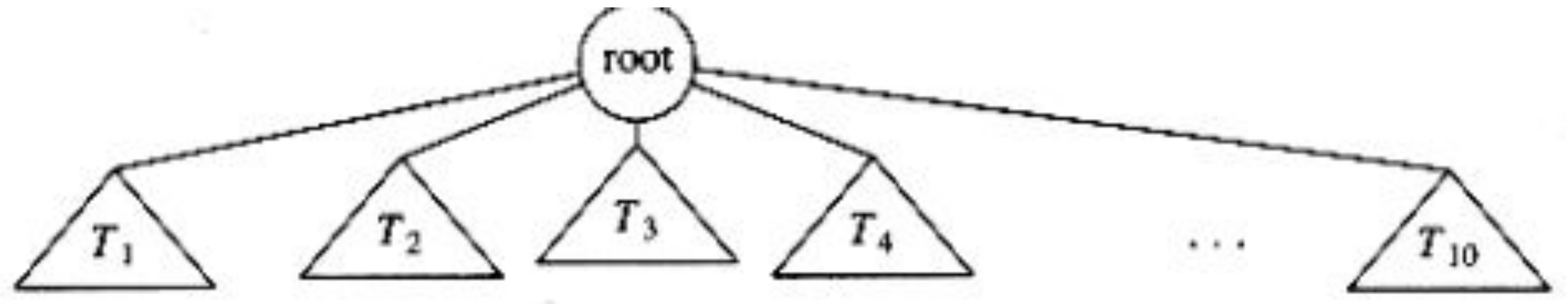
Trees

Introduction

- A tree is a data structure consisting of nodes organized as a hierarchy.
- In Tree nodes are connected by edges.
- A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.
- A tree is a collection of nodes connected by directed (or undirected) edges.
- The collection can be empty, which is sometimes denoted as A.
- Otherwise, a tree consists of a distinguished node r, called the root, and zero or more (sub)trees T1, T2, . . . , Tk, each of whose roots are connected by a edge to r.
- The root of each subtree is said to be a child of r, and r is the parent of each subtree root.

Trees

- Figure shows a typical tree using the recursive definition.



- From the recursive definition, we find that a tree is a collection of n nodes, one of which is the root, and n - 1 edges.

Trees

Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

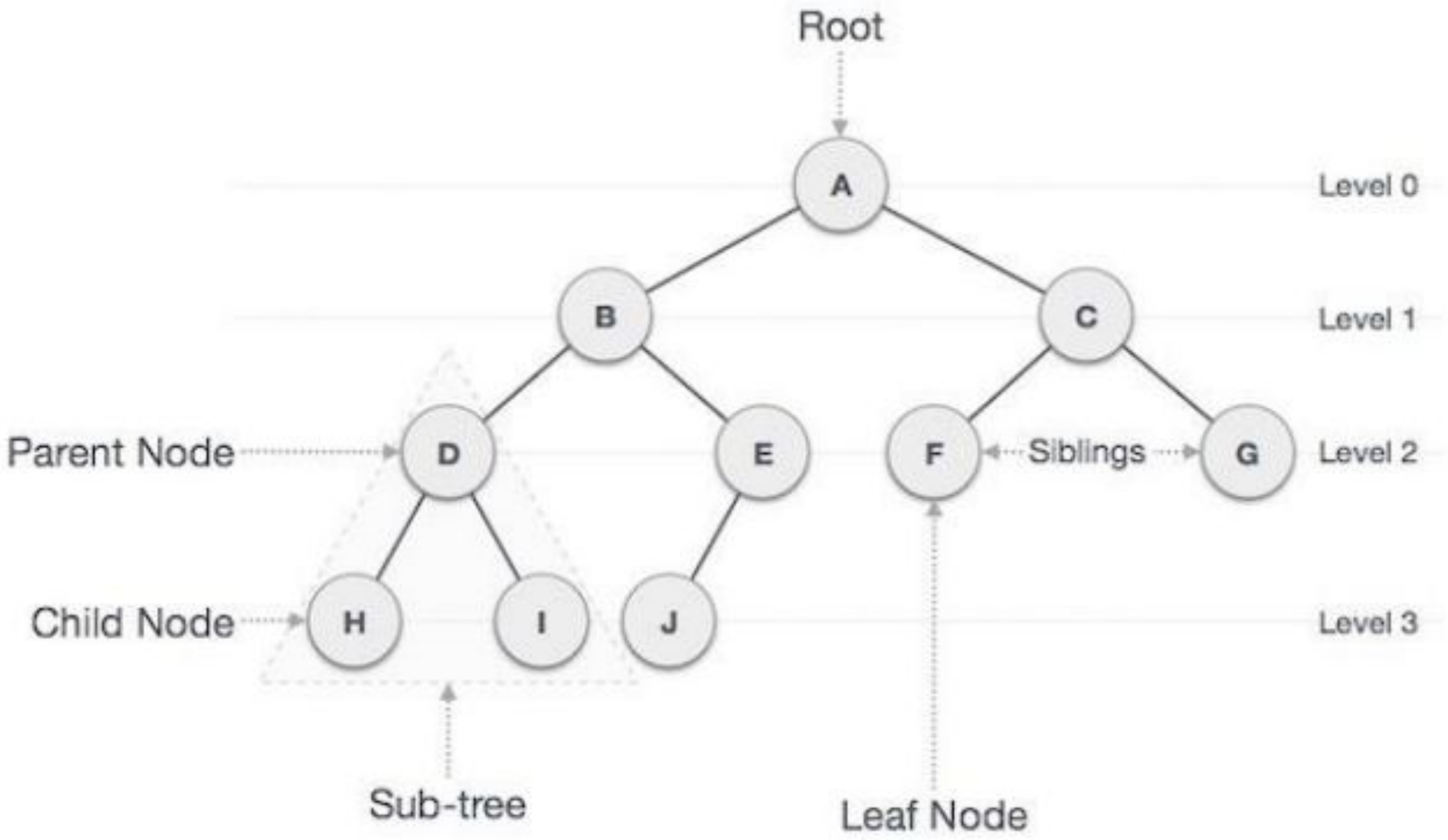
Terminologies used in Trees

- **Root** − Node at the top of the tree is called root. There is only one root per tree.
- **Parent** − Any node except root node has one edge upward to a node called parent.
- **Child** − Node below a given node connected by its edge downward is called its child node.
- **Leaf** − Node which does not have any child node is called leaf node.
- **Path** − Path refers to sequence of nodes along the edges of a tree.
  - A path from node n1to nk is defined as a sequence of nodes n1, n2, .. . , nk such that ni is the parent of ni+1for 1 i < k.
  - The length of this path is the number of edges on the path, namely k -1. There is a path of length zero from every node to itself.
  - Notice that in a tree there is exactly one path from the root to each node.
- **Siblings** - A group of nodes with the same parent.

Terminologies used in Trees

- Subtree − Subtree represents descendants of a node.
- Degree – The degree of a node is the total number of branches of that node.
- Height of node - The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).
  - The height of ni is the longest path from ni to a leaf.
  - Thus all leaves are at height 0.
- Height of tree - The height of a tree is the height of its root node.
- Depth - The depth of a node is the number of edges from the node to the tree's root node.
  - For any node ni, the depth of ni is the length of the unique path from the root to ni.
  - Thus, the root is at depth 0.
- Levels − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- Forest - A forest is a set of $n \geq 0$ disjoint trees.

Terminologies used in Trees



Types of Trees

1. Binary Trees
2. Binary search trees
3. AVL Tree
4. B-Tree

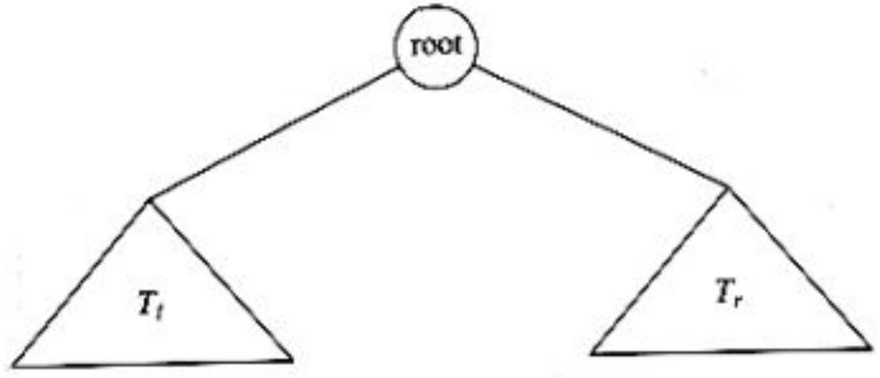
Binary Tree

- A binary tree is a tree in which no node can have more than two

children.

- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- Figure shows that a binary tree consists of a root and two subtrees,

Tl and Tr, both of which could possibly be empty.



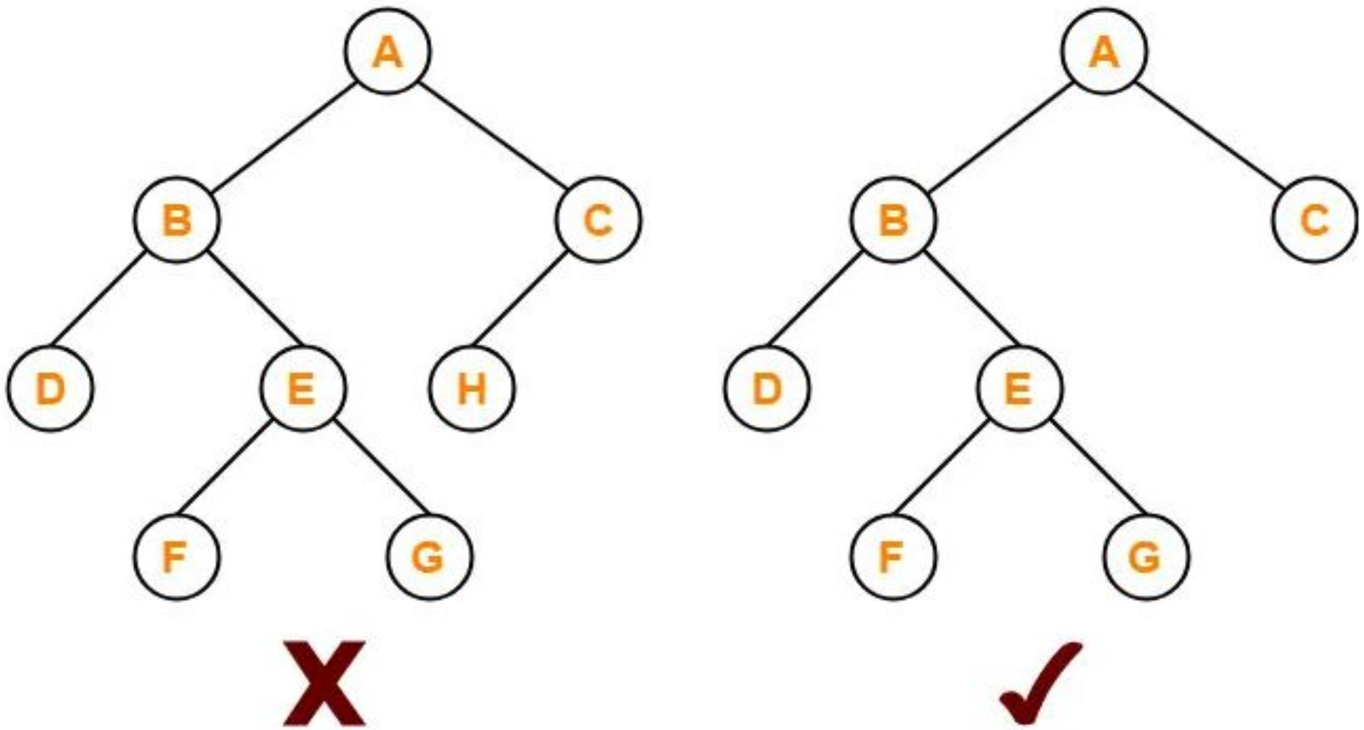
Types of Binary Tree

- Full/Strictly Binary tree
- Perfect binary tree
- Skewed binary tree
- Complete Binary tree

Full/Strictly Binary tree

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree**.

