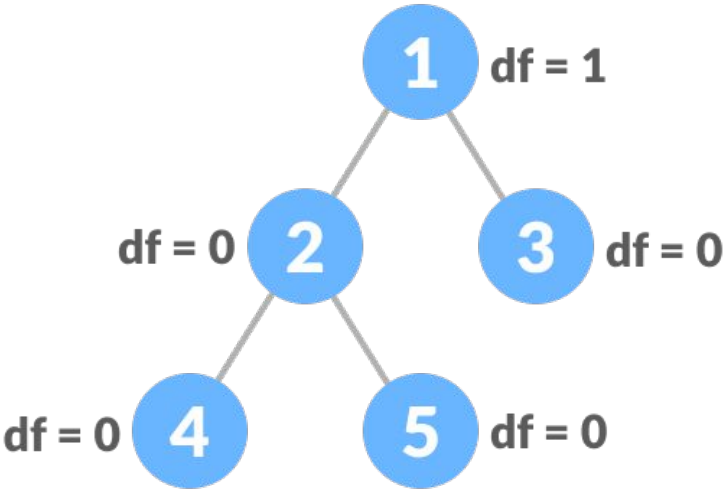
Types of Tree

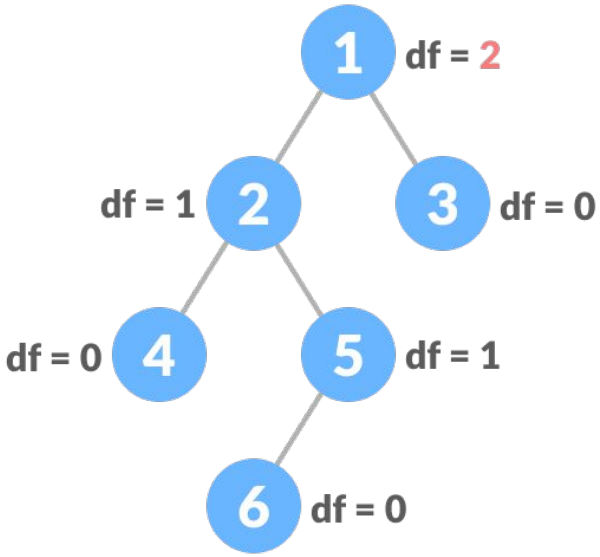
- **Complete Binary Tree :** A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.
  - A complete binary tree is just like a full binary tree, but with two major differences
  - All the leaf elements must lean towards the left.
  - The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Types of Tree

- **Balanced Binary Tree** : A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.
- Following are the conditions for a height-balanced binary tree:
  - difference between the left and the right subtree for any node is not more than one
  - the left subtree is balanced
  - the right subtree is balanced





df = |height of left child - height of right child|

Un balance binary tree

Applications of Binary Tree

- Searching algorithm
- arithmetic expressions
- decision processes
- Populating voluminous, relational, hierarchical data into memory
- Expression evaluation
- AI

Binary Tree Creation: Recursive Algorithm

Create_Tree (Info, Node)

Step 1: [Check whether the tree is empty]

If Node = NULL

Node = Create a node

Left_child[Node] = NULL

Right_child[Node] = NULL

Step 2 : [Test for the left child]

If Info[Node] >= Info

Left_child[Node] = Call Create_Tree (Info, Left_Child[Node])

Else

Right_child[Node] = Call Create_Tree (Info, Right_Child[Node])

Step 3:

Return (Node)

Example: Expression Trees

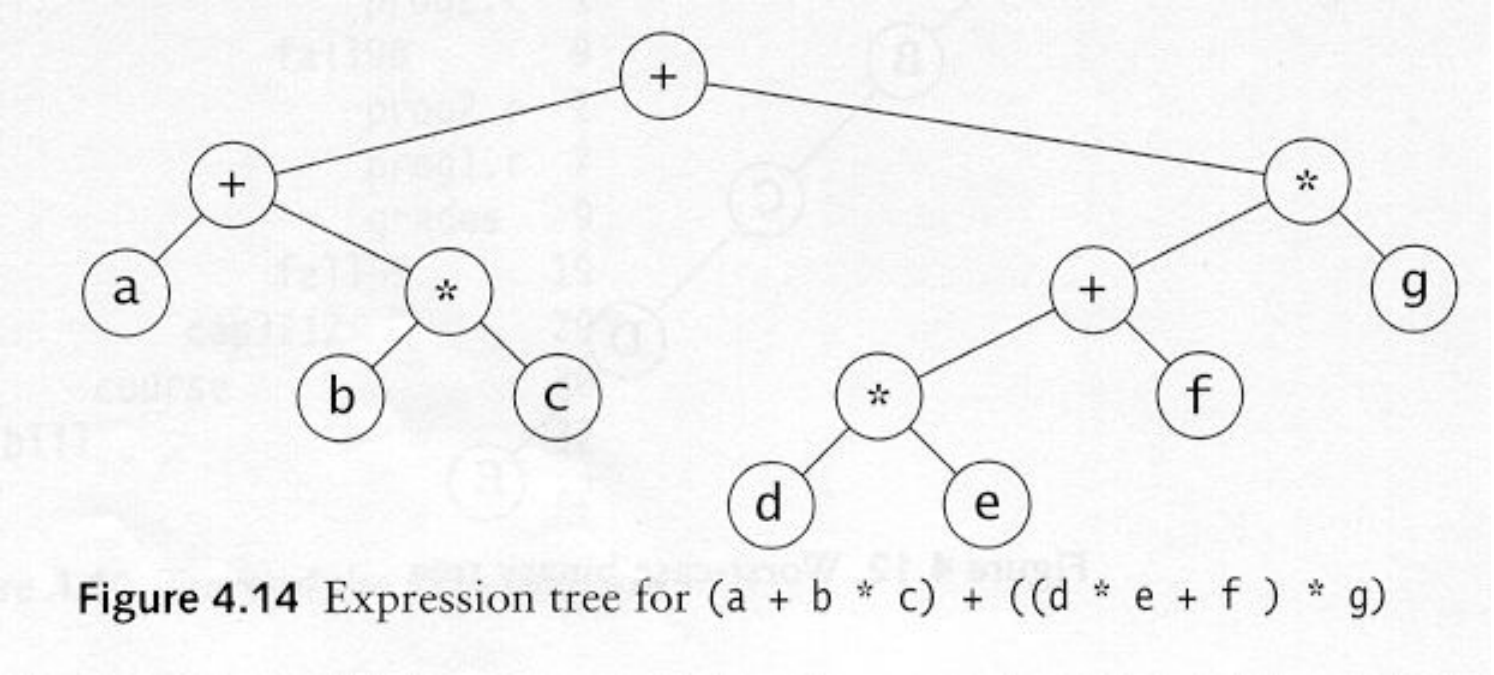
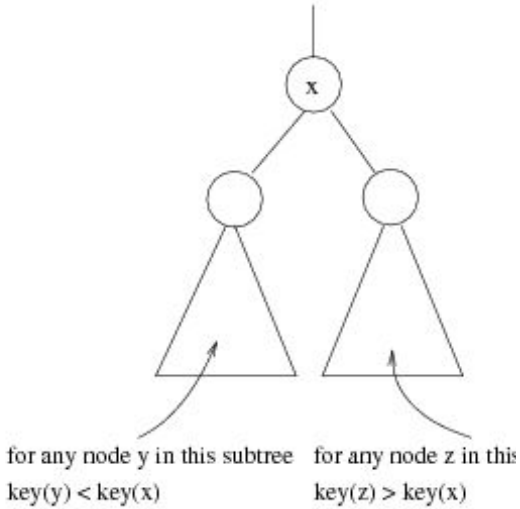


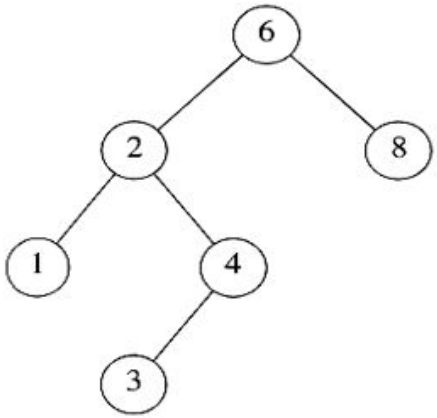
Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary

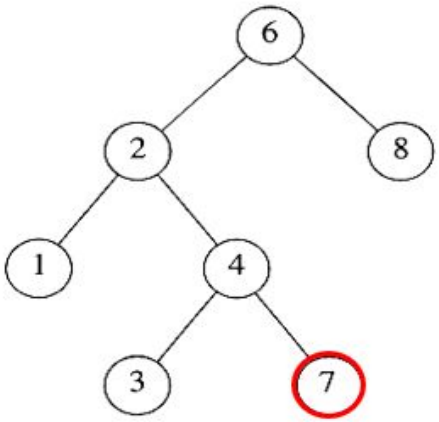
Types of Tree

- **Binary Search Tree** : Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
- **Binary search tree property** : For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X





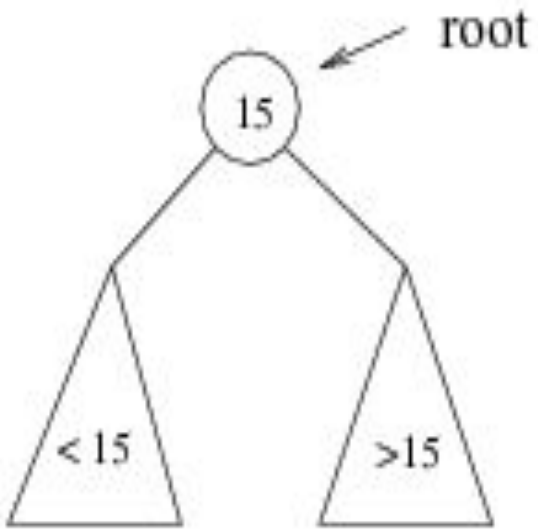
A binary search tree



Not a binary search tree

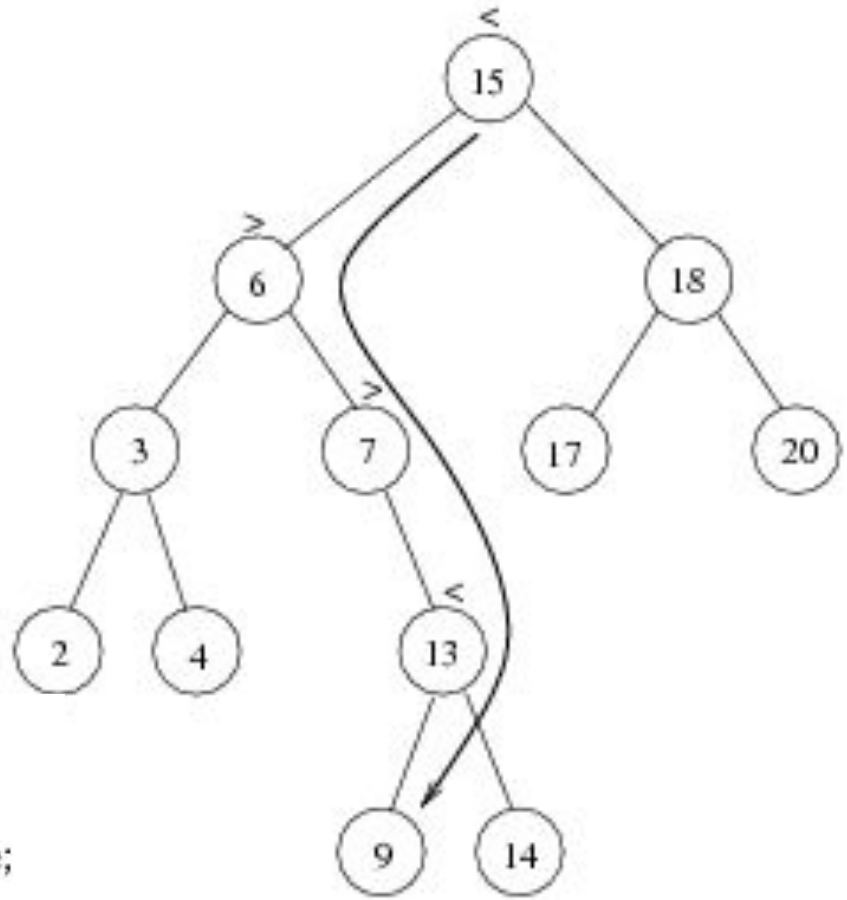
Searching Binary Tree

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Searching Binary Tree

Example: Search for 9 ...

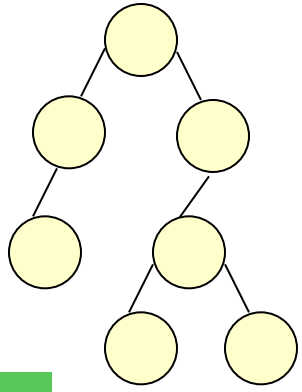


Search for 9:

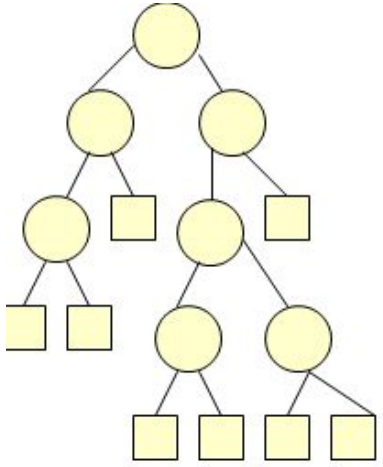
1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Types of Tree

- **Extended binary tree :** A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children.
- In an extended binary tree nodes that have two children are called internal nodes and nodes that have no child or zero children are called external nodes. In the figure internal nodes are represented using a circle and external nodes are represented using squares.
- To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes and the new nodes added are called the external nodes.



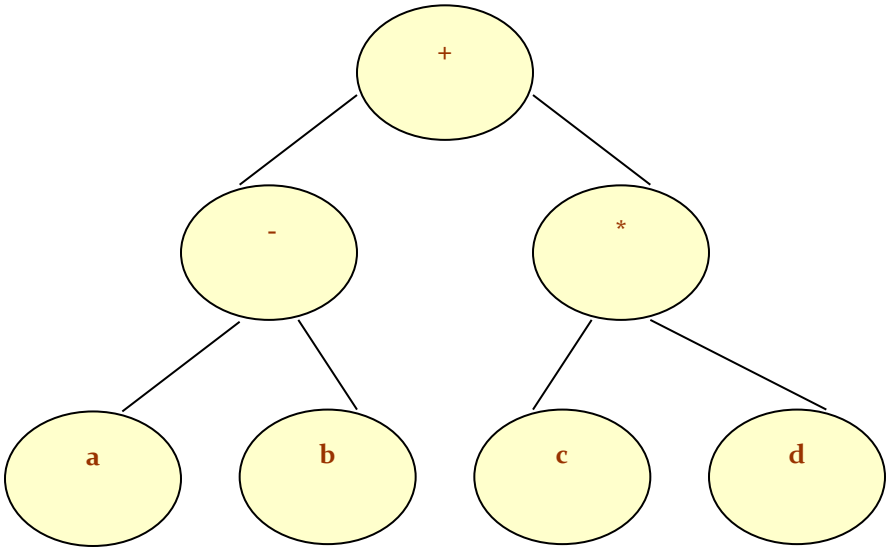
Binary tree



Extended binary tree

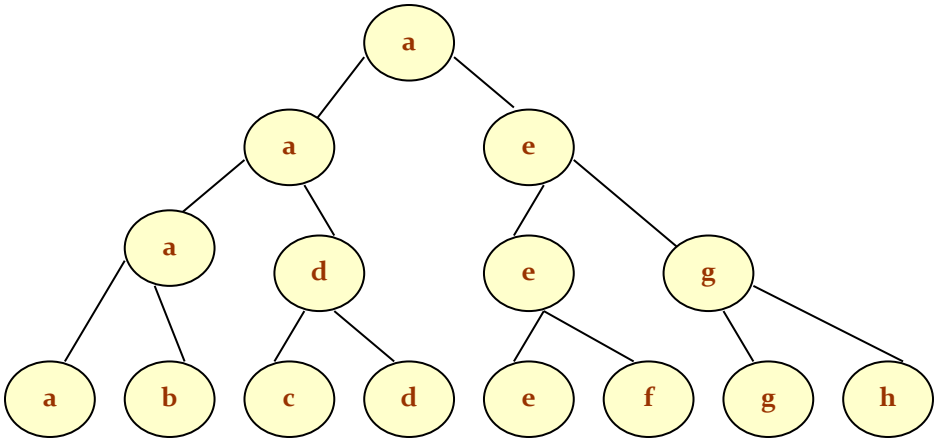
Types of Tree

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



Types of Tree

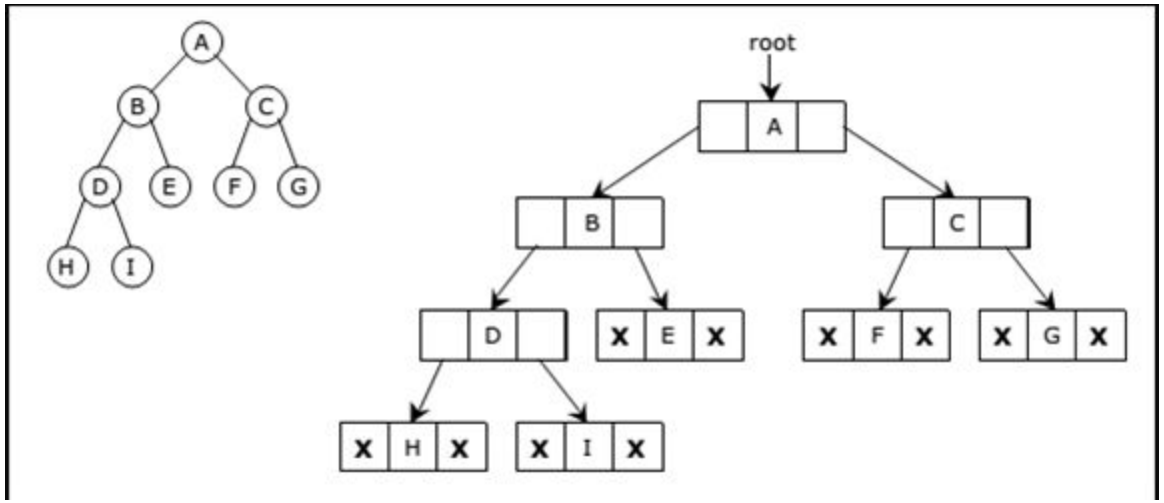
- **Tournament Tree**: In a tournament tree (also called a selection tree), each external node represents a player and each internal node represents the winner of the match played between the players represented by its children nodes.
- These tournament trees are also called winner trees because they are being used to record the winner at each level.
- We can also have a loser tree that records the loser at each level.



Representation of Binary Tree

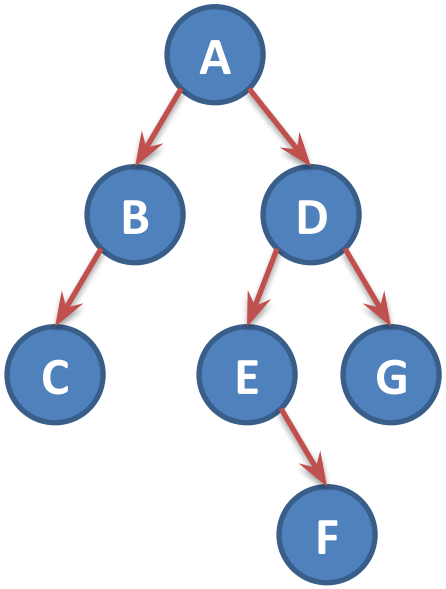
- In computer's memory, a binary tree can be maintained either using a
  - linked representation
  - sequential representation.
- **Linked Representation** : In linked representation of binary tree, every node will have three parts: the data element, a pointer to the left node and a pointer to the right node.
- So in C, the binary tree is built with a node type given as below.

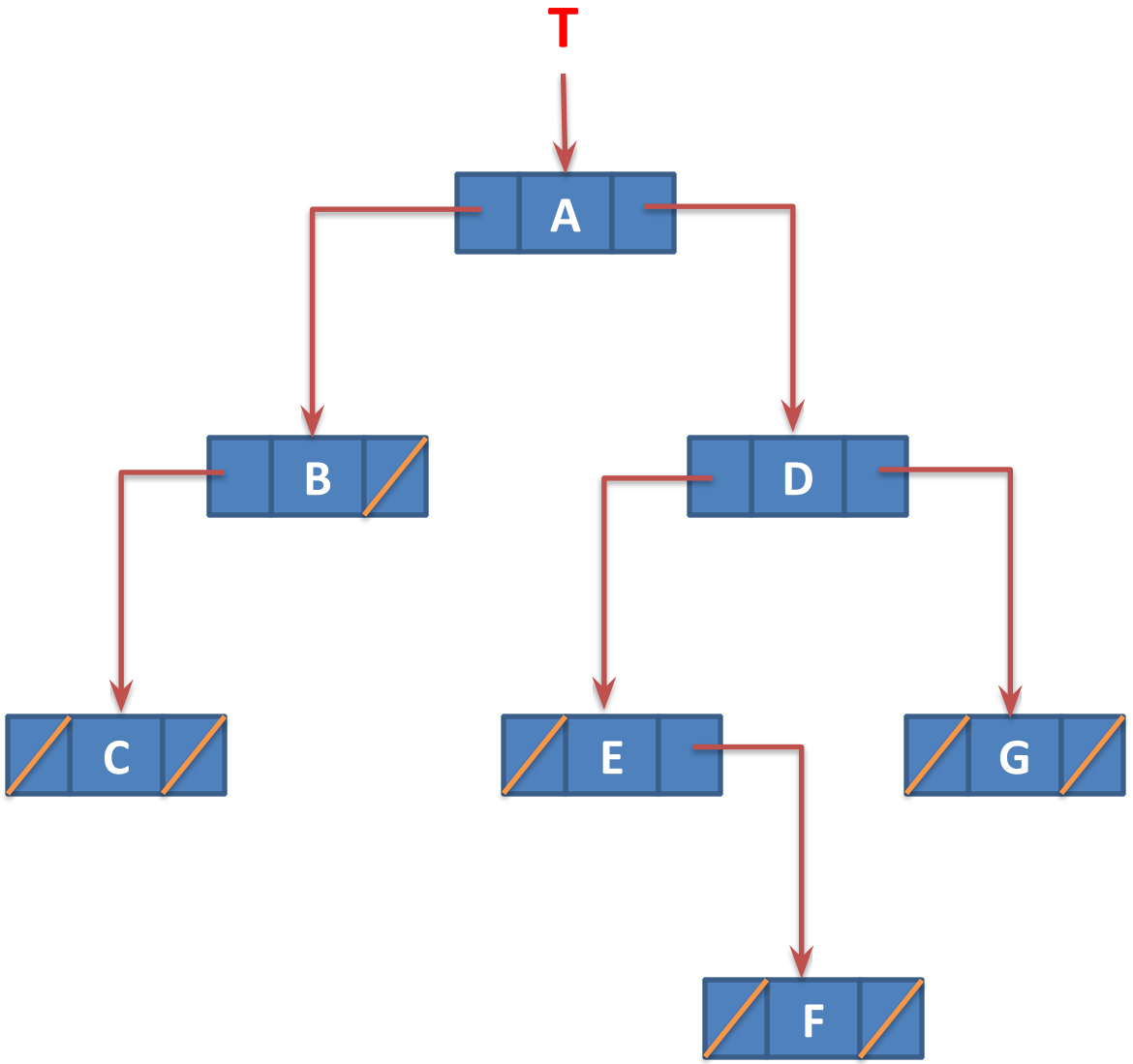
```
struct node
{
   struct node* left;
   int data;
   struct node* right;
};
```



Linked Representation of Binary Tree

Typical node of Binary Tree



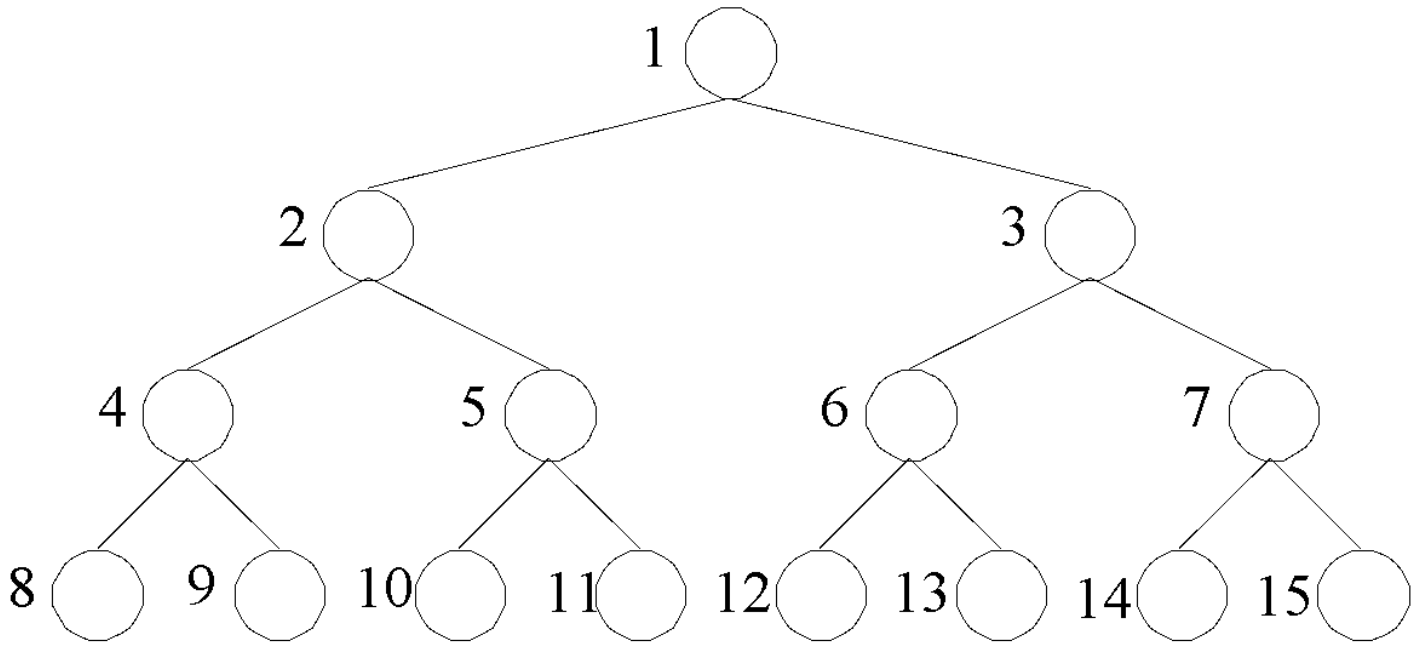


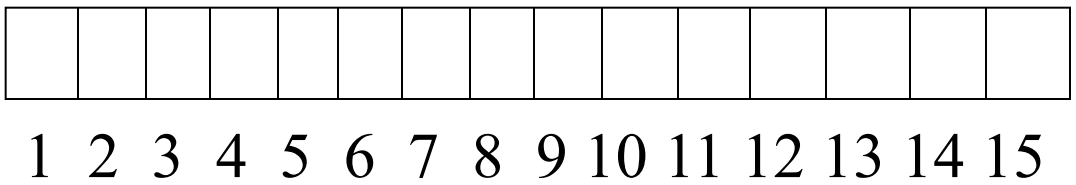
Representation of Binary Tree

- **Sequential representation** : Sequential representation of trees is done using a single or one dimensional array. Though, it is the simplest technique for memory representation, it is very inefficient as it requires a lot of memory space.
- A sequential binary tree follows the rules given below:
- One dimensional array called TREE is used.
- The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- The children of a node K will be stored in location (2*K) and (2*K+1).
- The maximum size of the array TREE is given as (2^h-1), where h is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

Sequential Representation of Binary Tree

Since a binary tree is an ordered tree and has levels, it is convenient to assign a number to each node.

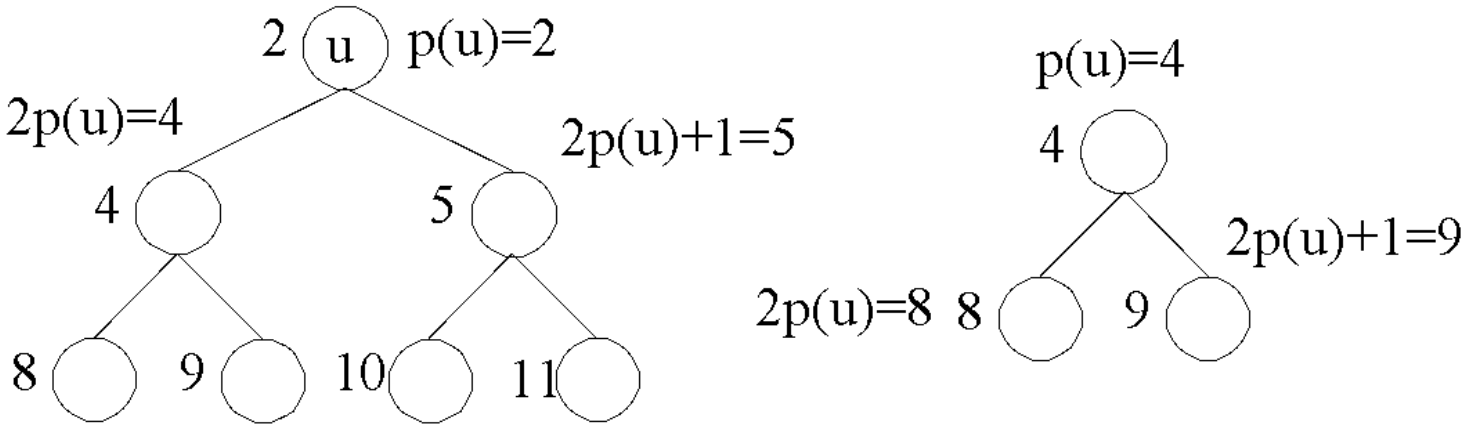




Sequential Representation of Binary Tree

Formally, for every node v of a tree T, let $p(v)$ be the integer defined as follows.

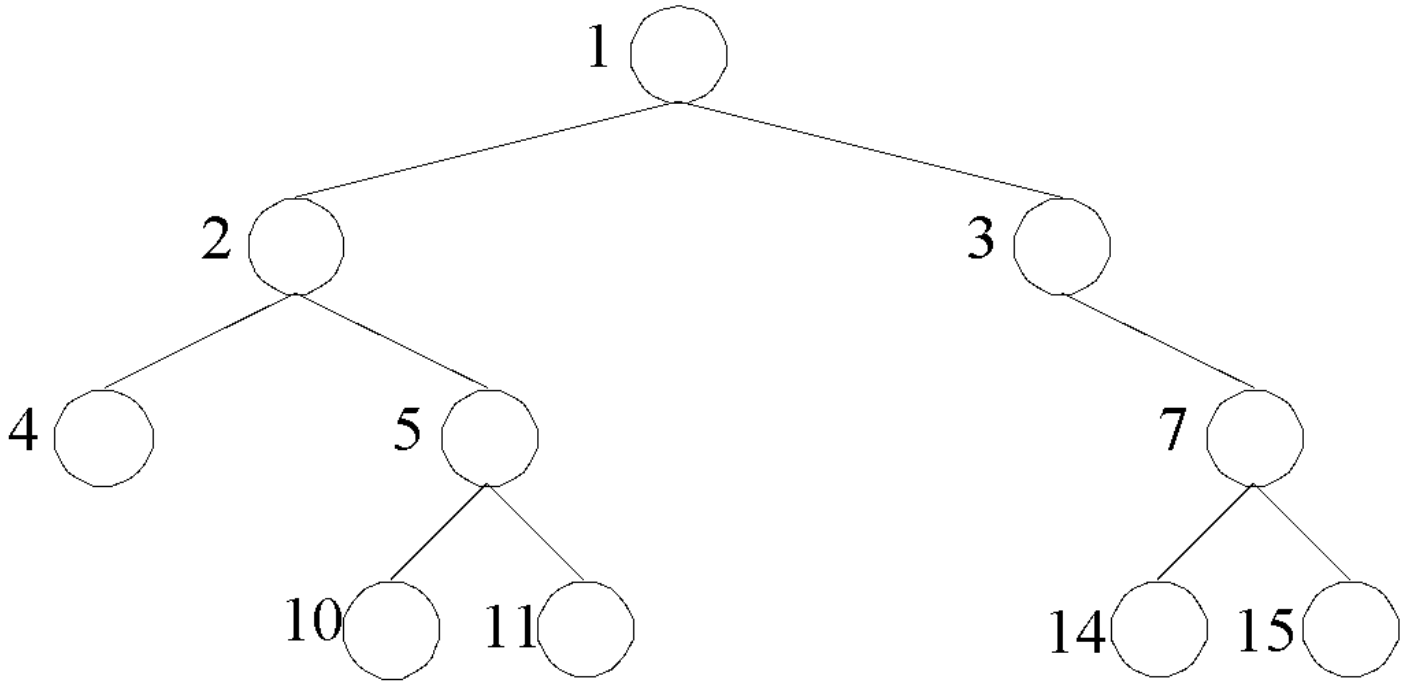
- If v is the root of T, then $p(v) = 1$.
- If v is the left child of the node u, then $p(v) = 2p(u)$
- If v is the right child of the node u, then $p(v) = 2p(u) + 1$.



The function $p()$ is called a level numbering of the nodes in a binary tree T.

Sequential Representation of Binary Tree

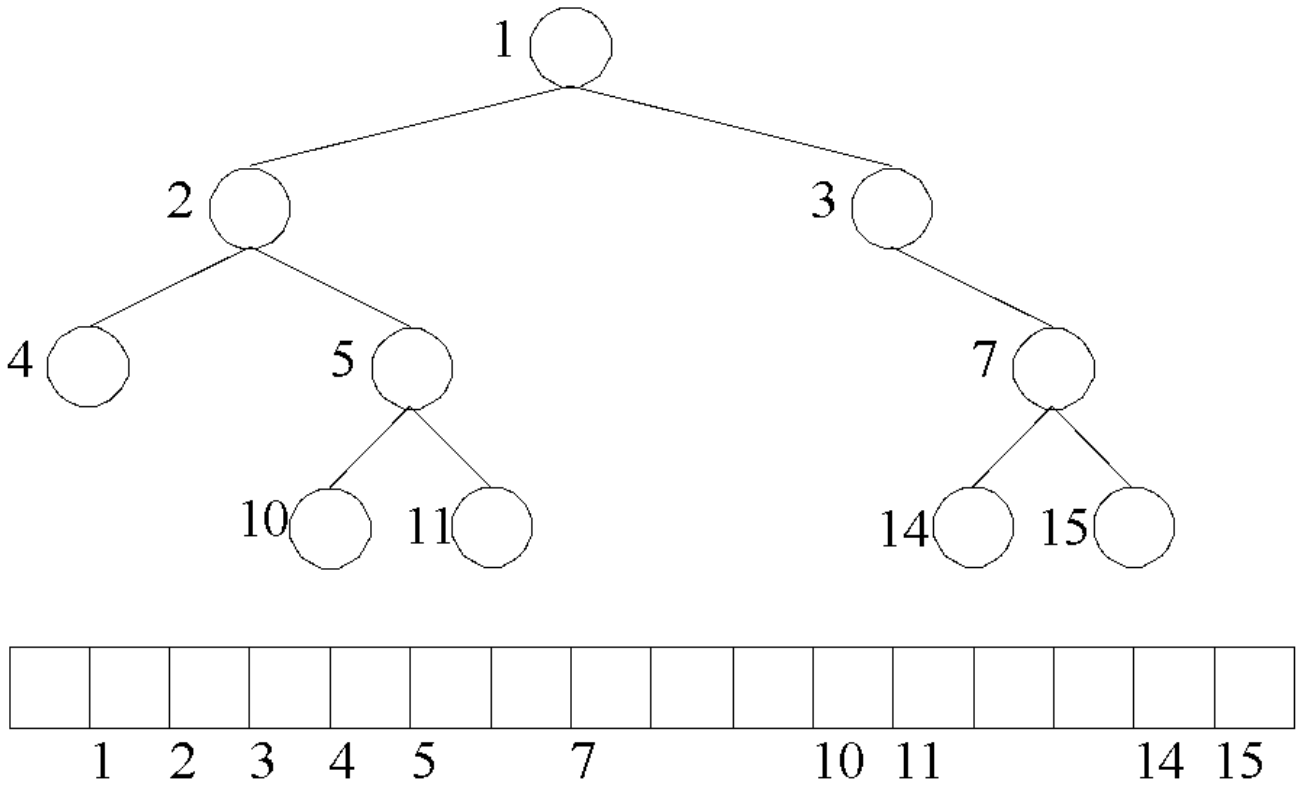
In general, the numbers for nodes may not be consecutive when the number of nodes at some level does not reach its maximum.



Sequential Representation of Binary Tree

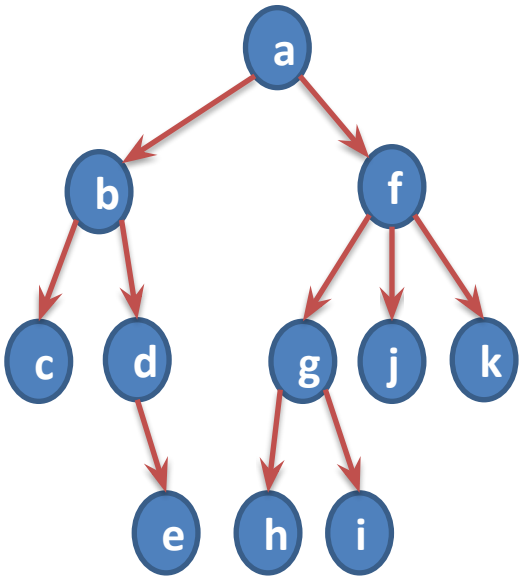
With a level numbering of the nodes, it is convenient to use a vector S to represent a tree.

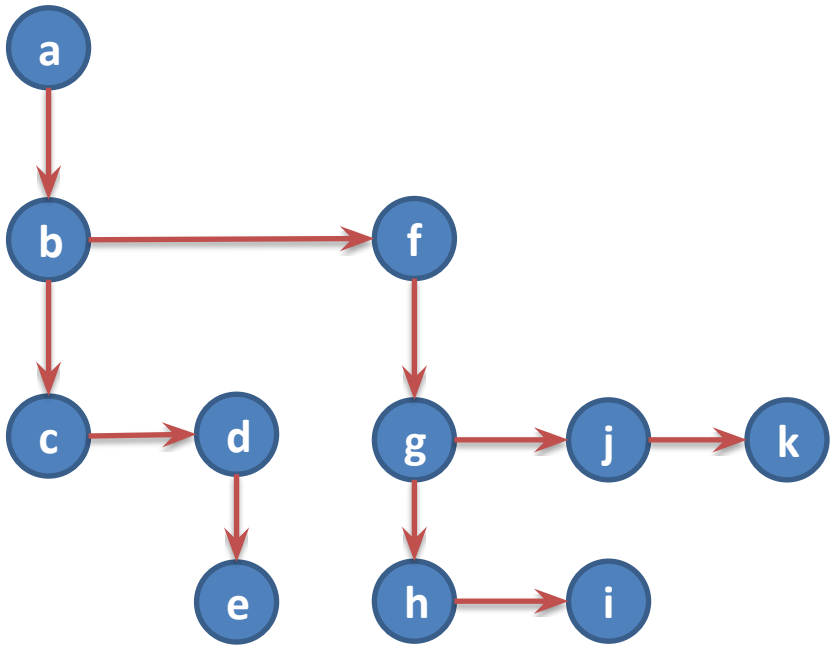
Recall that we use ranks to access elements in a vector. So we can place the node v in a tree T at rank $p(v)$.