Example-



Here,

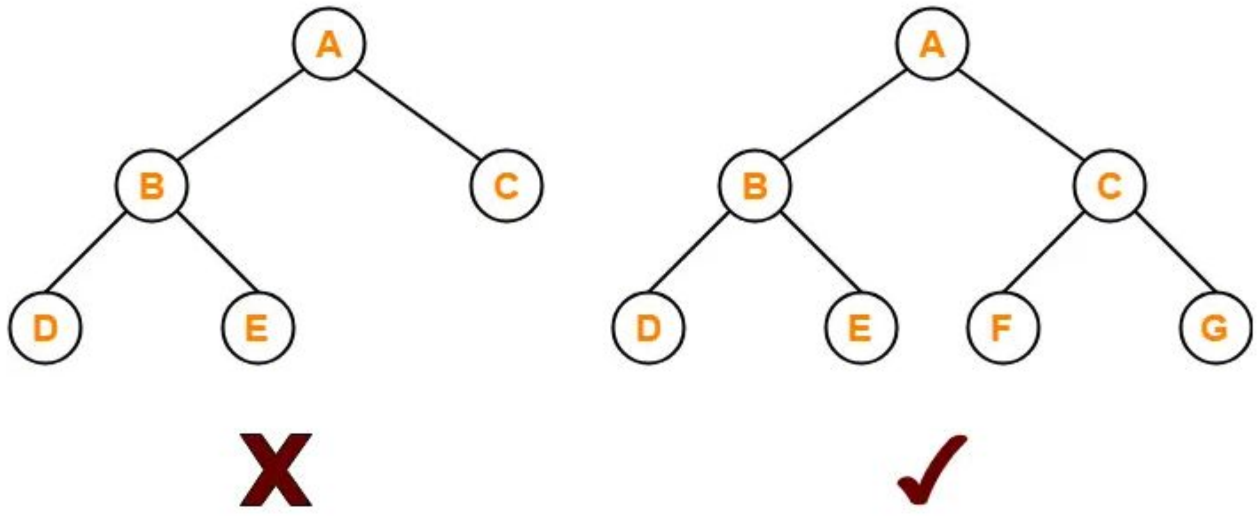
•First binary tree is not a full binary tree because node C has only 1 child.

Perfect binary tree

A **Perfect binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Example-

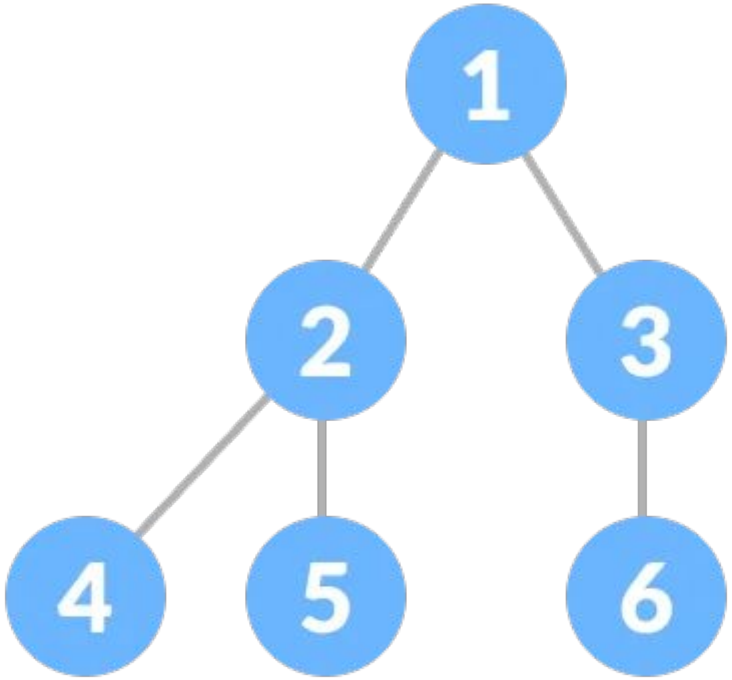


Here,

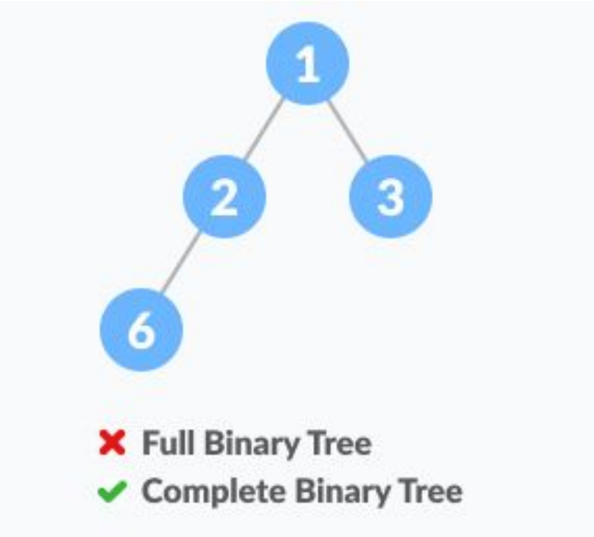
- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

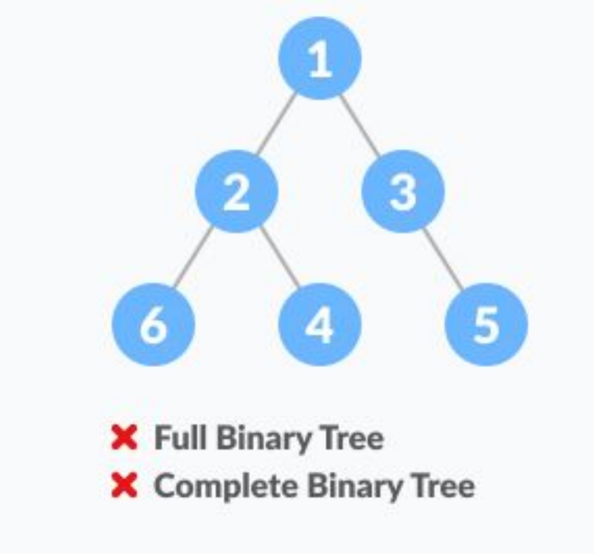
Complete Binary tree

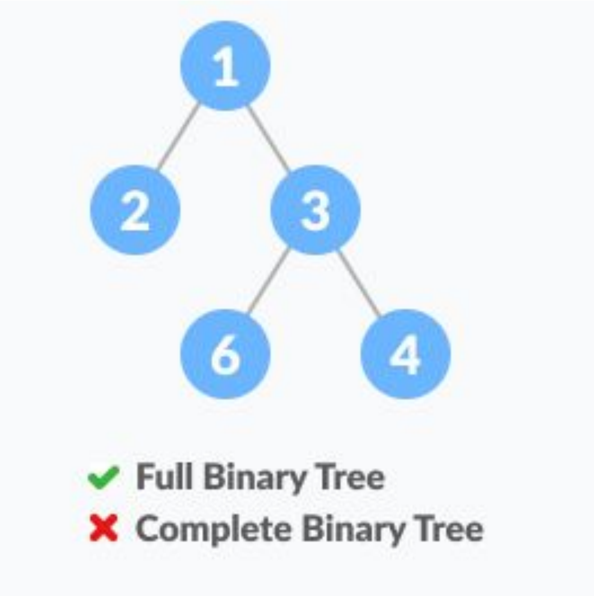
- A complete binary tree is just like a full binary tree, but with two major differences
- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

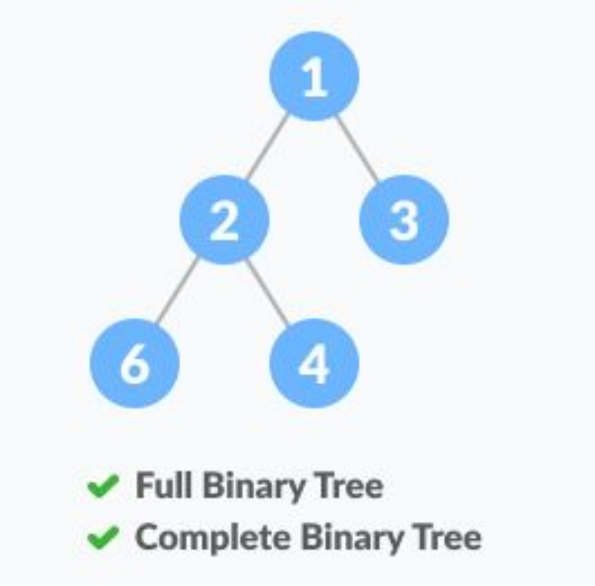


Full Binary Tree vs Complete Binary Tree









Skewed Binary Tree

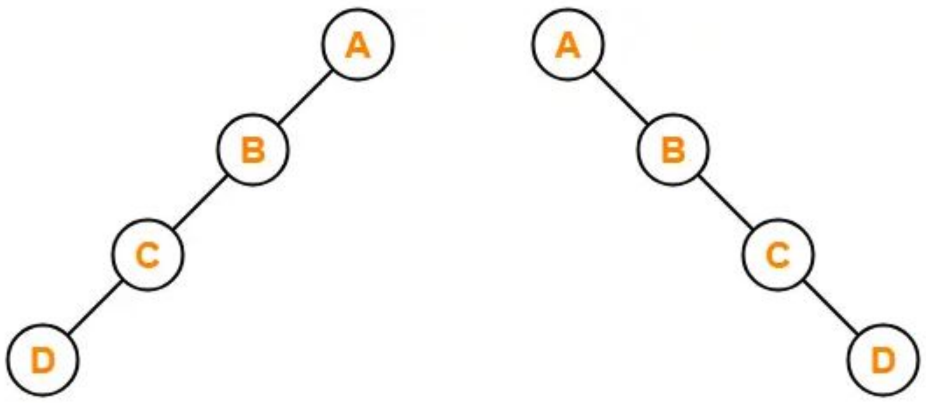
A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

- **OR**

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).

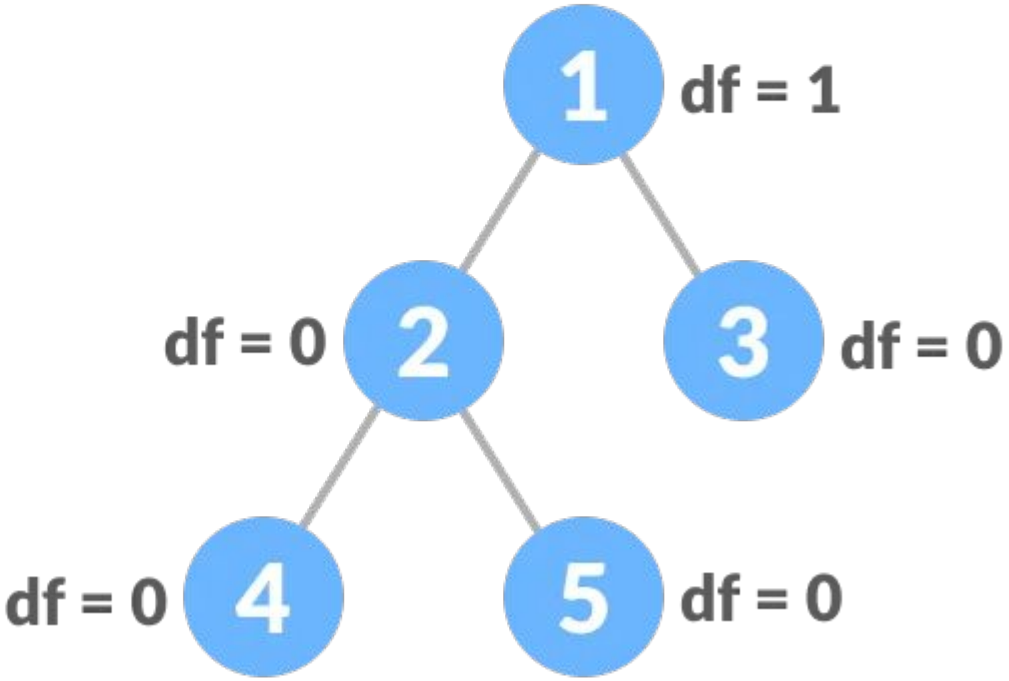
Example-



Left Skewed Binary Tree   Right Skewed Binary Tree

Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1

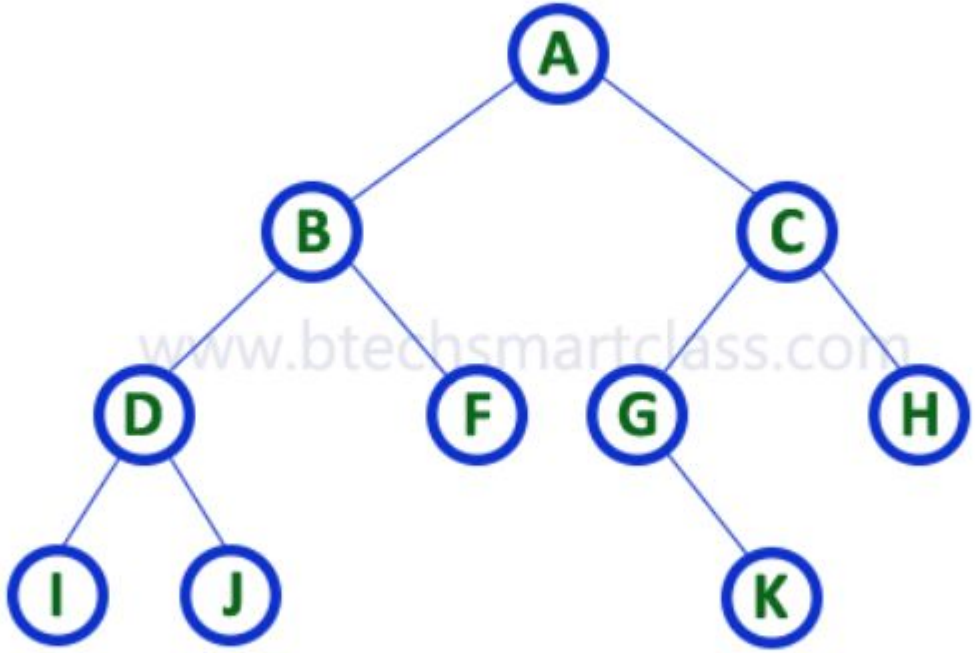


Binary Tree Representation

- A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows...

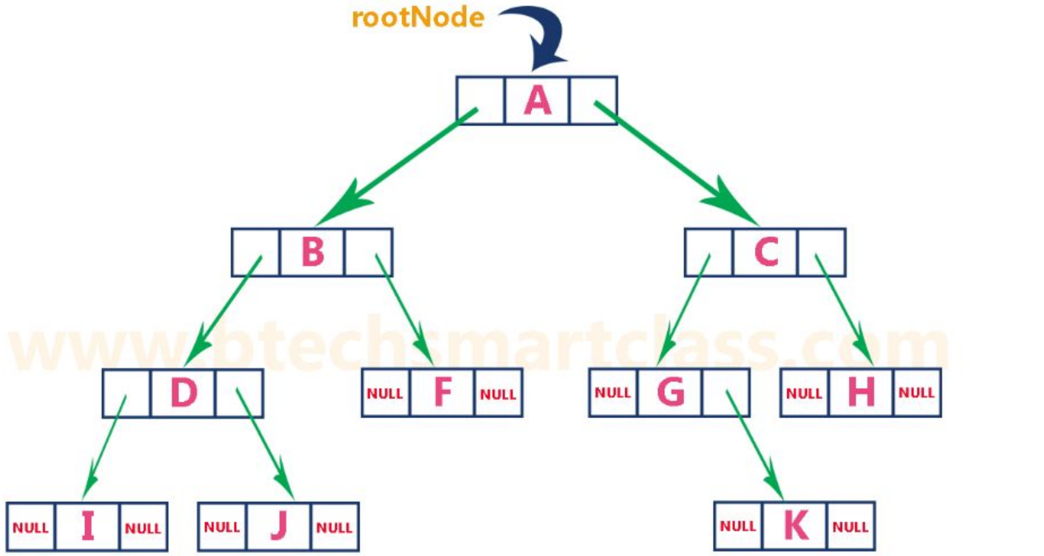
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth '**n**' using array representation, we need one dimensional array with a maximum size of **2n + 1**.

2. Linked List Representation of Binary Tree

- We use a double linked list to represent a binary tree.
- In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
- In this linked list representation, a node has the following structure...

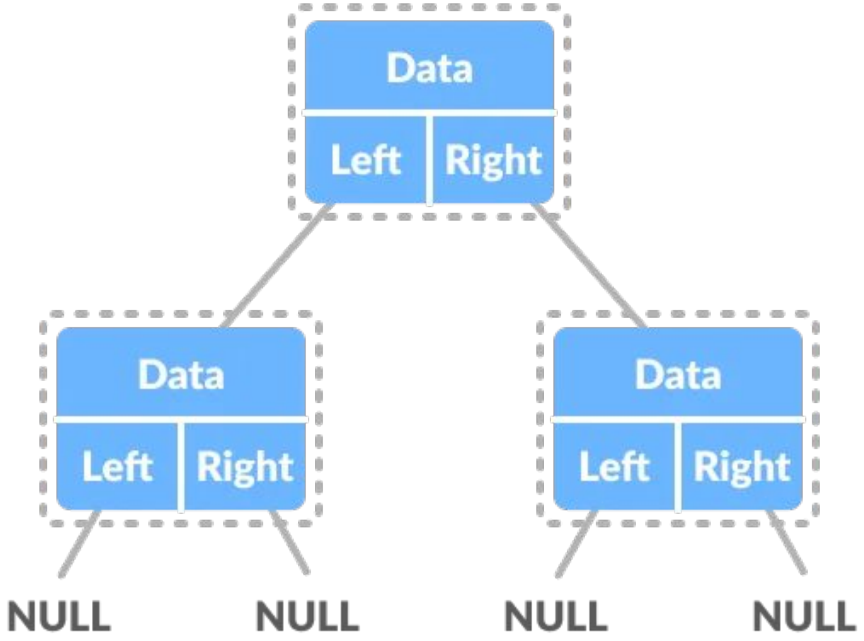
- The above example of the binary tree represented using Linked list representation is shown as follows...



Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

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



Binary Tree Traversal

- Traversal is a process to visit all the nodes of a tree and may print their values.
- Because, all nodes are connected via edges (links) we always start from the root node.
- That is, we cannot random access a node in tree.
- There are three ways which we use to traverse a tree

i. Pre-order Traversal

ii. In-order Traversal

iii. Post-order Traversal

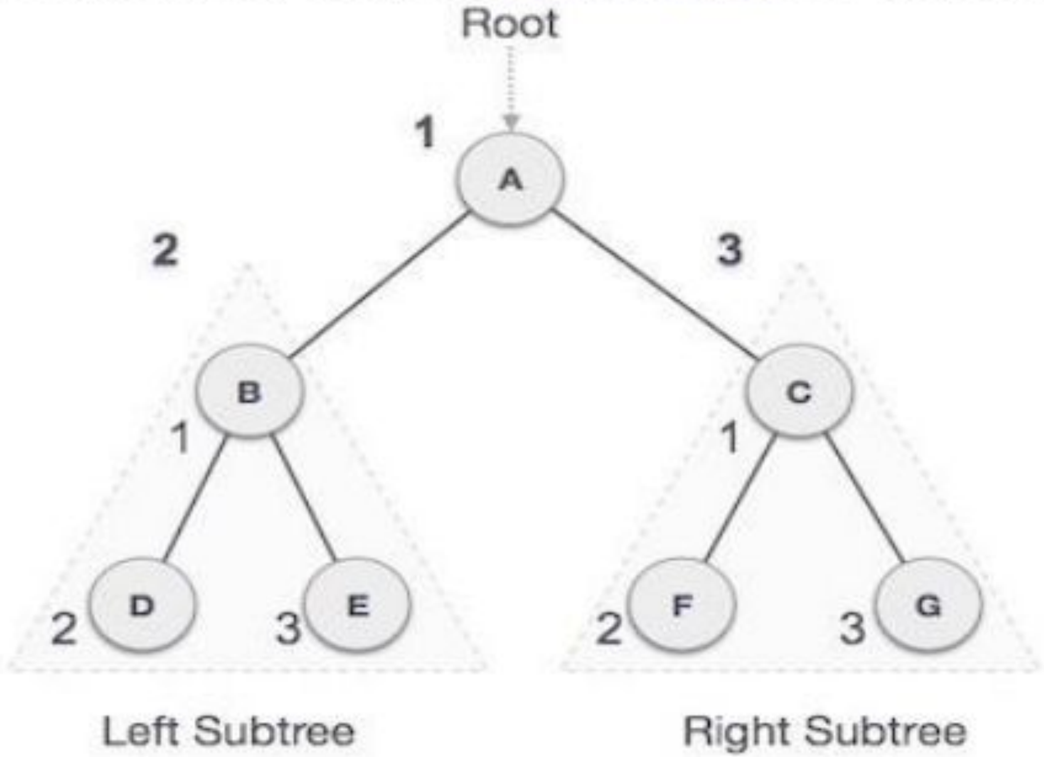
Preorder Traversal

Until all nodes are traversed −

Step 1 − Visit root node and process.

Step 2 − Recursively traverse left subtree in preorder.

Step 3 − Recursively traverse right subtree in preorder.