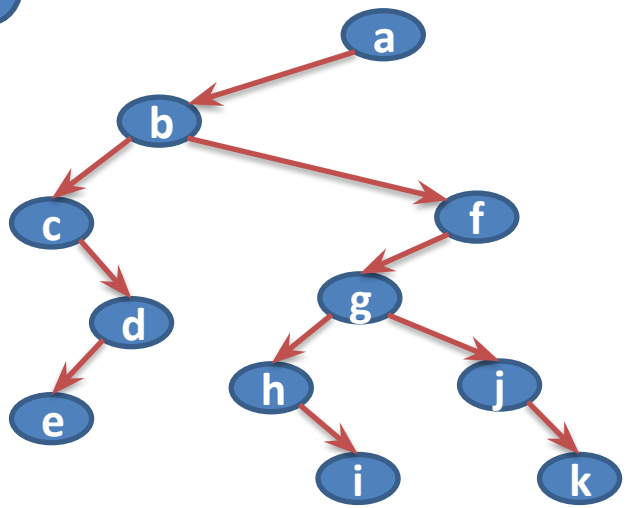


Convert any tree to binary tree

- Every Tree can be Uniquely represented by binary tree
- Let's have an example to convert given tree into binary tree

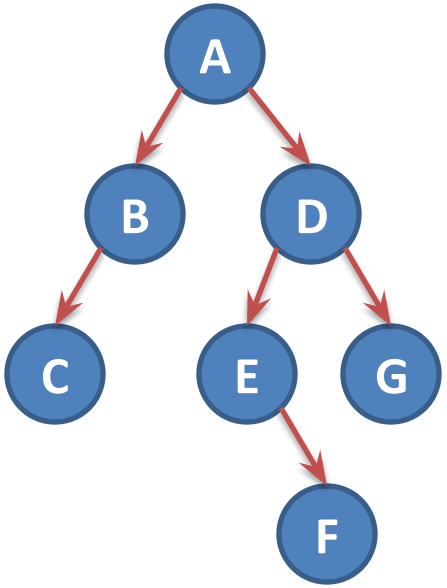






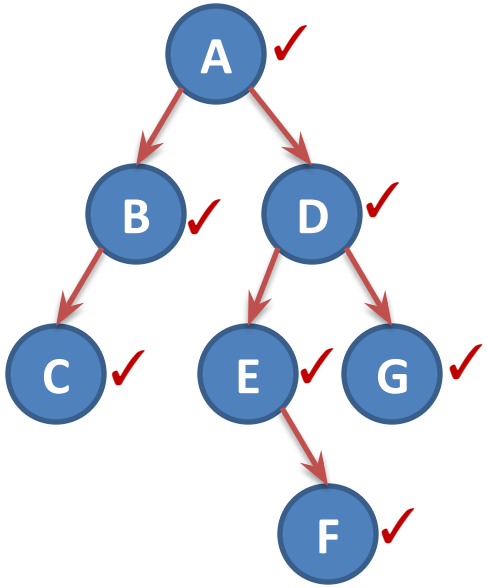
Tree Traversal

- The most common operations performed on tree structure is that of traversal.
- **This is a procedure by which each node in the tree is processed exactly once in a systematic manner.**
- There are three ways of traversing a binary tree.
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal



Preorder Traversal [NLR]

- Preorder traversal of a binary tree is defined as follow
  1. Process the **root** node
  2. **Traverse** the **left subtree** in preorder
  3. **Traverse** the **right subtree** in preorder
- If particular subtree is empty (i.e., node has no left or right descendant) the traversal is performed by doing nothing
- In other words, a null subtree is considered to be fully traversed when it is encountered

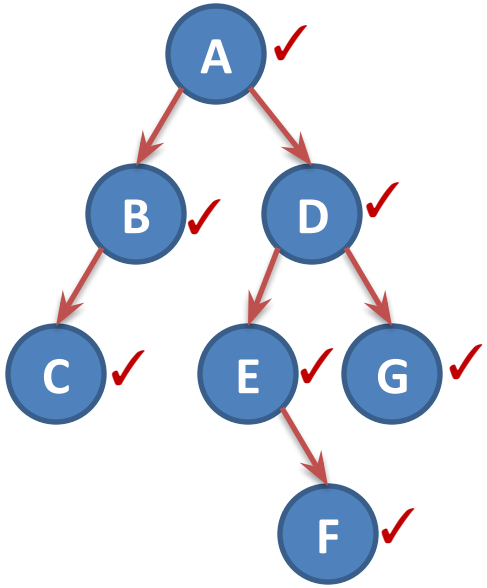


Preorder traversal of a given tree as

A B C D E F G

Inorder Traversal [LNR]

- Inorder traversal of a binary tree is defined as follow
  1. **Traverse** the **left subtree** in Inorder
  2. Process the **root** node
  3. **Traverse** the **right subtree** in Inorder

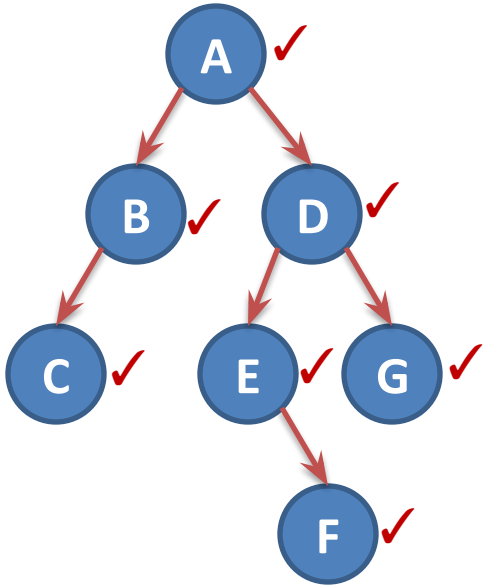


Inorder traversal of a given tree as

C B A E F D G

Postorder Traversal [LRN]

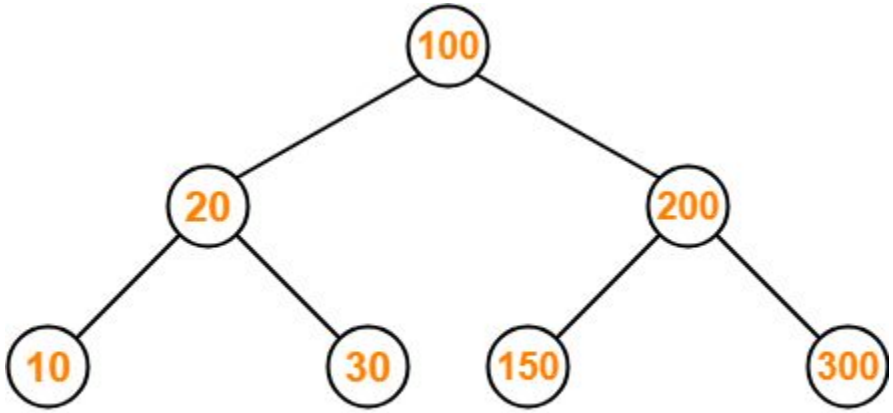
- Postorder traversal of a binary tree is defined as follow
  1. **Traverse** the **left subtree** in Postorder
  2. **Traverse** the **right subtree** in Postorder
  3. Process the **root** node



Postorder traversal of a given tree as

C B F E G D A

Write Preorder Inorder and Postorder



Binary Search Tree

- Preorder Traversal-

  100 , 20 , 10 , 30 , 200 , 150 , 300
- Inorder Traversal-

  10 , 20 , 30 , 100 , 150 , 200 , 300
- Postorder Traversal-

  10 , 30 , 20 , 150 , 300 , 200 , 100

Algorithm of Binary Tree Traversal

- Preorder Traversal - Procedure: RPREORDER(T)
- Inorder Traversal - Procedure: RINORDER(T)
- Postorder Traversal - Procedure: RPOSTORDER(T)

Procedure: RPREORDER(T) [NLR]

- This procedure **traverses the tree** in **preorder**, in a recursive manner.
- T is root node address of given binary tree
- Node structure of binary tree is described as below

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write ('Empty Tree')
          return
   ELSE   write (DATA(T))
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RPREORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RPREORDER (RPTR (T))
4. [Finished]
   Return
```

Typical node of Binary Tree

Procedure: RINORDER(T) [LNR]

- This procedure **traverses the tree** in **InOrder**, in a recursive manner.
- **T is root node address** of given binary tree
- Node structure of binary tree is described as below

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write ('Empty Tree')
          return
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RINORDER (LPTR (T))
3. [Process the Root Node]
   write (DATA(T))
4. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RINORDER (RPTR (T))
5. [Finished]
   Return
```

Typical node of Binary Tree

Procedure: RPOSTORDER(T) [LRN]

- This procedure **traverses the tree** in **PostOrder**, in a recursive manner.
- **T is root node address** of given binary tree
- Node structure of binary tree is described as below

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write ('Empty Tree')
          return
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RPOSTORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN RPOSTORDER (RPTR (T))
4. [Process the Root Node]
   write (DATA(T))
5. [Finished]
   Return
```

Typical node of Binary Tree

Program for tree traversal

```
#include <stdio.h>
#include <stdlib.h>

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}
```

Program for tree traversal

```
// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
```

Program for tree traversal

```
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}
```

Program for tree traversal

```
// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}


int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);
  insertLeft(root->left, 5);
  insertRight(root->left, 6);
  printf("Inorder traversal \n");
  inorderTraversal(root);
  printf("\nPreorder traversal \n");
  preorderTraversal(root);
  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

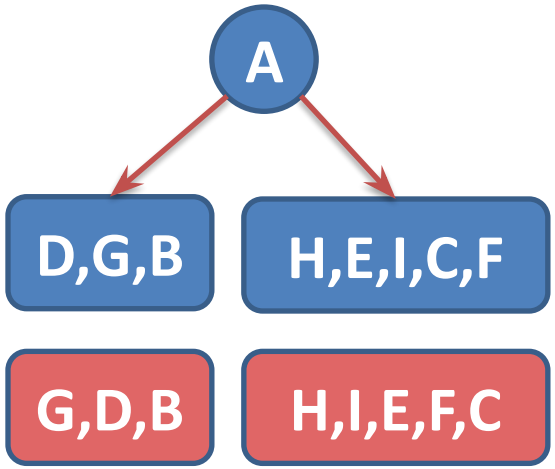
Construct Binary tree from traversal

Construct a Binary tree from the given **Inorder** and **Postorder** traversals

Inorder : D G B **A** H E I C F
Postorder : G D B H I E F C **A**
	• Step 1: Find the root node
	  • Preoder Traversal – first node is root node
	  • Postoder Traversal last node is root node
	• Step 2: Find Left & Right Sub Tree
	  • Inorder traversal gives Left and right sub tree

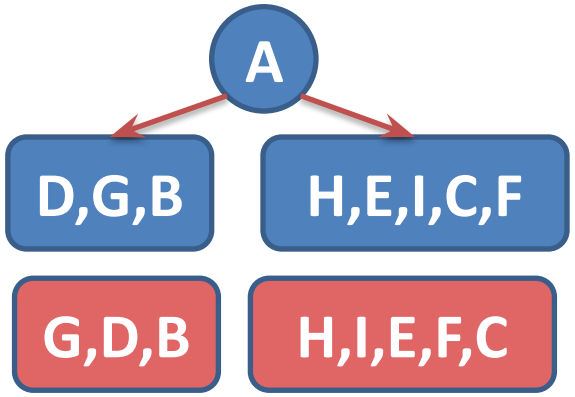
Postorder : G D B H I E F C **A**

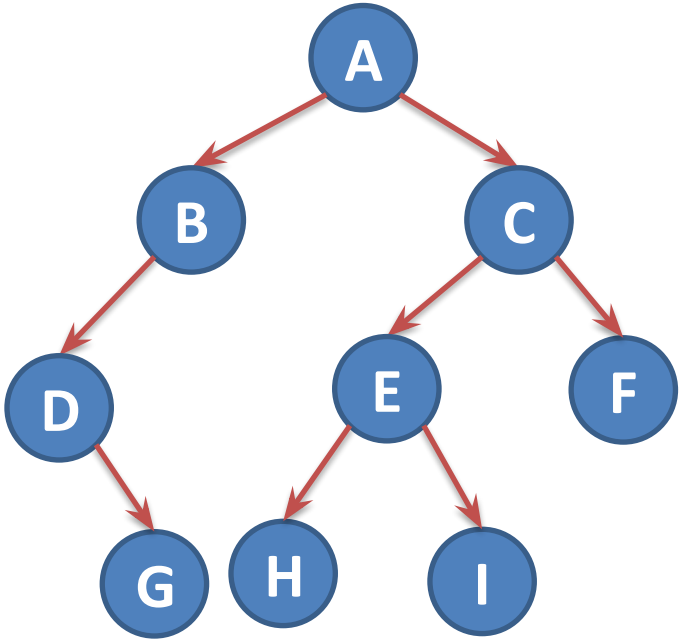
Inorder : D G B **A** H E I C F

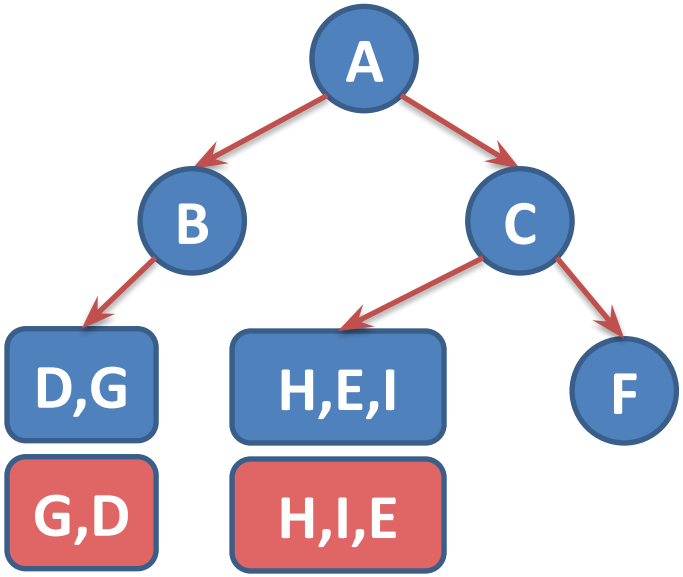


Construct Binary tree from traversal

Postorder : G D B H I E F C A | **Inorder :** D G B A H E I C F



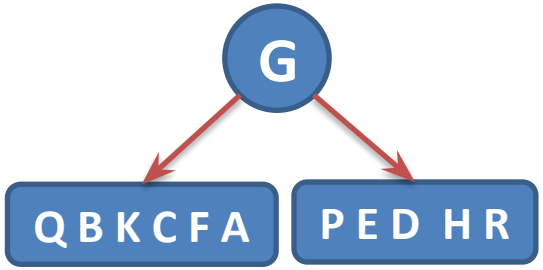


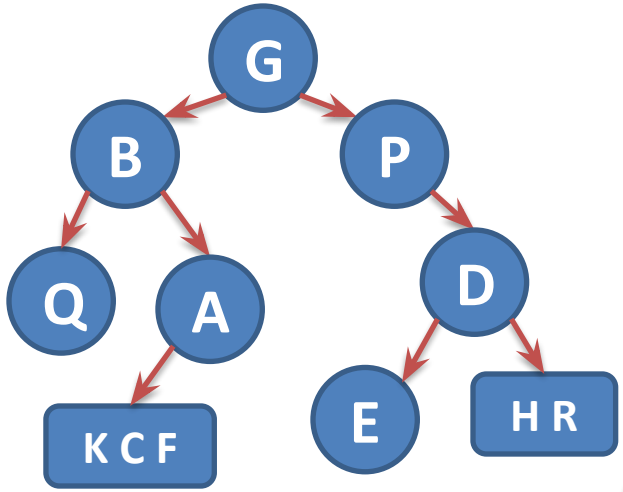


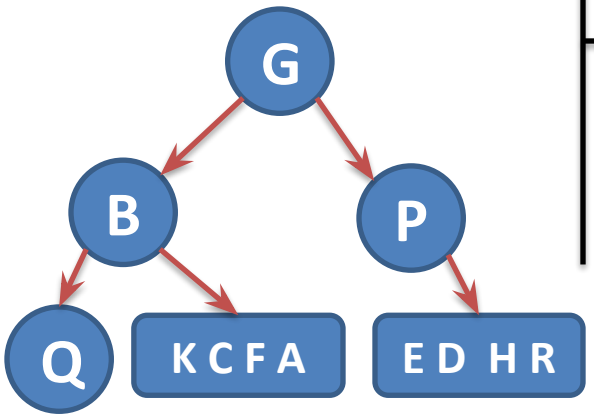
Construct Binary tree from traversal

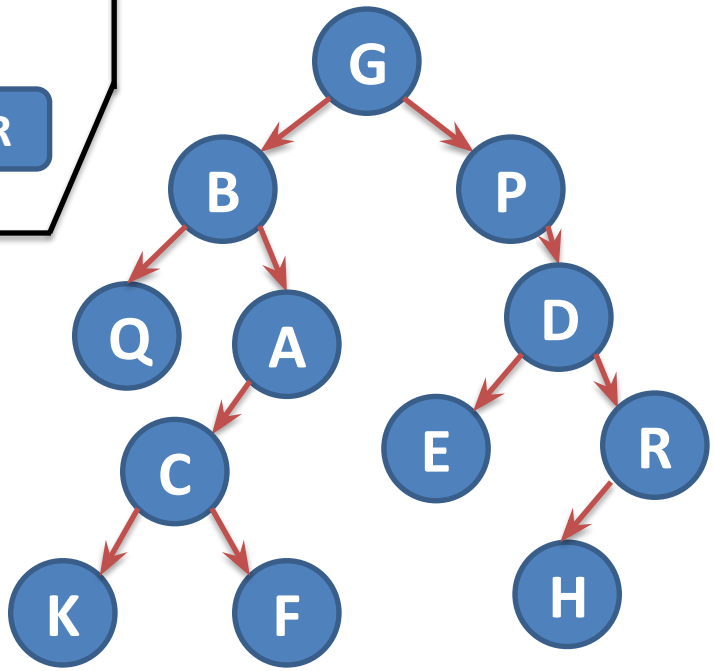
Preorder : G B Q A C K F P D E R H

Inorder : Q B K C F A G P E D H R







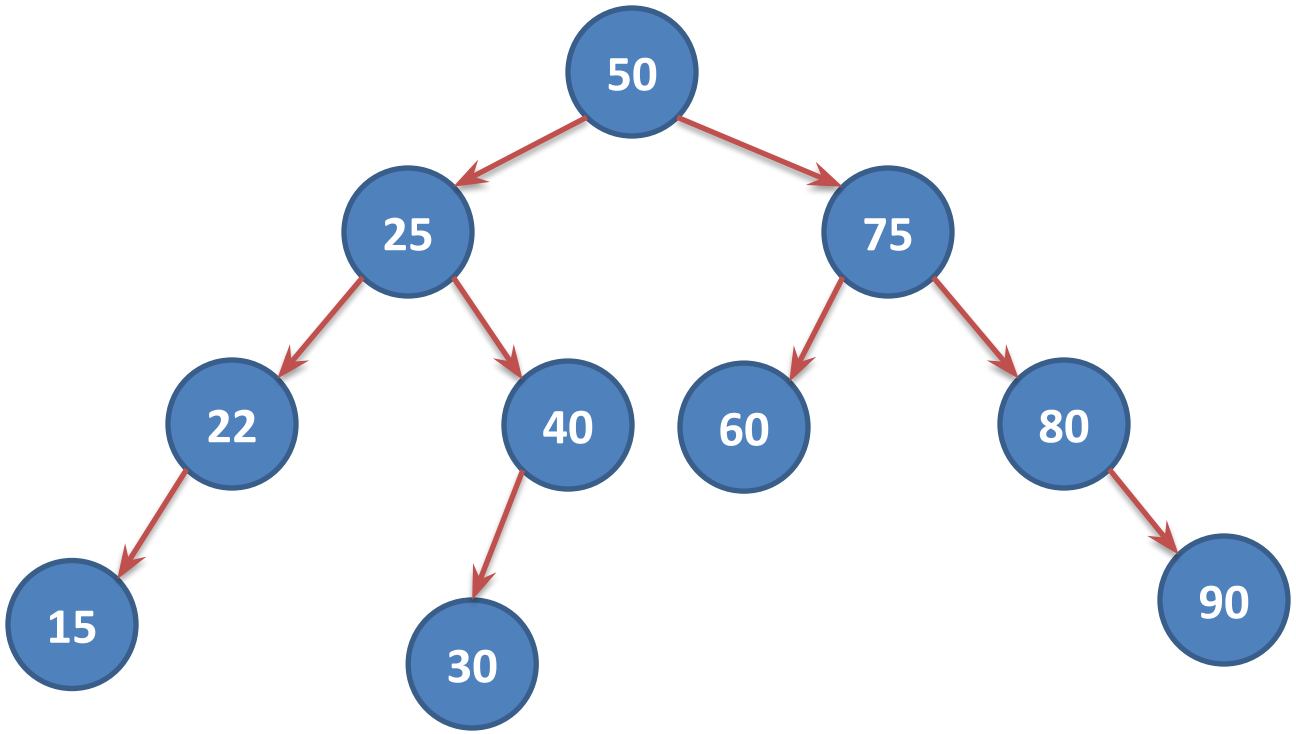


Binary Search Tree (BST)

- A **binary search tree** is a **binary tree** in which **each node** possessed a key that satisfy the following conditions
  1. All **key** (if any) in the **left sub tree** of the root **precedes the key** in the root
  2. The **key in the root precedes** all **key** (if any) in the **right sub tree**
  3. The **left and right sub trees** of the root are again **search trees**

Construct Binary Search Tree (BST)

Construct binary search tree for the following data
50 , 25 , 75 , 22 , 40 , 60 , 80 , 90 , 15 , 30



Construct binary search tree for the following data
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5

Search a node in Binary Search Tree (BST)

- To search for target value
- we first compare it with the key at root of the tree
- If it is not same, we go to either Left sub tree or Right sub tree as appropriate and repeat the search in sub tree.
- If we have **In-Order List** & we want to search for specific node it requires **O(n)** time
- In case of **Binary tree** it requires **$O(Log_2 n)$** time to search a node.

Algorithm to search a value in Binary Search Tree (BST)

- SearchElement (TREE, VAL)

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

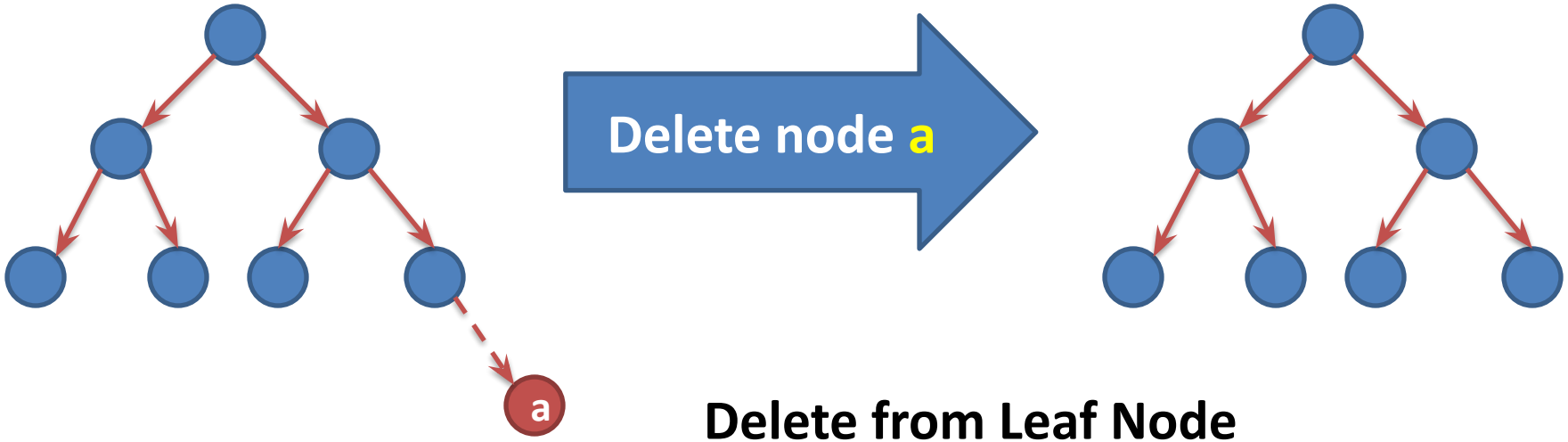
2. Step 2: End

Algorithm to insert a value in Binary Search Tree (BST)

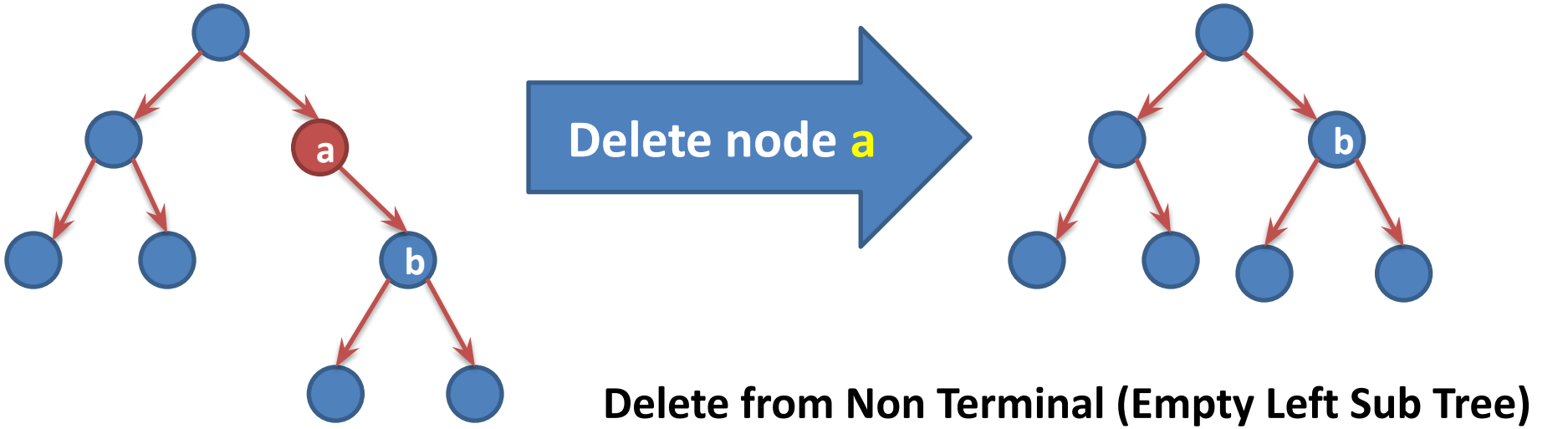
- Insert (TREE, VAL)

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

Delete a node in Binary Search Tree (BST)

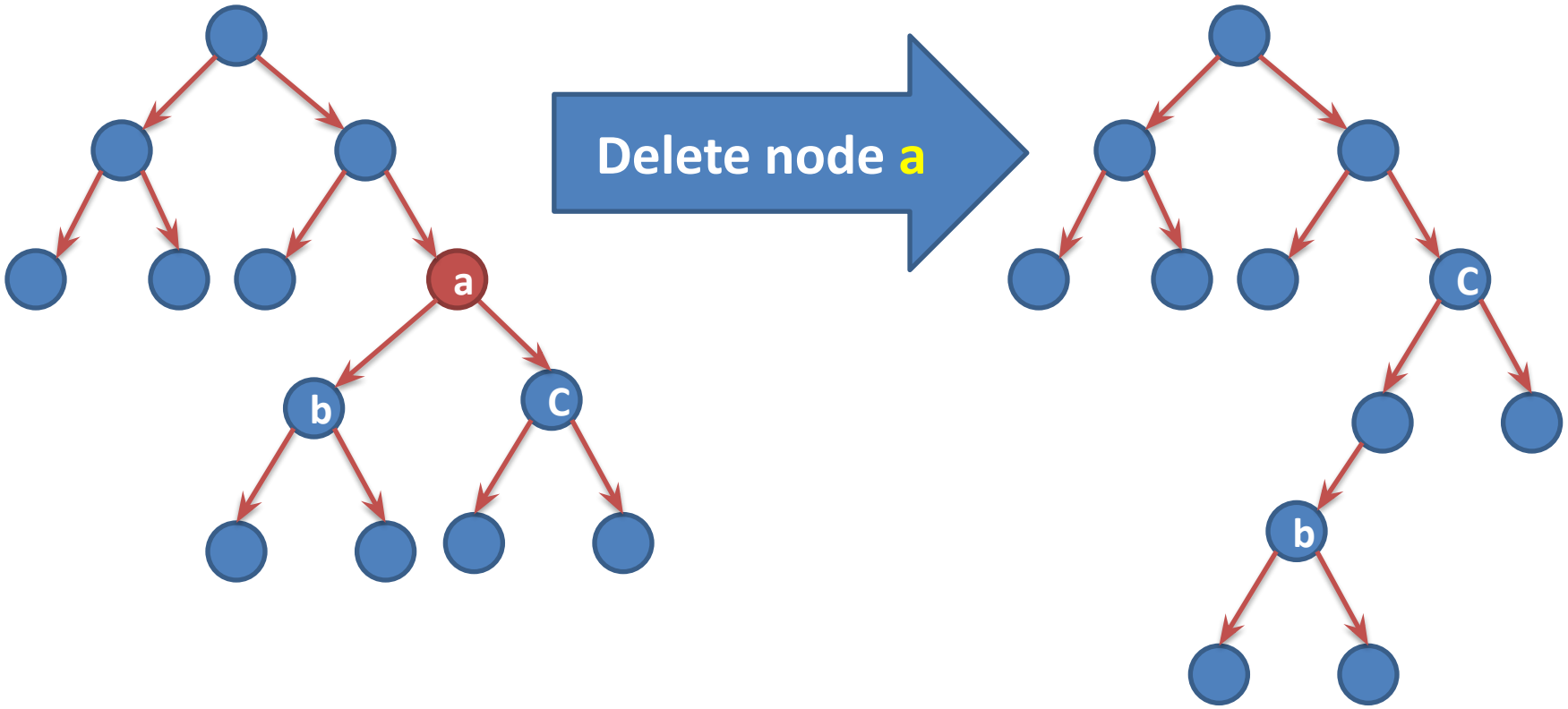


Delete from Leaf Node



Delete from Non Terminal (Empty Left Sub Tree)

Delete a node in Binary Search Tree (BST)



Delete from Non Terminal (Neither Sub Tree is Empty)

Algorithm to delete a value in Binary Search Tree (BST)

- Delete (TREE, VAL)

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

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass

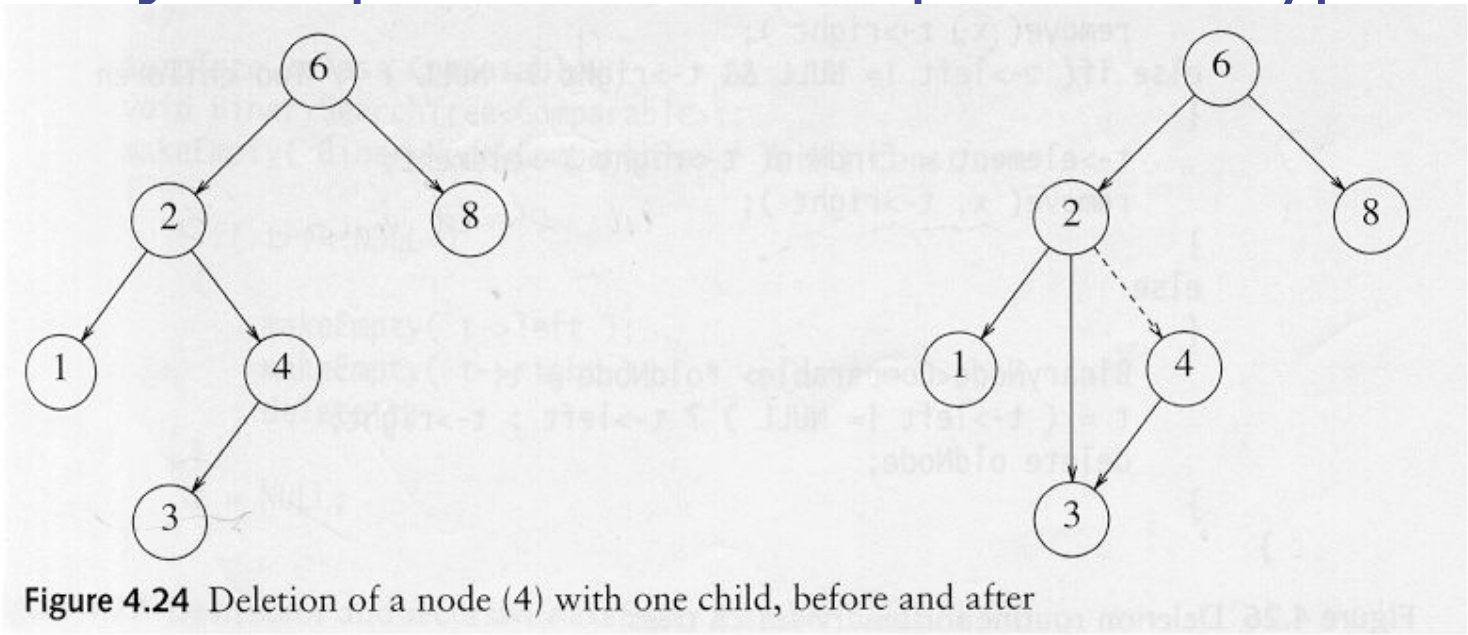


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

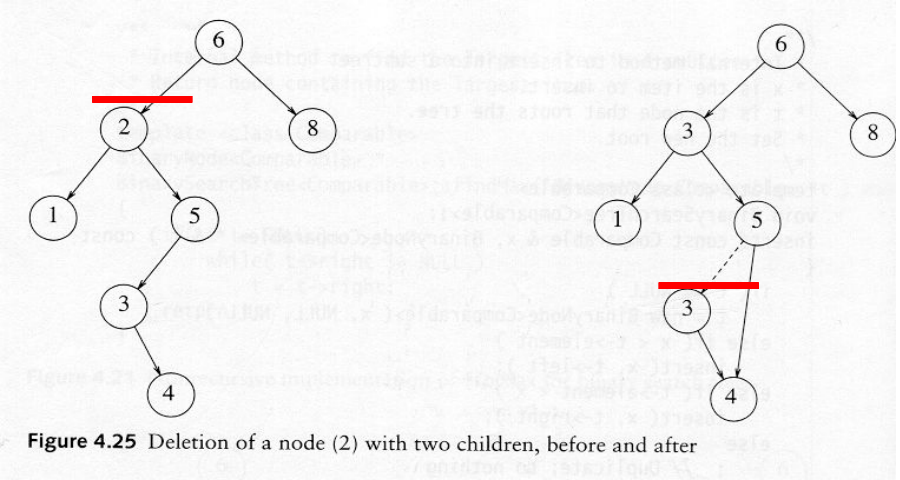


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Deleting an element from a binary tree

a) If the node being deleted is a terminal node then deletion is a straightforward procedure of just making the parent node point to

NULL.

b) If the node being deleted has only one child then the parent node will have to point to the child node of the node being deleted.

c) If the node being deleted has both the child's then we first need to find the inorder successor of the node being deleted and replace the node being deleted with this node. (The inorder successor of a node can be obtained by taking the right node of the current node and traversing in the left till we reach the left most node.)

Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

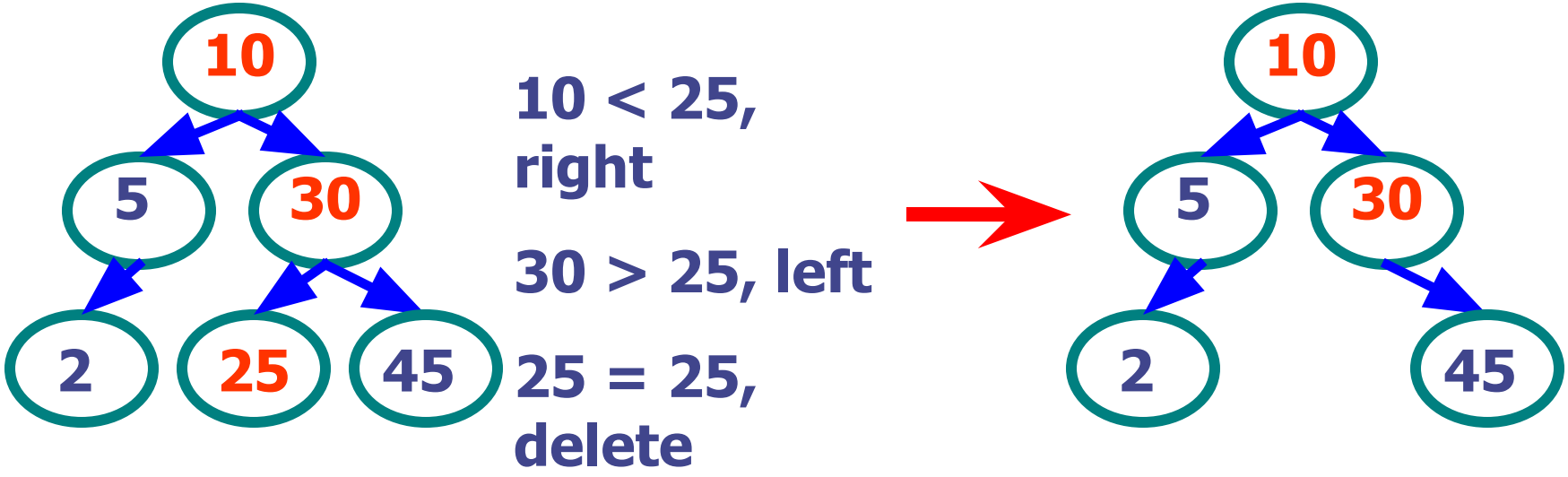
     a) Replace with largest value Y on left subtree

     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

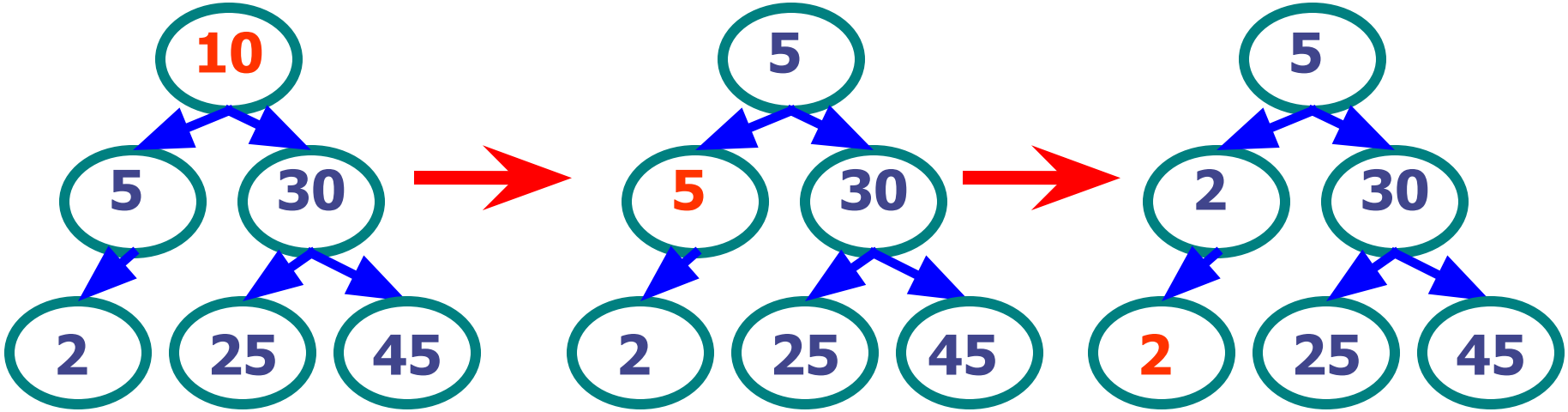
Example Deletion (Leaf)

- Delete (25)



Example Deletion (Internal Node)

- Delete (10)



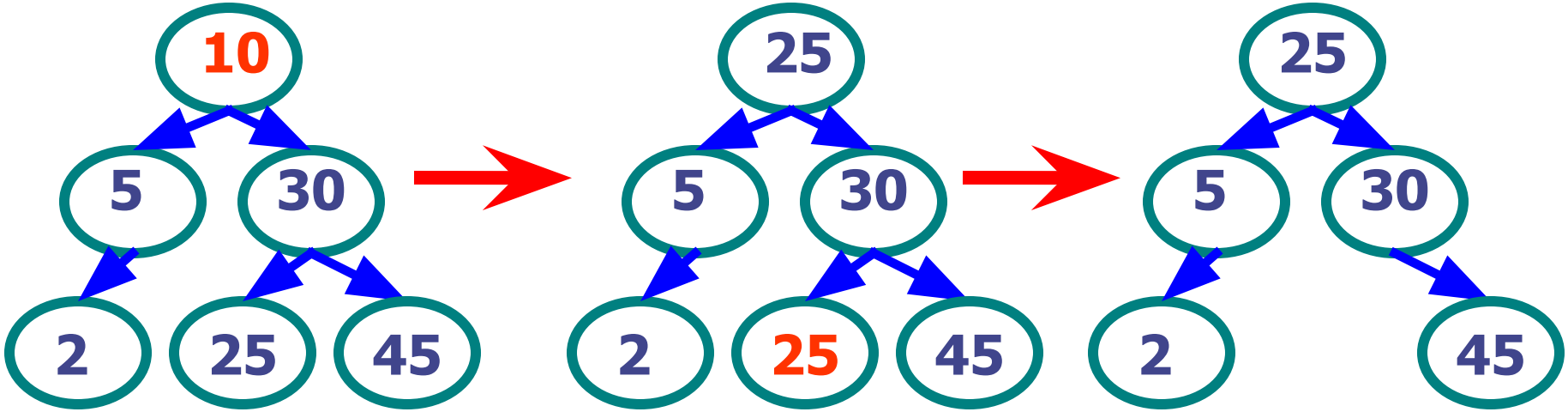
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Binary Search Tree Program

```
/*  C Program to Construct a Binary Search Tree and perform deletion, inorder traversal on it */
#include <stdio.h>
#include <stdlib.h>
struct btnode{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);
int flag = 1;
```

Binary Search Tree Program

```
void main(){
   int ch;
   while(1)    {
      printf("\nOPERATIONS ---");
      printf("\n1 - Insert an element into tree\n");
      printf("2 - Delete an element from the tree\n");
      printf("3 - Inorder Traversal\n");
      printf("4 - Preorder Traversal\n");
      printf("5 - Postorder Traversal\n");
      printf("6 - Smallest Element\n");
      printf("7 - Largest Element\n");
      printf("8 - Exit\n");
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)       {
      case 1:
        insert();
         break;
      case 2:
         delete();
         break;
```

Binary Search Tree Program

```
case 3:
   printf("\nThe inorder traversal is : ");
   inorder(root);
   break;
case 4:
    printf("\nThe preorder traversal is : ");
   preorder(root);
   break;
case 5:
   printf("\nThe postorder traversal is : ");
   postorder(root);
   break;
case 6:
   printf("\n The smallest element is %d\n",smallest(root));
   break;
case 7:
   printf("\n The largest element is %d\n",largest(root));
   break;
case 8:
   exit(0);
default :
   printf("Wrong choice, Please enter correct choice  ");
   break;        }   }    }
```

Binary Search Tree Program

```
/* To insert a node in the tree */
void insert(){
   create();
   if (root == NULL)
      root = temp;
   else
      search(root);
}

/* To create a node */
void create(){
   int data;
   printf("Enter data of node to be inserted : ");
   scanf("%d", &data);
   temp = (struct btnode *)malloc(1*sizeof(struct btnode));
   temp->value = data;
   temp->l = temp->r = NULL;
}
```

Binary Search Tree Program

```
/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
   if ((temp->value > t->value) && (t->r != NULL))
        /* value more than root node value insert at right */
       search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
       t->r = temp;
   else if ((temp->value < t->value) && (t->l != NULL))
         /* value less than root node value insert at left */
       search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
       t->l = temp;
}
```

Binary Search Tree Program

```
/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   if (t->l != NULL)
      inorder(t->l);
   printf("%d -> ", t->value);
   if (t->r != NULL)
      inorder(t->r);
}
```

Binary Search Tree Program

```
/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
      preorder(t->l);
   if (t->r != NULL)
      preorder(t->r);
}
```

Binary Search Tree Program

```
/* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}
```

Binary Search Tree Program

```
/* To check for the deleted node */
void delete()
{
   int data;
   if (root == NULL)
   {
      printf("No elements in a tree to delete");
      return;
   }
   printf("Enter the data to be deleted : ");
   scanf("%d", &data);
   t1 = root;
   t2 = root;
   search1(root, data);
}
```

Binary Search Tree Program

```
/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
{
   if ((data>t->value))
    {
       t1 = t;
       search1(t->r, data);
   }
   else if ((data < t->value))
   {
      t1 = t;
      search1(t->l, data);
   }
   else if ((data==t->value))
   {
      delete1(t);
   }
}
```

Binary Search Tree Program

```
/* To delete a node */
void delete1(struct btnode *t)
{
   int k;
   /* To delete leaf node */
   if ((t->l == NULL) && (t->r == NULL))
   {
      if (t1->l == t)
     {
         t1->l = NULL;
      }
      else
      {
         t1->r = NULL;
      }
      t = NULL;
      free(t);
      return;
   }
```

Binary Search Tree Program

```
 /* To delete node having one left hand child */
else if ((t->r == NULL))
{
   if (t1 == t)
   {
      root = t->l;
      t1 = root;
   }
   else if (t1->l == t)
   {
      t1->l = t->l;
   }
   else
   {
      t1->r = t->l;
   }
   t = NULL;
   free(t);
   return;
}
```

Binary Search Tree Program

```
 /* To delete node having right hand child */
else if (t->l == NULL)
{
   if (t1 == t)
   {
      root = t->r;
      t1 = root;
   }
   else if (t1->r == t)
      t1->r = t->r;
   else
      t1->l = t->r;
   t == NULL;
   free(t);
   return;
}
```

Binary Search Tree Program

```
  /* To delete node having two child */
  else if ((t->l != NULL) && (t->r != NULL))
  {
     t2 = root;
     if (t->r != NULL)
     {
        k = smallest(t->r);
        flag = 1;
     }
     else
     {
        k =largest(t->l);
        flag = 2;
     }
     search1(root, k);
     t->value = k;
  }
}
```

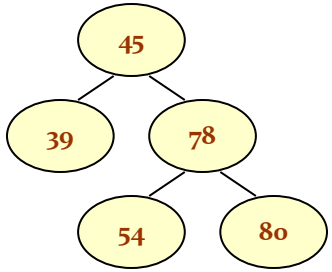
Binary Search Tree Program

```
/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
   t2 = t;
   if (t->l != NULL)
   {
      t2 = t;
      return(smallest(t->l));
   }
   else
      return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
   if (t->r != NULL)
   {
      t2 = t;
      return(largest(t->r));
   }
   else
      return(t->value); }
```

Determining the Height of a BST

- In order to determine the height of a BST, we will calculate the height of the left and right sub-trees. Whichever height is greater, 1 is added to it.
- Since height of right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 2 + 1 = 3



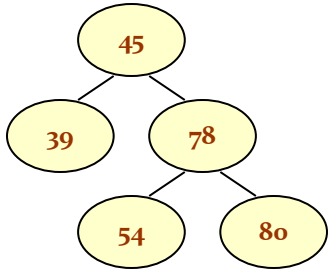
- **Algorithm Height (TREE)**

```
Step 1: IF TREE = NULL, then
        Return 0
        ELSE
        SET LeftHeight = Height(TREE->LEFT)
        SET RightHeight = Height(TREE->RIGHT)
        IF LeftHeight > RightHeight
              Return LeftHeight + 1
        ELSE
              Return RightHeight + 1
        [END OF IF]
        [END OF IF]
Step 2: End
```

Determining the Numbers of Nodes

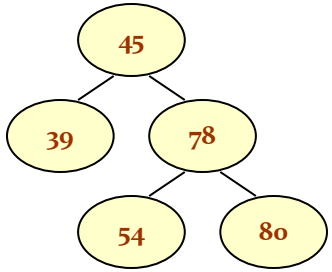
- To calculate the total number of elements/nodes in a BST, we will count the number of nodes in the left sub-tree and the right sub-tree.
- Number of nodes = totalNodes(left sub-tree) + total Nodes(right sub-tree) + 1[1 for root] = 1+3+1=5
- **Algorithm totalNodes (TREE)**

  Step 1: IF TREE = NULL, then
  Return 0
  ELSE
  Return totalNodes(TREE->LEFT) +
  totalNodes(TREE->RIGHT) + 1
  [END OF IF]
  Step 2: End



Determining the Number of Internal Nodes

- To calculate the total number of internal nodes or non-leaf nodes, we count the number of internal nodes in the left sub-tree and the right sub-tree and add 1 to it (1 is added for the root node).
- Number of internal nodes = totalInternalNodes(left sub–tree) + totalInternalNodes(right sub–tree) + 1 = 0+1+1=2

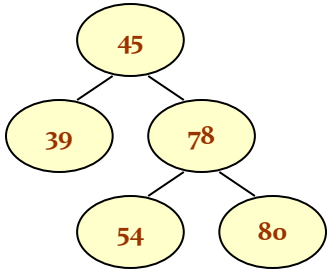


- **Algorithm totalInternalNodes(TREE)**

```
Step 1: IF TREE = NULL
        Return
        [END OF IF]
        IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
        Return
        ELSE
        Return totalInternalNodes(TREE->LEFT) +
             totalInternalNodes(TREE->RIGHT) + 1
        [END OF IF]
Step 2: END
```

Determining the Number of External /Leaf Nodes

- To calculate the total number of external nodes or leaf nodes, we add the number of external nodes in the left sub-tree and the right sub-tree. However if the tree is empty, that is TREE= NULL, then the number of external nodes will be zero. But if there is only one node in the tree, then the number of external nodes will be one.
- Number of external nodes = totalExternalNodes(left sub–tree) + totalExternalNodes (right sub–tree) = 1+2=3



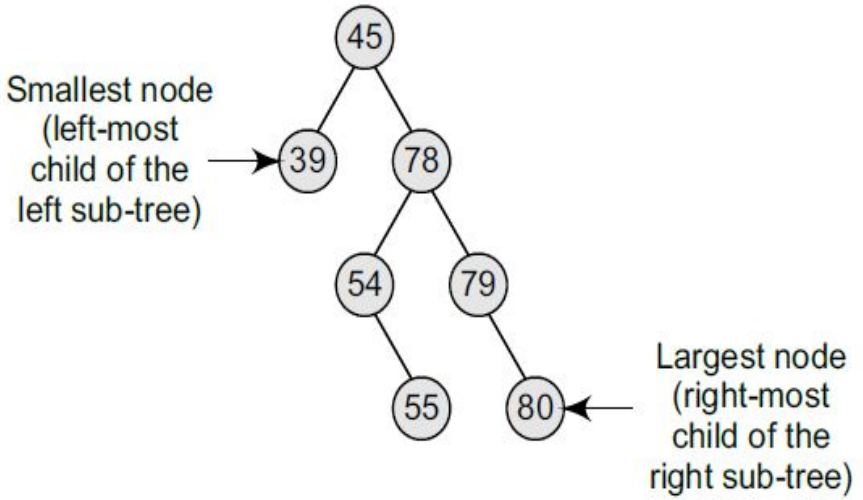
- **Algorithm totalExternalNodes(TREE)**

```
Step 1: IF TREE = NULL
          Return
        ELSE IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
          Return 1
        ELSE
                  Return totalExternalNodes(TREE->LEFT) +
                           totalExternalNodes(TREE->RIGHT)
              [END OF IF]
Step 2: END
```

Finding the Smallest Node in a Binary Search Tree

- The very basic property of the binary search tree states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of the root node will be smallest as compared to the nodes in the right sub-tree.
- **Algorithm findSmallestElement(TREE)**

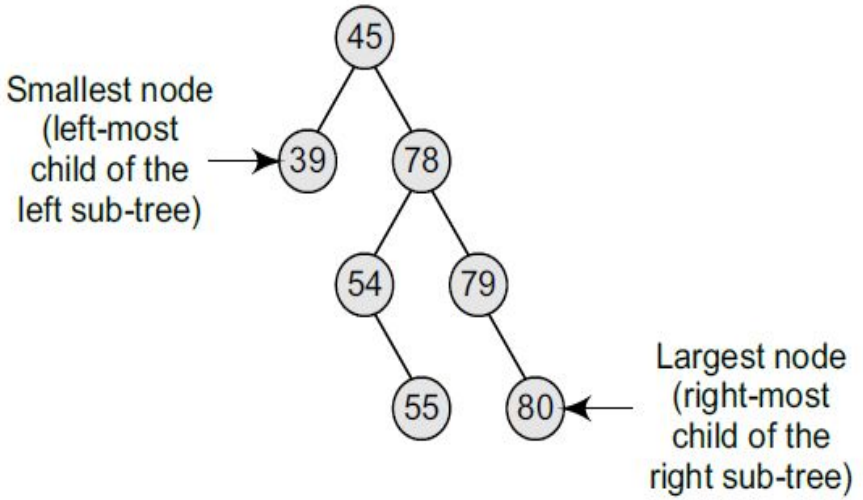
  Step 1: IF TREE = NULL OR TREE->LEFT = NULL
  Return TREE
  ELSE
  Return findSmallestElement(TREE->LEFT)
  [END OF IF]
  Step 2: END



Finding the Largest Node in a Binary Search Tree

- To find the node with the largest value, we find the value of the rightmost node of the right subtree.
- However, if the right sub-tree is empty, then the root node will be the largest value in the tree.
- **Algorithm findLargestElement(TREE)**

  Step 1: IF TREE = NULL OR TREE->RIGHT = NULL
  Return TREE
  ELSE
  Return findLargestElement(TREE->RIGHT)
  [END OF IF]
  Step 2: END



Binary Search Tree Program

```
/*  C Program to Construct a Binary Search Tree and perform traversal as well as find height, no. of leaf nodes, no. of internal nodes and total no. of nodes */
#include <stdio.h>
#include <stdlib.h>
struct btnode{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL;

void insert();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
int totalNodes(struct btnode *t);
int totalExternalNodes(struct btnode *t);
int totalInternalNodes(struct btnode *t);
int Height(struct btnode *t);

//int flag = 1;
```

Binary Search Tree Program

```
void main(){
   int ch;
   while(1)    {
      printf("\nOPERATIONS ---");
      printf("\n1 - Insert an element into tree\n");
          printf("2 - Height of the tree\n");
      printf("3 - Inorder Traversal\n");
      printf("4 - Preorder Traversal\n");
      printf("5 - Postorder Traversal\n");
      printf("6 - Count Total number of nodes\n");
      printf("7 - Count Internal Node\n");
      printf("8 - Count External Node\n");
      printf("9 - Exit\n");
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)       {
      case 1:
        insert();
          break;
      case 2:
          printf("\n The Height of the tree is :  %d\n",Height(root));
          break;
```

Binary Search Tree Program

```
case 3:
   printf("\nThe inorder traversal is : ");
   inorder(root);
   break;
case 4:
    printf("\nThe preorder traversal is : ");
   preorder(root);
   break;
case 5:
   printf("\nThe postorder traversal is : ");
   postorder(root);
   break;
case 6:
   printf("\n Total number of nodes in the tree is : %d\n",totalNodes(root));
   break;
case 7:
   printf("\n Number of Internal Nodes are : %d\n",totalInternalNodes(root));
   break;
case 8:
   printf("\n Number of External Nodes are : %d\n",totalExternalNodes(root));
   break;
```

Binary Search Tree Program

```
    case 9:
        exit(0);
     default :
        printf("Wrong choice, Please enter correct choice  ");
        break;
     }
  }
}
```

Binary Search Tree Program

```
/* To find total number of nodes in the tree */
int totalNodes(struct btnode *tree)
{
   if(tree==NULL)
      return 0;
   else
      return(totalNodes(tree->l) + totalNodes(tree->r) + 1);
}


/* to find external nodes / leaf nodes in the tree */
int totalExternalNodes(struct btnode *tree)
{
   if(tree==NULL)
      return 0;
   else if((tree->l==NULL) && (tree->r==NULL))
      return 1;
   else
      return (totalExternalNodes(tree->l) + totalExternalNodes(tree->r));
}
```

Binary Search Tree Program

```
/* To find internal nodes in the tree */
int totalInternalNodes(struct btnode *tree){
   if( (tree==NULL) || ((tree->l==NULL) && (tree->r==NULL)))
      return 0;
   else
   return (totalInternalNodes(tree->l)+ totalInternalNodes(tree->r) + 1);
}

/* to find height of tree */
int Height(struct btnode *tree){
   int leftheight, rightheight;
   if(tree==NULL)
      return 0;
   else
   {
      leftheight = Height(tree->l);
      rightheight = Height(tree->r);
      if(leftheight > rightheight)
         return (leftheight + 1);
   else
      return (rightheight + 1);
   }
}
```

Binary Search Tree Program

```
/* To insert a node in the tree */
void insert(){
   create();
   if (root == NULL)
      root = temp;
   else
      search(root);
}

/* To create a node */
void create(){
   int data;
   printf("Enter data of node to be inserted : ");
   scanf("%d", &data);
   temp = (struct btnode *)malloc(1*sizeof(struct btnode));
   temp->value = data;
   temp->l = temp->r = NULL;
}
```

Binary Search Tree Program

```
/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
   if ((temp->value > t->value) && (t->r != NULL))
        /* value more than root node value insert at right */
       search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
       t->r = temp;
   else if ((temp->value < t->value) && (t->l != NULL))
         /* value less than root node value insert at left */
       search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
       t->l = temp;
}
```

Binary Search Tree Program

```
/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   if (t->l != NULL)
      inorder(t->l);
   printf("%d -> ", t->value);
   if (t->r != NULL)
      inorder(t->r);
}
```

Binary Search Tree Program

```
/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
      preorder(t->l);
   if (t->r != NULL)
      preorder(t->r);
}
```

Binary Search Tree Program

```
/* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}
```

Height Balanced Tree (AVL Tree)

- A tree is called **AVL tree (Height Balanced Tree)**, if each node possessed one of the following properties
  - A **node** is called **left heavy**, if the **longest path in its left sub tree** is **one** longer than the **longest path of its right sub tree**
  - A **node** is called **right heavy,** if the **longest path in its right subtree** is **one** longer than the **longest path of its left sub tree**
  - A **node** is called **balanced**, if the longest path in **both the right and left sub-trees** are equal
- In height balanced tree, each node must be in one of these states
- If there exists a node in a tree where this is not true, then such a tree is called **Unbalanced**
- Introduced by Adelson-Velskii and Landis in 1962.

Representation of AVL Tree

- For representing an AVL tree we require four fields-
  - One for data.
  - One for address of the right sub tree.
  - One for the address of the left sub tree.
  - An additional field for storing the balancing factor.
- Taking the example above as reference, we can se that,
  - Data value = 30.
  - Height = 3.
  - Left child = 17.
  - Right child = 35.
- The structure of a node of an AVL tree is as follows-

```
struct AVL {
        struct AVL *left;
        int data;
        struct AVL *right;
        int bfact;
}
```

Height Balanced Tree (AVL Tree)

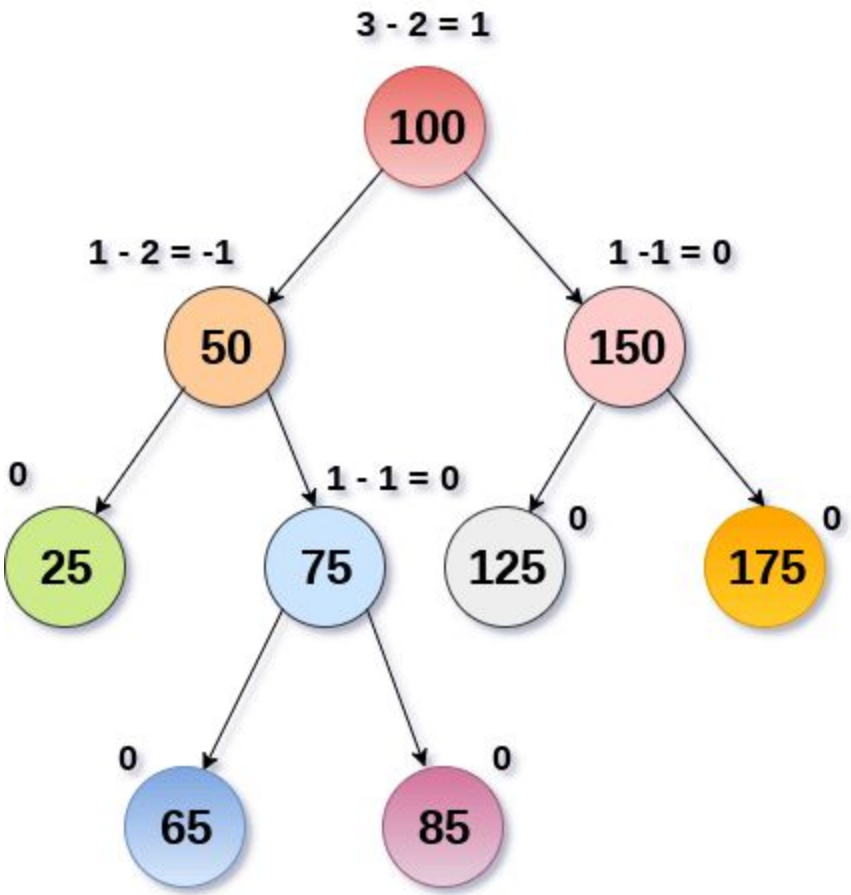
- AVL tree is a self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one. Because of this property, AVL tree is also known as a height-balanced tree.
- The key advantage of using an AVL tree is that it takes O(logn) time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to O(logn)).
- 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.
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

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

Height Balanced Tree (AVL Tree)

- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be **height balanced.** A node with any other balance factor is considered to be **unbalanced** and requires rebalancing.
- If the **balance factor** of a node is **1**, then it means that the **left sub-tree of the tree is one level higher** than that of the **right sub-tree**. Such a tree is called **Left-heavy tree.**
- If the **balance factor** of a node is **0**, then it means that the **height of the left sub-tree** is **equal** to the **height of its right sub-tree**.
- If the **balance factor** of a node is -**1**, then it means that the **left sub-tree** of the tree is **one level lower than that of the right sub-tree**. Such a tree is called **Right-heavy tree**.

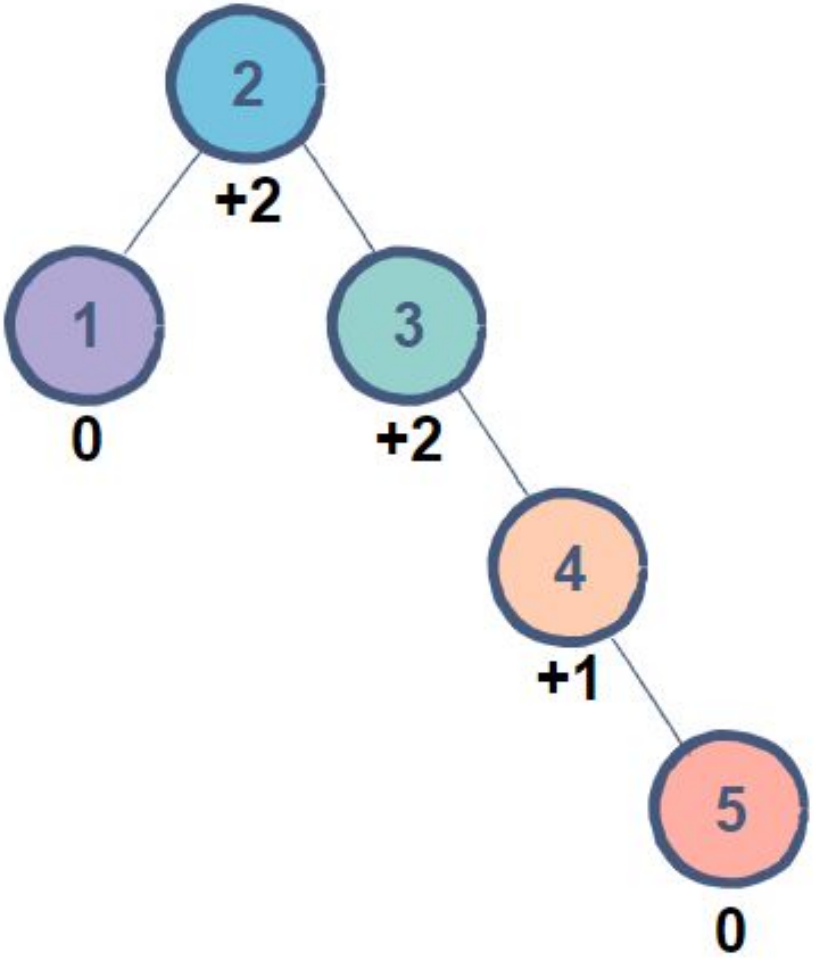
Balance Factor of AVL Tree

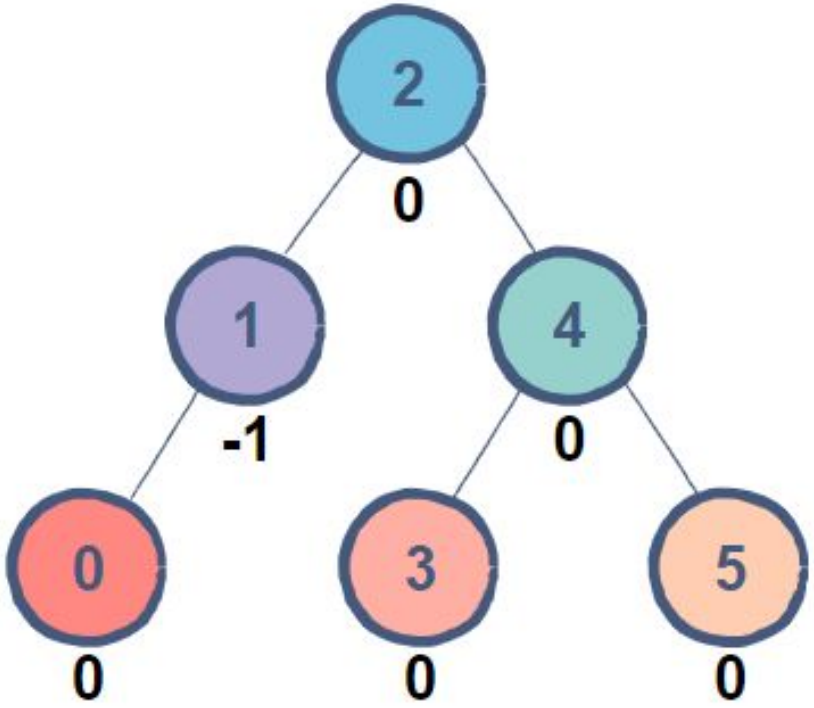


AVL Tree

Balance Factor (k) = height (left(k)) - height (right(k))

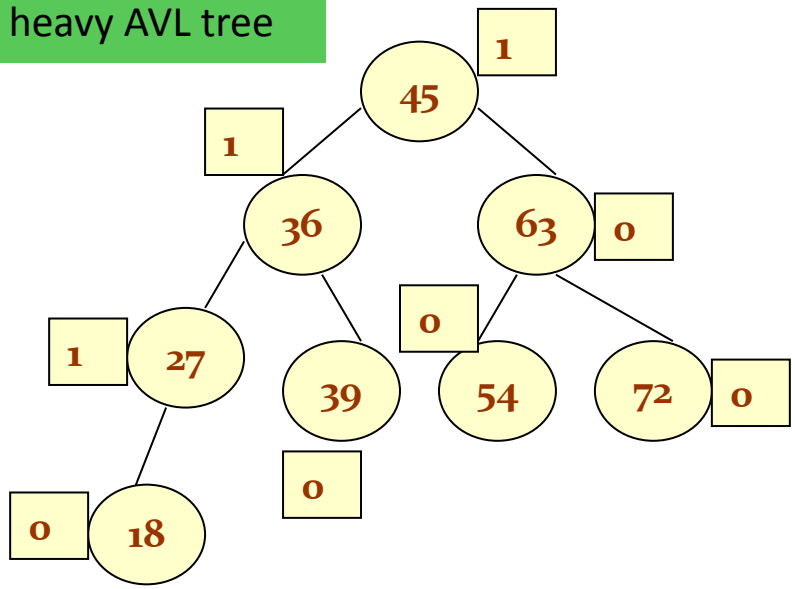
Balance Factor of AVL Tree



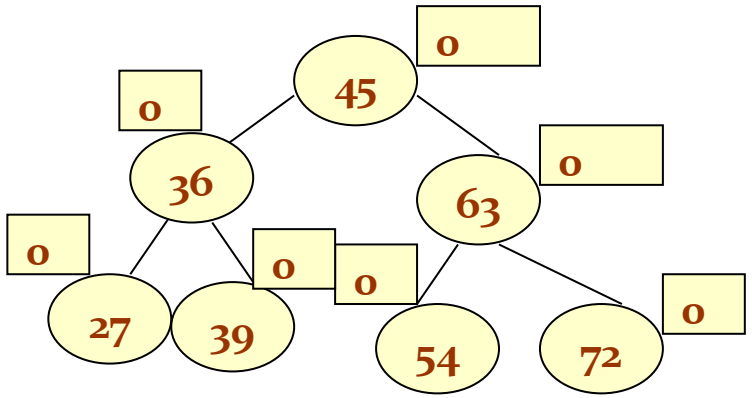


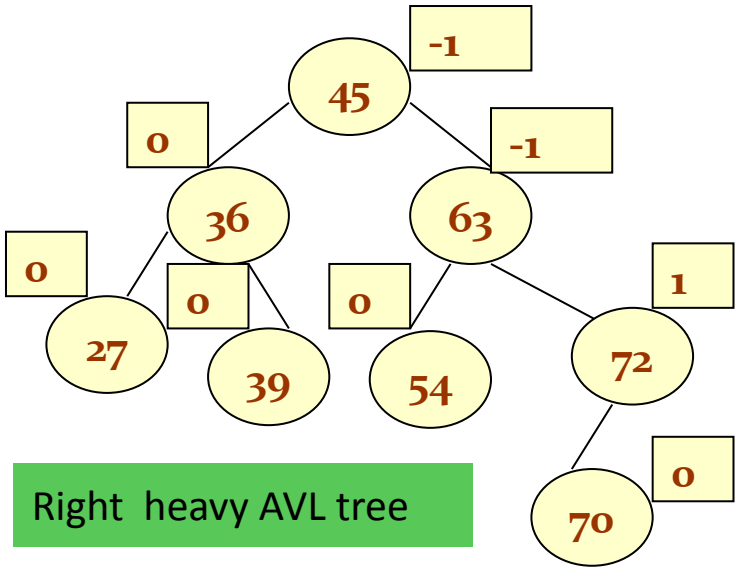
AVL Tree

Left heavy AVL tree



Balanced AVL tree

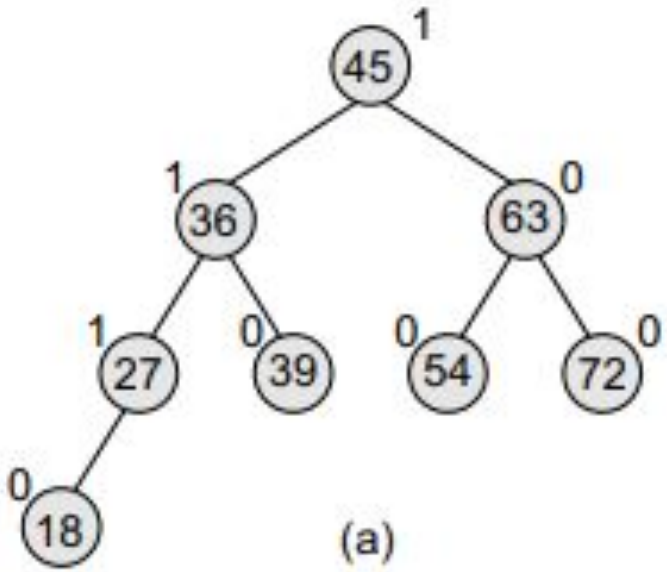




Right heavy AVL tree

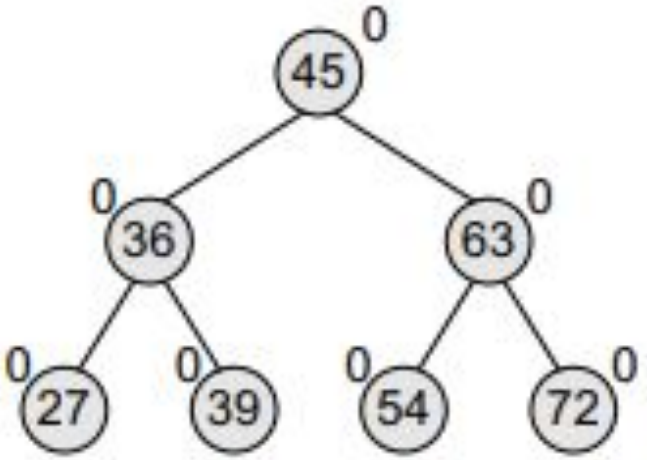
AVL Tree

Left heavy AVL tree

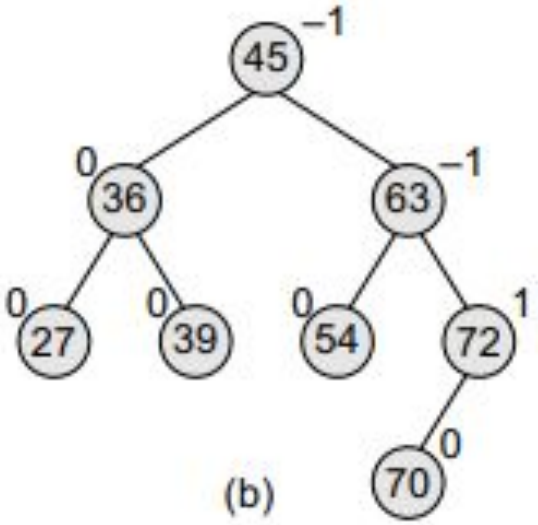


(a)

Balanced AVL tree



(c)



(b)

Right heavy AVL tree

WHY AVL

The primary issue with binary search trees is that they can be unbalanced as worst as performing operations such as insertions, deletions, and searches in O(n) time.

- Since skewed or unbalanced BSTs provide inefficient search operations,

AVL Trees prevent unbalancing by defining what we call a balance factor for each node.

- AVL Trees were developed to achieve logarithmic time complexity in BSTs irrespective of the order in which the elements were inserted.
- AVL Tree implemented a Balancing Criteria (For all nodes, the subtrees' height difference should be at most 1) to overcome the limitations of BST.
- It maintains its height by performing rotations whenever the balance factor of a node violates the Balancing Criteria. As a result, it has self-balancing properties.
- It exists as a balanced BST at all times, providing logarithmic time complexity for operations such as searching.