The output of pre-order traversal of this tree will be −

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

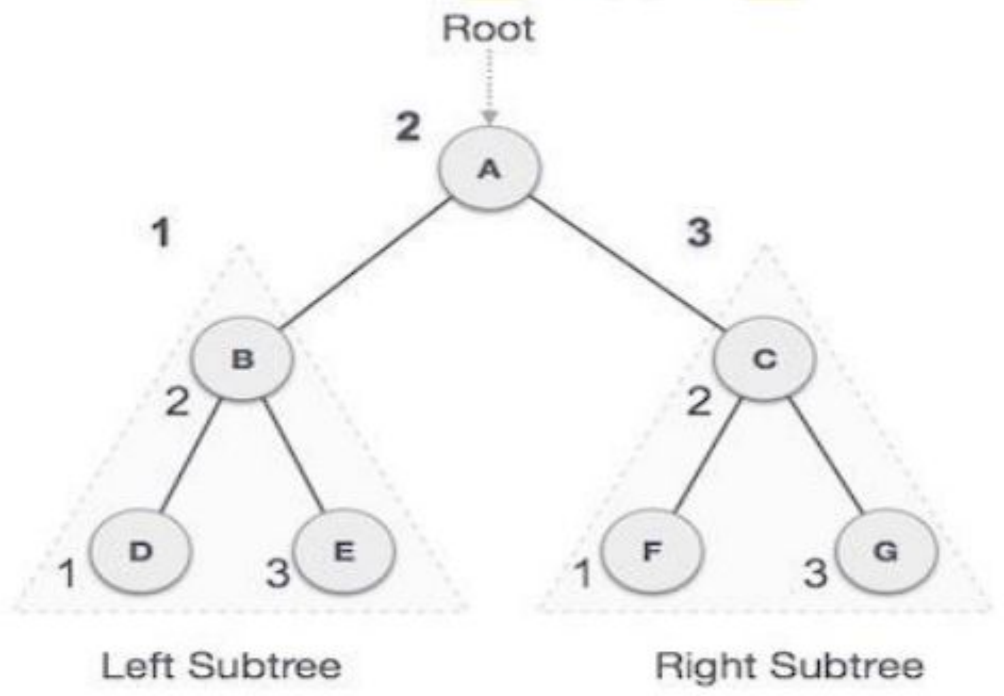
In order Traversal

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree in in-oeder.

Step 2 − Visit root node and process.

Step 3 − Recursively traverse right subtree in in-order.



The output of in-order traversal of this tree will be −

D → B → E → A → F → C → G

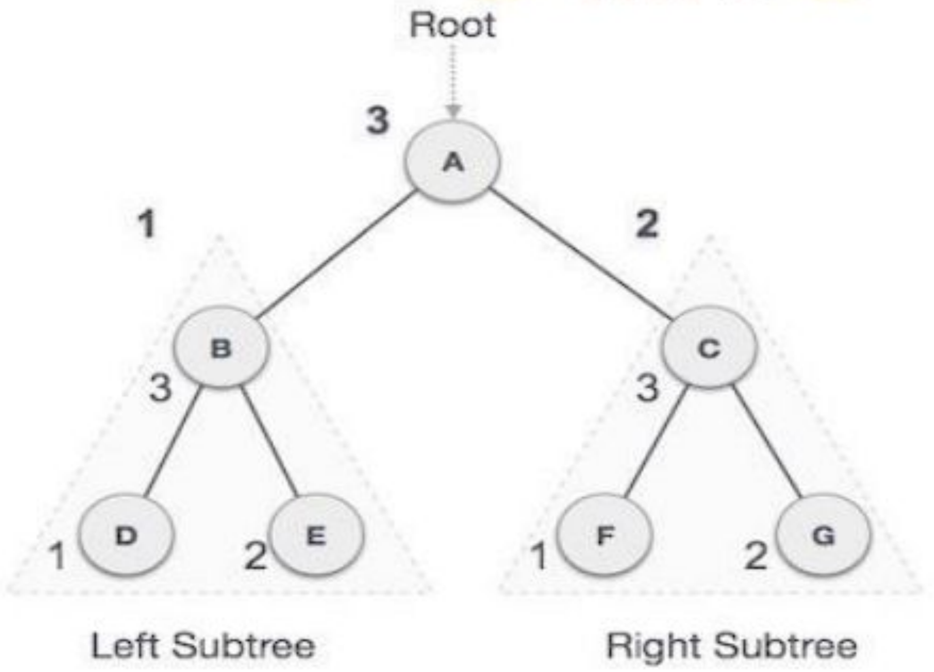
Post Order Traversal

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree in post-order.

Step 2 − Recursively traverse right subtree in post-order.

Step 3 − Visit root node and process.



The output of post-order traversal of this tree will be −

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Creation of Binary Tree from Traversal sequence

eg.1. Inorder- EACKFHDBG
Preorder- FAEKCDHGB

Solu.

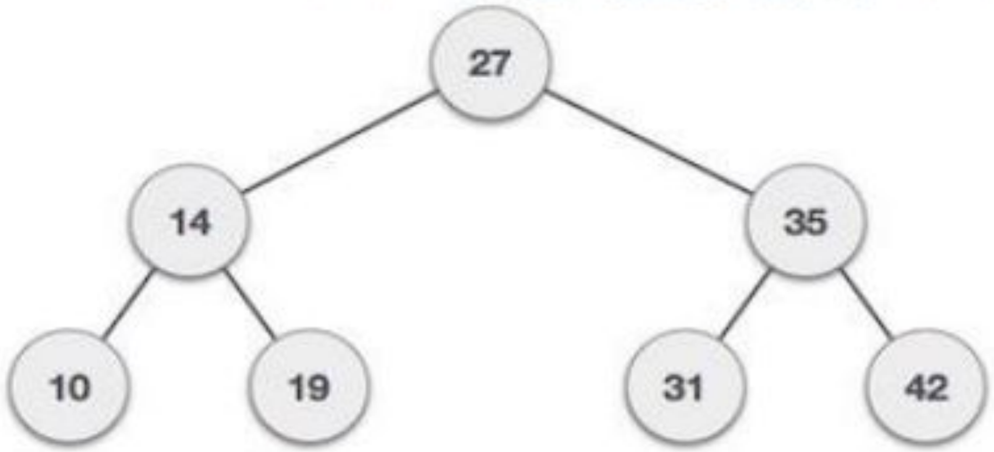
1. In preoder traversal root comes first. Hence F is root.
2. From Inoreder traversal we can find left and right descendants. That is EACK and HDBG
3. Among EACK, A comes first in preorder, therefore A is root of left subtree.
4. Similarly in HDBG, D comes first in preorder hence D is root of right subtree.
5. And soon.

Eg.2 Inorder- BIDACGEHF and
Postorder- IDBGCHFEA

3. 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

- A binary search tree (BST) is a tree in which all nodes has keys that follows the below mentioned properties −
- All keys are distinct.
- For every node X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X.
- For every node X, in the tree, the values of all the keys in the right subtree are greater than the key value in X.



Operations on Binary Search Tree

- Search
- Insert
- Delete

Search Operation

- Whenever an element is to be search;
  - Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree.
- This operation generally requires returning a pointer to the node in tree T that has key x, or NULL if there is no such node.

search() function:

```
struct node* search(struct node * root,int n)
{
struct node *p = root;
while(p!=NULL)
{
 if(n > p->data)
{
return(search(p->right_ptr, n));
}
else if (n < p->data)
{
return(search(p->left_ptr, n));
}
else return (p);
}
return NULL;
 }
```

Insert Operation

- Whenever an element is to be inserted.
- First locate its proper location.
- Start search from root node then if data is less than key value, search empty location in left subtree and insert the data.
- Otherwise search empty location in right subtree and insert the data.

```c
struct node *newNode(int item)
{
  struct node *temp = (struct node
*)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
```

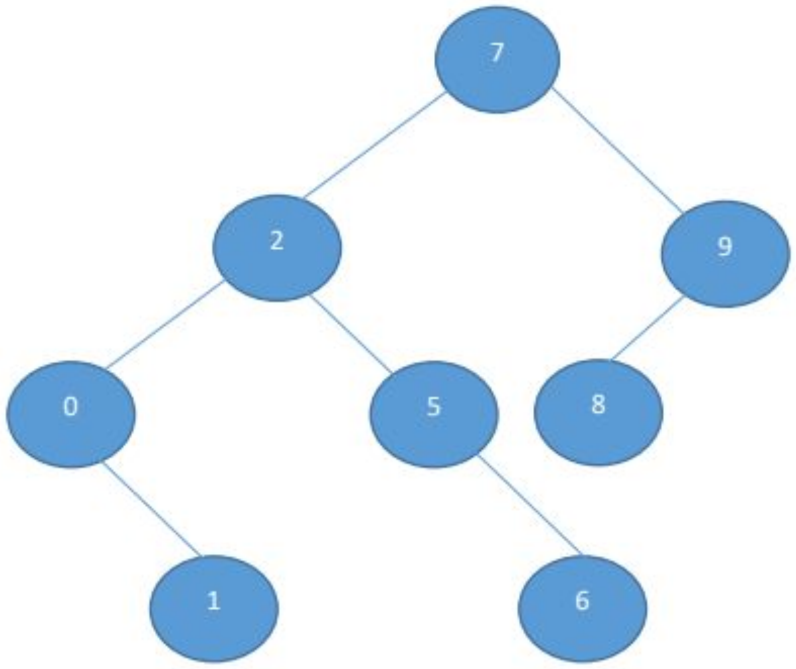
```c
struct node *insert(struct node *node, int key)
{
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the
node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}
```

Example:

Create BST

- BST can be created by using repeated insert operation.
- eg. Create BST for following sequence

7 2 9 0 5 6 8 1



Delete Operation

- Once we have found the node to be deleted, we need to consider several possibilities.

  i. A leaf node

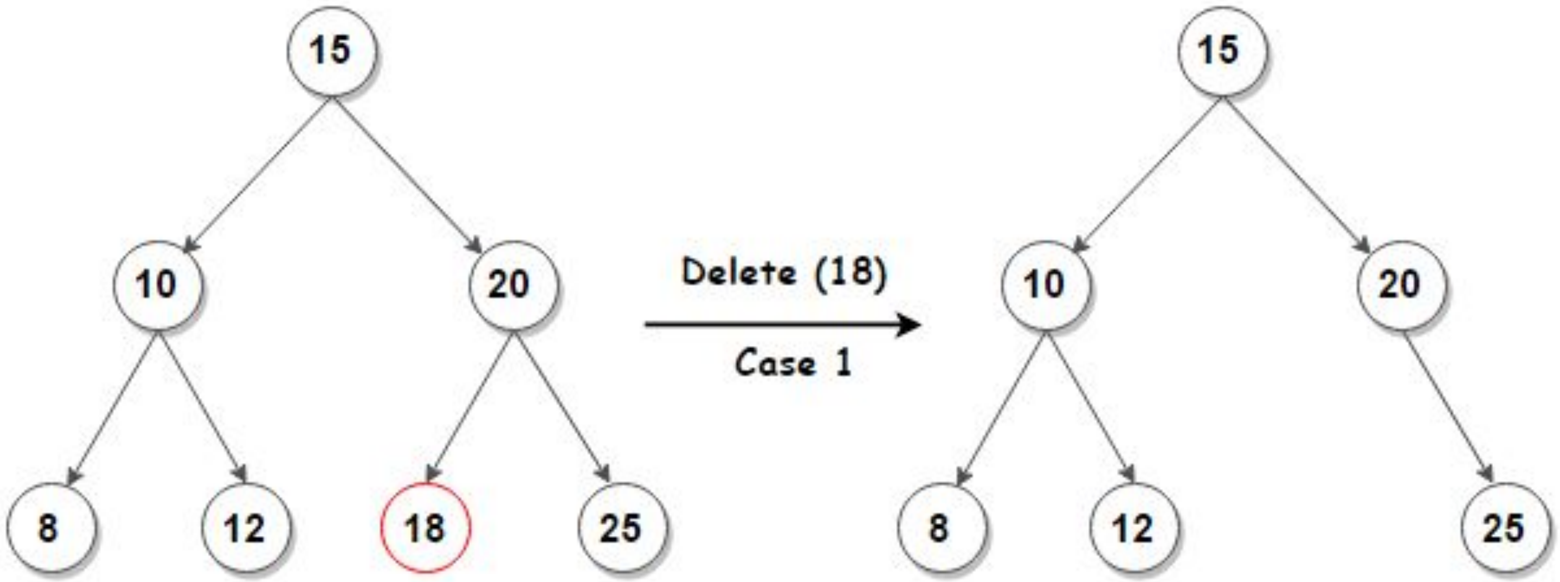
  ii. A node with one child

  iii. A node with two children

Delete..

A leaf node

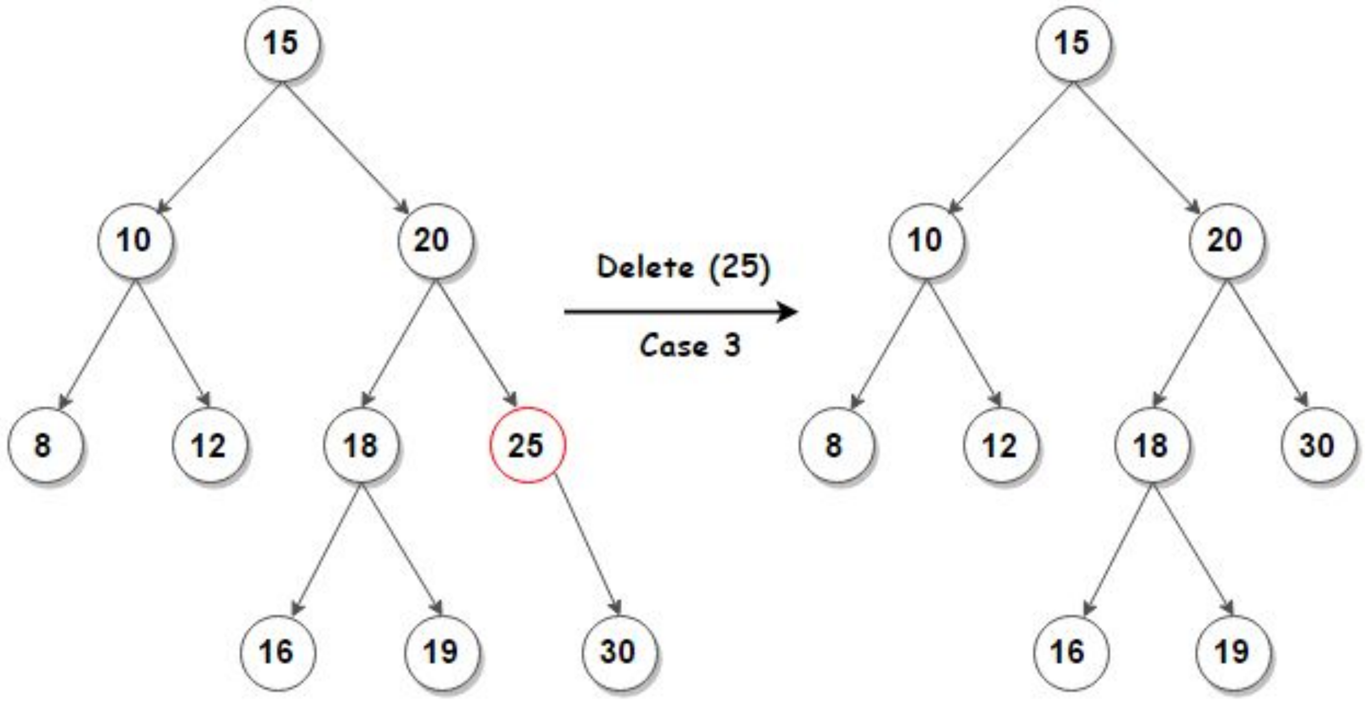
If the node is a leaf, it can be deleted immediately by setting the corresponding parent pointer to NULL.



Delete..

A node with one child

If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node.

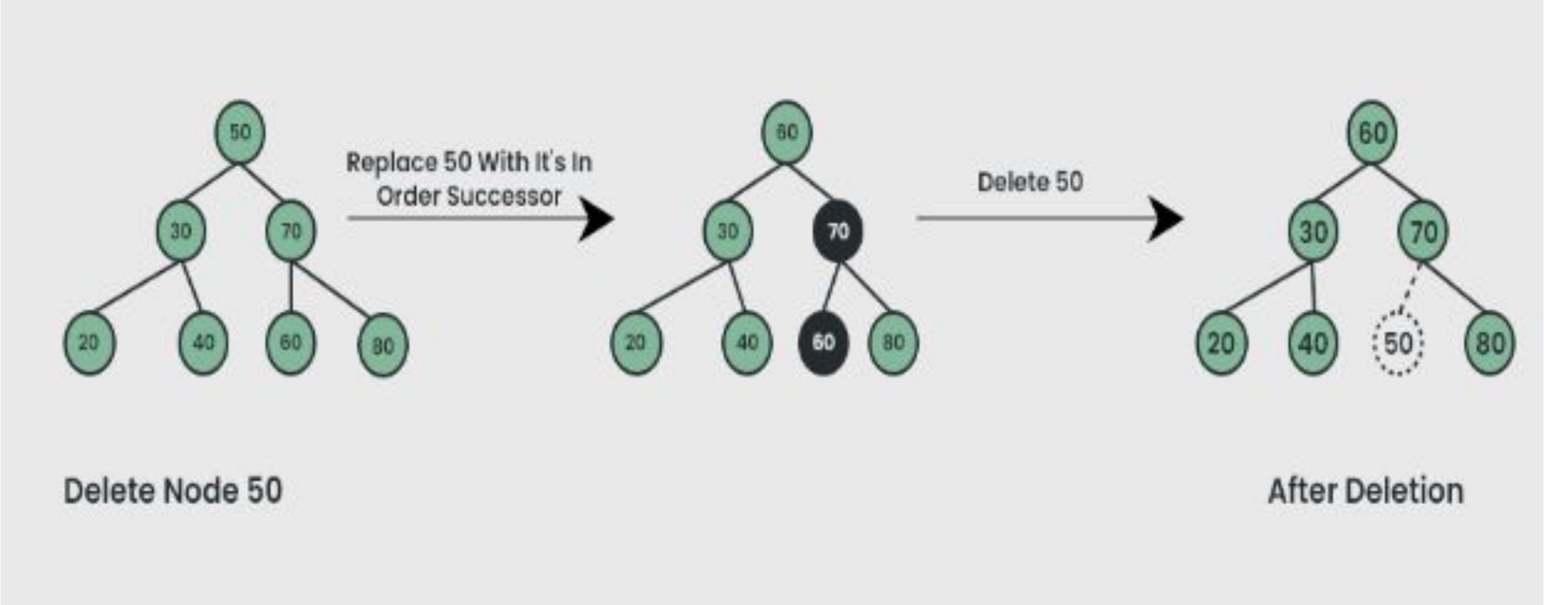


Delete..

A node with two children

• The general strategy is to replace the key of this node with the smallest key of the right subtree.

• The smallest child in right subtree will either be leaf node or a node with single right child.

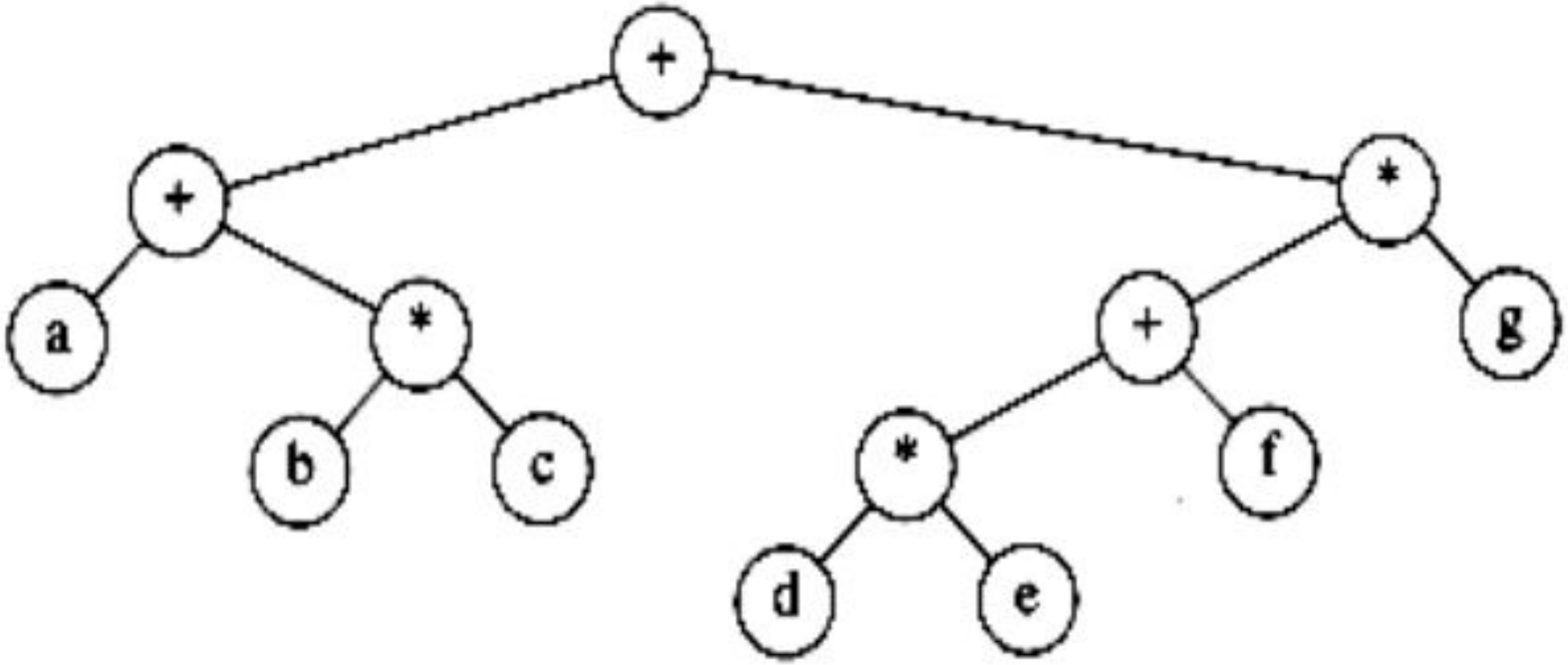


Application of Binary Tree- Expression Tree

- The leaves of an expression tree are operands, such as constants or variable names and the other nodes contain operators.
- This tree happens to be binary, because all the operations are binary.
- It is also possible for a node to have only one child, as is the case with the unary minus operator.
- We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

Example

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



Huffman Coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details.
- It was first developed by David Huffman.
- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

How Huffman Coding works?

- Suppose the string below is to be sent over a network.

Initial string

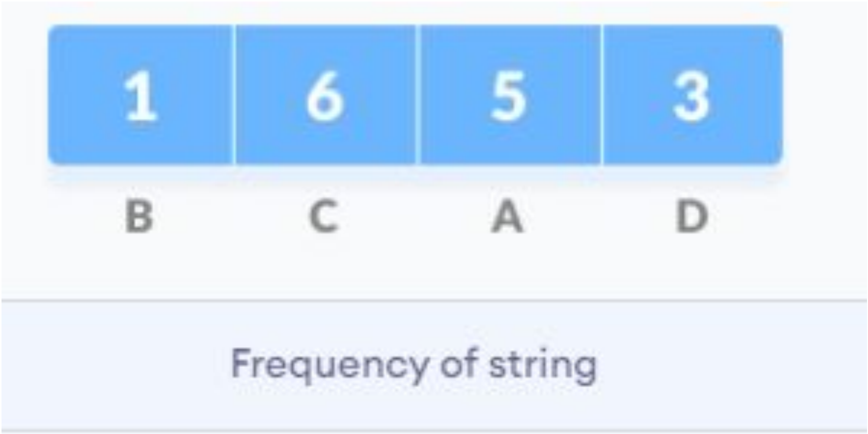
- Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.