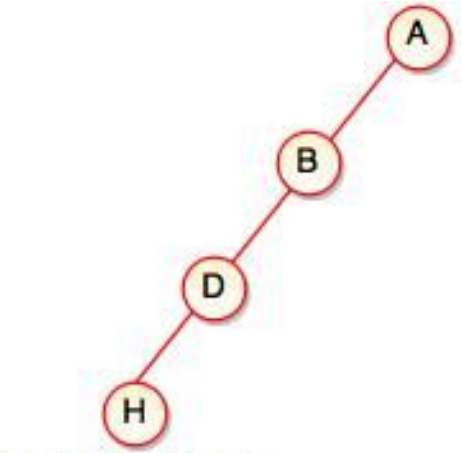


Fig. Left Skewed Binary Tree

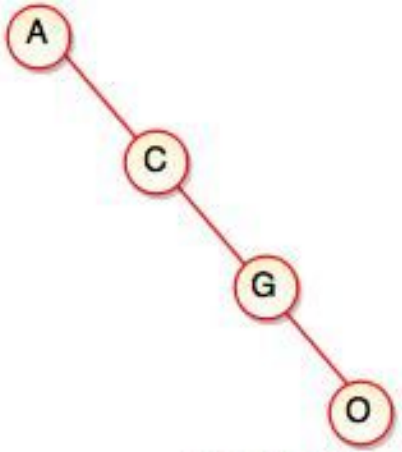
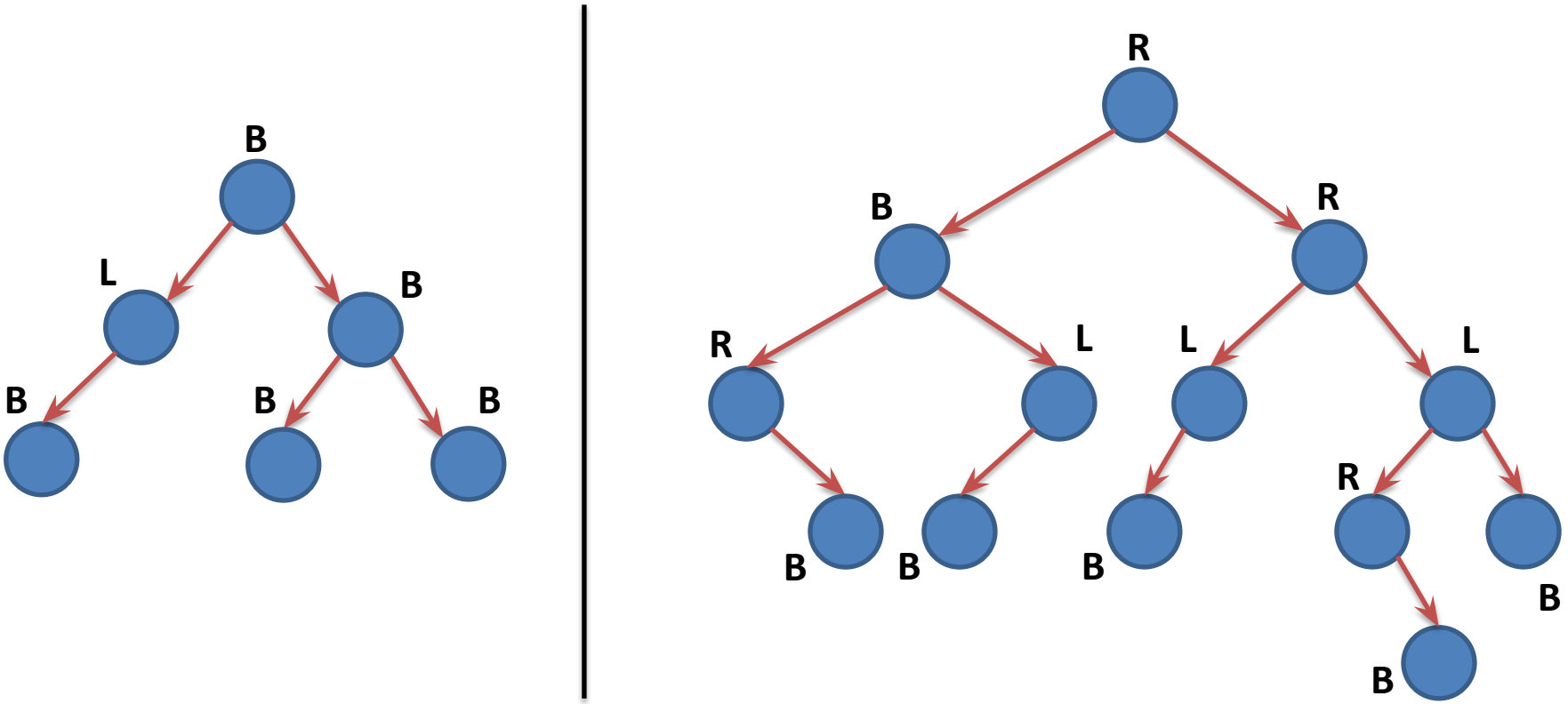


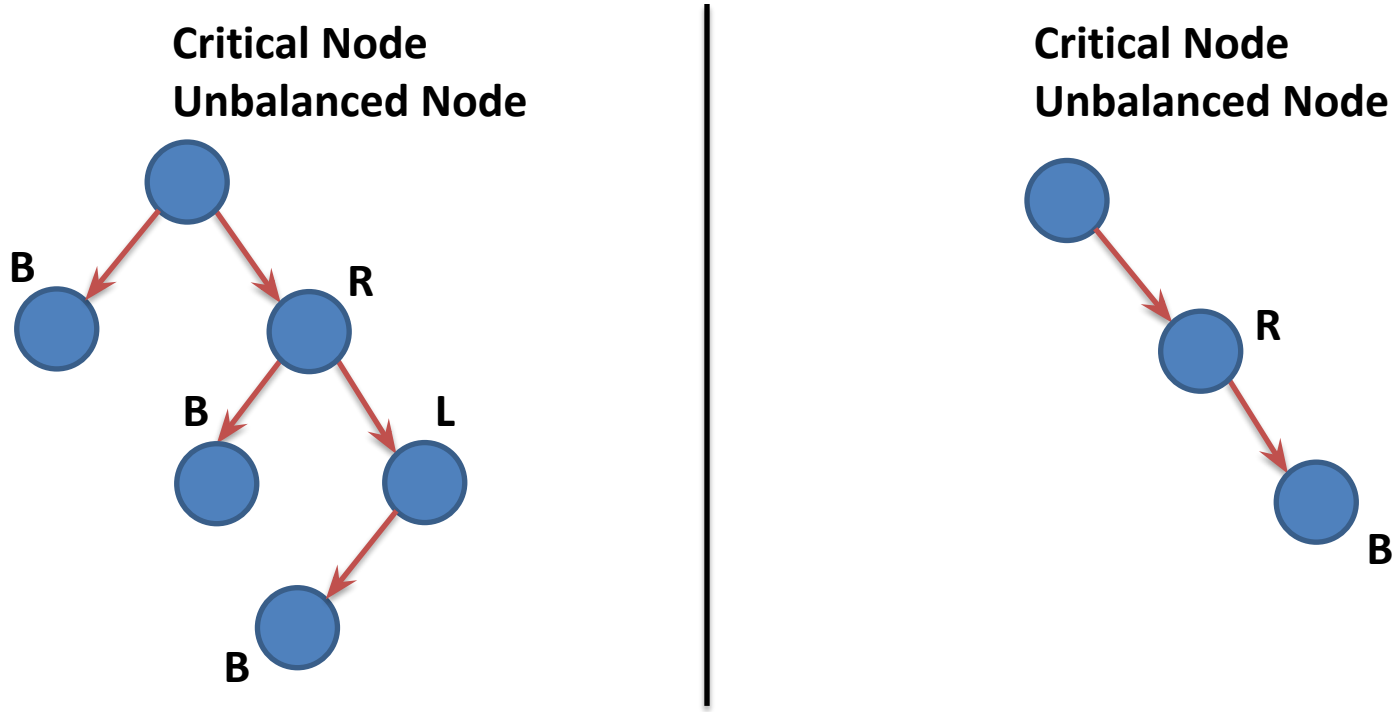
Fig. Right Skewed Binary Tree

AVL Tree



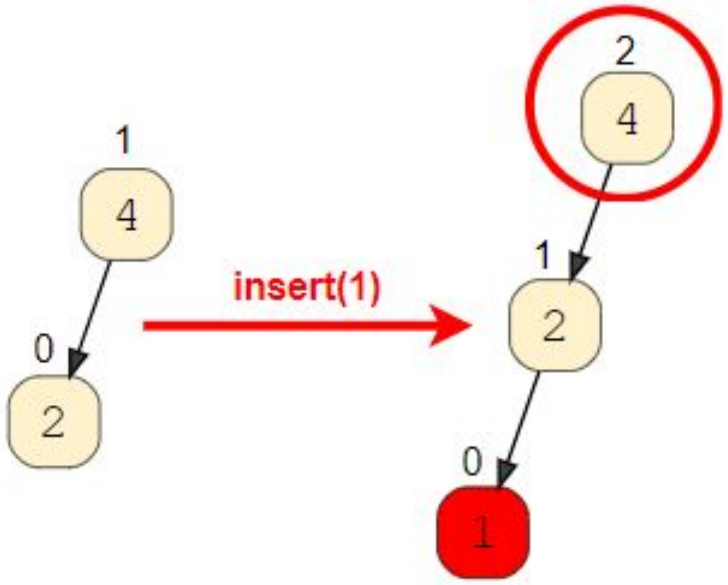
Balanced Trees

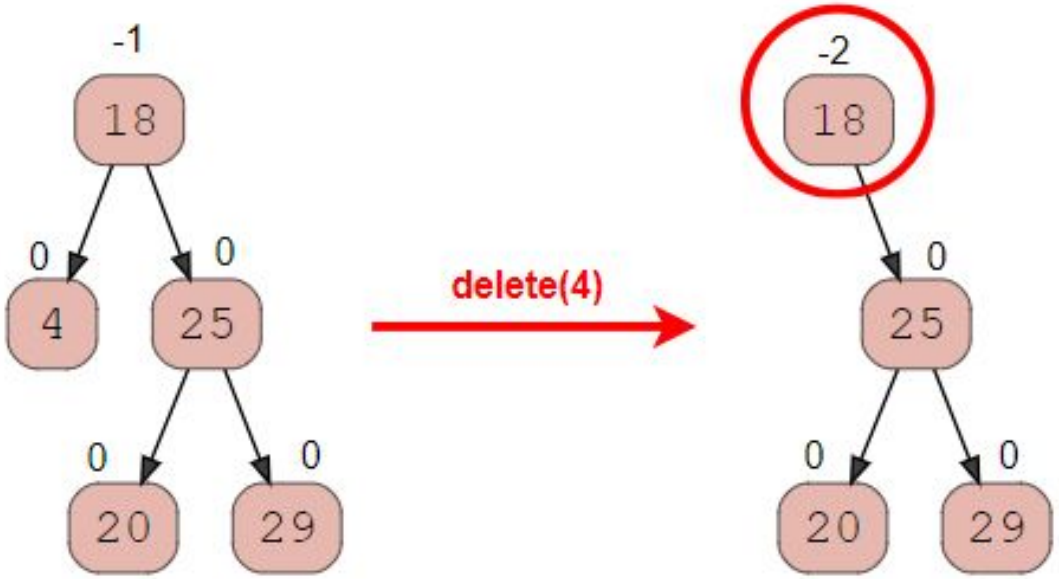
AVL Tree



- Sometimes tree becomes unbalanced by inserting or deleting any node
- Then based on position of insertion, we need to rotate the unbalanced node
- **Rotation** is the **process** to **make tree balanced**

- Each time a node is inserted into an AVL tree or deleted from an AVL tree, the balance factor of a node(s) may be disrupted (it will lie outside the range [-1, 1], which causes the AVL tree to become unbalanced).





Operations on AVL Tree

- **Insertion :** Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

- **Deletion** : Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Rotations on AVL Tree

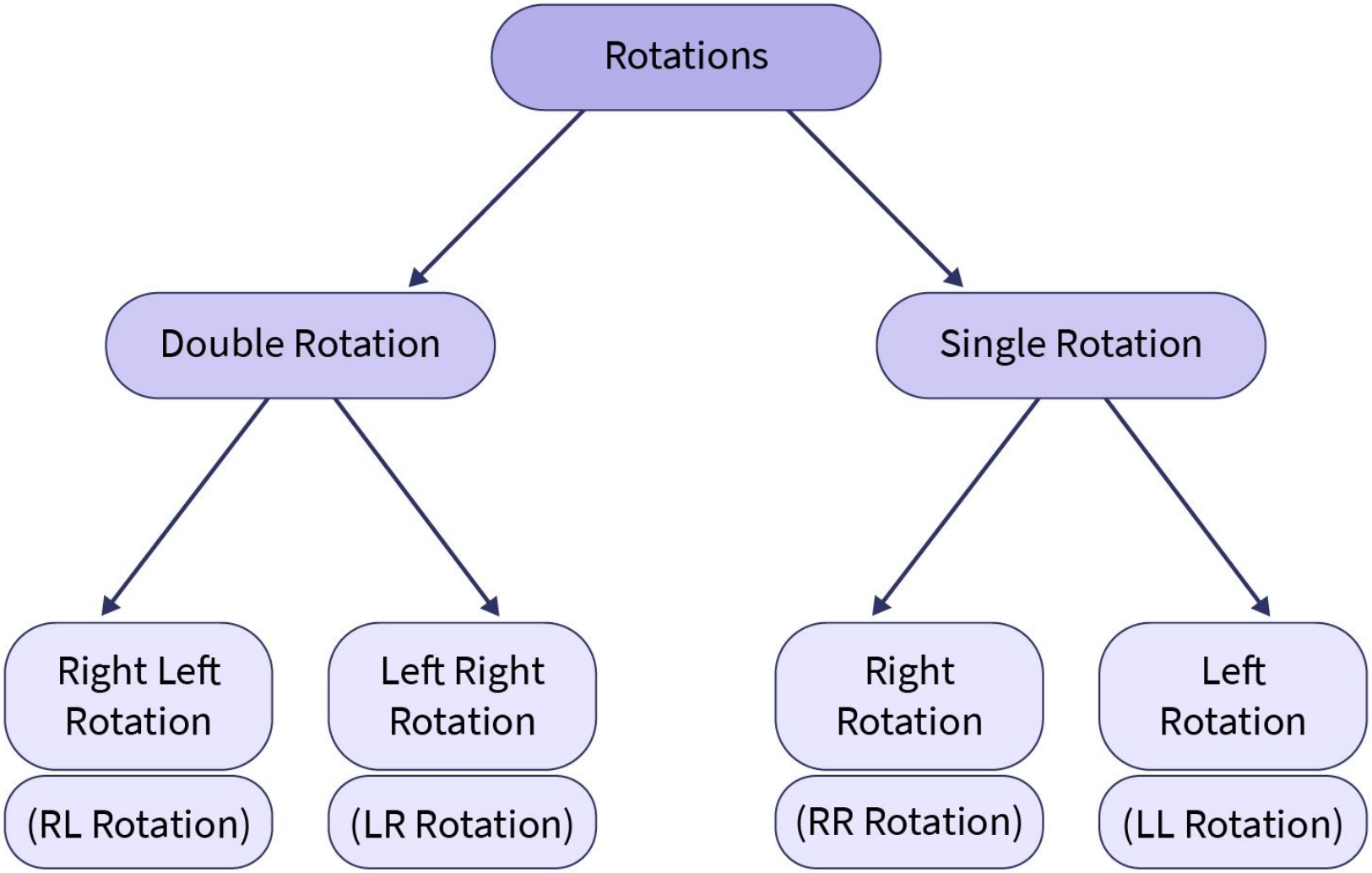
- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:
  1. L L rotation: Inserted node is in the left subtree of left subtree of A
  2. R R rotation : Inserted node is in the right subtree of right subtree of A
  3. L R rotation : Inserted node is in the right subtree of left subtree of A
  4. R L rotation : Inserted node is in the left subtree of right subtree of A
- Where node A is the node whose balance Factor is other than -1, 0, 1.
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

Rotations on AVL Tree

Tree Rotations:

- It is the process of changing the structure of the tree by moving smaller subtrees down and larger subtrees up, without interfering with the order of the elements.
- If the balance factor of any node doesn't follow the AVL Balancing criterion, the AVL Trees make use of 4 different types of Tree rotations to re-balance themselves.
- These rotations are classified based on the node imbalance cured by them i.e., a specific rotation is applied to counter the change that occurred in the balance factor of a node making it unbalanced.

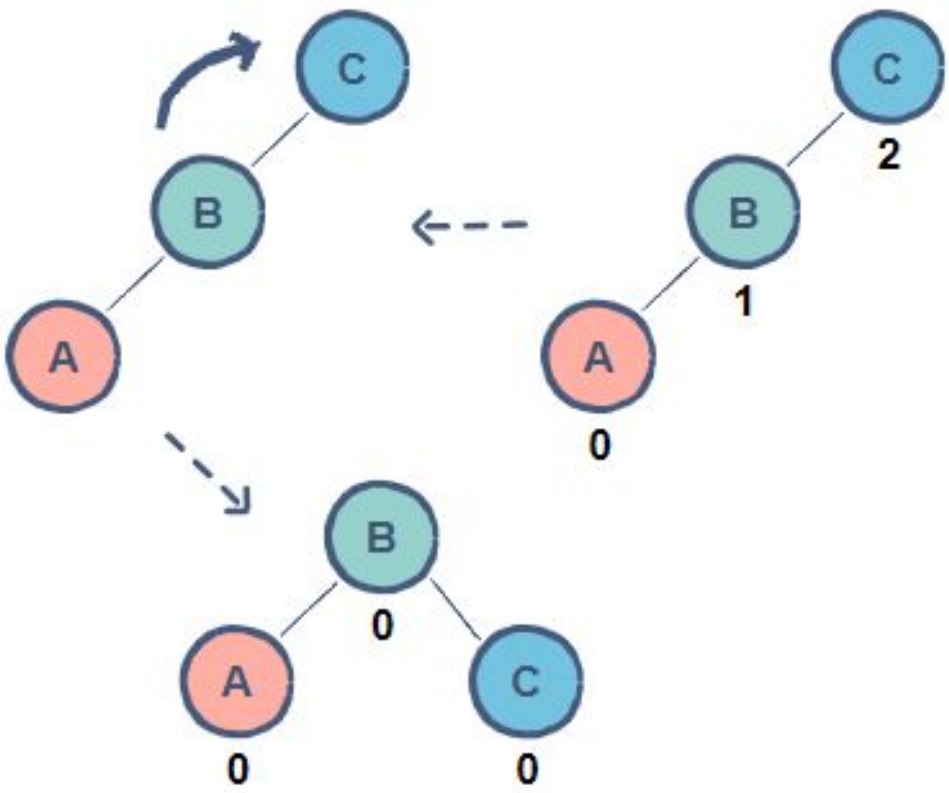
TREE ROTATIONS



TREE ROTATIONS

Right Rotation

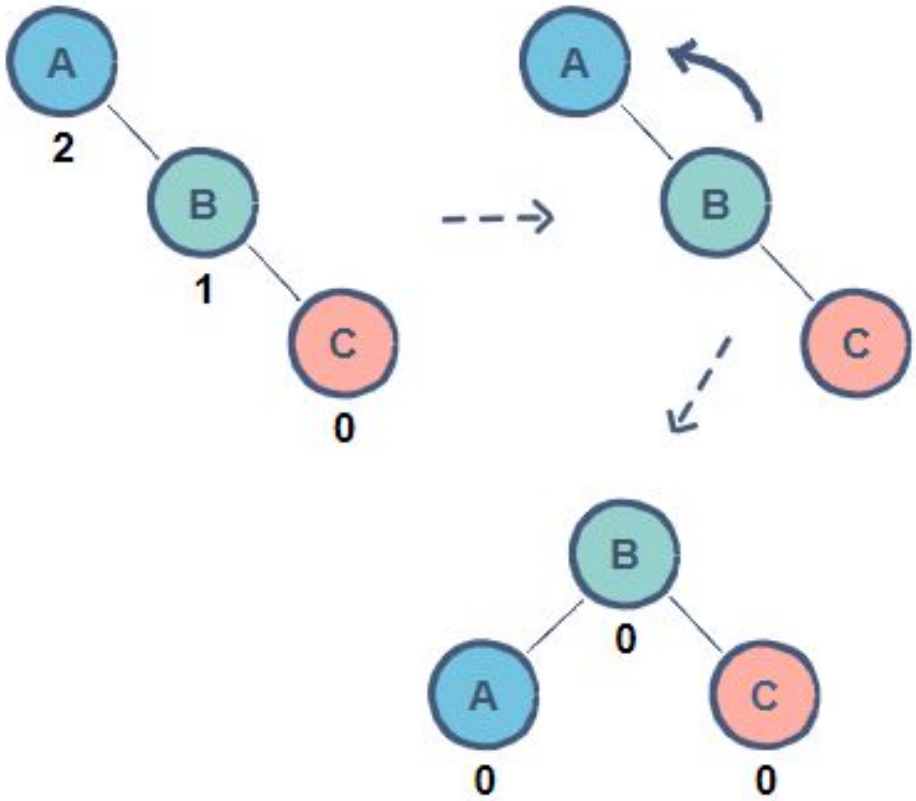
- A single rotation applied when a node is inserted in the left subtree of a left subtree.
- In the given example, node C now has a balance factor of 2 after the insertion of node A.
- By rotating the tree right, node B becomes the root resulting in a balanced tree.



TREE ROTATIONS

Left Rotation

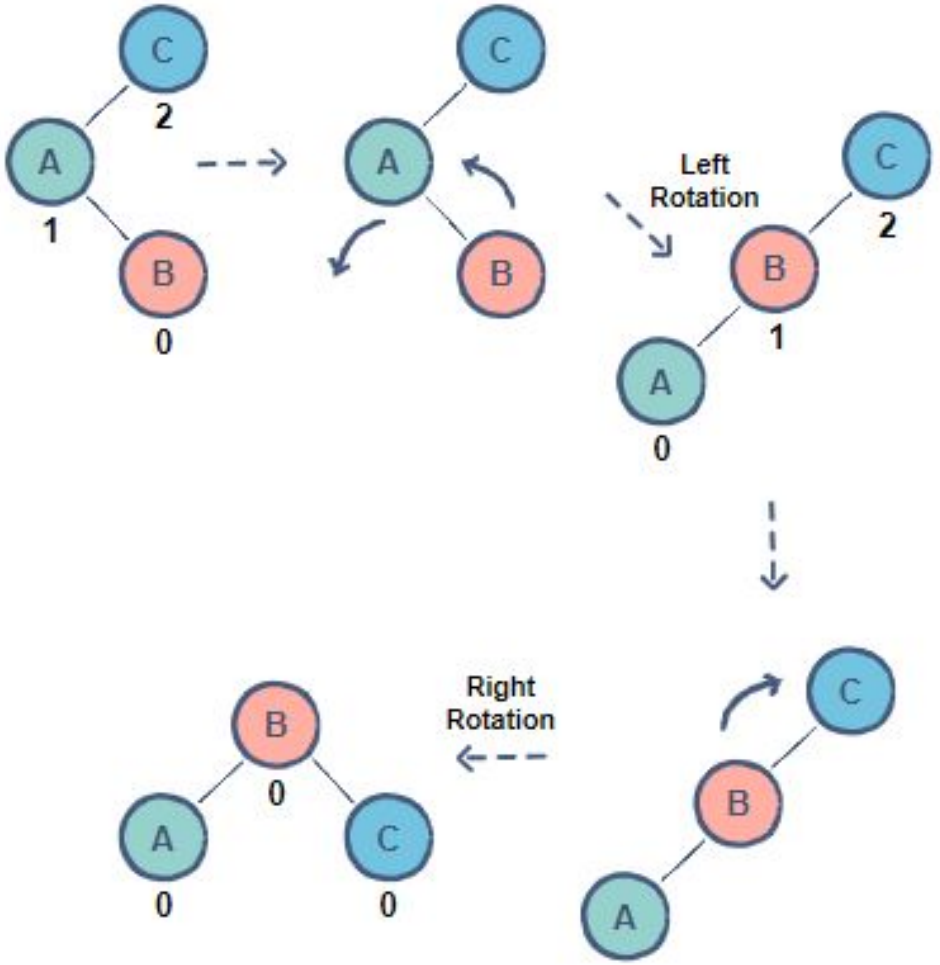
- A single rotation applied when a node is inserted in the right subtree of a right subtree.
- In the given example, node A has a balance factor of 2 after the insertion of node C.
- By rotating the tree left, node B becomes the root resulting in a balanced tree.



TREE ROTATIONS

Left-Right Rotation

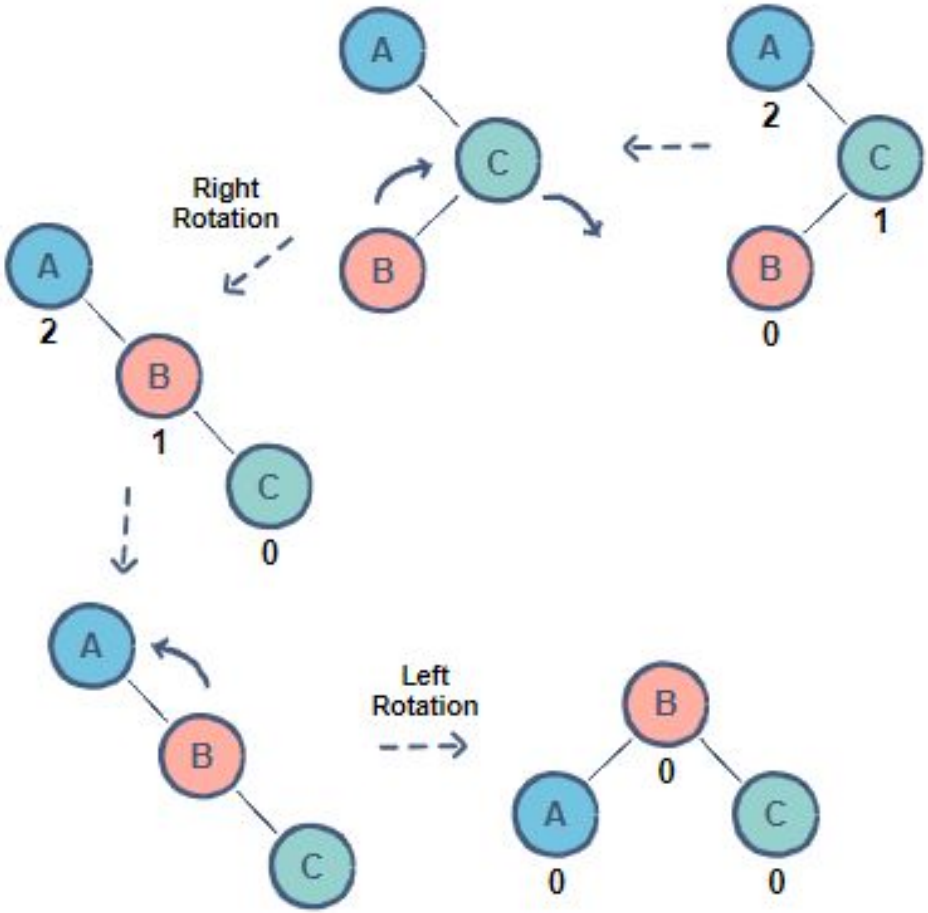
- A double rotation in which a left rotation is followed by a right rotation.
- In the given example, node B is causing an imbalance resulting in node C to have a balance factor of 2.
- As node B is inserted in the right subtree of node A, a left rotation needs to be applied.
- However, a single rotation will not give us the required results.
- Now, all we have to do is apply the right rotation as shown before to achieve a balanced tree.



TREE ROTATIONS

Right-Left Rotation

- A double rotation in which a right rotation is followed by a left rotation.
- In the given example, node B is causing an imbalance resulting in node A to have a balance factor of 2.
- As node B is inserted in the left subtree of node C, a right rotation needs to be applied.
- However, just as before, a single rotation will not give us the required results.
- Now, by applying the left rotation as shown before, we can achieve a balanced tree.



Rotations on AVL Tree

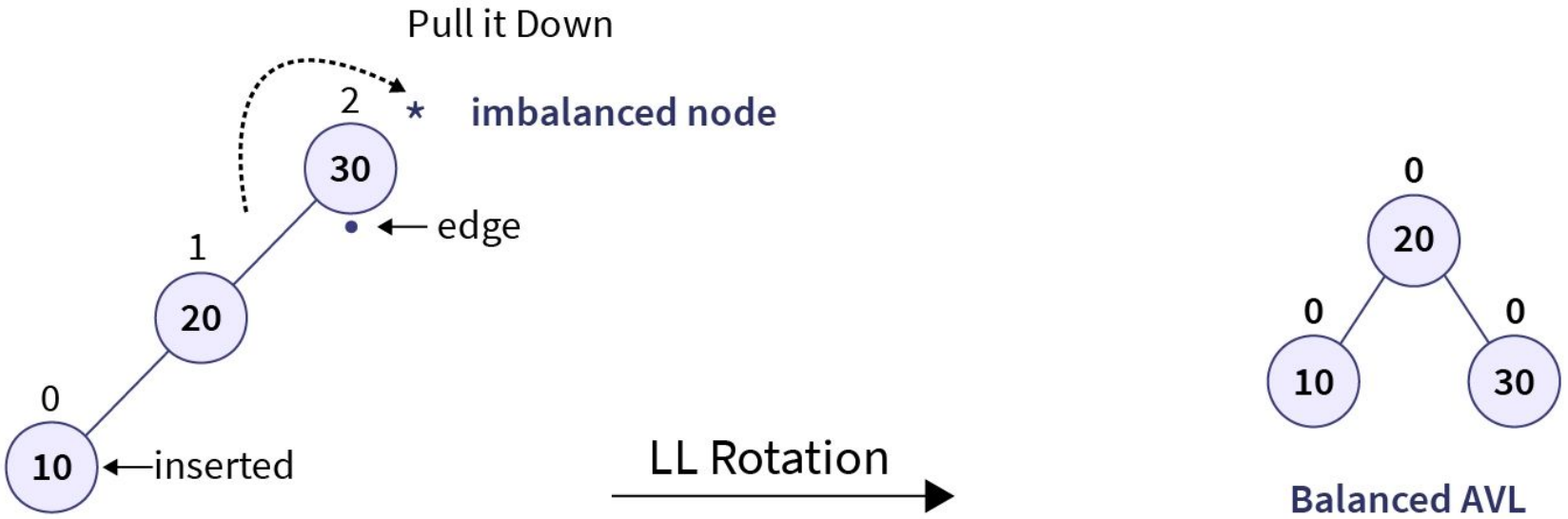
LL Rotation

- It is a type of single rotation that is performed when the tree gets unbalanced, upon insertion of a node into the left subtree of the left child of the imbalance node i.e., upon Left-Left (LL) insertion.
- This imbalance indicates that the tree is heavy on the left side. Hence, a right rotation (or clockwise rotation) is applied such that this left heaviness imbalance is countered and the tree becomes a balanced tree.

Rotations on AVL Tree

LL Rotation

- Consider, a case when we wish to create a BST using elements 30, 20, and 10. Now, since these elements are given in a sorted order, the BST so formed is a left-skewed tree as shown below:



Rotations on AVL Tree

LL Rotation

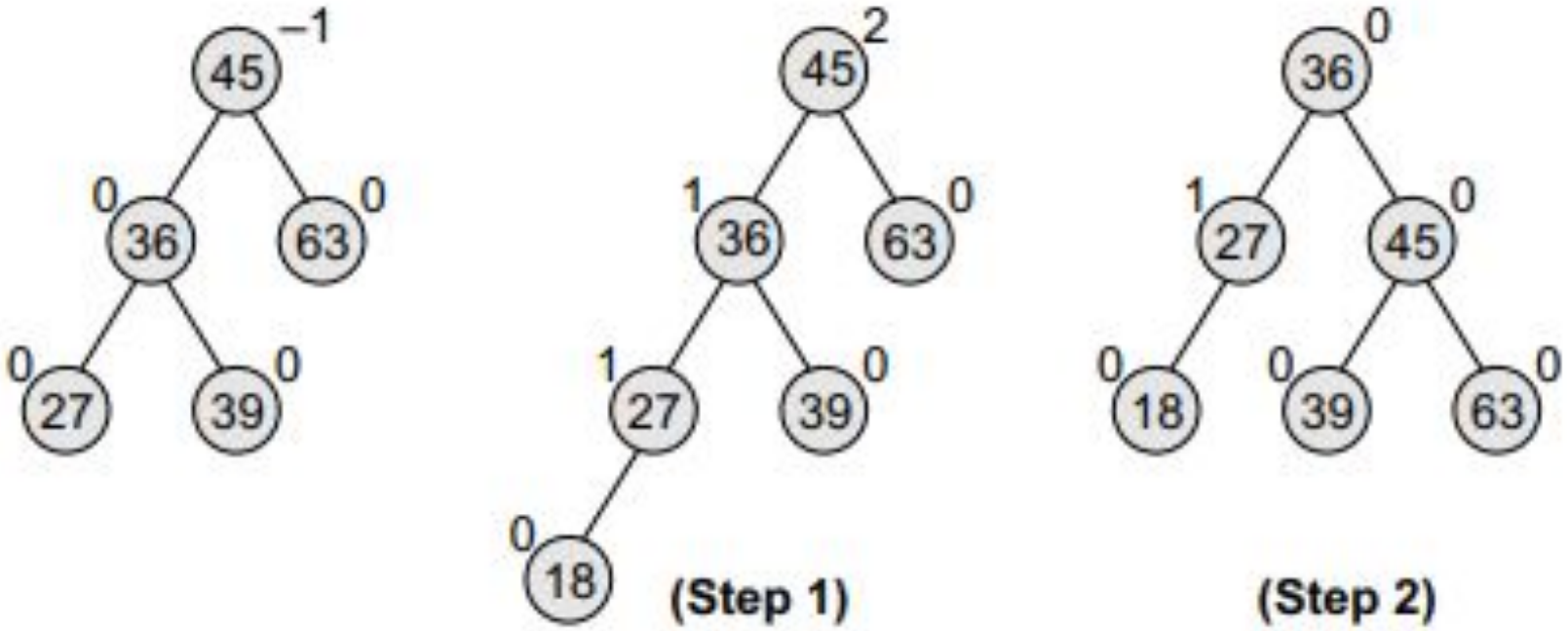
- This is confirmed after calculating the balance factor of all the nodes present in the tree. As you can observe, when we insert element 10 in the tree, the root node becomes imbalanced (balance factor = 2) because the tree becomes left-skewed upon this operation.
- Also, notice that element 10 is inserted as a left child in the left subtree of the imbalanced node (here, the root node of the tree).
- Hence, this is the case of L-L insertion and we will have to perform a certain operation to counter-act this left skewness.
- Now, imagine a weighing scale in which we only have 5 kg on the left plate and nothing on the right plate. This is the case of left heavy since there is nothing on the right plate to balance the weight present in the left plate. Now, to balance this scale we can just add some weight to the right plate.

Rotations on AVL Tree

LL Rotation

- Hence, to balance the weight on one side we try to increase the weight on the other side.
- In the case of trees, instead of adding a new node (weight) on the lighter side, we try to rotate the structure of the tree around a pivot point thereby shifting the nodes from the heavier side to the lighter side.
- In our example, we have extra weight on the left subtree (LL insertion) therefore we will perform right rotation or clockwise rotation on the imbalanced node to transfer this node on the right side to retrieve a balanced tree i.e., we will pull the imbalanced node down by rotating the tree in a clockwise direction along the edge of the imbalanced or in this case, the root node.

LL Rotation


45
−1
0
36
63
0
0
27
39
0
45
2
1
36
63
0
1
27
39
0
0
18
(Step 1)
36
0
1
27
45
0
0
18
0
39
63
0
(Step 2)

Rotations on AVL Tree

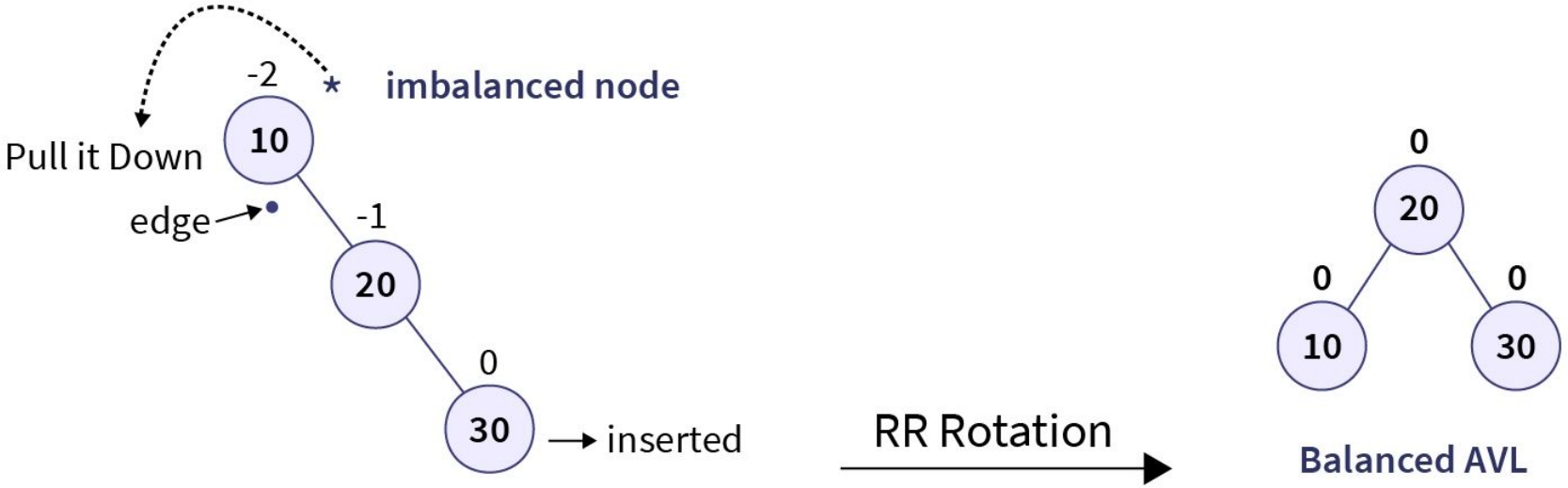
RR Rotation

- It is similar to that of LL Rotation but in this case, the tree gets unbalanced, upon insertion of a node into the right subtree of the right child of the imbalance node i.e., upon Right-Right (RR) insertion instead of the LL insertion.
- In this case, the tree becomes right heavy and a left rotation (or anti-clockwise rotation) is performed along the edge of the imbalanced node to counter this right skewness caused by the insertion operation.

Rotations on AVL Tree

RR Rotation

- Consider a case where we wish to create a BST using the elements 10, 20, and 30. Now, since the elements are given in sorted order, the BST so created becomes right-skewed as shown below:

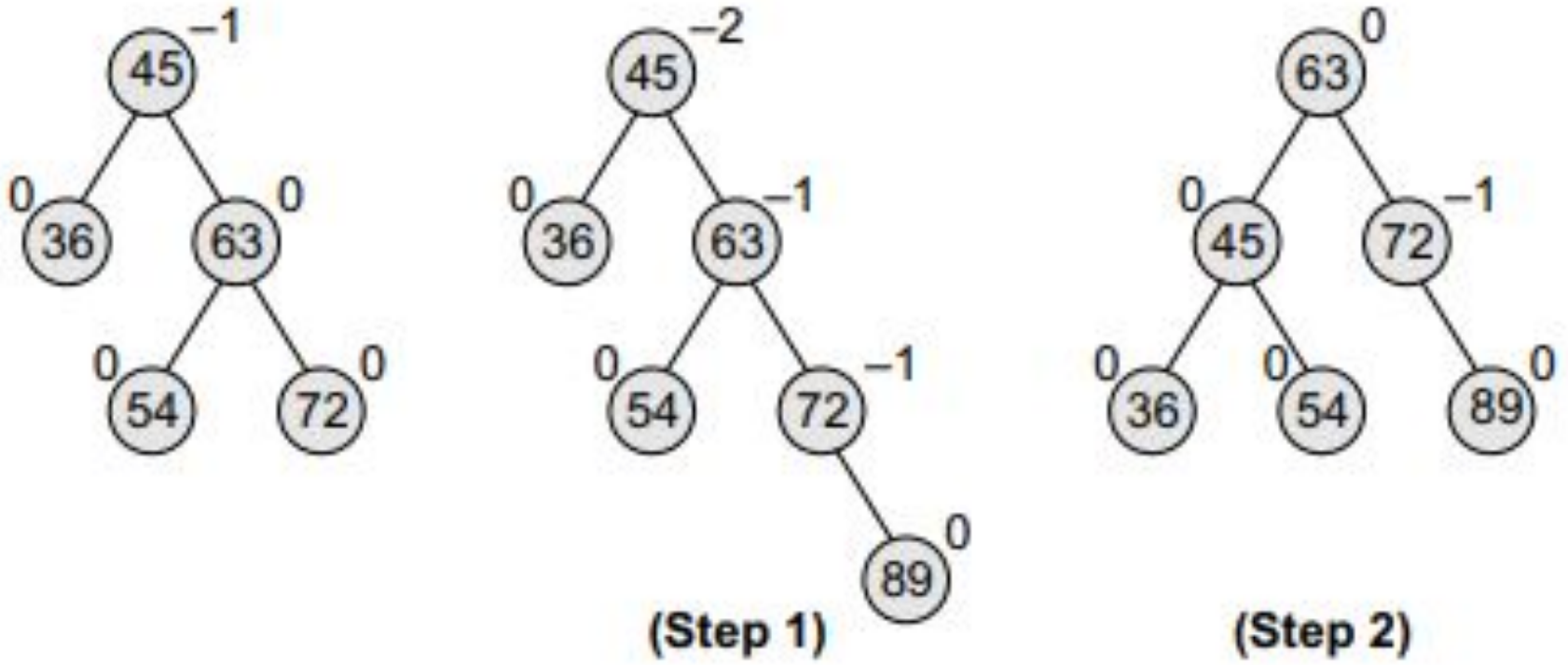


Rotations on AVL Tree

RR Rotation

- Upon calculating the balance factor of all the nodes, we can confirm that the root node of the tree is imbalanced (balance factor = 2) when the element 30 is inserted using RR-insertion.
- Hence, the tree is heavier on the right side and we can balance it by transferring the imbalanced node on the left side by applying an anti-clockwise rotation around the edge (pivot point) of the imbalanced node or in this case, the root node.

RR Rotation


45
−1
0
36
63
0
0
54
72
0
45
−2
0
36
63
−1
0
54
72
−1
89
0
(Step 1)
63
0
0
45
72
−1
0
36
0
54
89
0
(Step 2)

Rotations on AVL Tree

LR Rotation

- There also exist some cases where a single tree rotation isn't enough to balance the tree i.e., we may need to perform one more rotation to finally counter the effects of the height-affecting operation.
- One such case is of Left-Right (LR) insertion i.e., the tree gets unbalanced, upon insertion of a node into the right subtree of the left child of the imbalance node.

Rotations on AVL Tree

LR Rotation

- Consider a situation where you create a BST using elements 30, 10, and 20. Now, when the elements are inserted, the element 30 becomes the root, 10 becomes its left child, and when the element 20 is inserted, it is inserted as the right child of the node having the value 10. This causes an imbalance in the tree as the root node's balance factor becomes equal to 2.
- You may have noticed that a positive balance factor indicates that the given node is left-heavy while a negative one indicates that the node is right-heavy.
- Now, if we notice the immediate parent of the inserted node, we notice that its balance factor is negative i.e., its right-heavy.
- Hence, we may say that we should perform a left rotation (RR rotation) on the immediate parent of the inserted node to counter this effect.

Rotations on AVL Tree

LR Rotation

- As we observe, upon applying the RR rotation the BST becomes left-skewed and is still unbalanced. This is now the case of LL rotation and by rotating the tree along the edge of the imbalanced node in the clockwise direction, we can retrieve a balanced BST.
- Hence, a simple rotation won't fully balance the tree but it may flip the tree in such a manner that it gets converted into a single rotation scenario, after which we can balance the tree by performing one more tree rotation.
- This process of applying two rotations sequentially one after another is known as double rotation and since in our example the insertion was Left-Right (LR) insertion, this combination of RR and LL rotation is known as LR rotation.