- Using the Huffman Coding technique, we can compress the string to a smaller size.

- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

- Huffman coding is done with the help of the following steps.

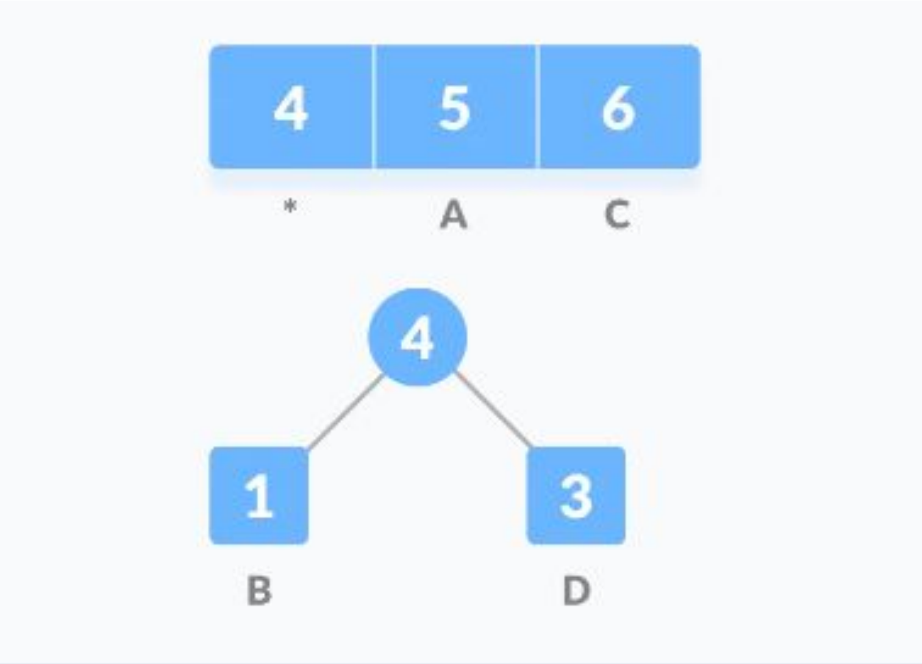
1. Calculate the frequency of each character in the string.



2. Sort the characters in increasing order of the frequency.

3.Make each unique character as a leaf node.

4.Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

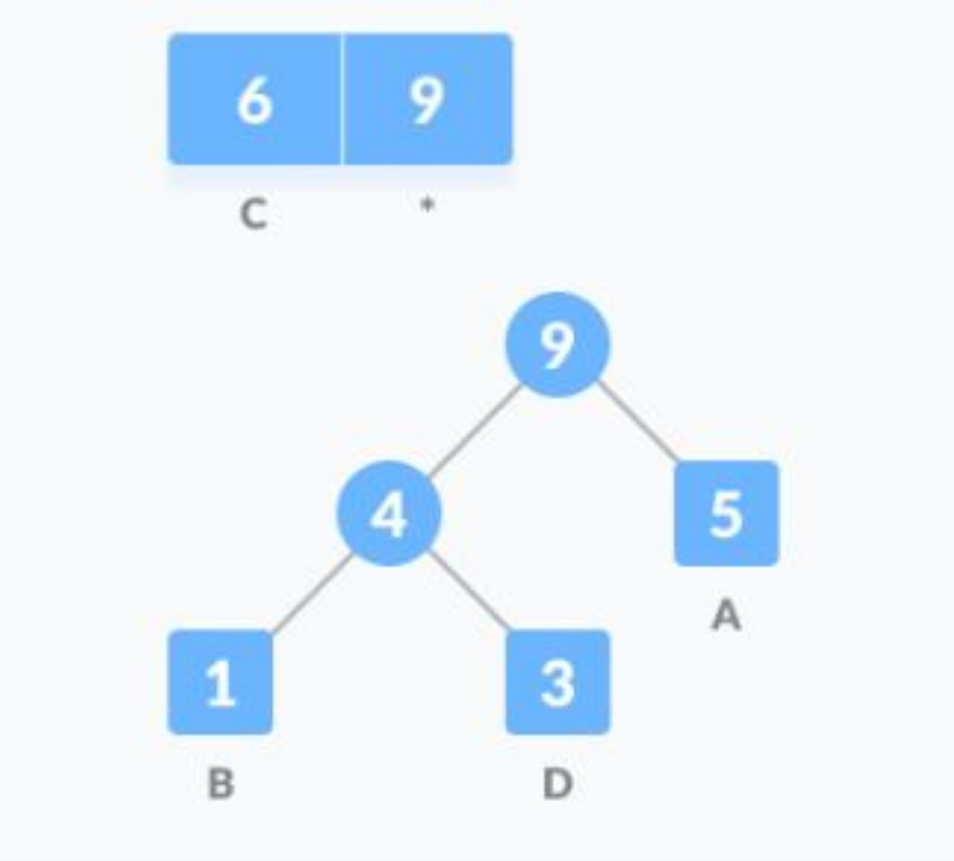


Getting the sum of the least numbers

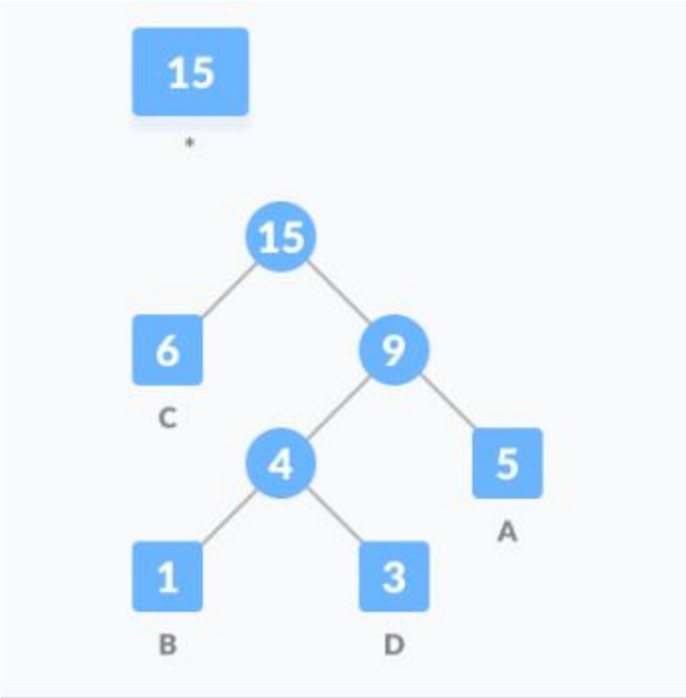
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies.

6. Insert node z into the tree.

7. Repeat steps 3 to 5 for all the characters.

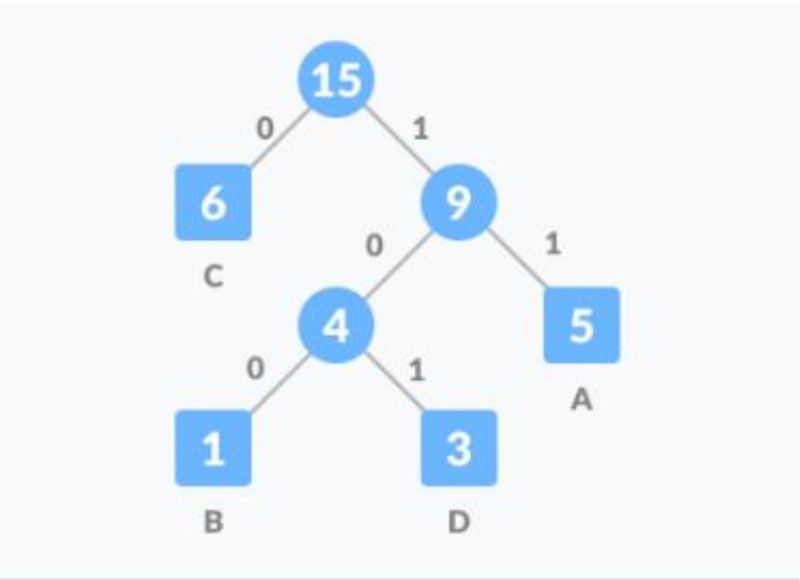


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. For each non-leaf node, assign 0 to the left edge and 1 to the right e



Assign 0 to the left edge and 1 to the right edge

1. Huffman code for character :

- To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.
- Following this rule, the Huffman Code for each character is-

C= 0

B=100

D=101

A=11

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

2. Average Code Length :

Using formula-01, we have-

Average code length

$= \sum (\text{frequency}_i \text{ x code length}_i) / \sum (\text{frequency}_i)$

= { (1 x 3) + (6x1) + (5 x 2) + (3 x 3) } / (1+6+5+3)

=1.87

3.Length of Huffman encoded message:

Total number of bits in Huffman encoded message

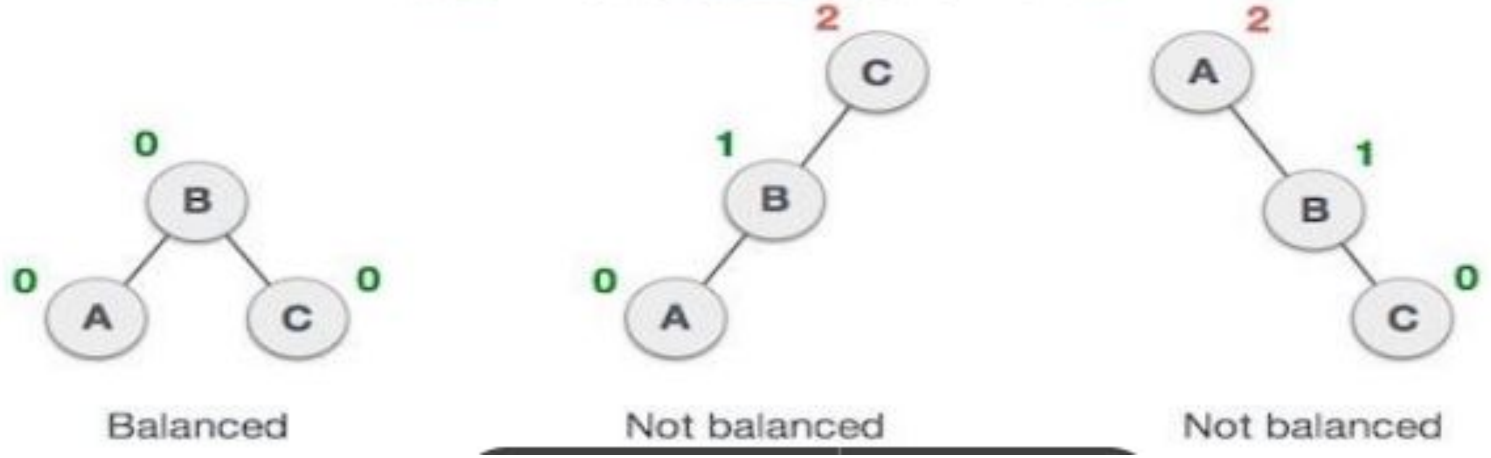
= Total number of characters in the message x Average code length per character

= 15x 1.87

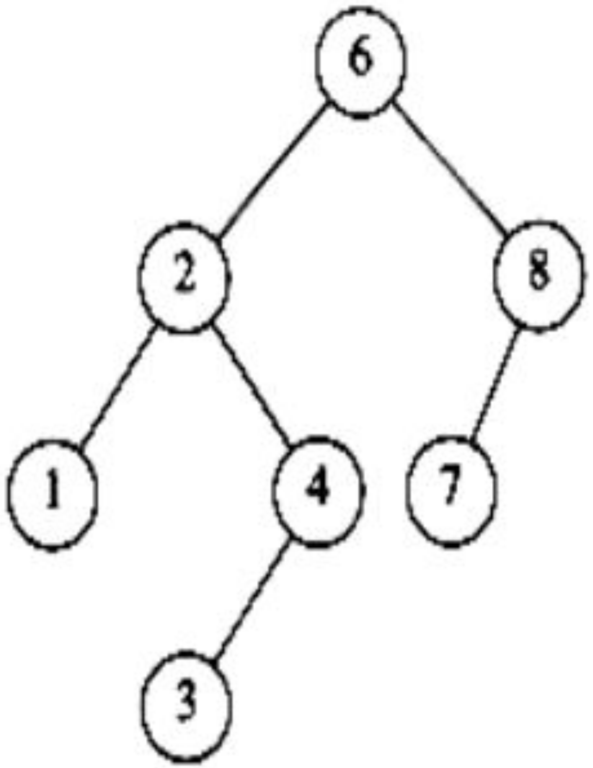
= 28.05

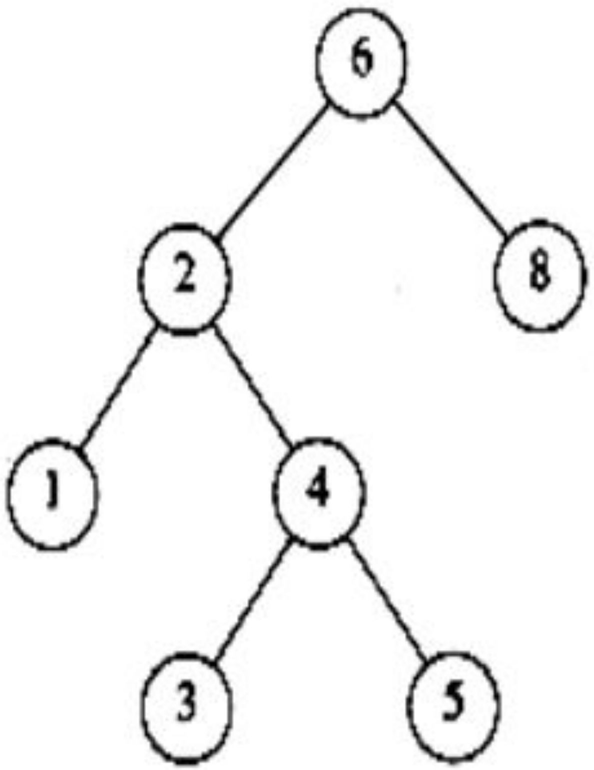
AVL Trees

- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a height balance condition.
- height balance condition
  - every node in the tree, the height of the left and right subtrees can differ by at most 1.
  - i.e. | **height of left subtree – height right subtree | <= 1**
  - This difference is called ***Balance Factor***.



Exercise - AVL Trees??





AVL Rotations

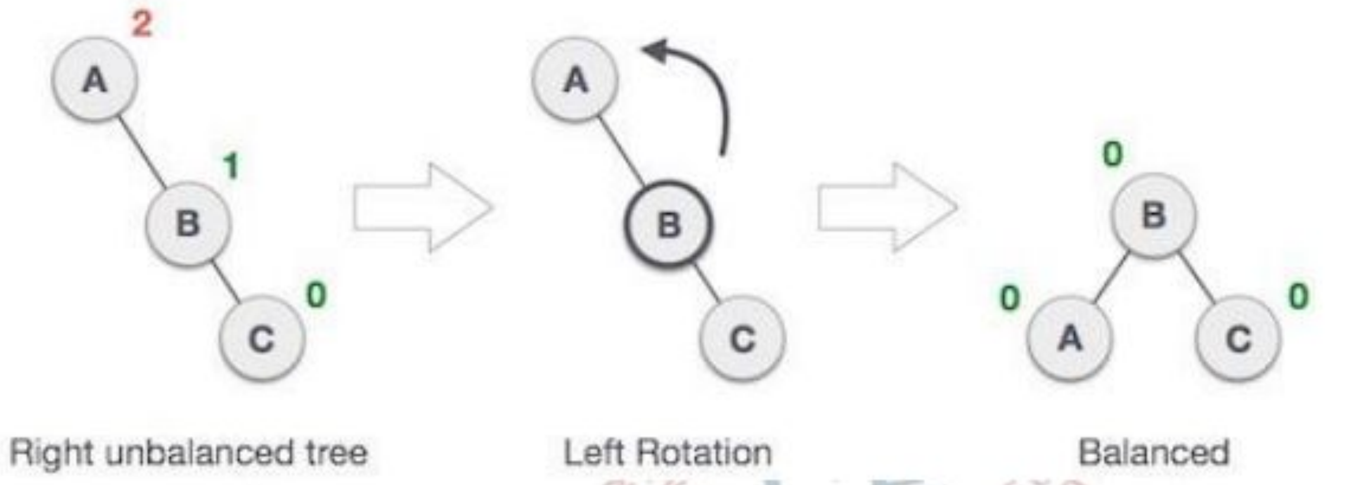
To make itself balanced, an AVL tree may perform four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

- First two rotations are single rotations and next two rotations are double rotations.

AVL Rotations - Left Rotation

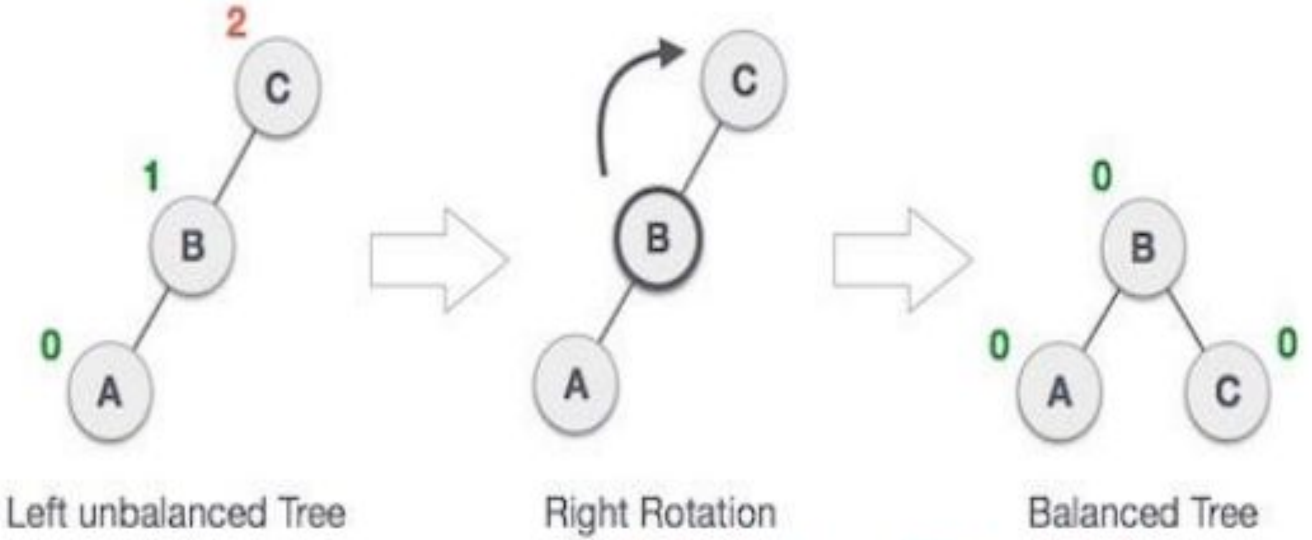
- If a tree become unbalanced, when a node is inserted into the right subtree tree, then we perform single left rotation,



- Here, node **A** has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making **A** left-subtree of B.

AVL Rotations - Right Rotation

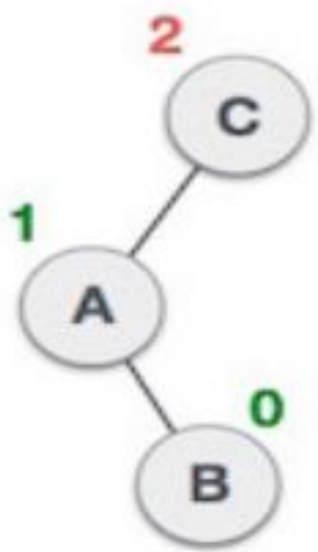
- AVL tree may become unbalanced if a node is inserted in the left subtree of tree. The tree then needs a right rotation.



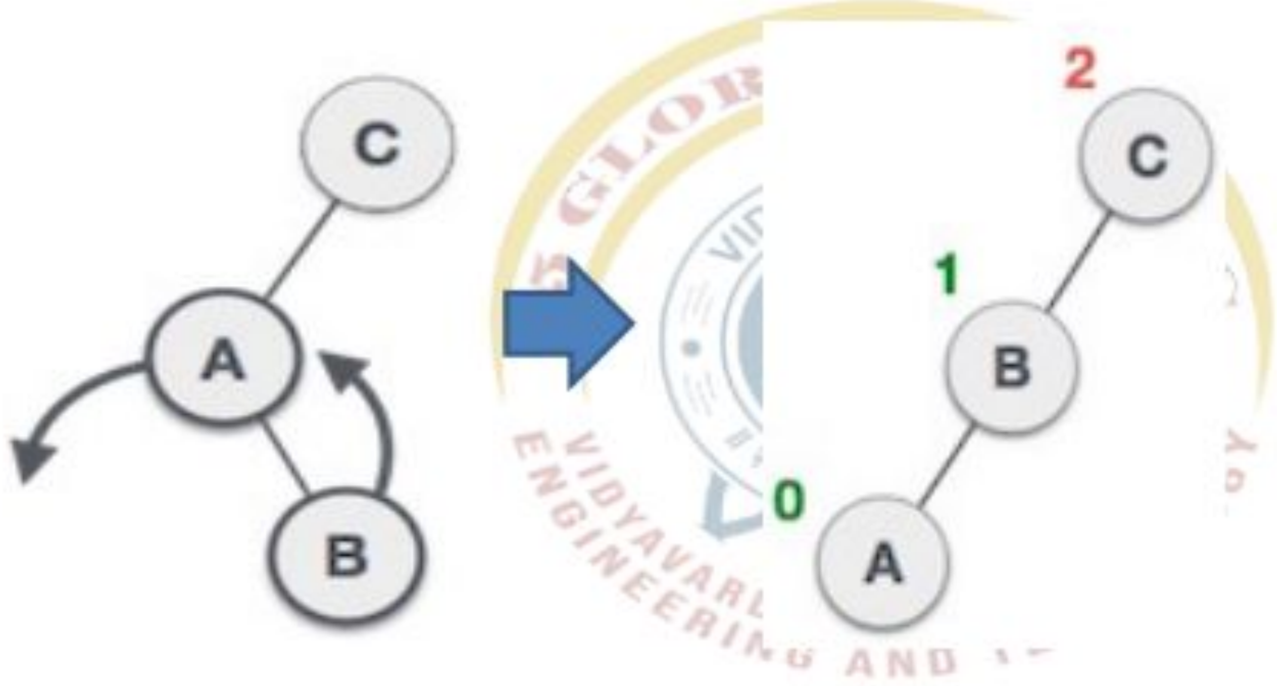
- The unbalanced node becomes right child of its left child by performing a right rotation.