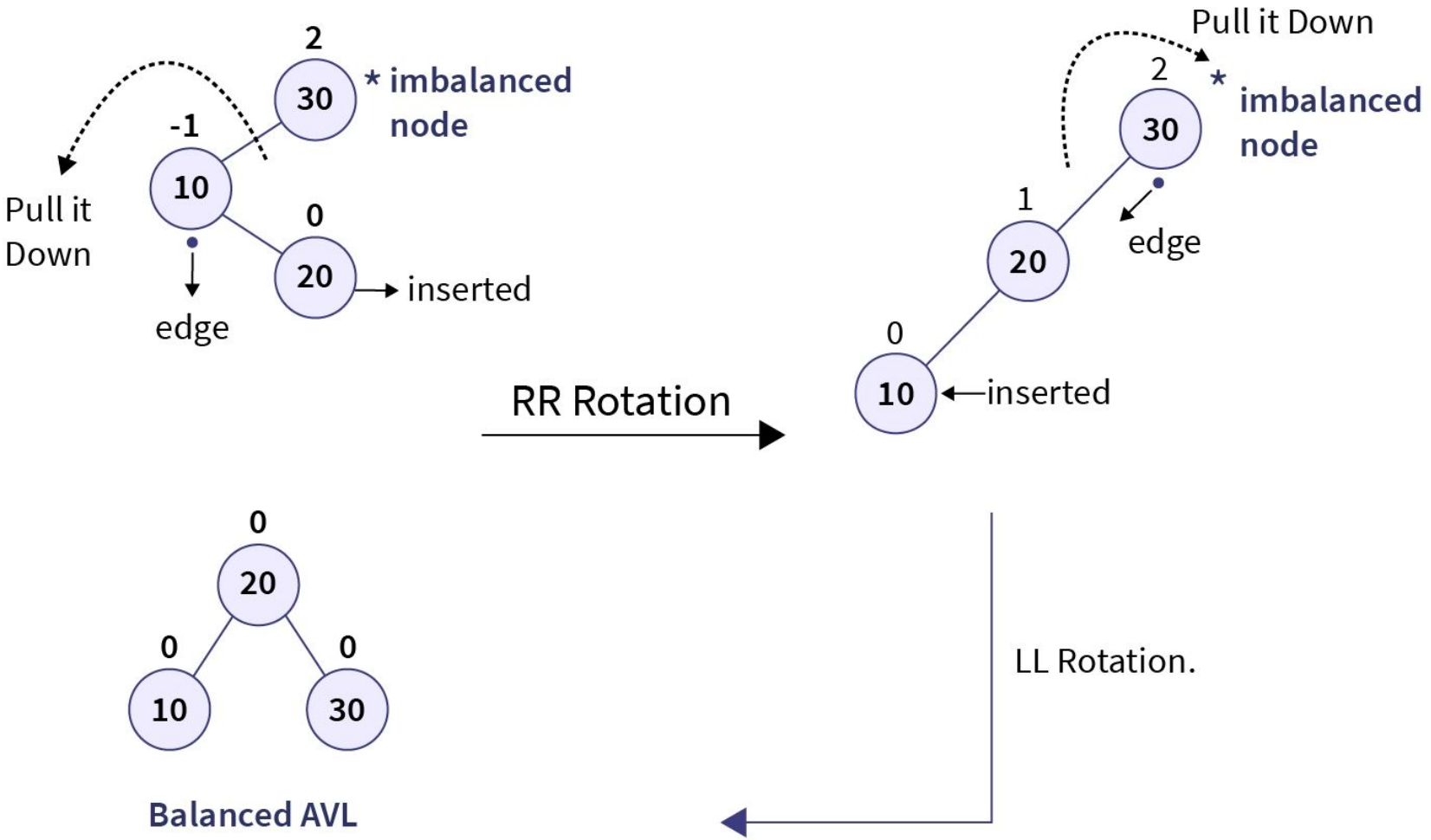
Rotations on AVL Tree

LR Rotation

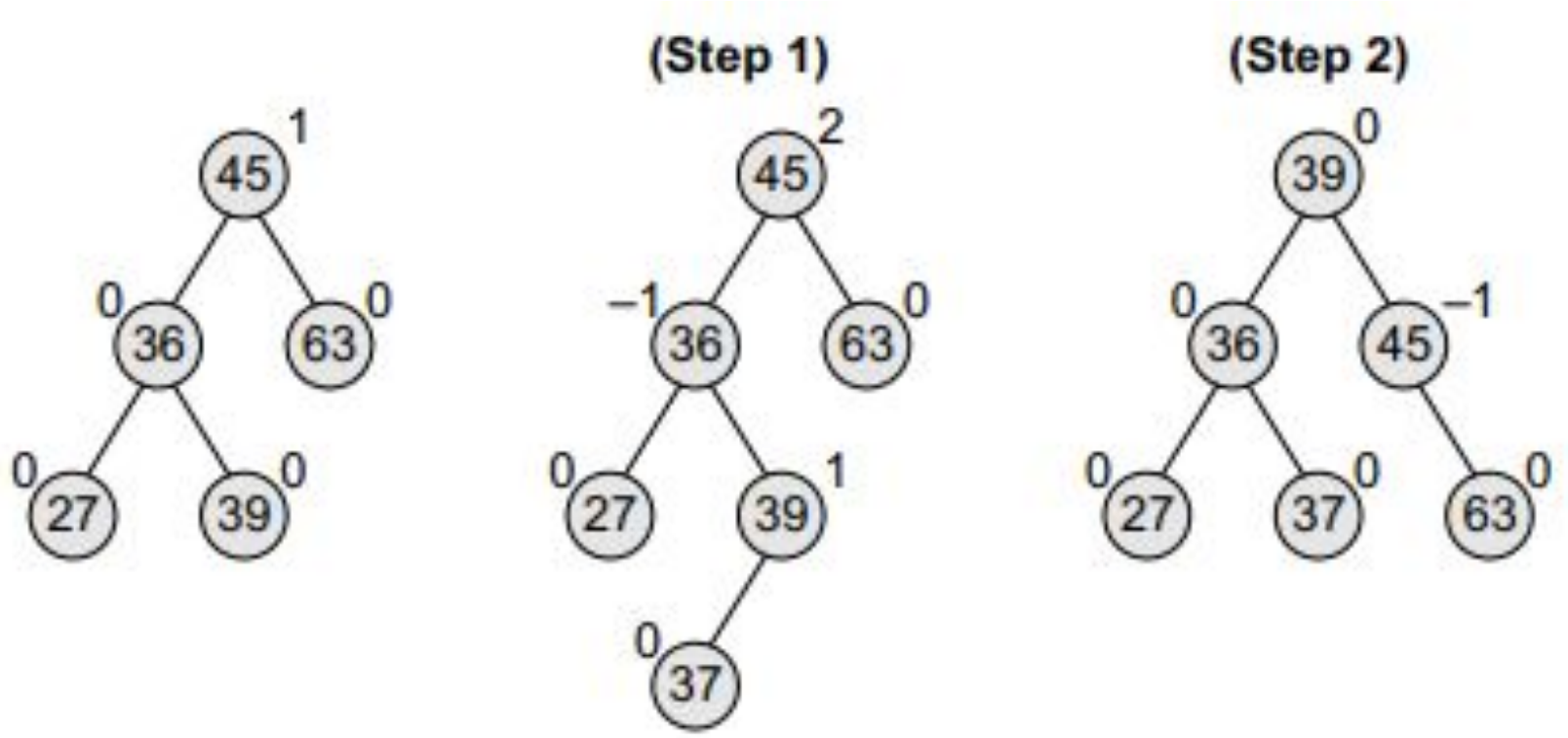
- The LR rotation consists of 2 steps:
  - Apply RR Rotation (anti-clockwise rotation) on the left subtree of the imbalanced node as the left child of the imbalanced node is right-heavy. This process flips the tree and converts it into a left-skewed tree.
  - Perform LL Rotation (clock-wise rotation) on the imbalanced node to balance the left-skewed tree.
- Hence, LR rotation is essentially a combination of RR and LL Rotation.

Rotations on AVL Tree

LR Rotation



LR Rotation



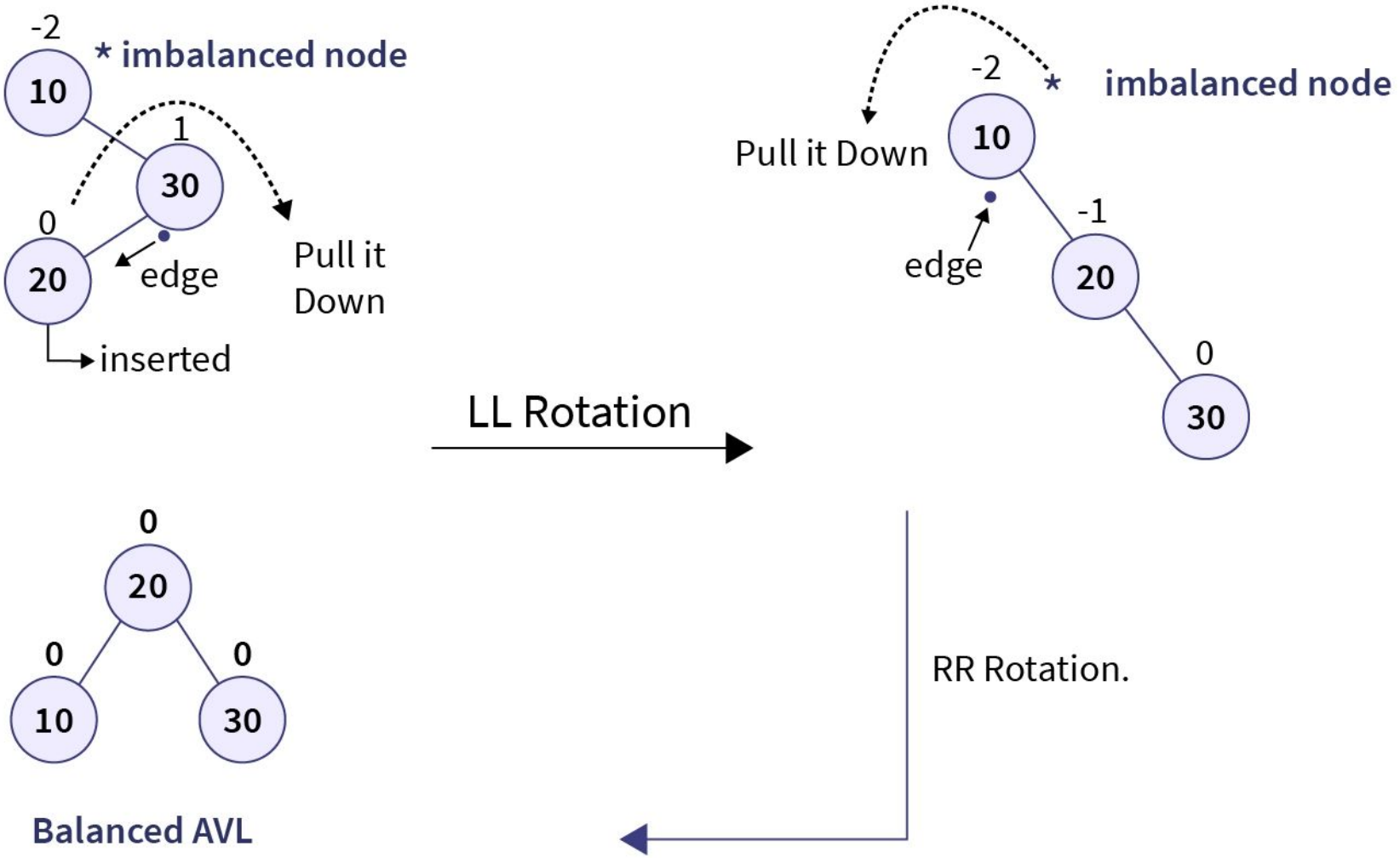
Rotations on AVL Tree

RL Rotation

- It is similar to LR rotation but it is performed when the tree gets unbalanced, upon insertion of a node into the left subtree of the right child of the imbalance node i.e., upon Right-Left (RL) insertion instead of LR insertion.
- In this case, the immediate parent of the inserted node becomes left-heavy i.e., the LL rotation (right rotation or clockwise rotation) is performed that converts the tree into a right-skewed tree.
- After which, RR rotation (left rotation or anti-clockwise rotation) is applied around the edge of the imbalanced node to convert this right-skewed tree into a balanced BST.

Rotations on AVL Tree

RL Rotation



Rotations on AVL Tree

RL Rotation

- In the above example, we can observe that the root node of the tree becomes imbalanced upon insertion of the node having the value 20. Since this is a type of RL insertion, we will perform LL rotation on the immediate parent of the inserted node thereby retrieving a right-skewed tree.
- Finally, we will perform RR Rotation around the edge of the imbalanced node (in this case the root node) to get the balanced AVL tree.

Hence, RL rotation consists of two steps:

1. Apply LL Rotation (clockwise rotation) on the right subtree of the imbalanced node as the right child of the imbalanced node is left-heavy. This process flips the tree and converts it into a right-skewed tree.
2. Perform RR Rotation (anti-clockwise rotation) on the imbalanced node to balance the right-skewed tree.

Notes

- Rotations are done only on three nodes (including the imbalanced node) irrespective of the size of the Binary Search Tree. Hence, in the case of a large tree always focus on the two nodes around the imbalanced node and perform the tree rotations.

- Upon insertion of a new node, if multiple nodes get imbalanced then traverse the ancestors of the inserted node in the tree and perform rotations on the first occurred imbalanced node. Continue this process until the whole tree is balanced. This process is knowns as retracing which is discussed later in the article.

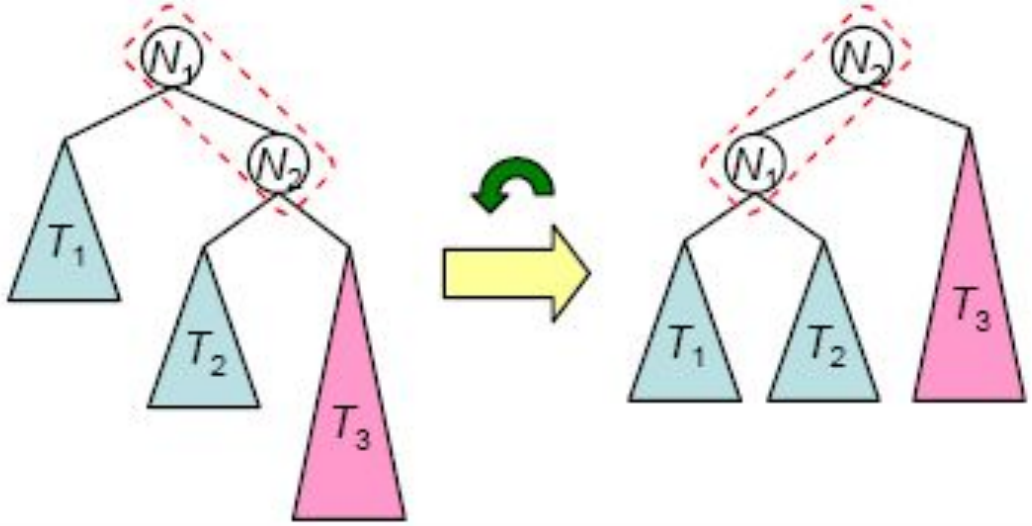
Notes

- Rotations are performed to maintain the AVL Balance criteria.
- Rotation is a process of changing the structure without affecting the elements' order.
- Rotations are done on an unbalanced node based on the location of the newly inserted node.
- Single rotations include LL (clockwise) and RR (anti-clockwise) rotations.
- Double rotations include LR (RR + LL) and RL (LL + RR) rotations.
- Rotations are done only on 3 nodes, including the unbalanced node.

Left-Rotation

Case 1: insertion to *right* subtree of *right* child

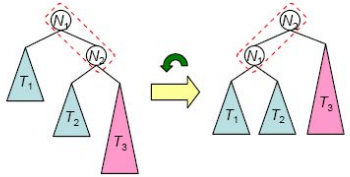
Solution: *Left* rotation

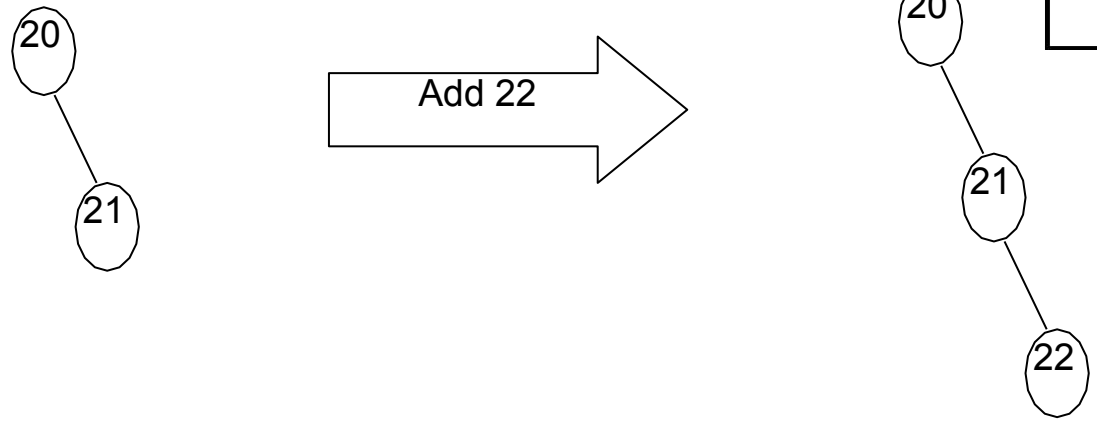


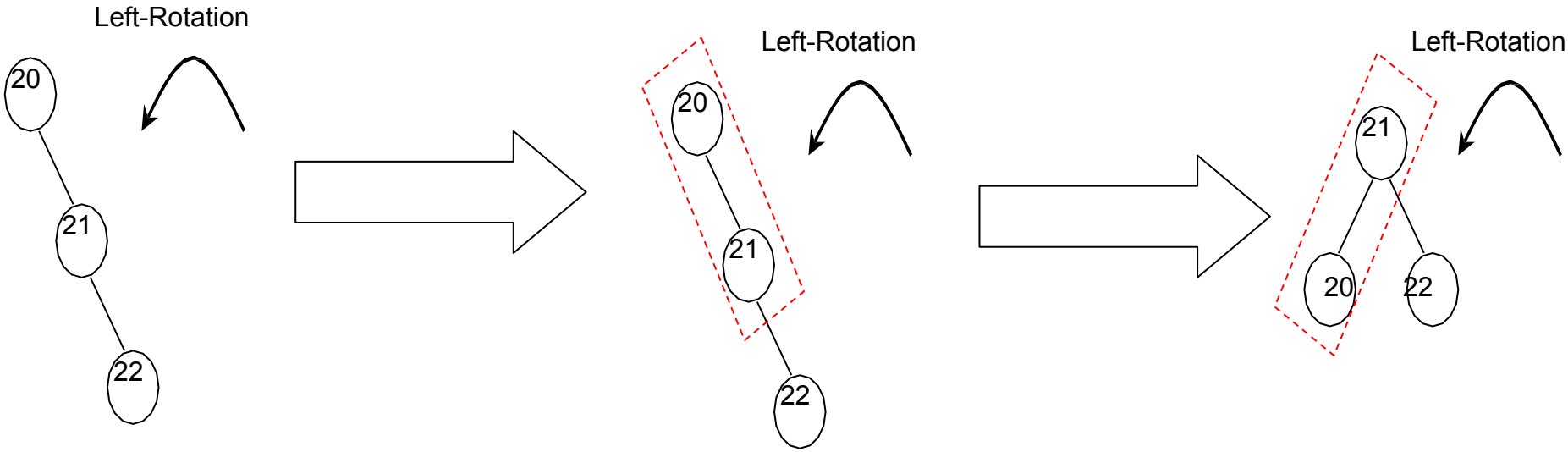
Left-Rotation

Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



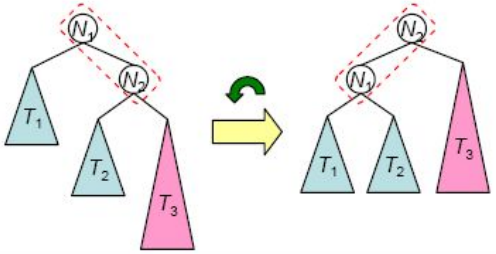


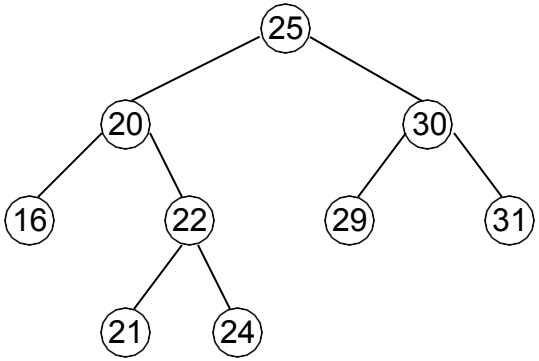


Left-Rotation

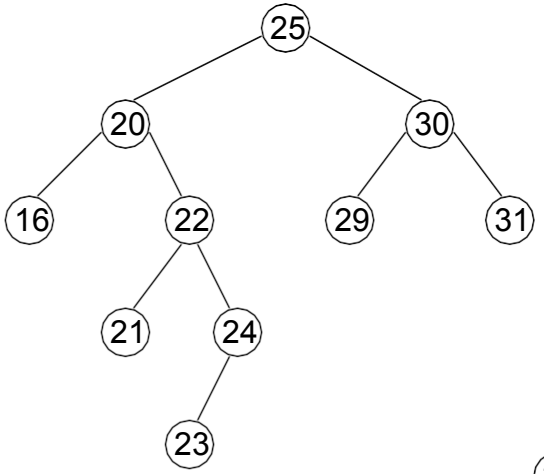
Case 1: insertion to *right* subtree of *right* child

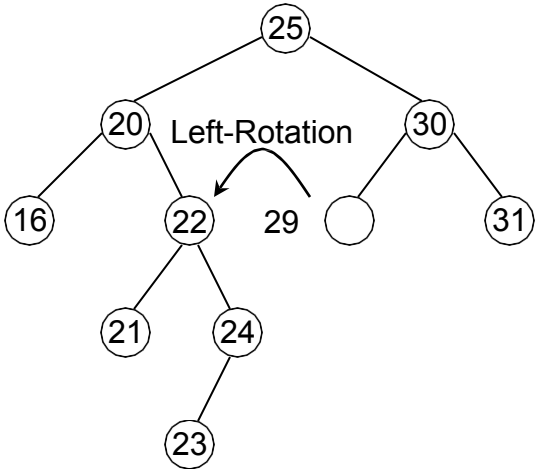
Solution: *Left* rotation

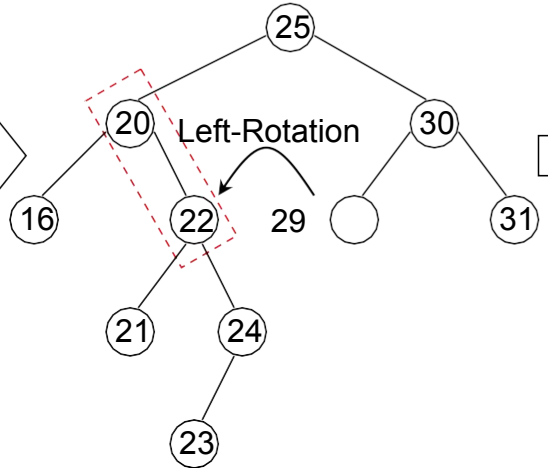


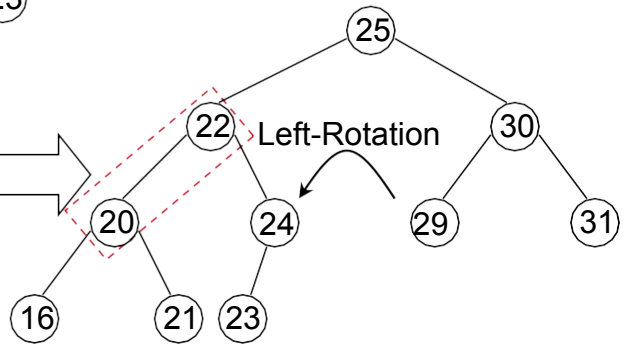


Add 23



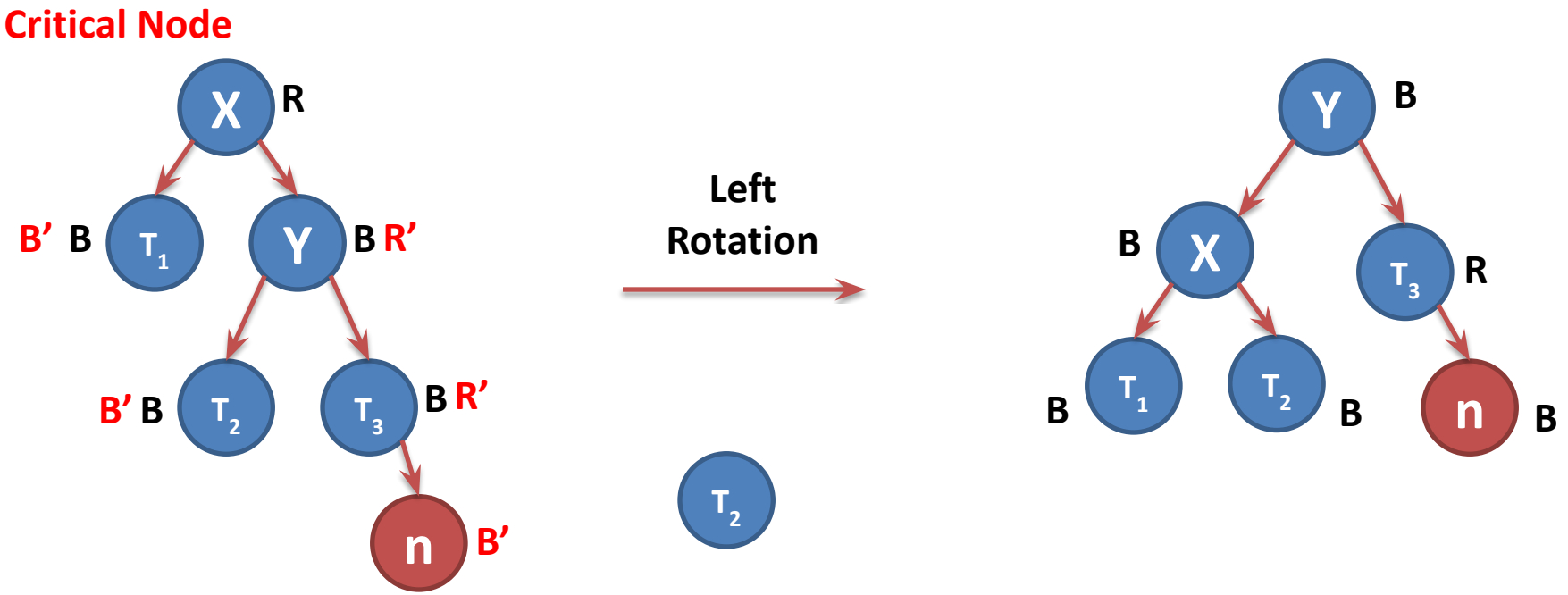






Left Rotations

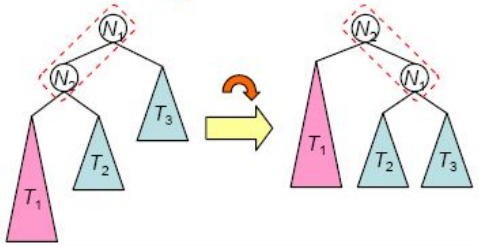
a) **Detach** right child's leaf sub-tree
b) Consider **right child** to be the **new parent**
c) **Attach old parent** onto **left of new parent**
d) **Attach old right child's old left sub-tree** as **right sub-tree of new left child.**

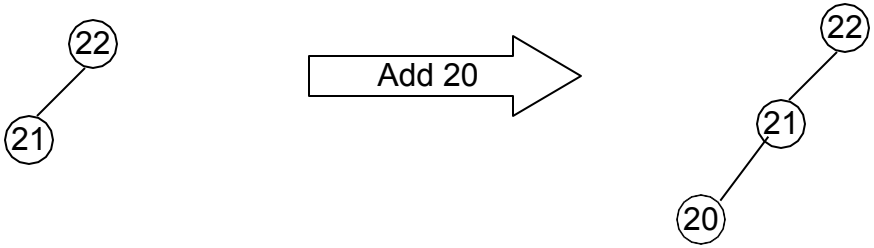


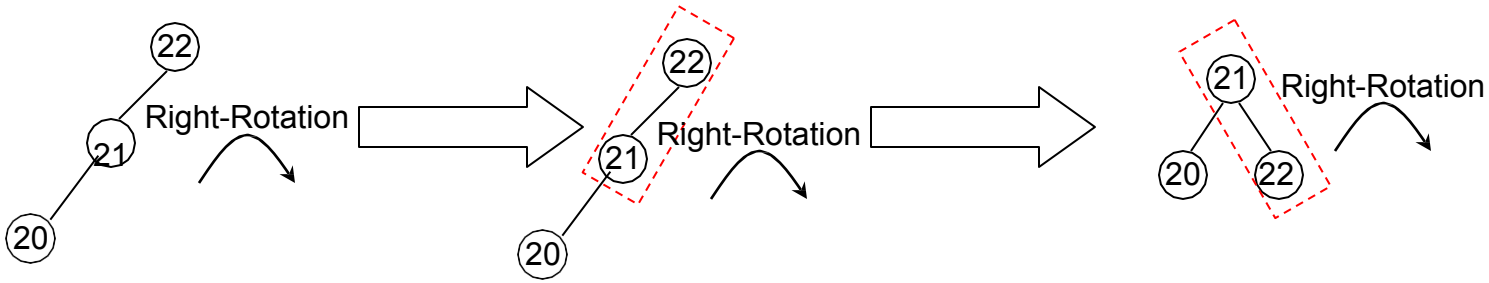
Right-Rotation

Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



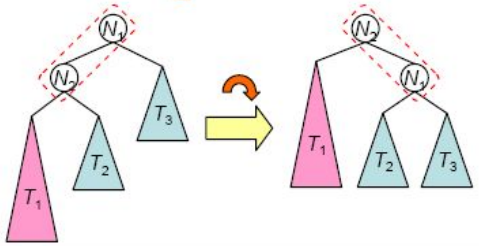


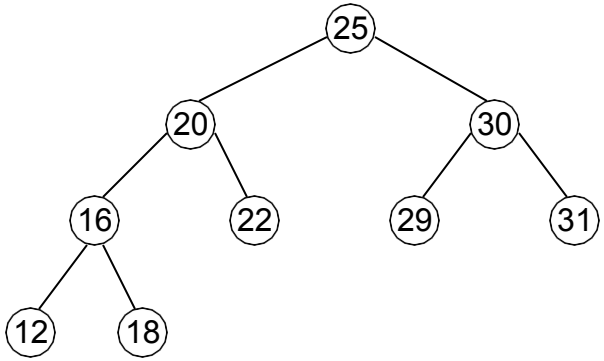


Right-Rotation

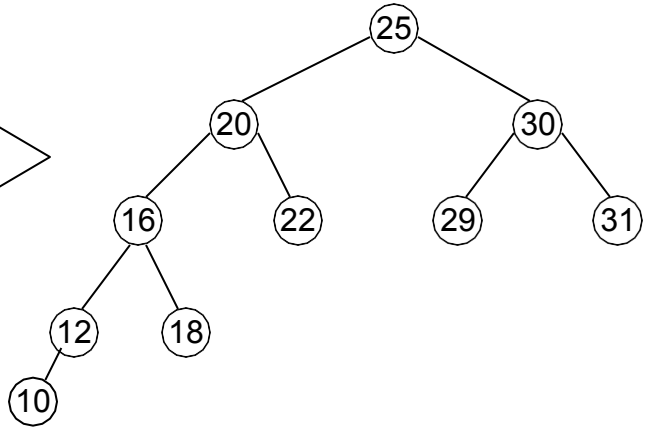
Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation

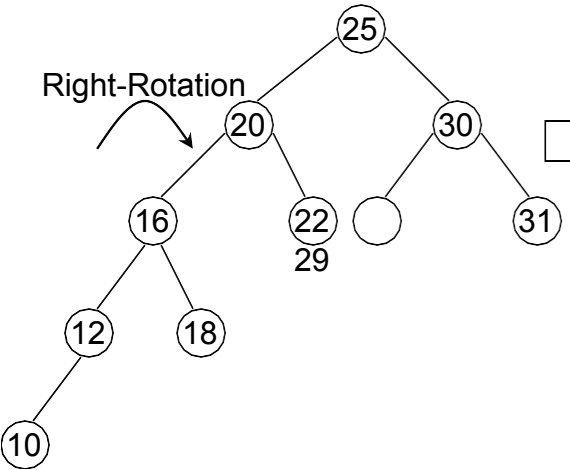




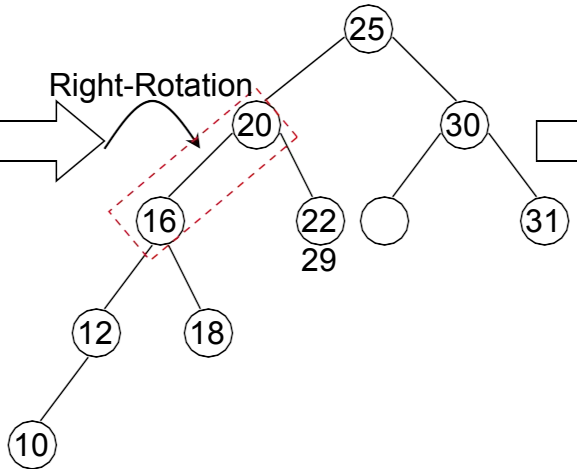
Add 10



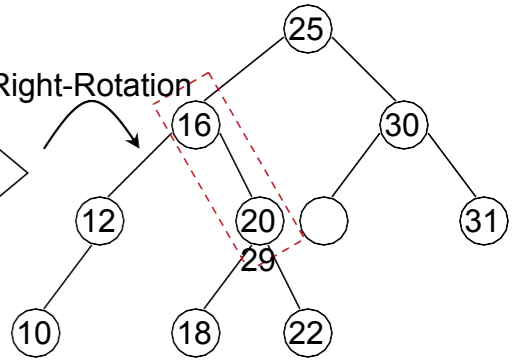
Right-Rotation



Right-Rotation

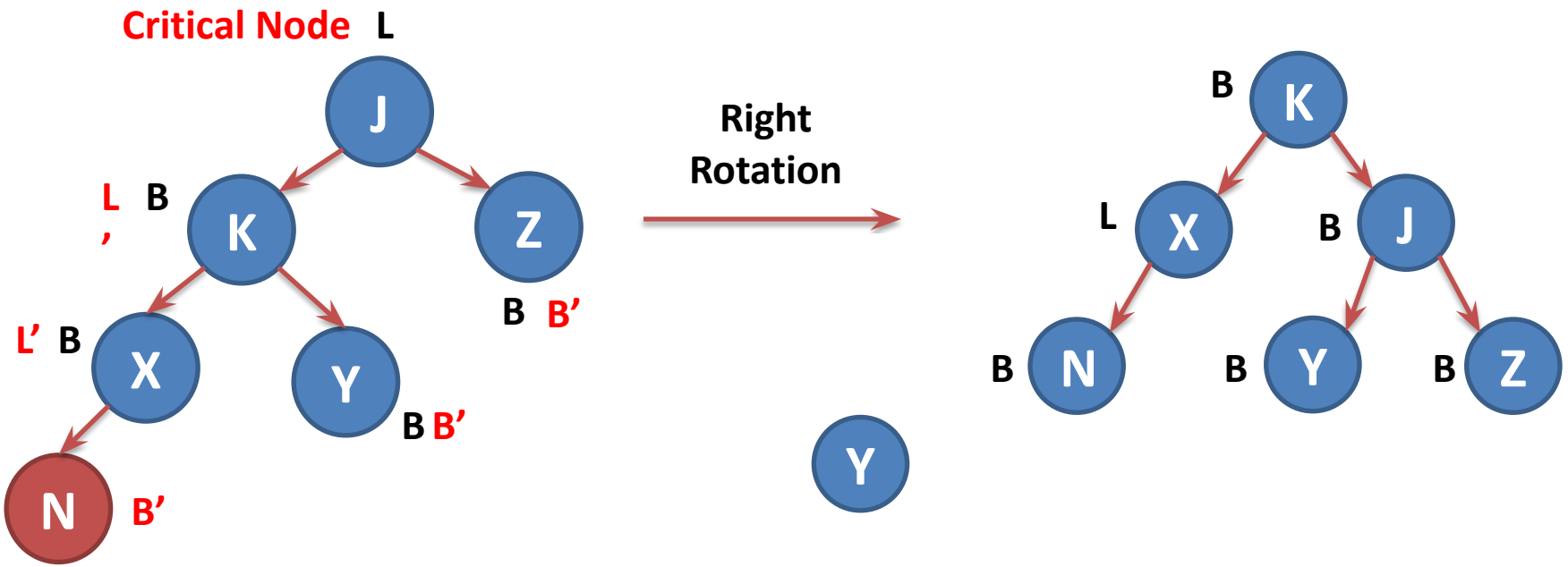


Right-Rotation



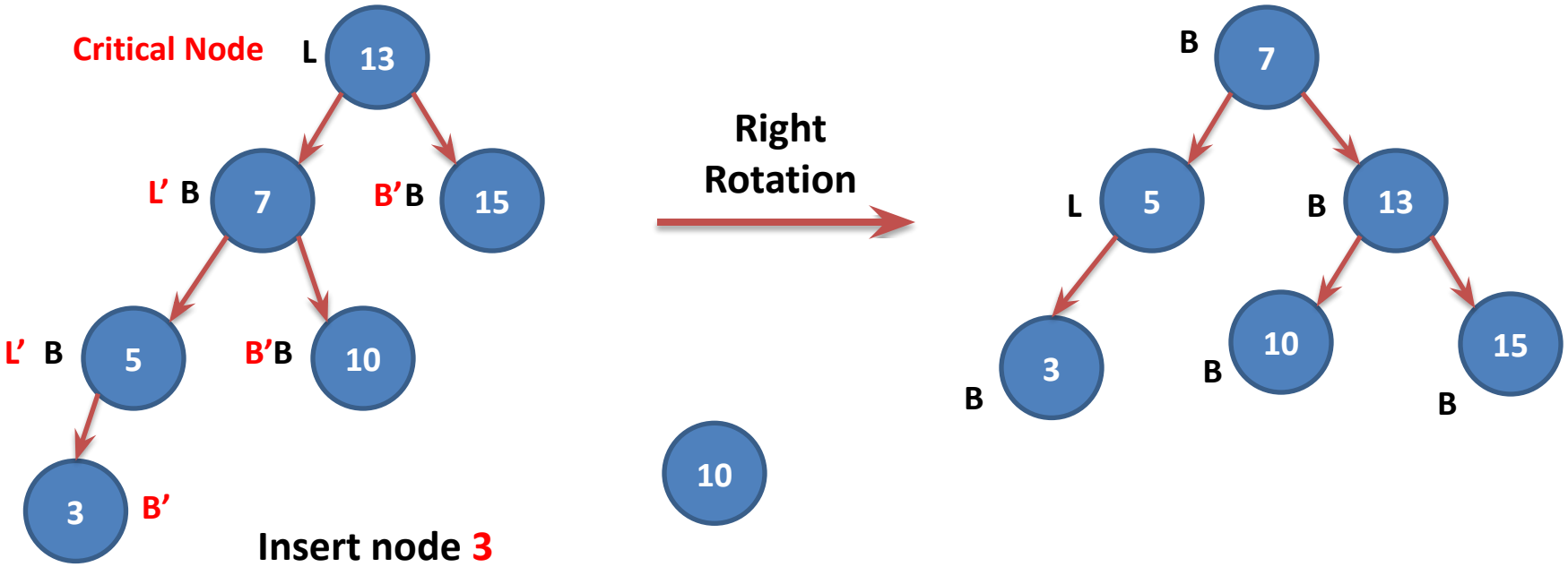
Right Rotations

a) **Detach** left child's right sub-tree
b) Consider **left child** to be the **new parent**
c) **Attach old parent** onto **right of new parent**
d) **Attach old left child's old right sub-tree** as **left sub-tree of new right child.**



Right Rotations

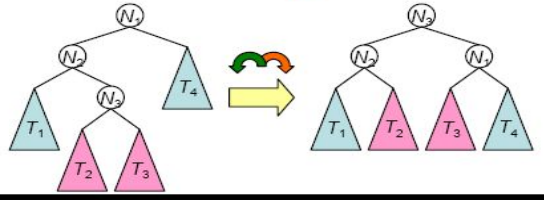
a) **Detach** left child's right sub-tree
b) Consider **left child** to be the **new parent**
c) **Attach old parent** onto **right of new parent**
d) **Attach old left child's old right sub-tree** as **left sub-tree of new right child.**

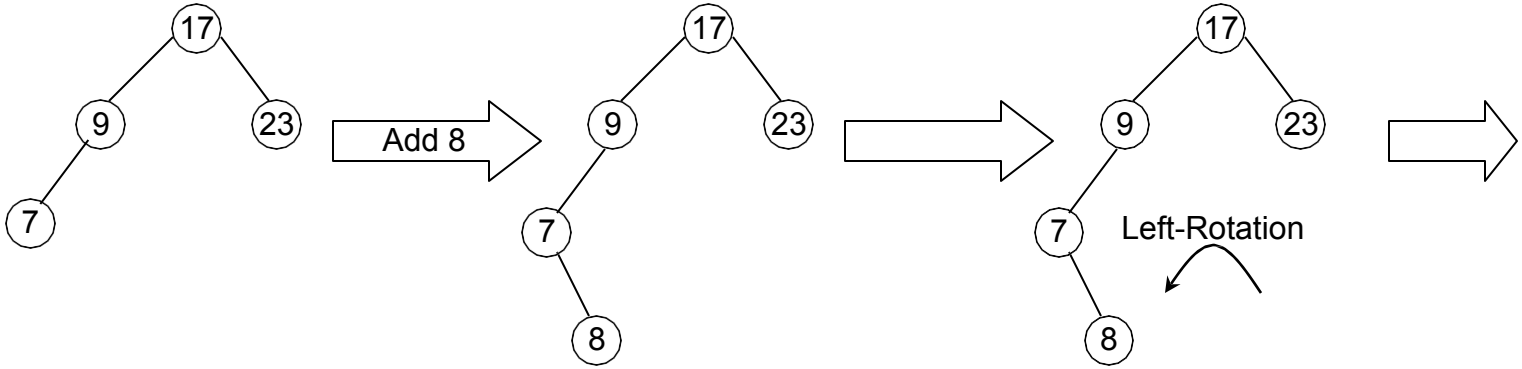


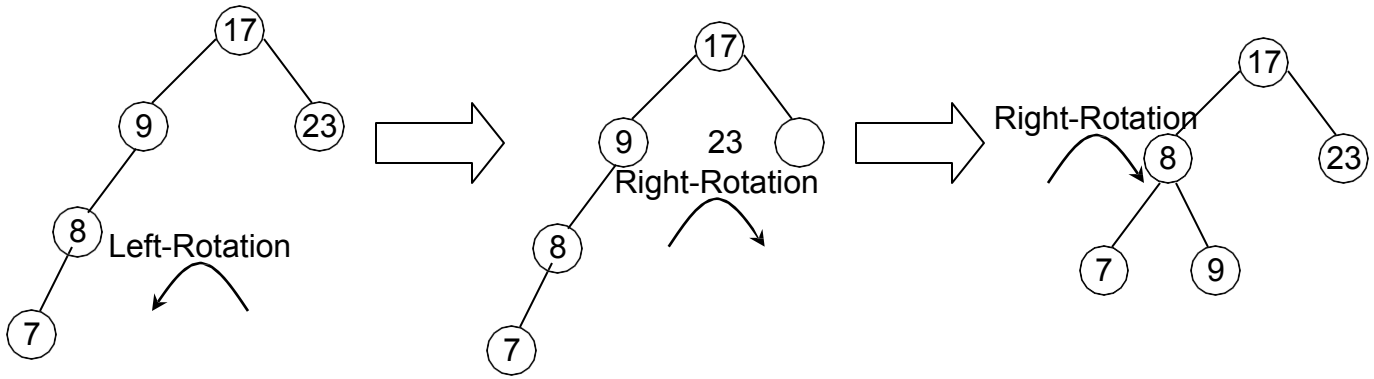
Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

Solution: *Left-right* rotation



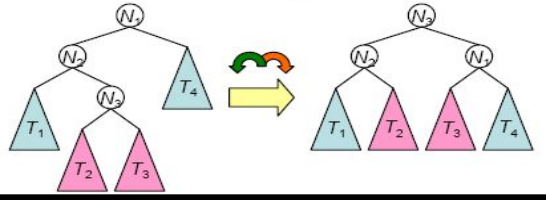


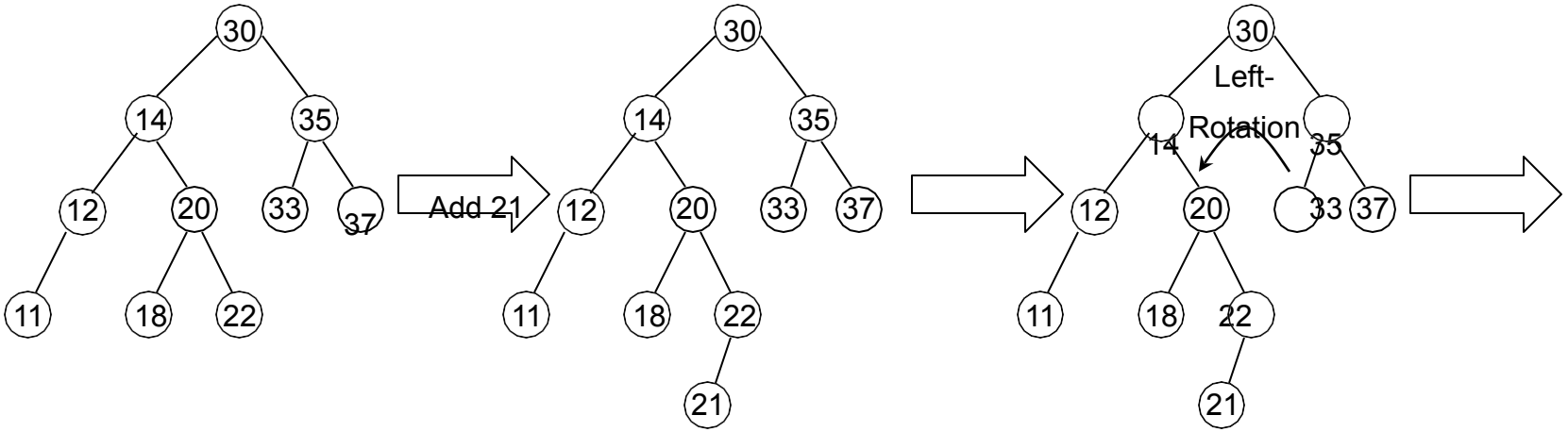


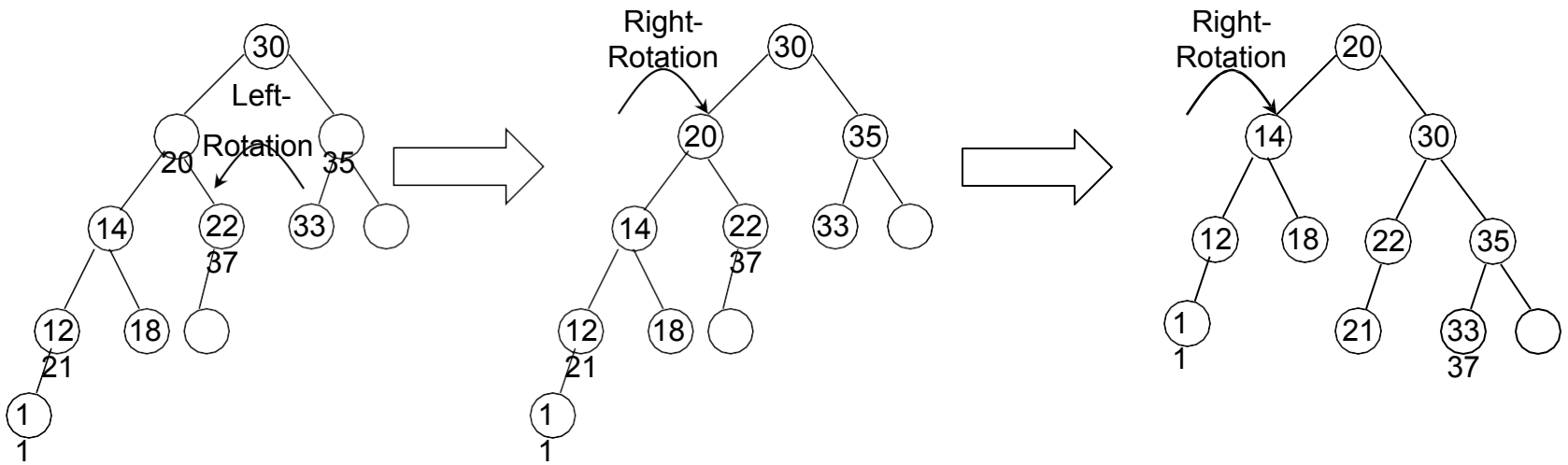
Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

Solution: *Left-right* rotation



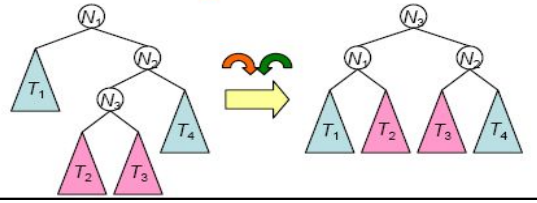




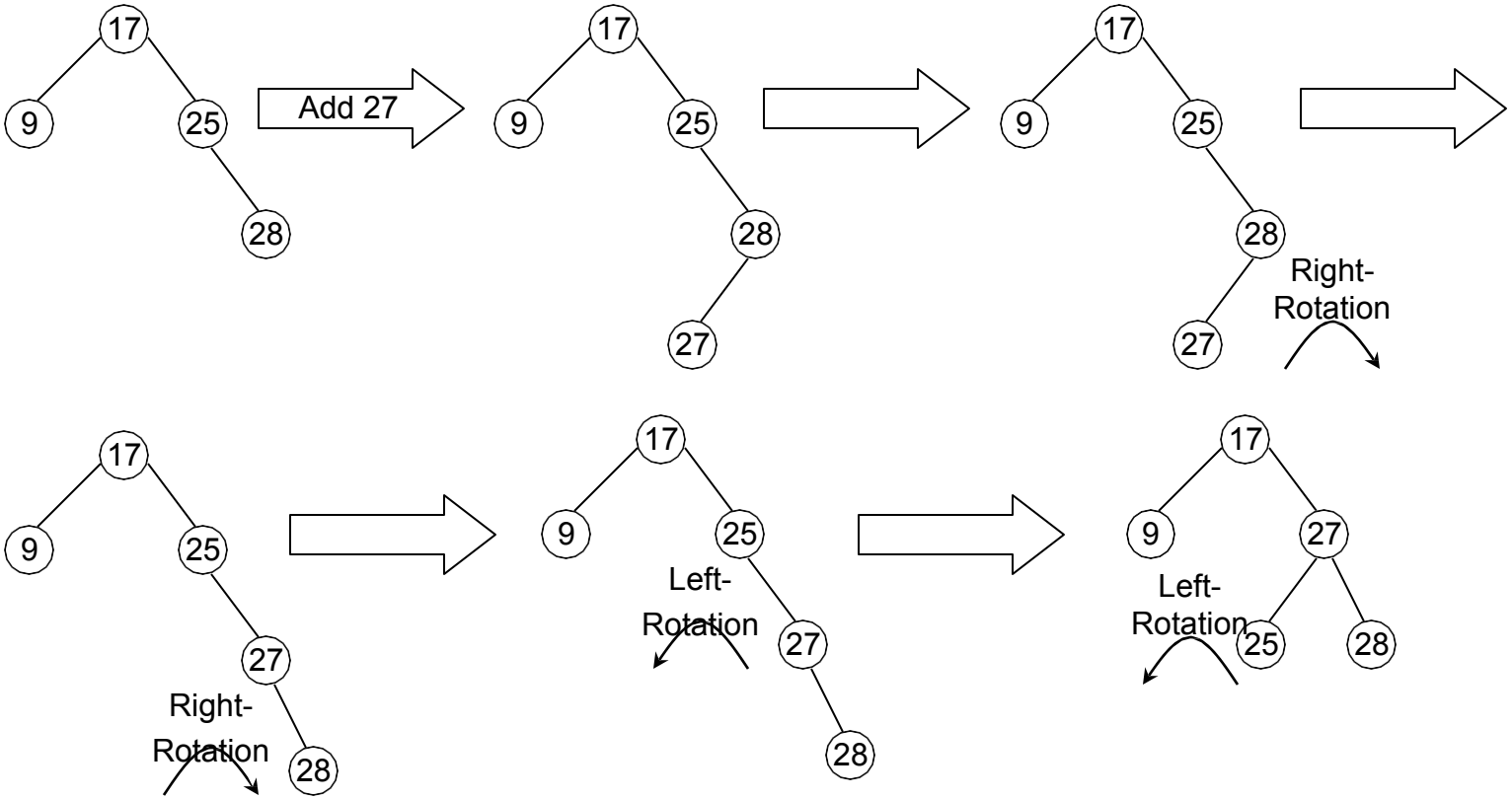
Right-Left Rotation

Case 4: insertion to *left* subtree of *right* child

Solution: *Right-left* rotation



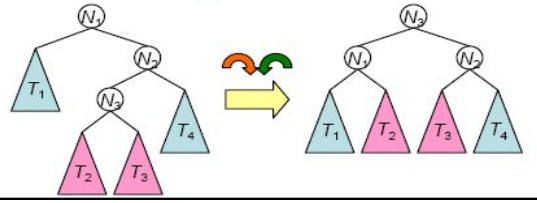
35



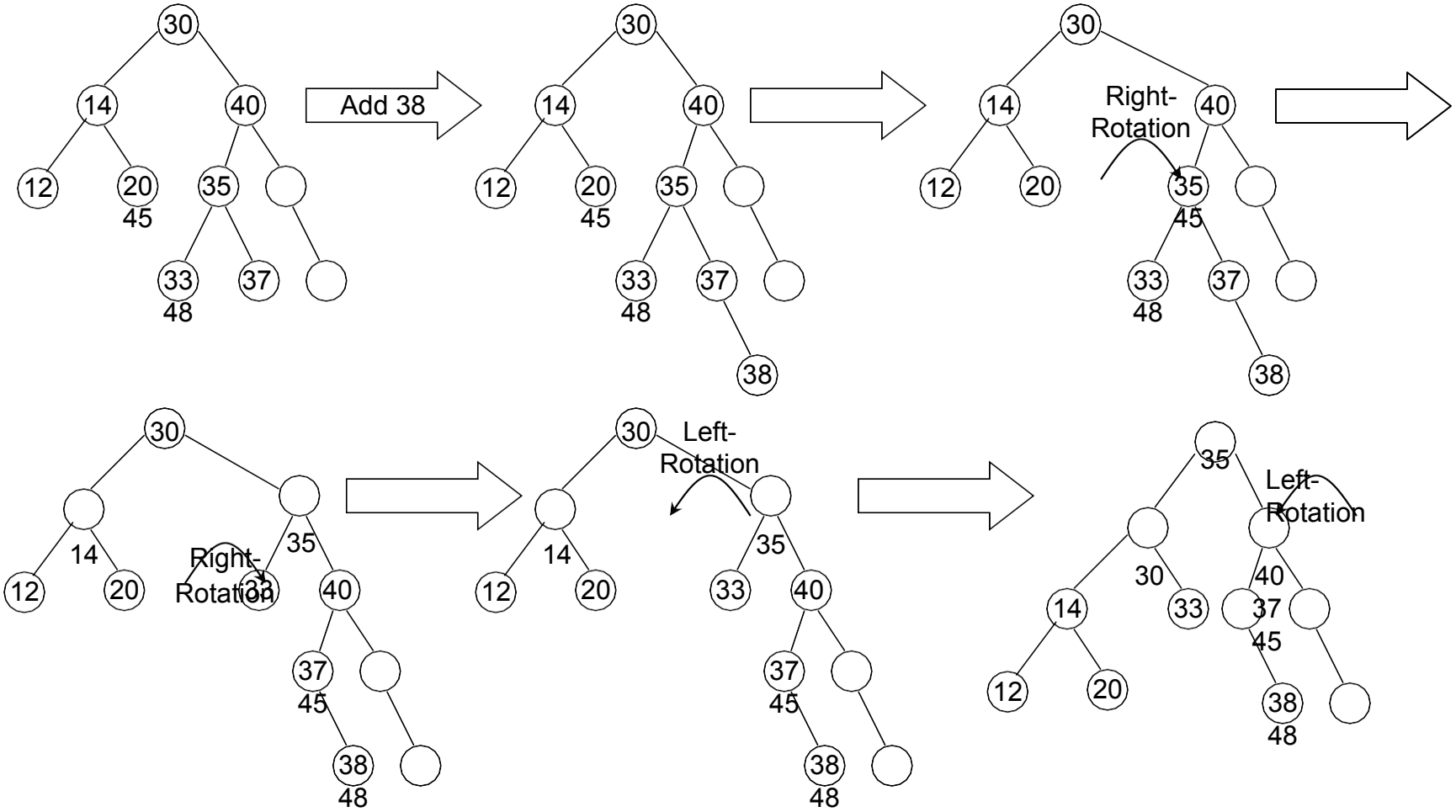
Right-Left Rotation

Case 4: insertion to *left* subtree of *right* child

Solution: *Right-left* rotation

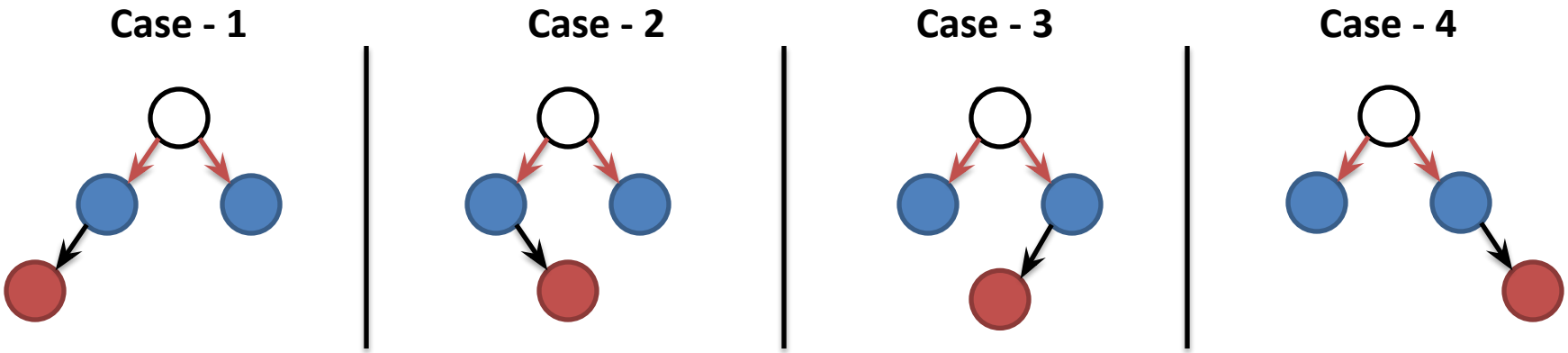


35



Select Rotation based on Insertion Position

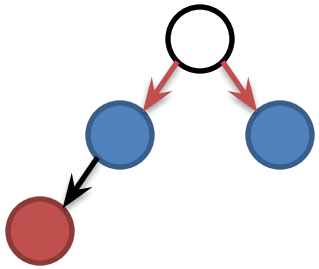
- Case 1: Insertion into **Left sub-tree** of **nodes Left child**
  - Single Right Rotation
- Case 2: Insertion into **Right sub-tree** of **node's Left child**
  - Left Right Rotation
- Case 3: Insertion into **Left sub-tree** of **node's Right child**
  - Right Left Rotation
- Case 4: Insertion into **Right sub-tree** of **node's Right child**
  - Single Left Rotation



Insertion into Left sub-tree of nodes Left child

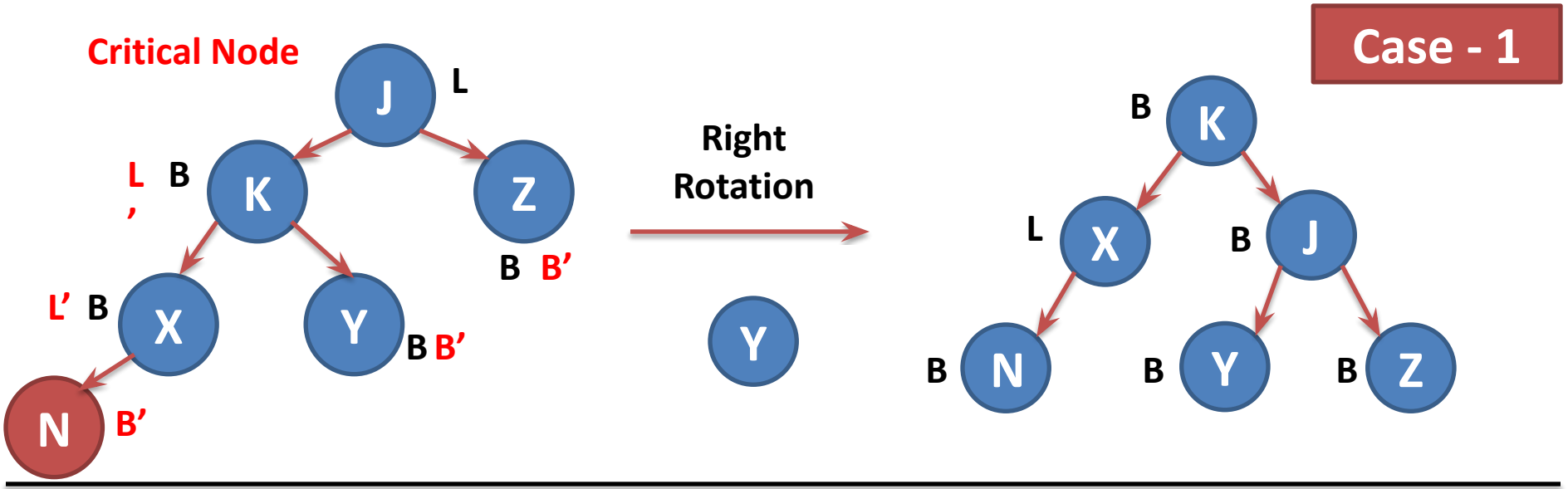
- Case 1: If node becomes **unbalanced** after **insertion** of new node at **Left sub-tree** of nodes **Left child**, then we need to perform **Single Right Rotation** of **unbalanced node** to balance the node Right Rotation
  a) Detach leaf child's right sub-tree
  b) Consider leaf child to be the new parent
  c) Attach old parent onto right of new parent
  d) Attach old leaf child's old right sub-tree as leaf sub-tree of new right child

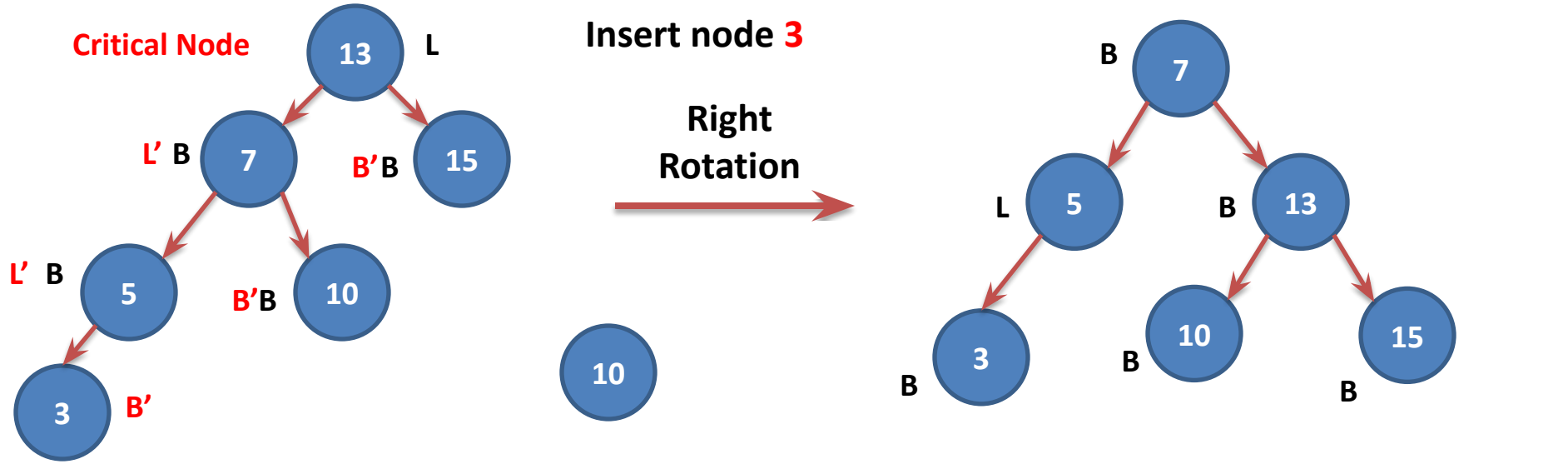
Case - 1



Single Right Rotation
of
unbalanced node

Insertion into Left sub-tree of nodes Left child

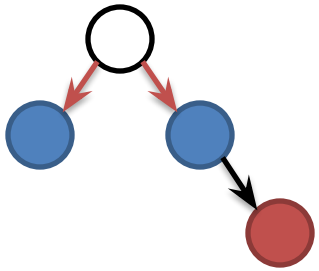




Insertion into Right sub-tree of node's Right child

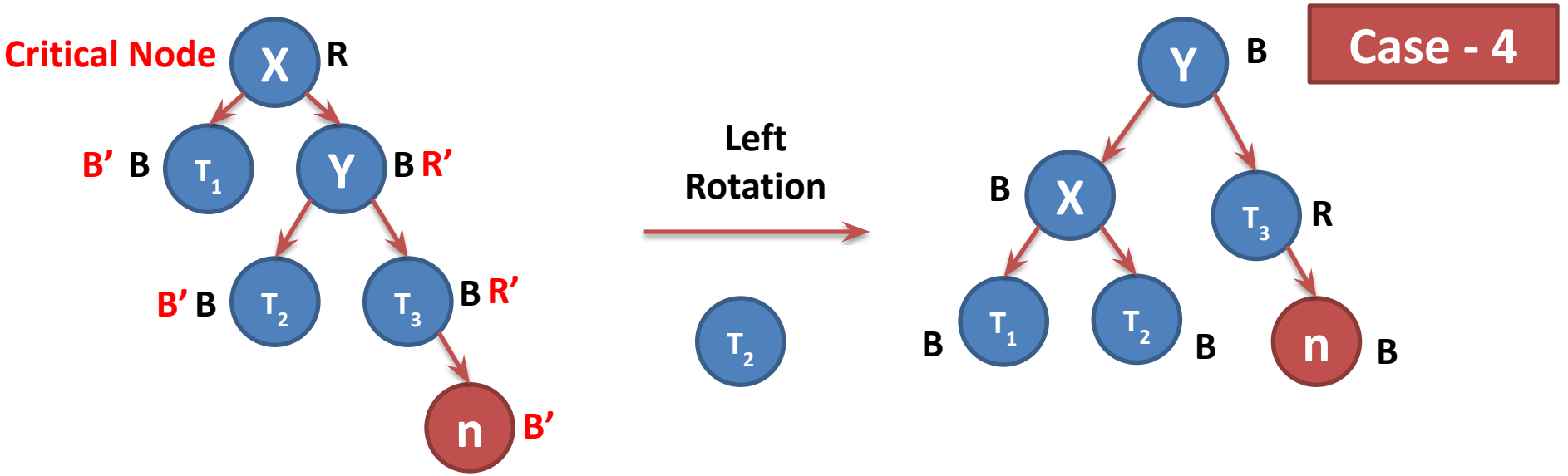
- Case 4: If node becomes **unbalanced** after **insertion of new node** at **Right sub-tree** of **nodes Right child**, then we need to perform **Single Left Rotation** of **unbalance node** to balance the node Left Rotation
  a) Detach right child's leaf sub-tree
  b) Consider right child to be new parent
  c) Attach old parent onto left of new parent
  d) Attach old right child's old left sub-tree as right sub-tree of new left child

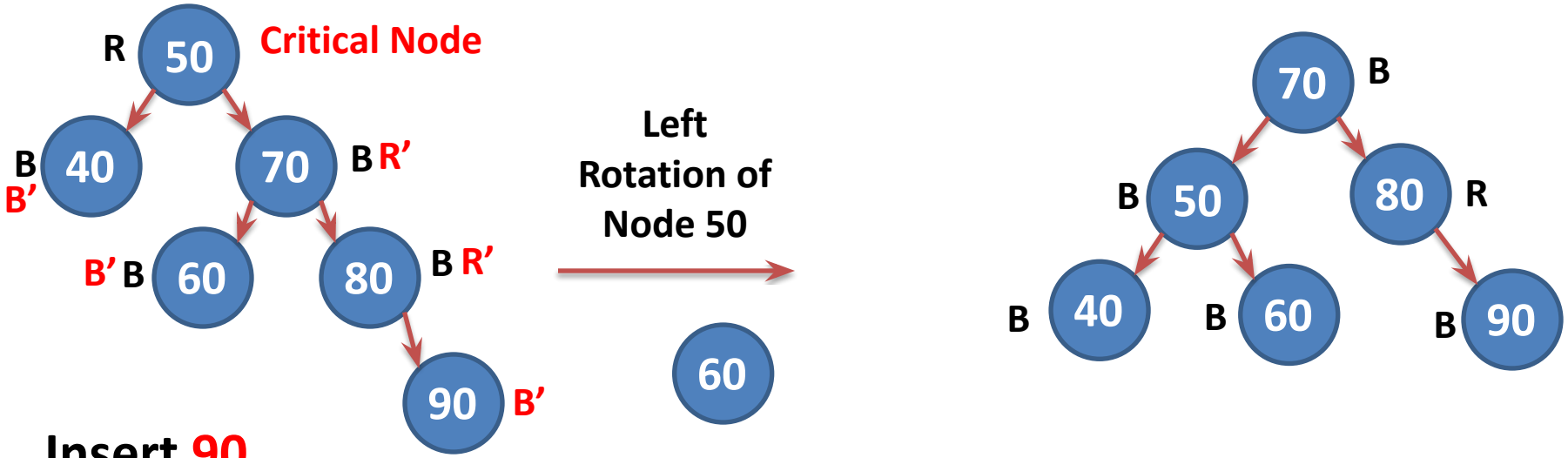
Case - 4



Single Left Rotation
of
unbalanced node

Insertion into Right sub-tree of node's Right child

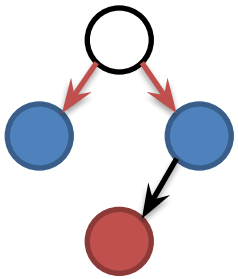




Insertion into Right sub-tree of node's Left child

- Case 2: If node becomes unbalanced **after insertion** of **new node** at **Right sub-tree** of **node's Left child**, then we need to perform **Left Right Rotation** for **unbalanced node**.
- Left Right Rotation
  - **Left Rotation** of **Left Child** followed by
  - **Right Rotation** of **Parent**

Case - 3

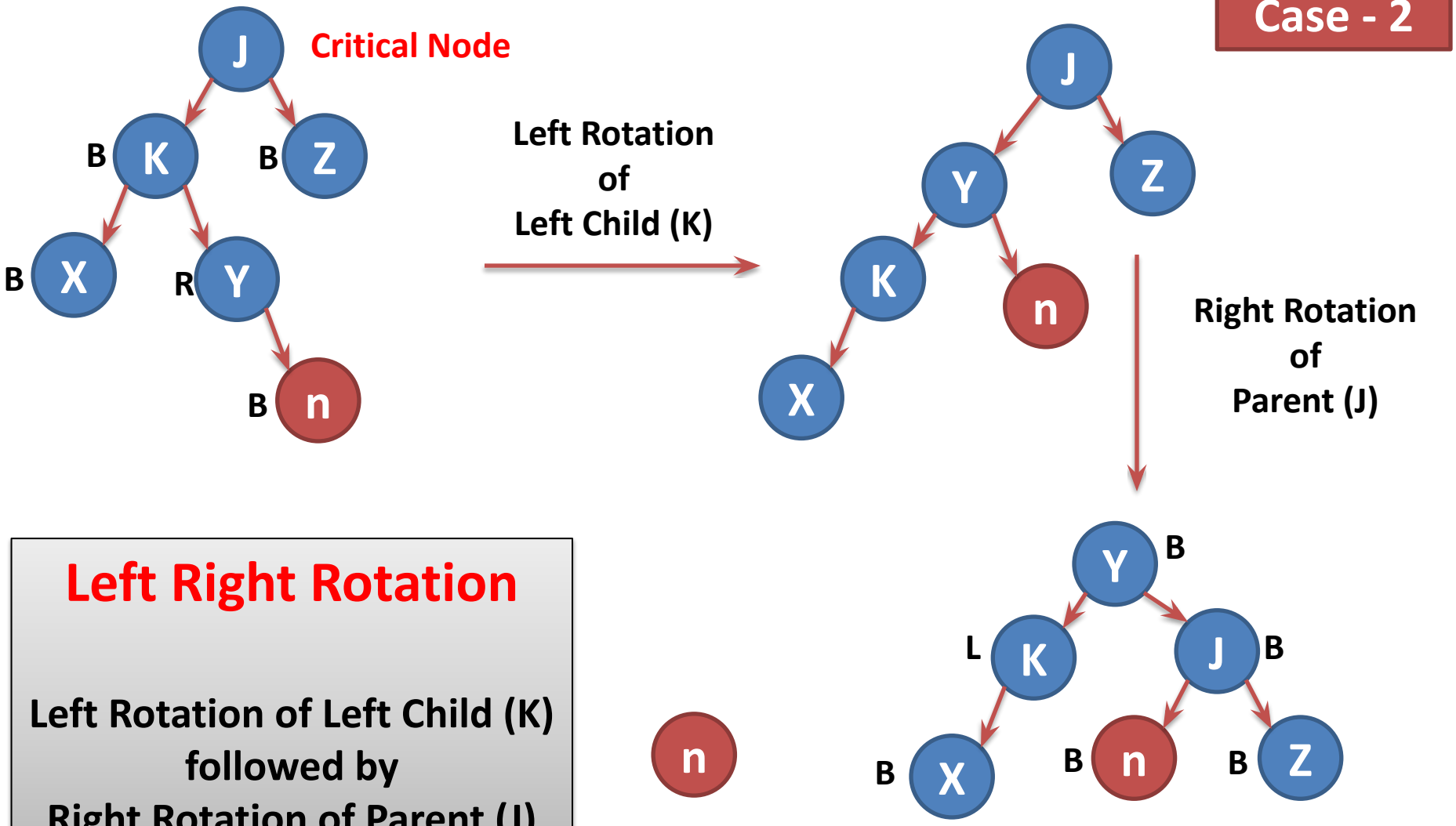


Left Right Rotation

Left Rotation of Left Child
followed by
Right Rotation of Parent

Insertion into Right sub-tree of node's Left child

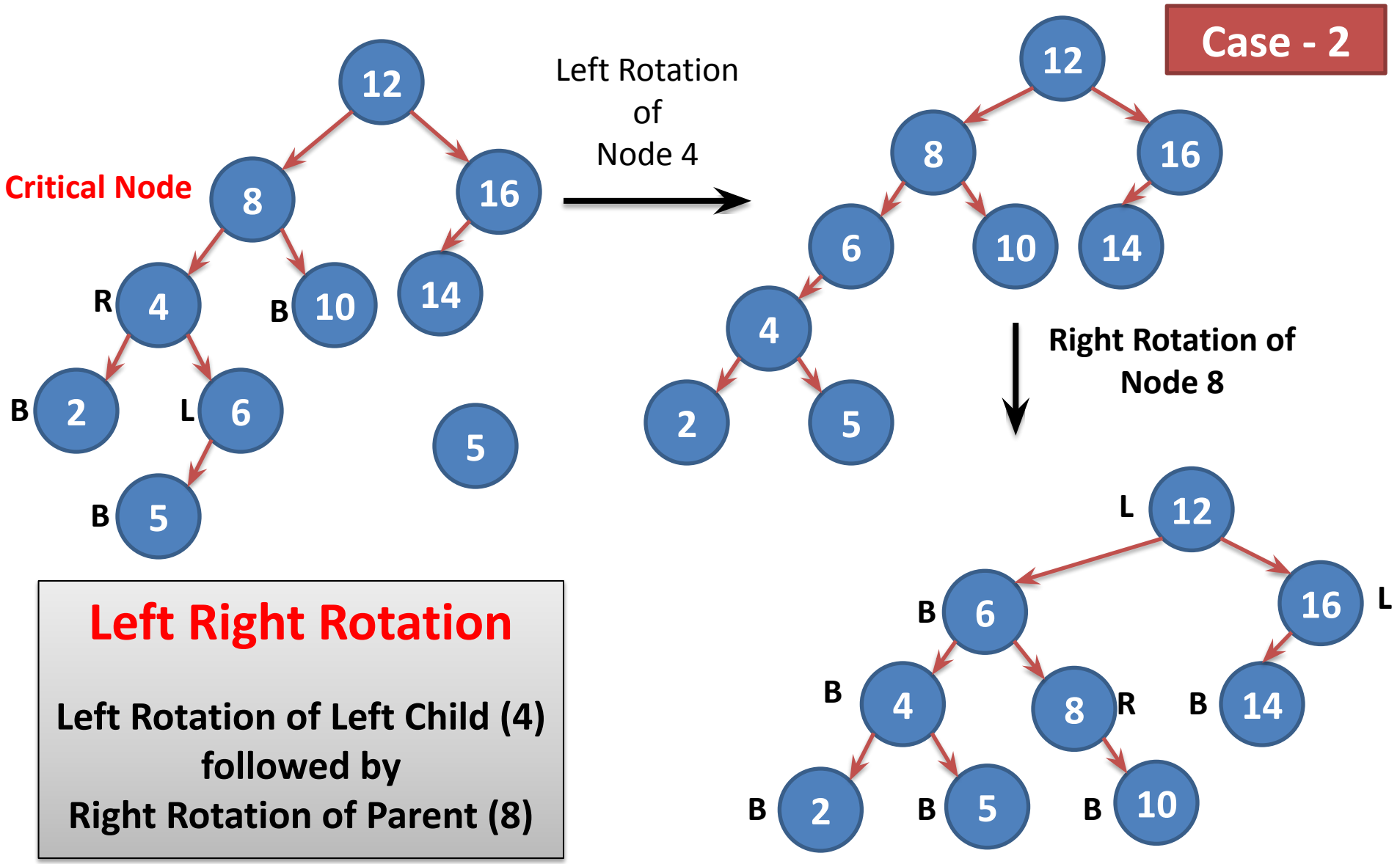
Case - 2



Left Right Rotation

Left Rotation of Left Child (K)
followed by
Right Rotation of Parent (J)

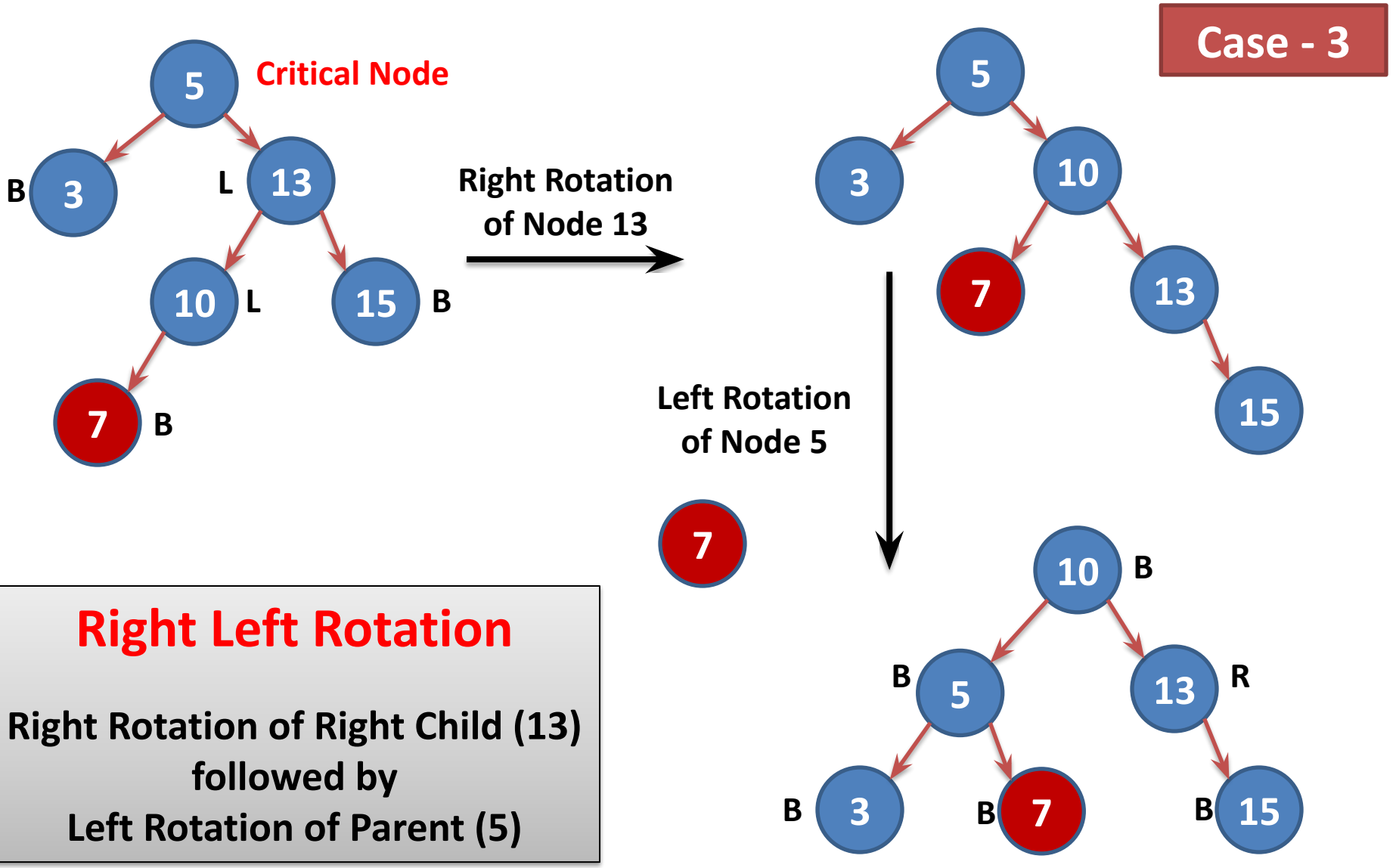
Insertion into Right sub-tree of node's Left child



Left Right Rotation

Left Rotation of Left Child (4)
followed by
Right Rotation of Parent (8)

Insertion into Left sub-tree of node's Right child

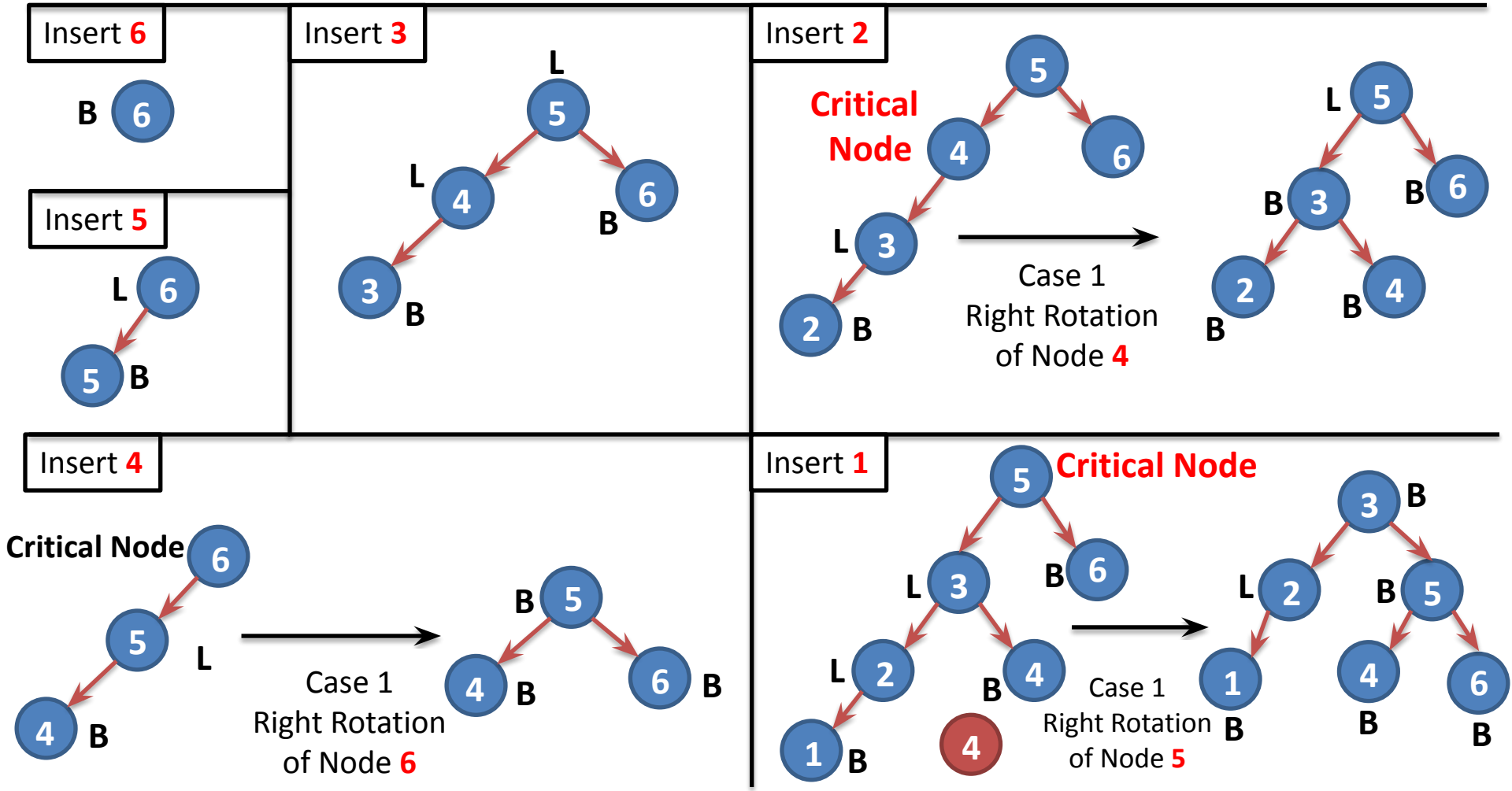


Right Left Rotation

Right Rotation of Right Child (13)
followed by
Left Rotation of Parent (5)

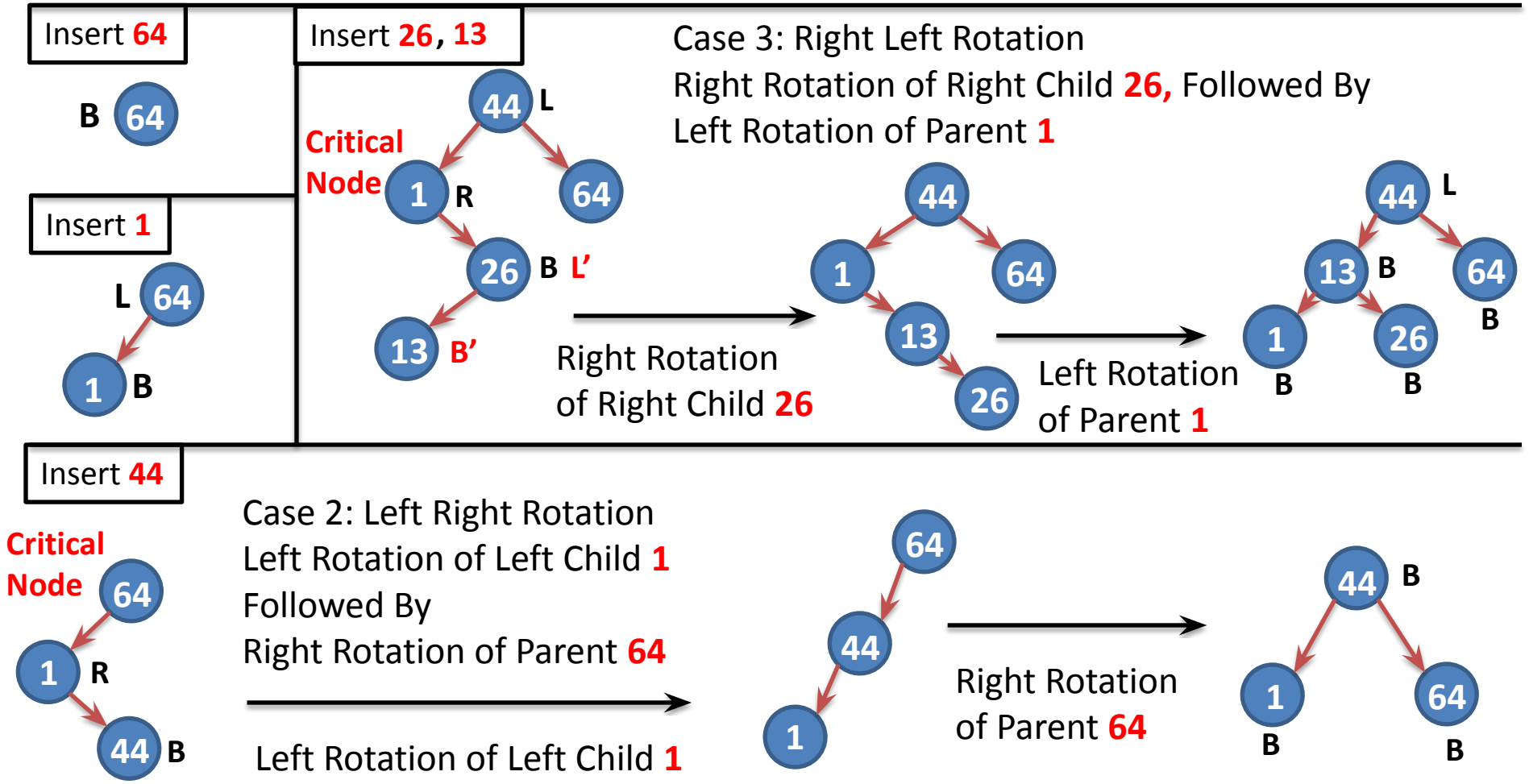
Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **6, 5, 4, 3, 2, 1**



Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

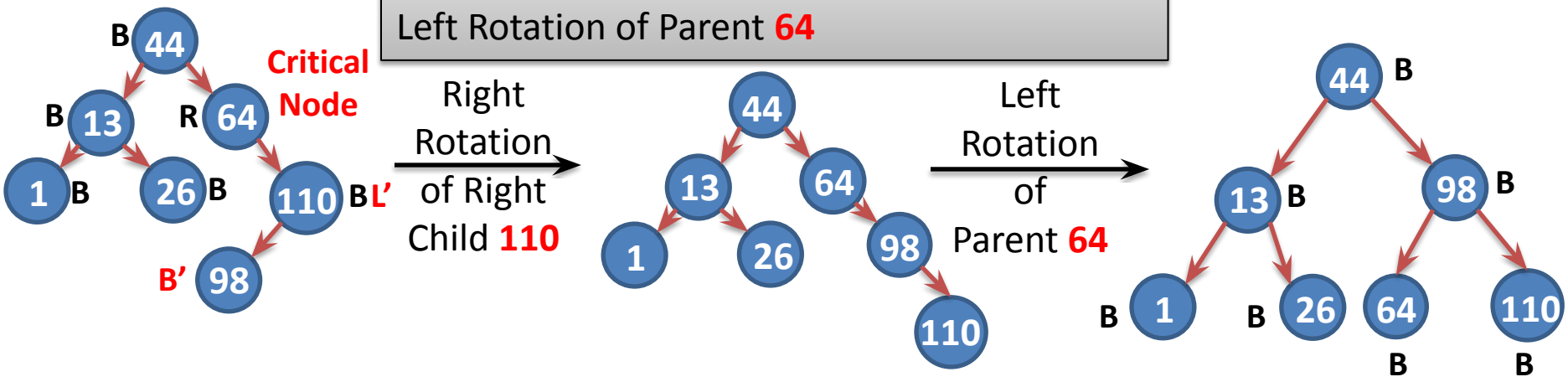


Construct AVL Search Tree

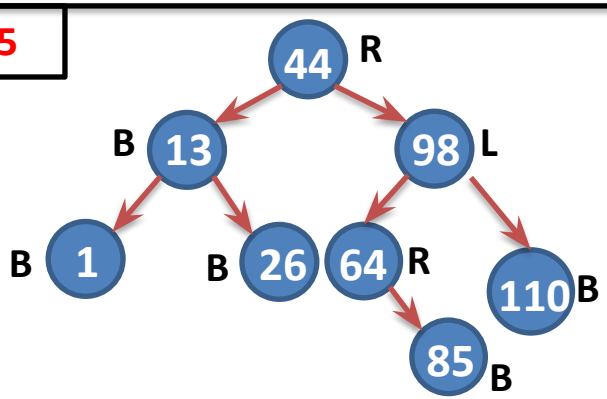
Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

Insert **110**, **98**

Case 3: Right Left Rotation
Right Rotation of Right Child **110**, Followed By
Left Rotation of Parent **64**

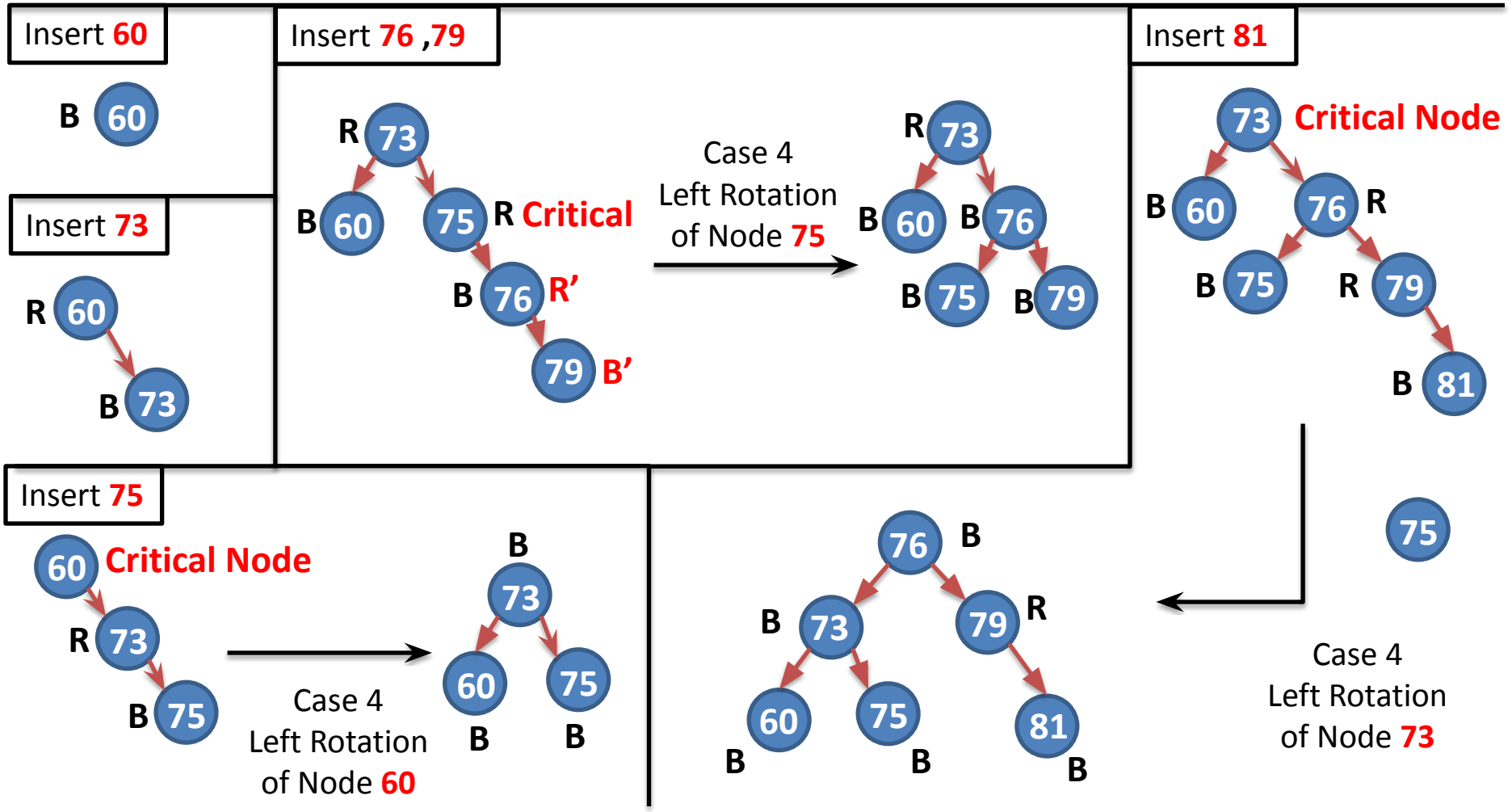


Insert **85**



Construct AVL Search Tree

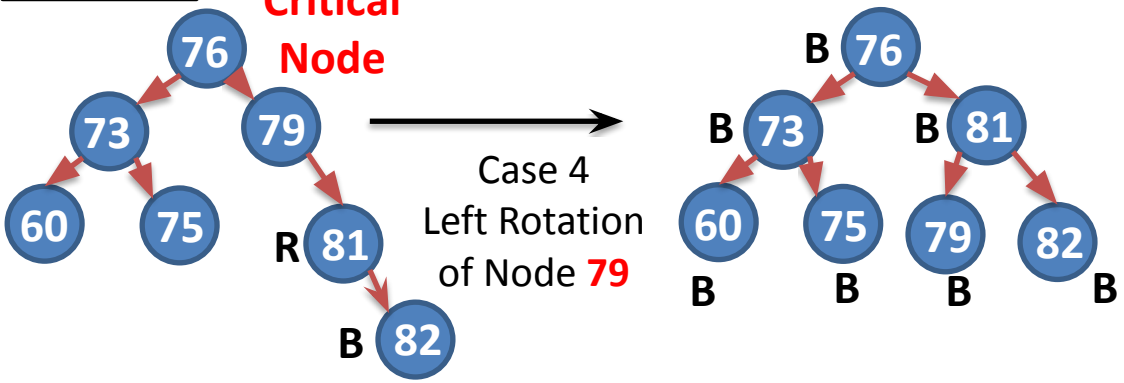
Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,75,82,300,0,5,73**



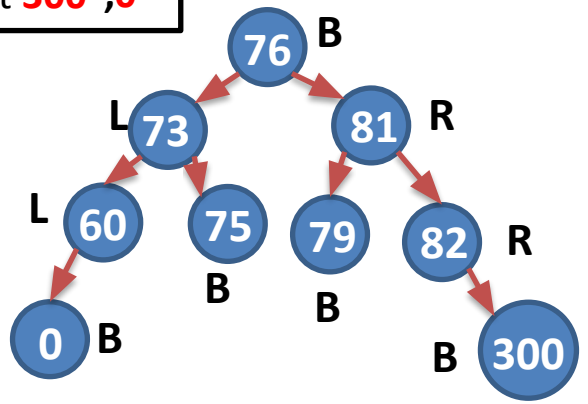
Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,75,82,300,0,5,73**

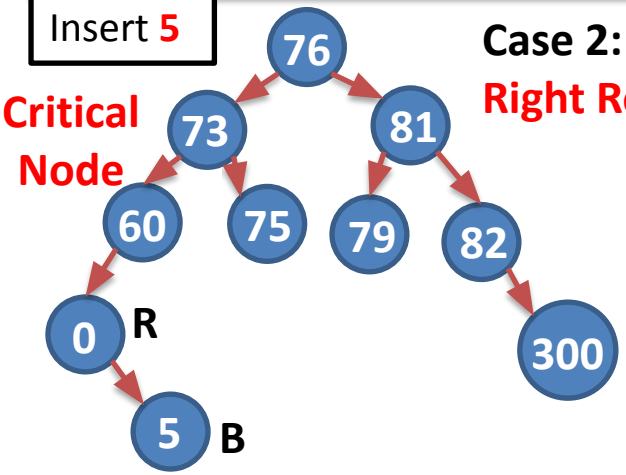
Insert **82**



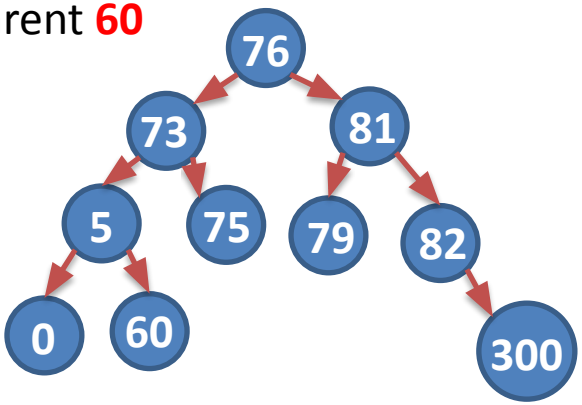
Insert **300** ,**0**



Insert **5**



Case 2: Left Right Rotation, **Left Rotation** of Left Child 0, Followed By **Right Rotation** of Parent **60**

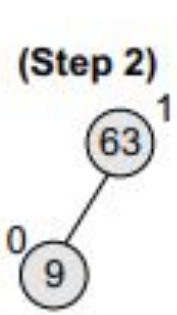


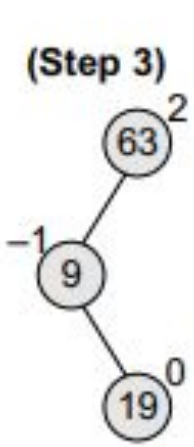
Insert **73**

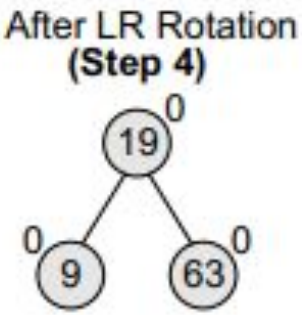
Can not Insert **73** as duplicate key found

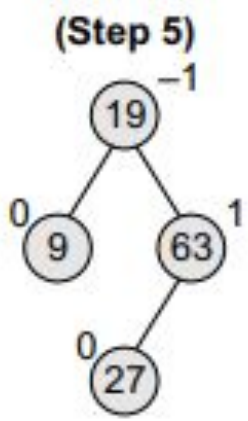
Construct AVL Search Tree

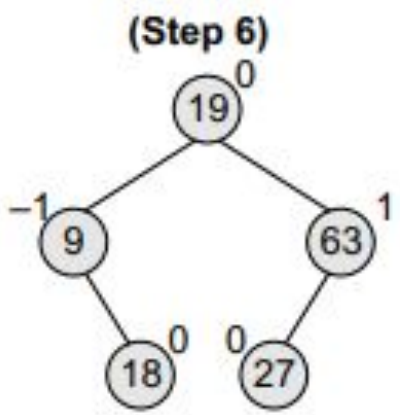
Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

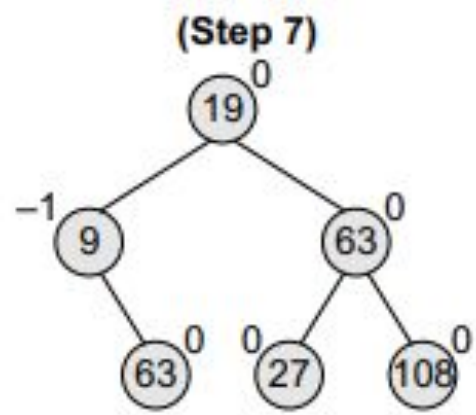






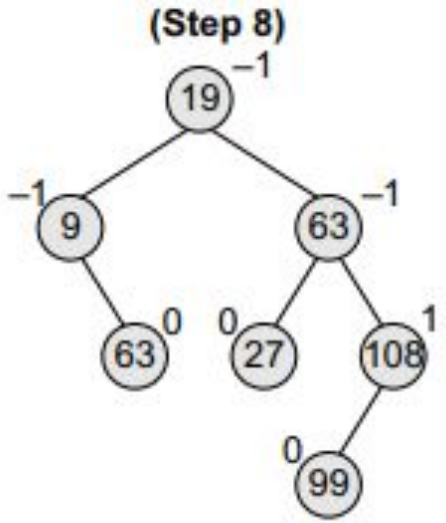


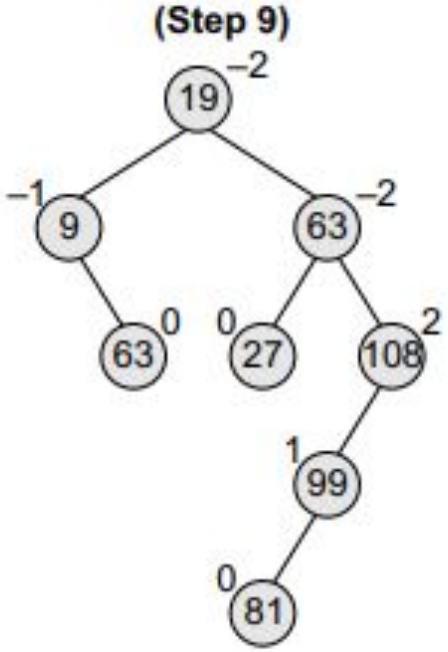


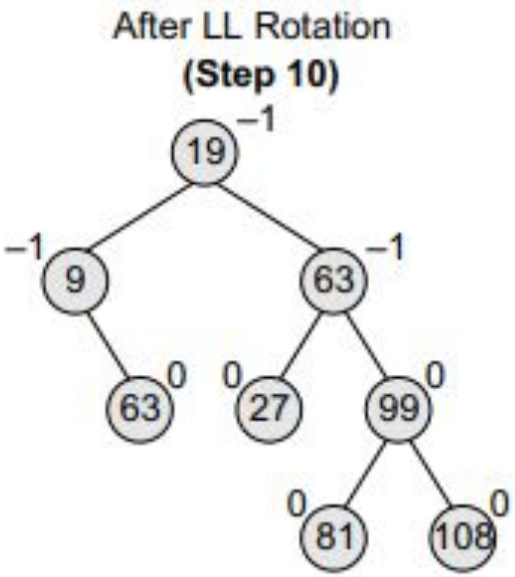


Construct AVL Search Tree

Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

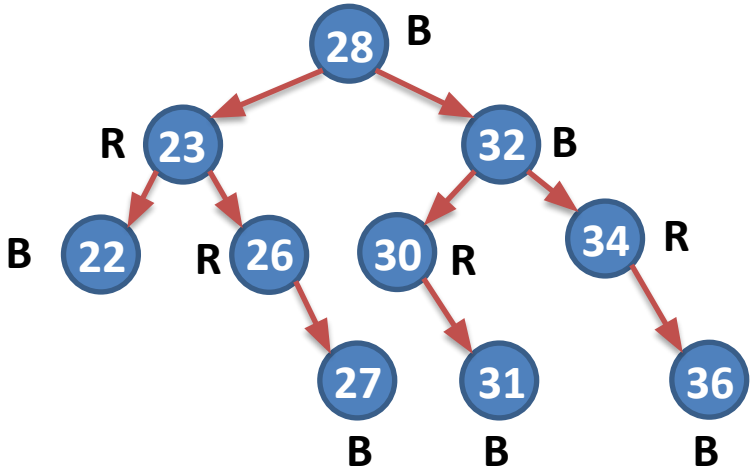




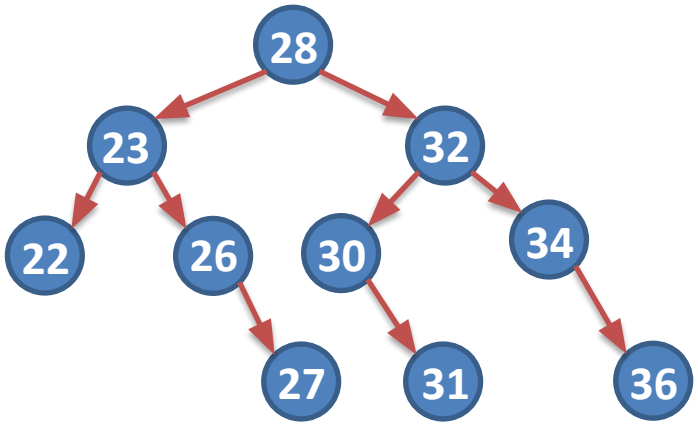


Deleting node from AVL Tree

- If element to be deleted **does not have empty right sub-tree**, then **element** is **replaced** with its **In-Order successor** and its **In-Order successor is deleted** instead
- During **winding up phase**, we need to **revisit every node** on the **path** from the **point of deletion** upto the **root**, rebalance the tree if require



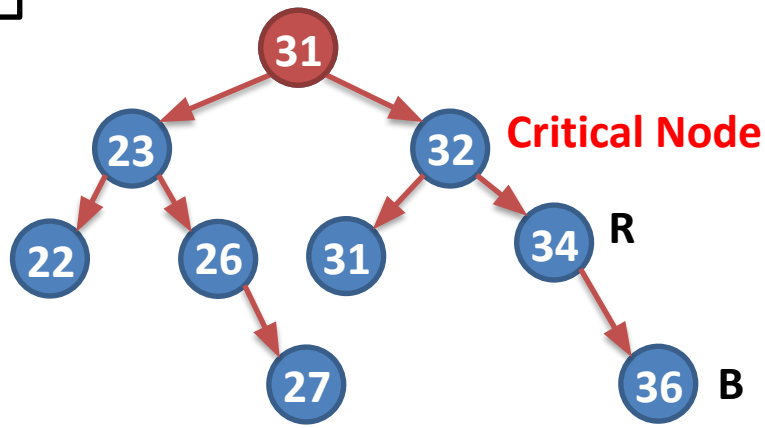
Deleting node from AVL Tree



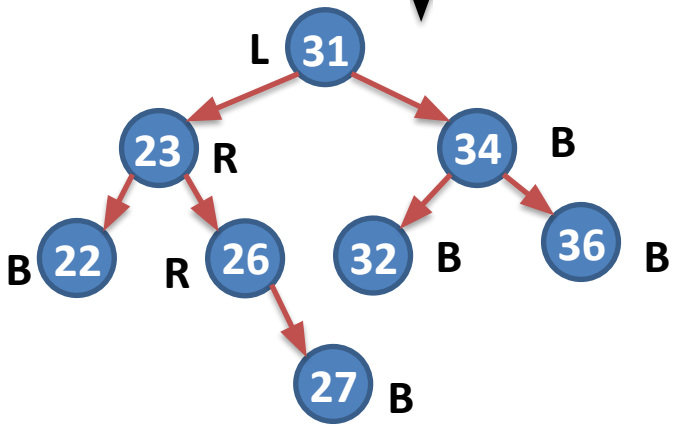
In-Order Traversal

22, 23, 26, 27, 28, 30, 31, 32, 34, 36

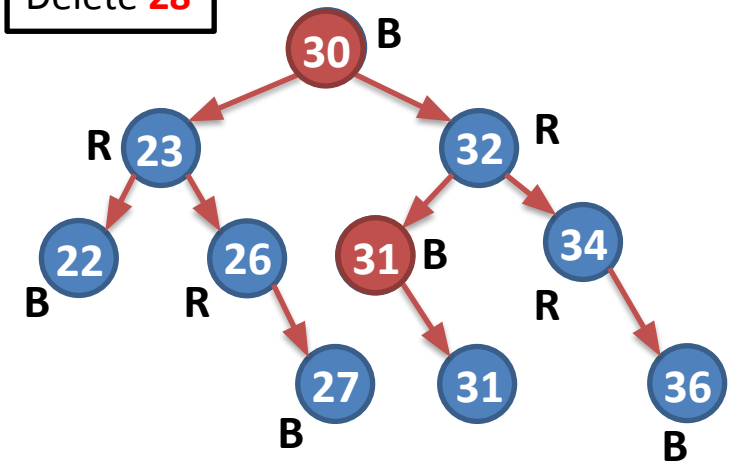
Delete **30**



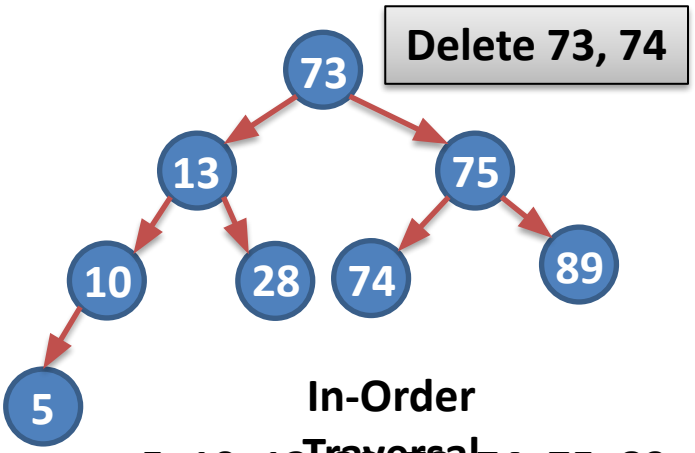
Case 4: Left Rotation of Node 32



Delete **28**



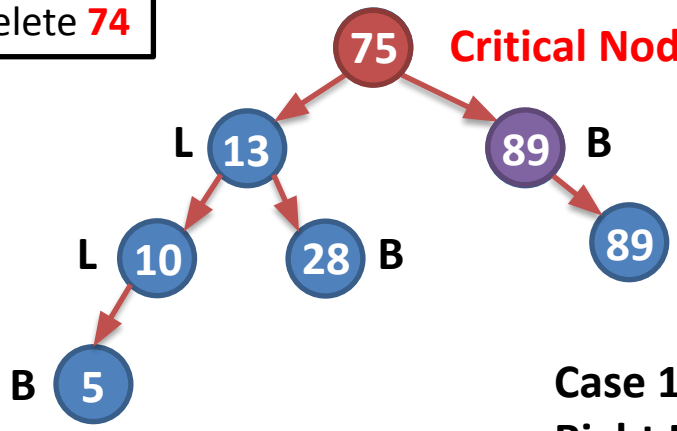
Deleting node from AVL Tree



In-Order Traversal

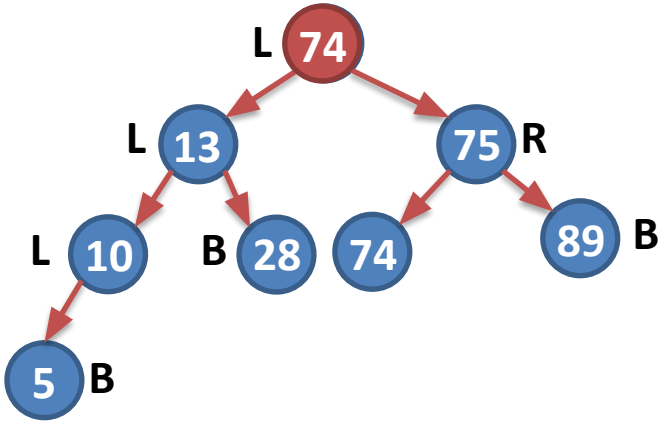
5, 10, 13, 28, 73, 74, 75, 89

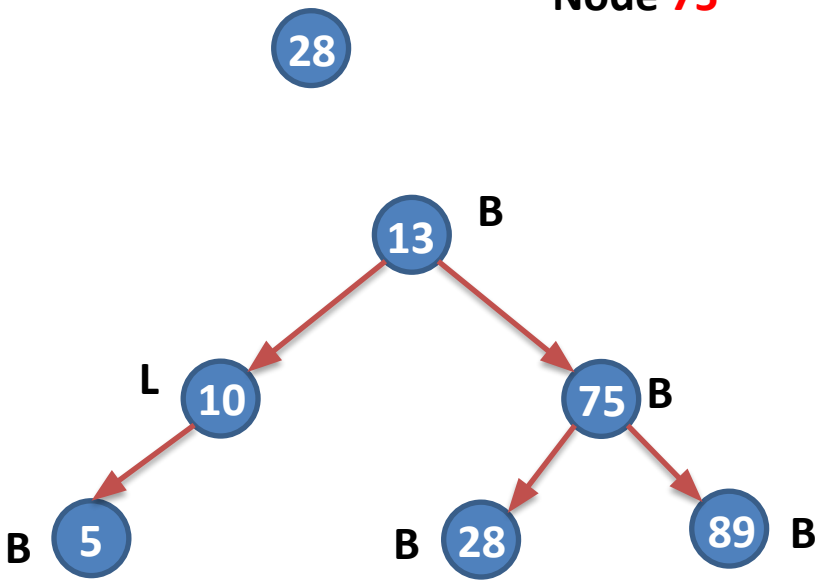
Delete **74**



Case 1:
Right Rotation of
Node **75**

Delete **73**

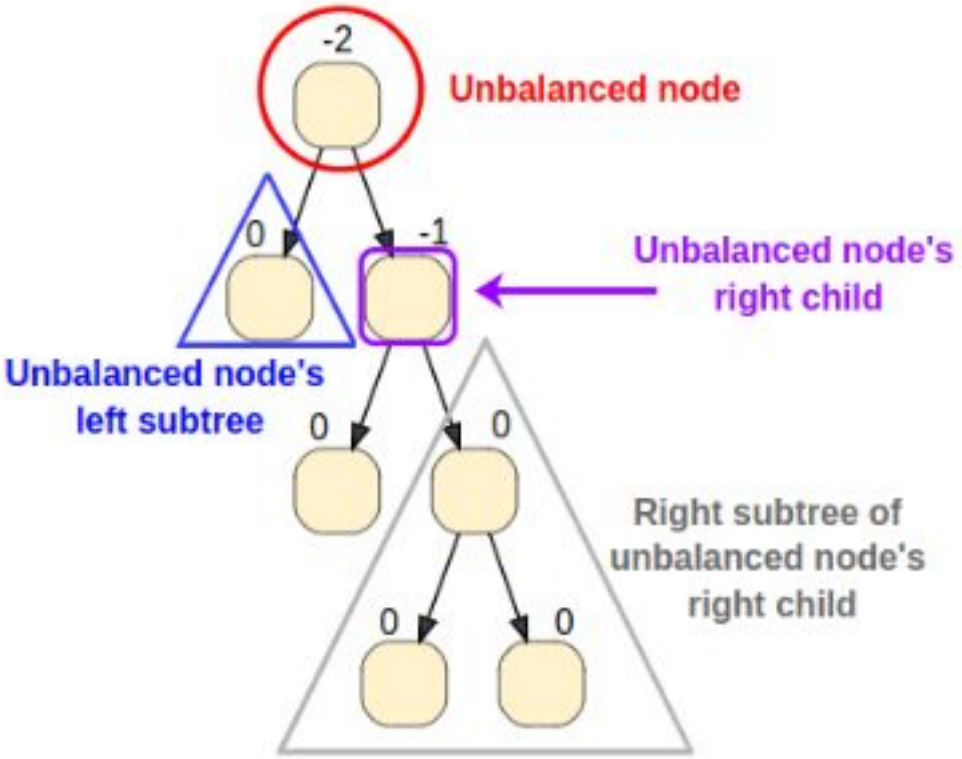




Types of Imbalances

Right-right imbalance:

- It occurs when the right sub-tree of a node's right child is heavier than the node's own left sub-tree. This is indicated by the balance factor of the unbalanced node being -2 and the balance factor of its right child being 0 or -1.

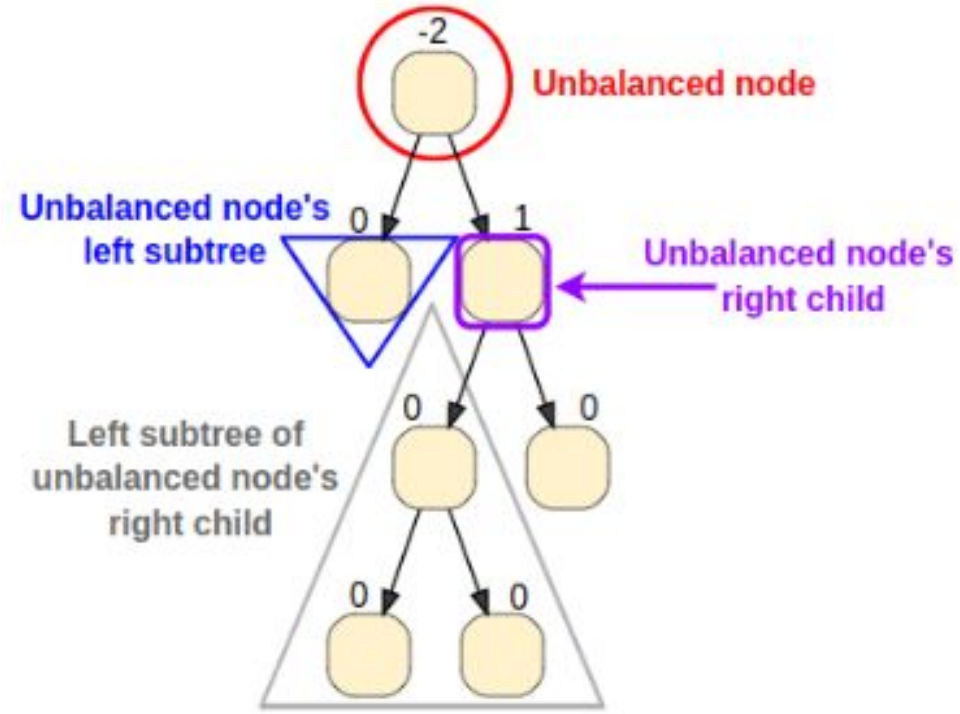


Right-Right imbalance

Types of Imbalances

Right-left imbalance:

- It occurs when the left sub-tree of a node's right child is heavier than the node's own left sub-tree. This is indicated by the balance factor of the unbalanced node being -2 and the balance factor of its left child being 1.

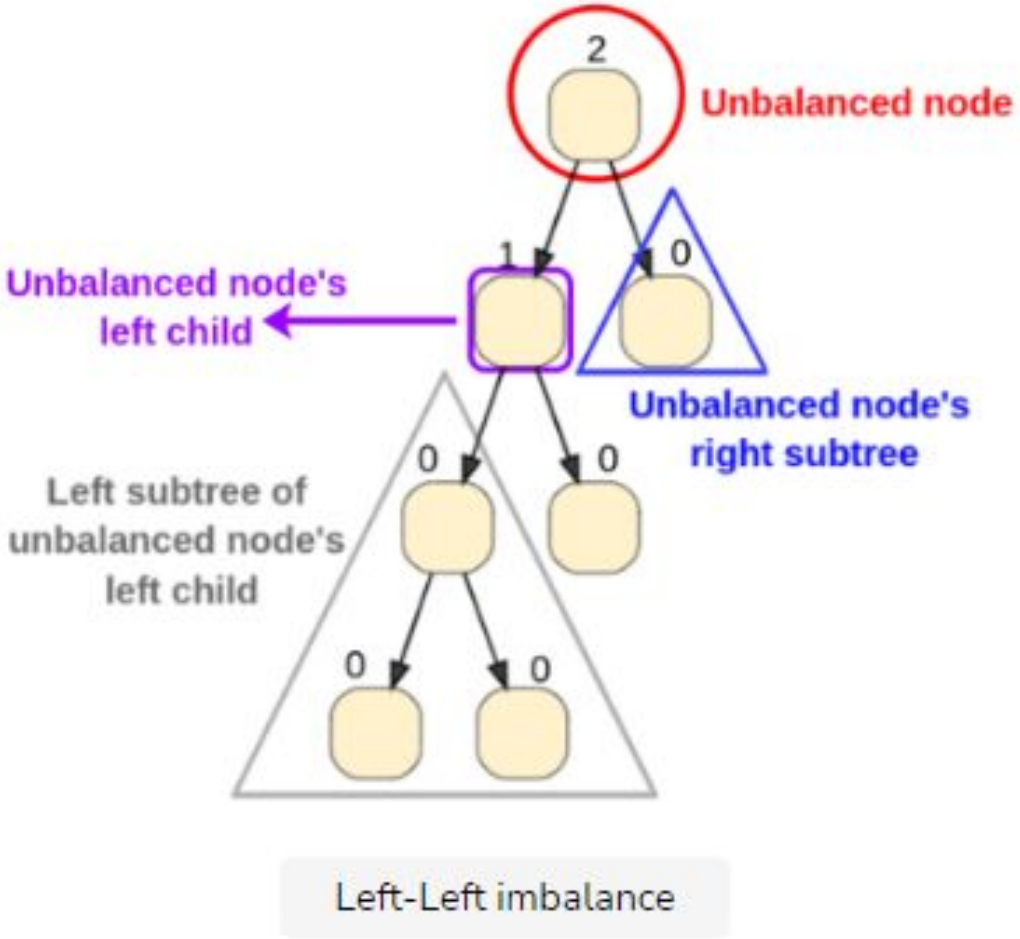


Right-Left imbalance

Types of Imbalances

Left-left imbalance:

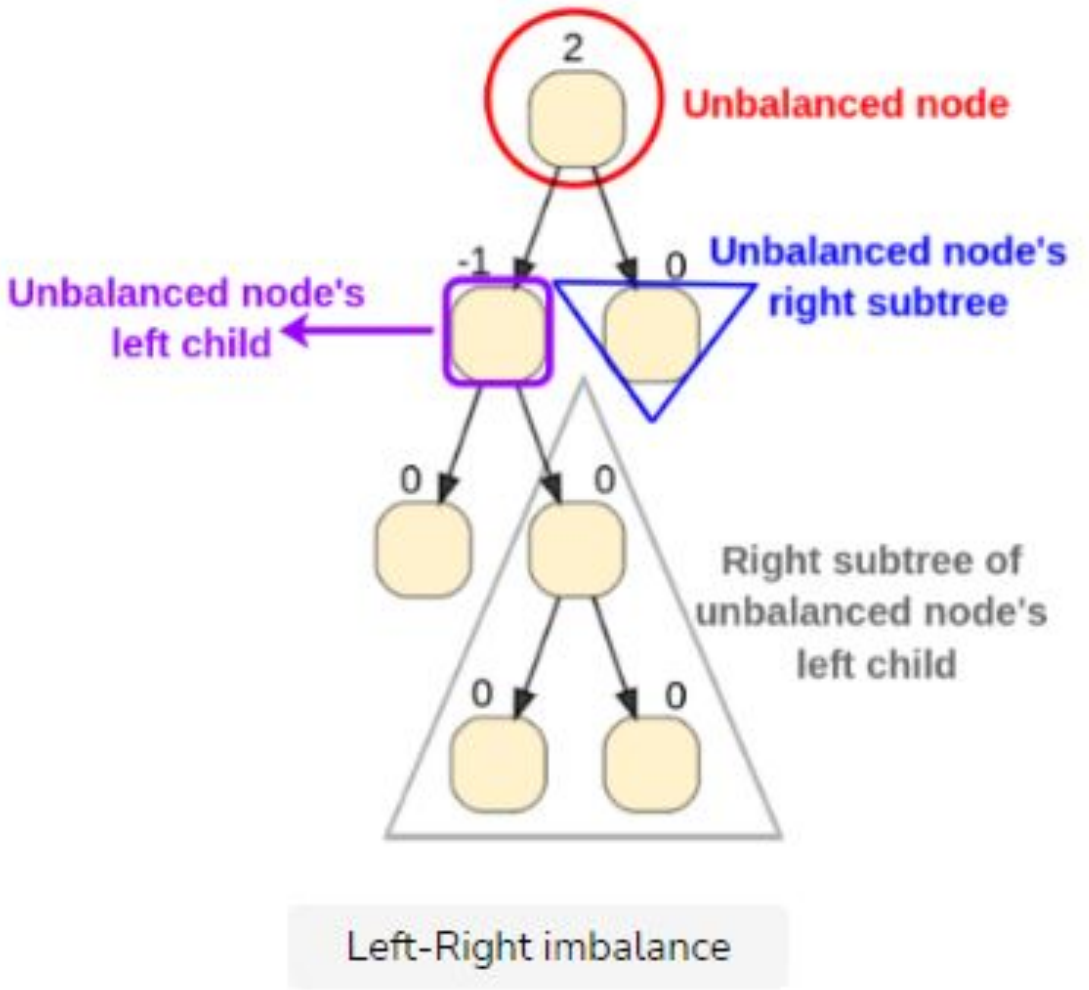
- It occurs when the left sub-tree of a node's left child is heavier than the node's own right sub-tree. This is indicated by the balance factor of the unbalanced node being 2 and the balance factor of its left child being 0 or 1.



Types of Imbalances

Left-right imbalance:

- It occurs when the right sub-tree of a node's left child is heavier than the node's own right sub-tree. This is indicated by the balance factor of the unbalanced node being 2 and the balance factor of its right child being -1.



Types of Imbalances

- To deal with each type of imbalance and fix it, AVL trees can perform rotations. These rotations involve switching around positions of the unbalanced node's sub-tree(s) to restore the balance factor of all affected nodes.
- By maintaining this balancing factor for each node, AVL trees allow the primary operations of a binary search tree (insert, delete, search) to run efficiently in O(log(n)) time in the worst case.

Types of Imbalances

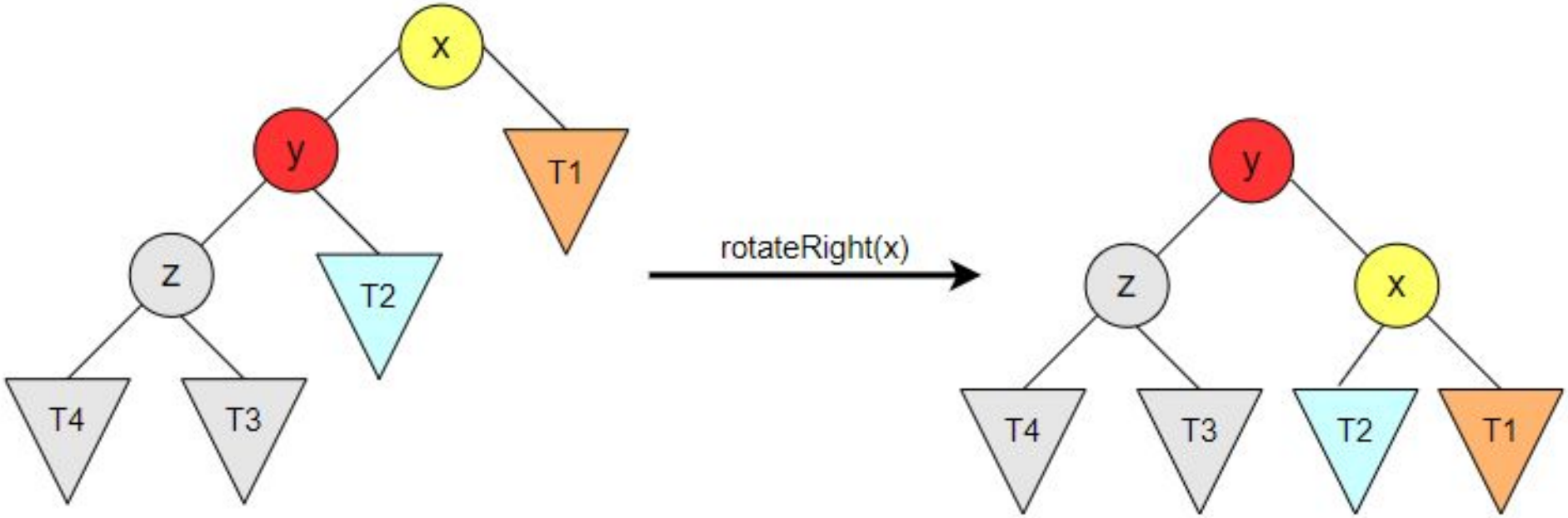
Single rotations

There are 2 rotations in this category:

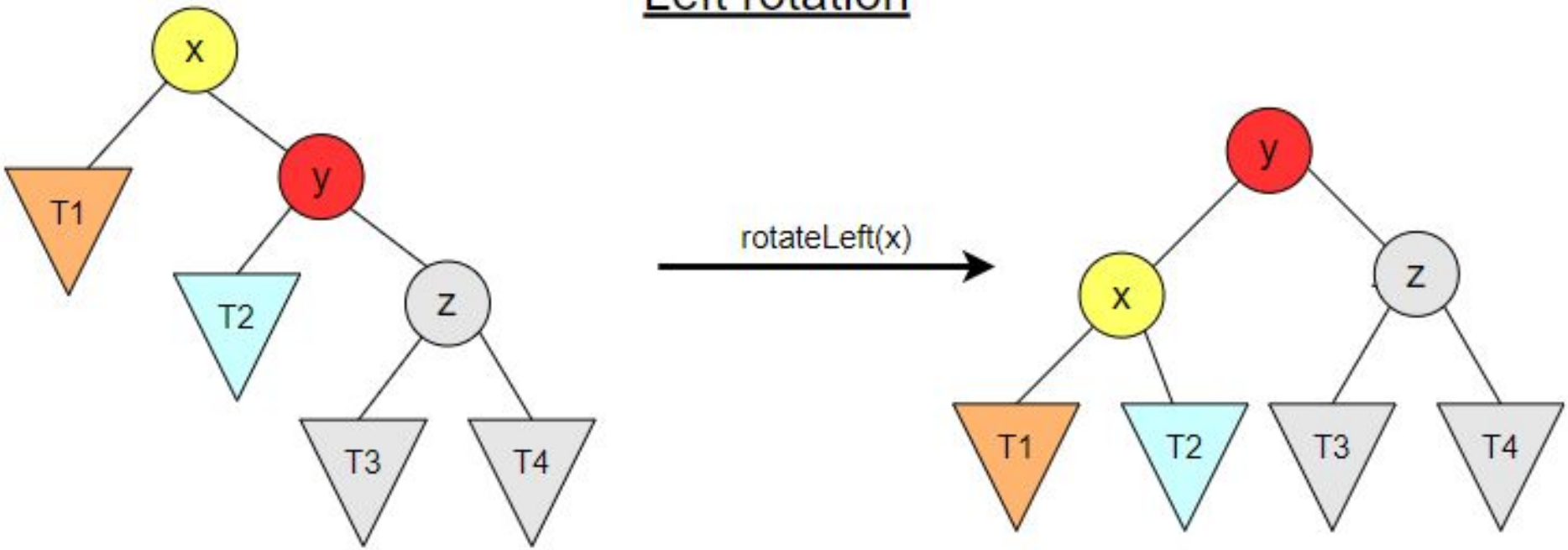
- Right rotation—to deal with the left-left imbalance.
- Left rotation—to deal with the right-right imbalance.

Given an unbalanced node x with children y and T_1, a single-rotation involves making y a parent of x. The diagrams below illustrate how right-rotation and left-rotation work:

Right rotation



Left rotation



Types of Imbalances

Double rotations

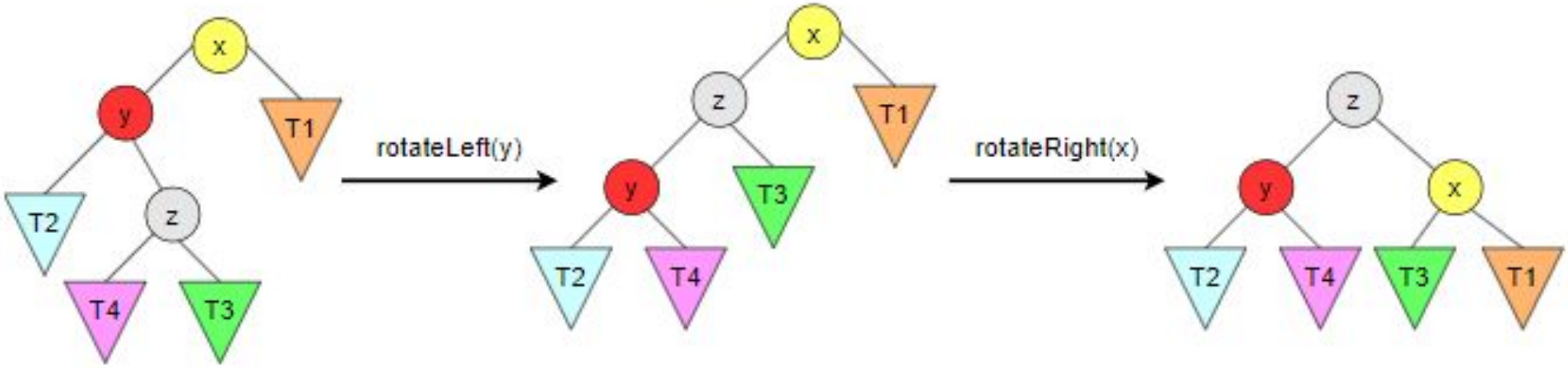
There are 2 rotations in this category:

- Right-left rotation—to deal with the right-left imbalance.
- Left-right rotation—to deal with the left-right imbalance.

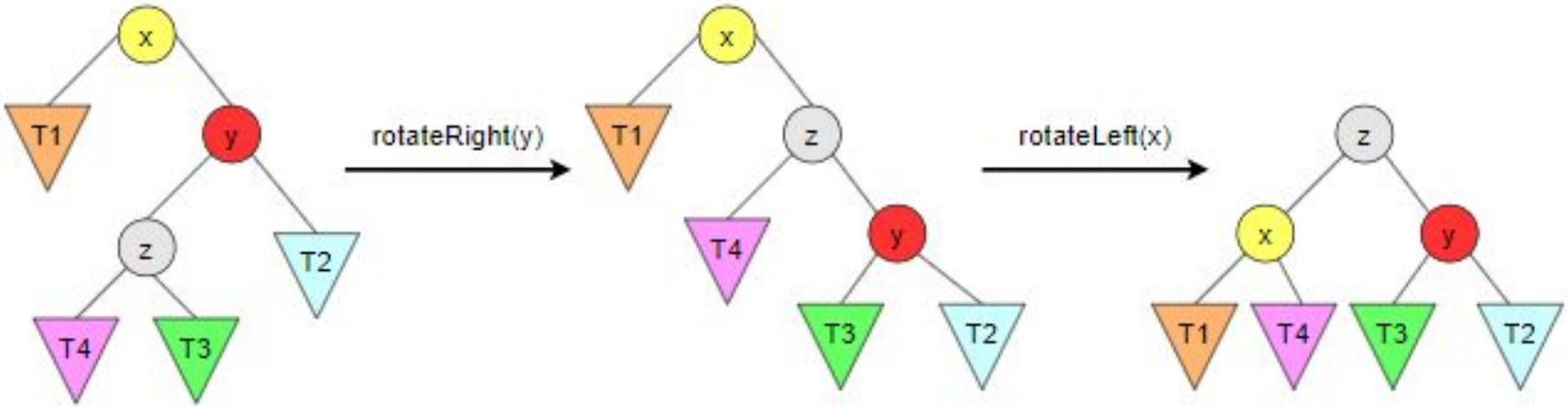
As the name suggests, these rotations are a combination of two single rotations applied after each other

1. Given an unbalanced node x with children y and T_1, and given that y is the parent of node z, the first rotation involves making z the parent of y. Now, z is a child of x.
2. Then, the second rotation involves making z the parent of x as well.

Left-Right rotation



Right-Left rotation

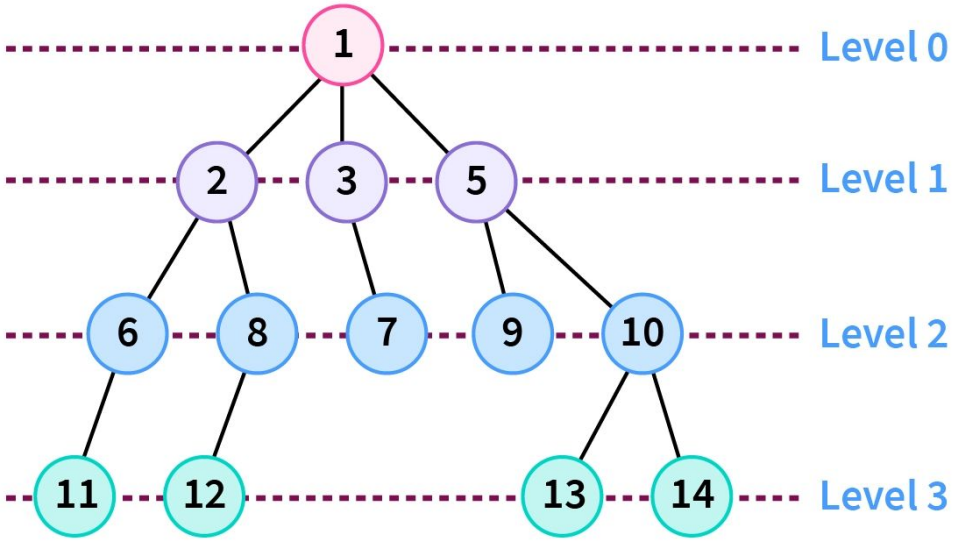


Level Order Traversal

- In the level order tree traversal, the level of the root node is considered to be level-0 (in some explanations, you can find level-1 also).
- So, we first print the root node and then increase the level (i.e. we move through the depth).
- In this way we can print the entire tree.
- Generally the level order traversal is done using a queue data structure.
- Firstly we insert the root into the queue and iterate over the queue until the queue is empty.
- In every iteration, we will pop the front element of the queue and print its value.
- Then, we add its left child and right child to the end of the queue.

Level Order Traversal

1. Initialize an empty queue data structure.
2. Insert the root into the queue.
3. If the queue is not empty, iterate over the queue.
4. Pop the front element of the queue and print it.
5. Add the left child and right child of the pooped node to the end of the queue.
6. Repeat the above process (3 - 5) until the queue becomes empty..



Level Order Traversal

```
void print_tree_level_order(Node* root) {
    int height = tree_height(root);
    for (int i=0; i<height; i++) {
        // Print the ith level
        print_level(root, i);
    }
}
```

Level Order Traversal

- Run a for loop for counter i, i.e. **current height** from 1 to **h** (height of the tree).
- Use DFS to traverse the tree and maintain height for the current node.
  - If the Node is NULL then return;
  - If level is 1 print(tree->data);
  - Else if the level is greater than 1, then
    - Recursively call to for tree->left, level-1.
    - Recursively call to for tree->right, level-1.

Level Order Traversal

```
void print_level(Node* root, int level_no) {
    // Prints the nodes in the tree
    // having a level = level_no

    // We have a auxiliary root node
    // for printing the root of every
    // sub-tree
    if (!root)
        return;
    if (level_no == 0) {
        // We are at the top of a sub-tree
        // So print the auxiliary root node
        printf("%d -> ", root->value);
    }
    else {
        // Make the auxiliary root node to
        // be the left and right nodes for
        // the sub-trees and decrease level by 1, since
        // you are moving from top to bottom
        print_level(root->left, level_no - 1);
        print_level(root->right, level_no - 1);
    }

}
```

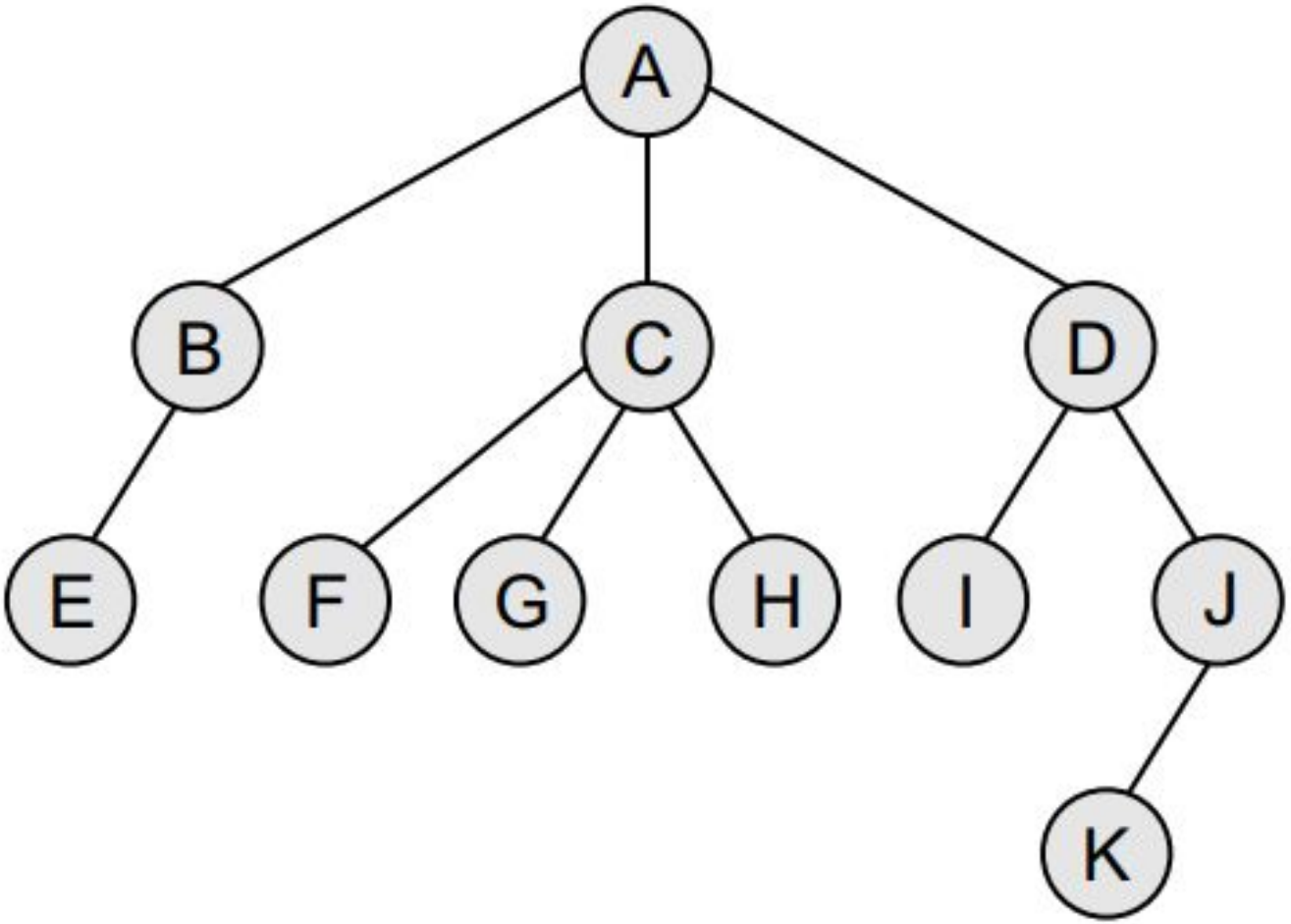
Creating a Binary Tree from a General Tree

Note that a general tree is converted into a binary tree and not a binary search tree.

- Rule 1: Root of the binary tree = Root of the general tree
- Rule 2: Left child of a node = Leftmost child of the node in the binary tree in the general tree
- Rule 3: Right child of a node in the binary tree = Right sibling of the node in the general tree

Creating a Binary Tree from a General Tree

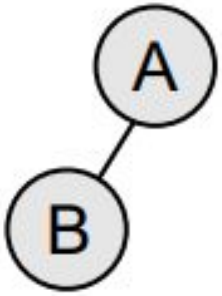
- Rule 1: Root of the binary tree = Root of the general tree
- Rule 2: Left child of a node = Leftmost child of the node in the binary tree in the general tree
- Rule 3: Right child of a node in the binary tree = Right sibling of the node in the general tree



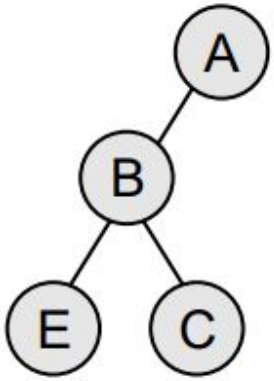
Creating a Binary Tree from a General Tree

- Step 1: Node A is the root of the general tree, so it will also be the root of the binary tree.
- Step 2: Left child of node A is the leftmost child of node A in the general tree and right child of node A is the right sibling of the node A in the general tree. Since node A has no right sibling in the general tree, it has no right child in the binary tree.
- Step 3: Now process node B. Left child of B is E and its right child is C (right sibling in general tree).

Step 1



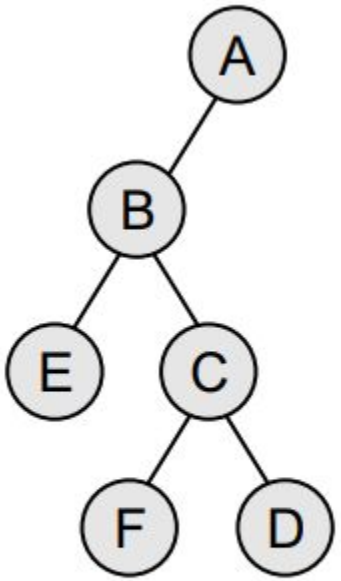
Step 2



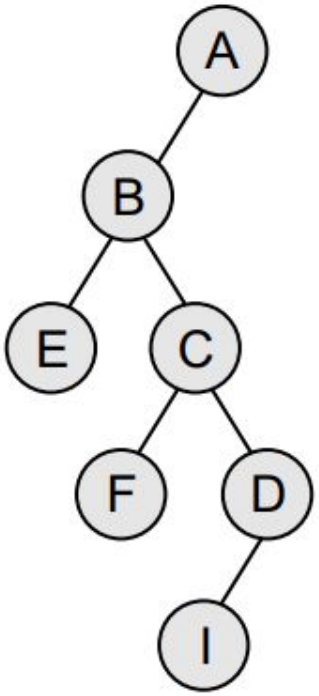
Step 3

Creating a Binary Tree from a General Tree

- Step 4: Now process node C. Left child of C is F (leftmost child) and its right child is D (right sibling in general tree).
- Step 5: Now process node D. Left child of D is I (leftmost child). There will be no right child of D because it has no right sibling in the general tree.



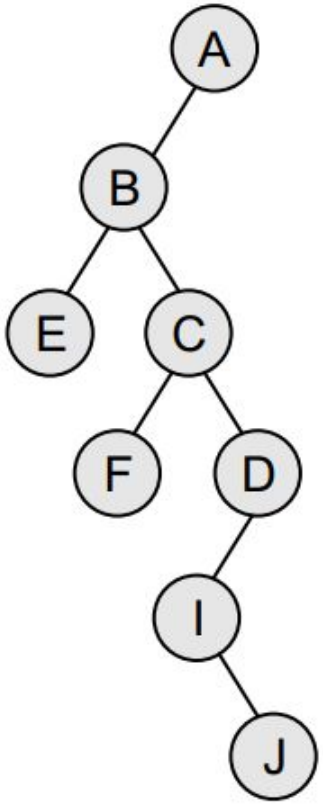
Step 4



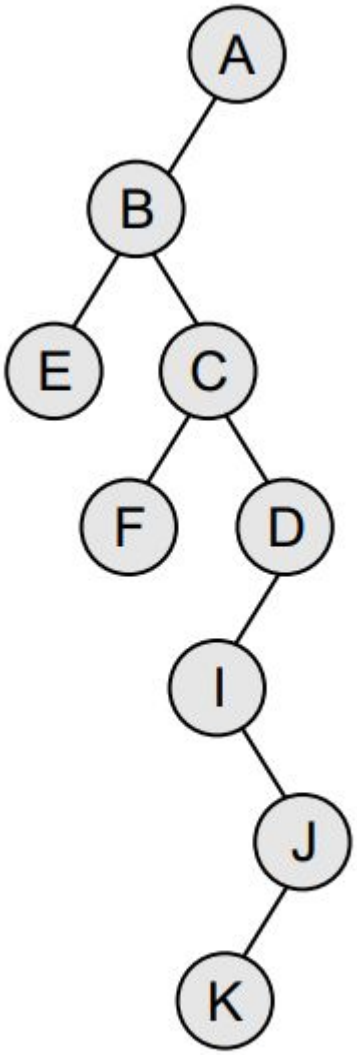
Step 5

Creating a Binary Tree from a General Tree

- Step 6: Now process node I. There will be no left child of I in the binary tree because I has no left child in the general tree. However, I has a right sibling J, so it will be added as the right child of I.
- Step 7: Now process node J. Left child of J is K (leftmost child). There will be no right child of J because it has no right sibling in the general tree.



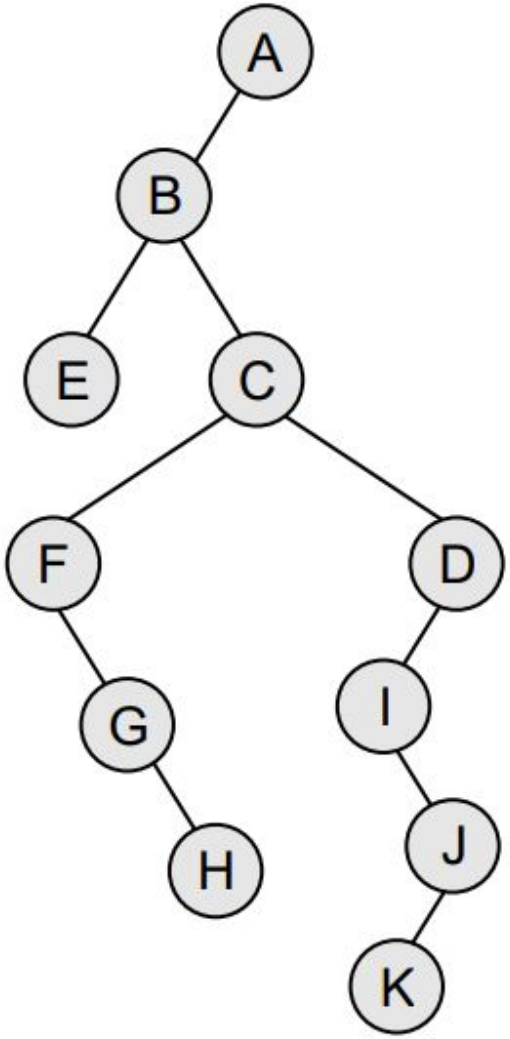
Step 6



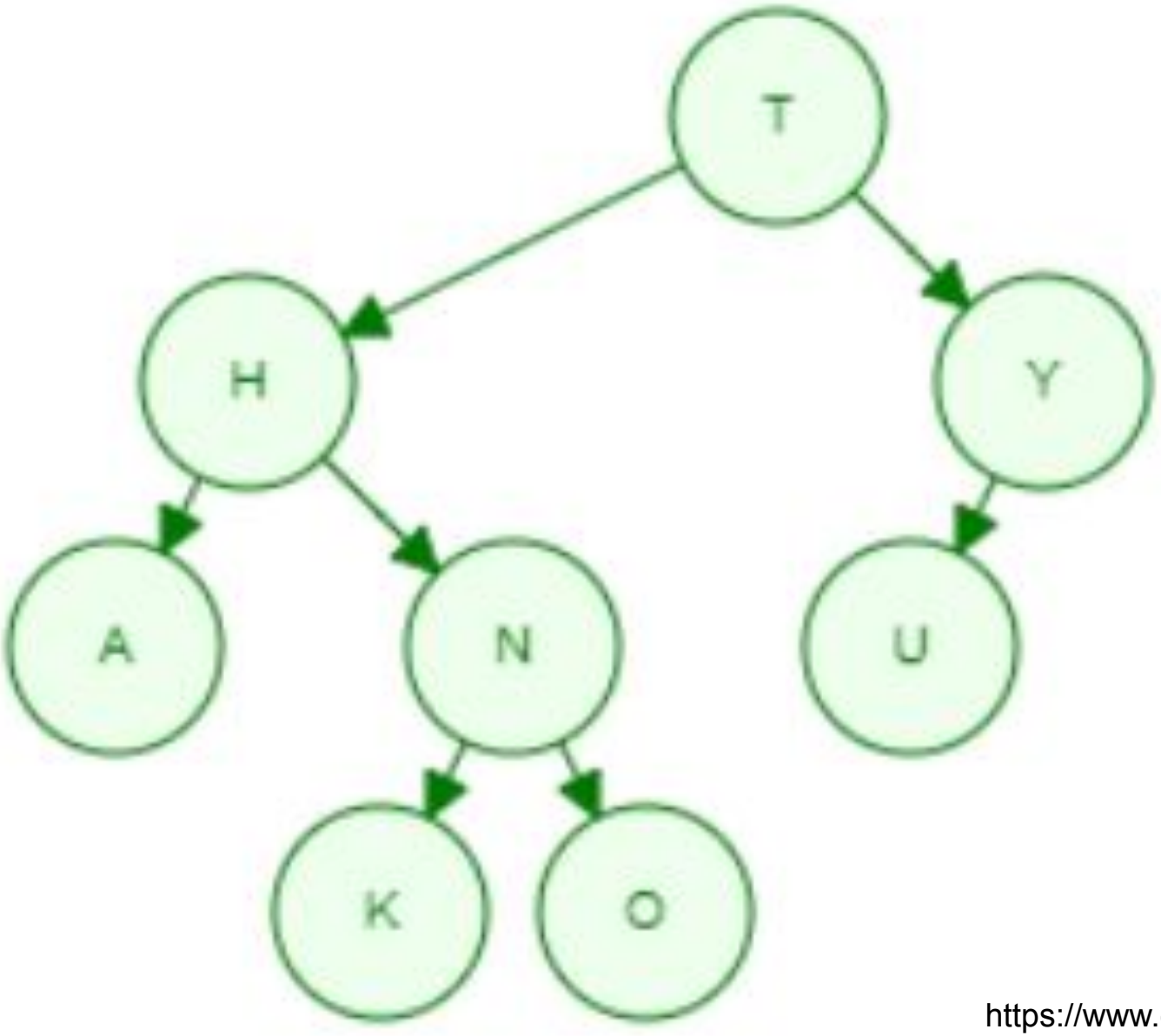
Step 7

Creating a Binary Tree from a General Tree

- Step 8: Now process all the unprocessed nodes (E, F, G, H, K) in the same fashion, so the resultant binary tree can be given as follows.



Step 8



https://www.cs.usfca.edu/~galles/visualization/BST.html