AVL Rotations – Left-Right Rotation

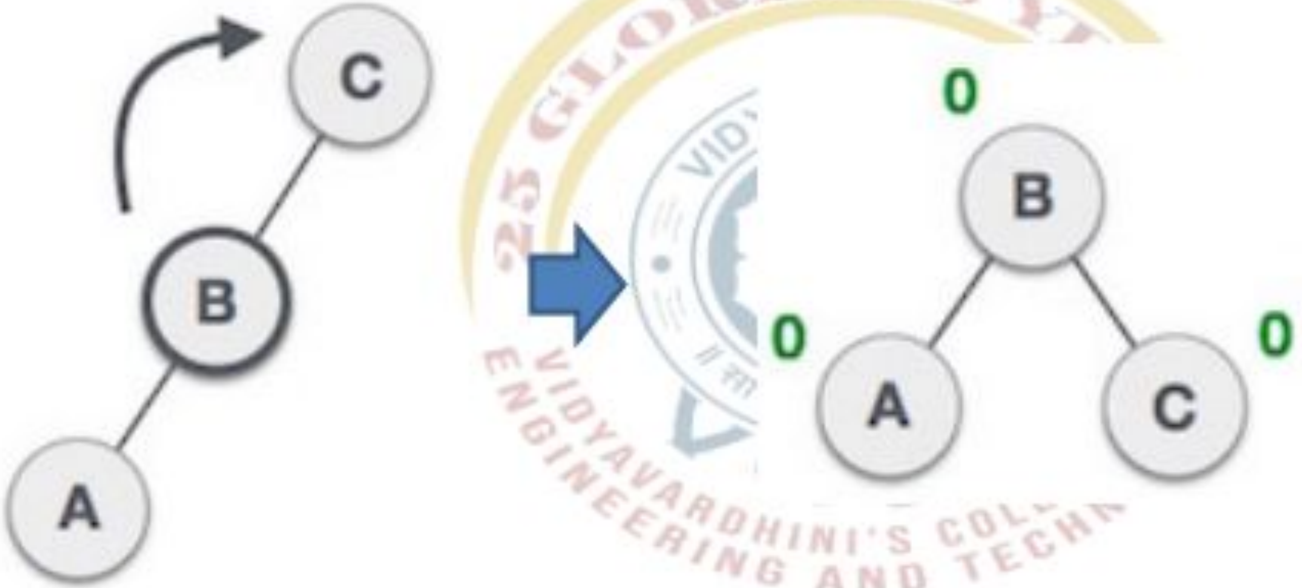
- Double rotations are slightly complex version of already explained versions of rotations.
- A left-right rotation is combination of left rotation followed by right rotation.
- A node has been inserted into right subtree of left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



- We first perform left rotation on left subtree of C.
- This makes A, left subtree of B.



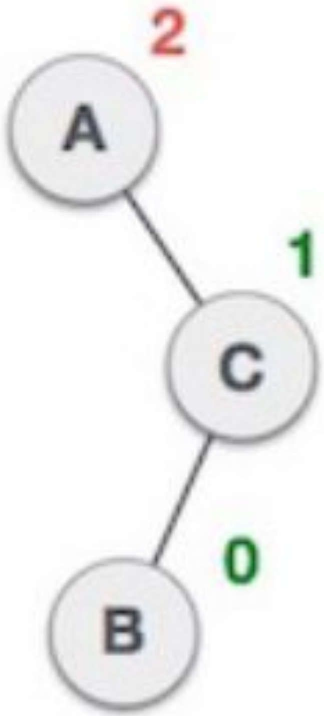
- Node C is still unbalanced but now, it is because of left-subtree of left-subtree.
- We shall now right-rotate the tree making **B** new root node of this subtree. **C** now becomes right subtree of its own left subtree.



- The tree is now balanced.

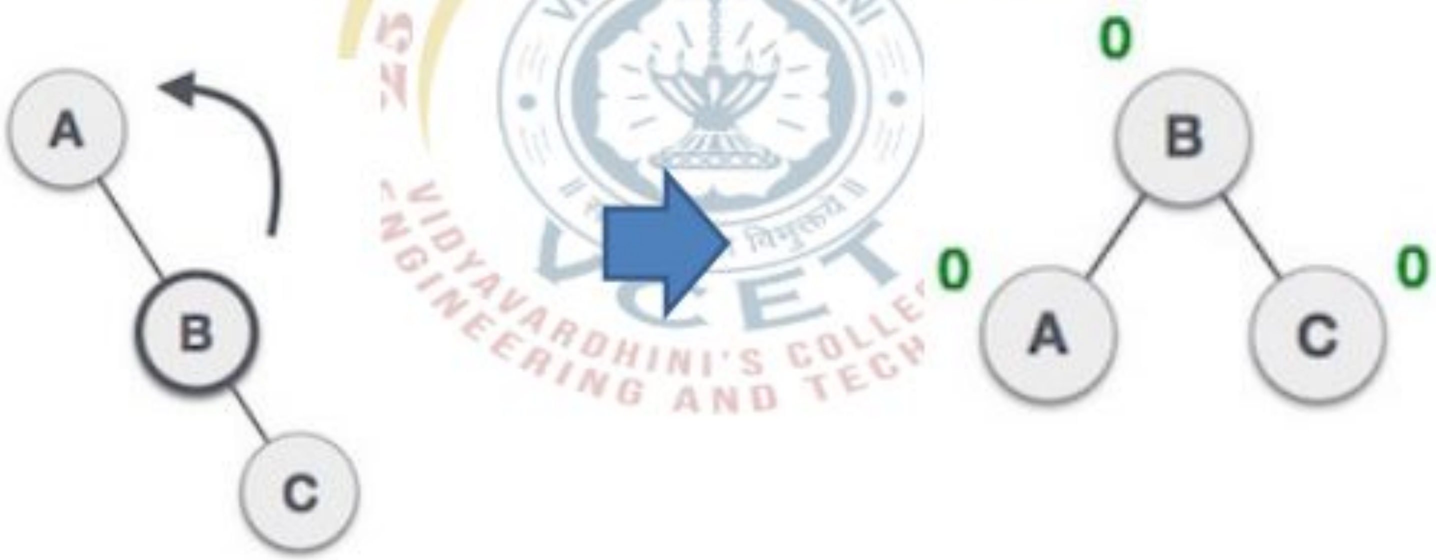
AVL Rotations – Right-Left Rotation

- It is a combination of right rotation followed by left rotation.
- A node has been inserted into left subtree of right subtree.
- This makes **A** an unbalanced node, with balance factor 2.

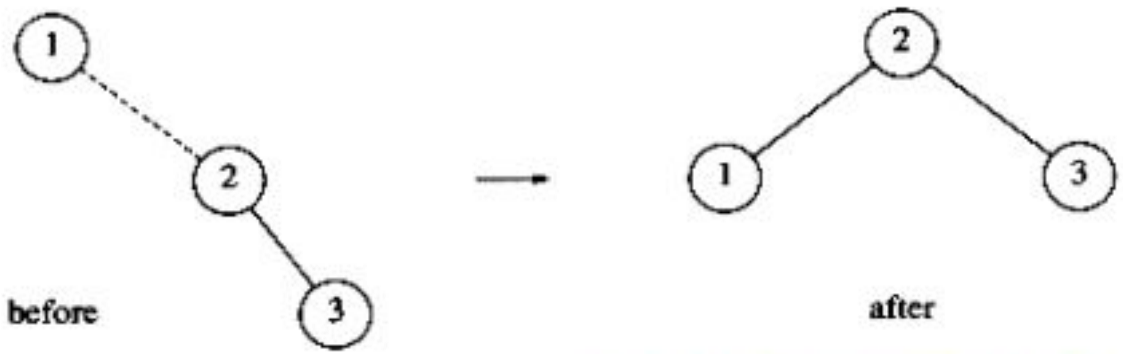


- First, we perform right rotation along C node, making C the right subtree of its own left subtree B.
- Now, B becomes right subtree of A.

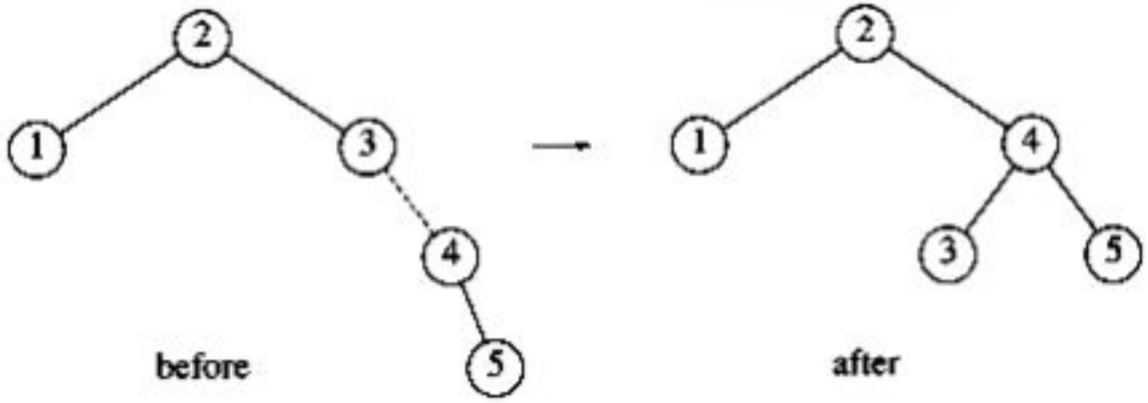
- Node **A** is still unbalanced because of right subtree of its right subtree and requires a left rotation.
- A left rotation is performed by making **B** the new root node of the subtree.
- **A** becomes left subtree of its right subtree **B**.



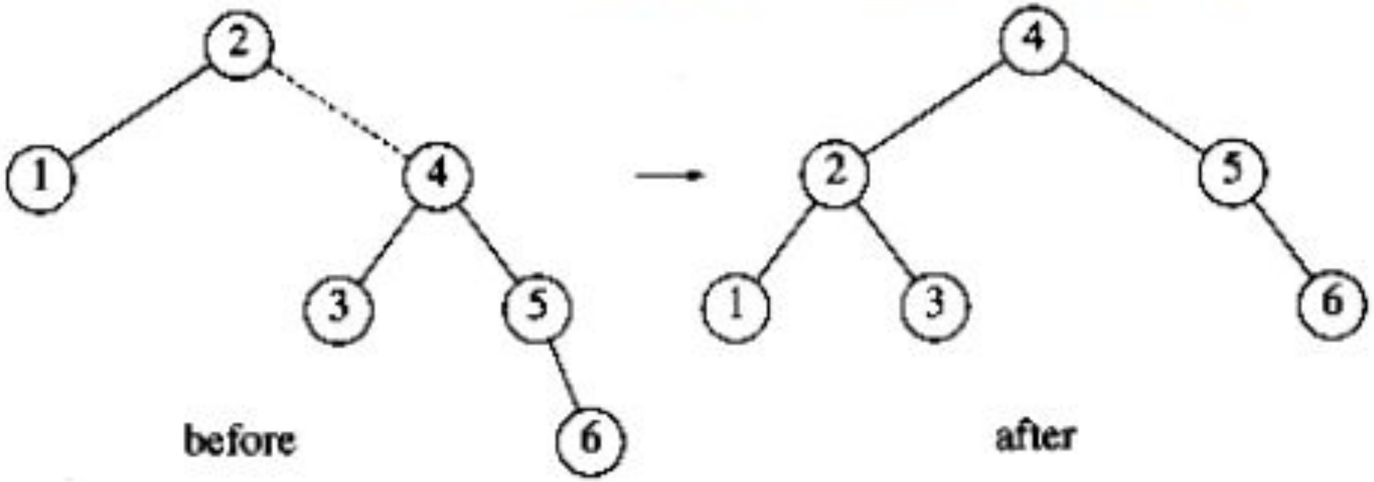
eg. Create AVL tree for sequence 1, 2, 3, 4, 5, 6, 7



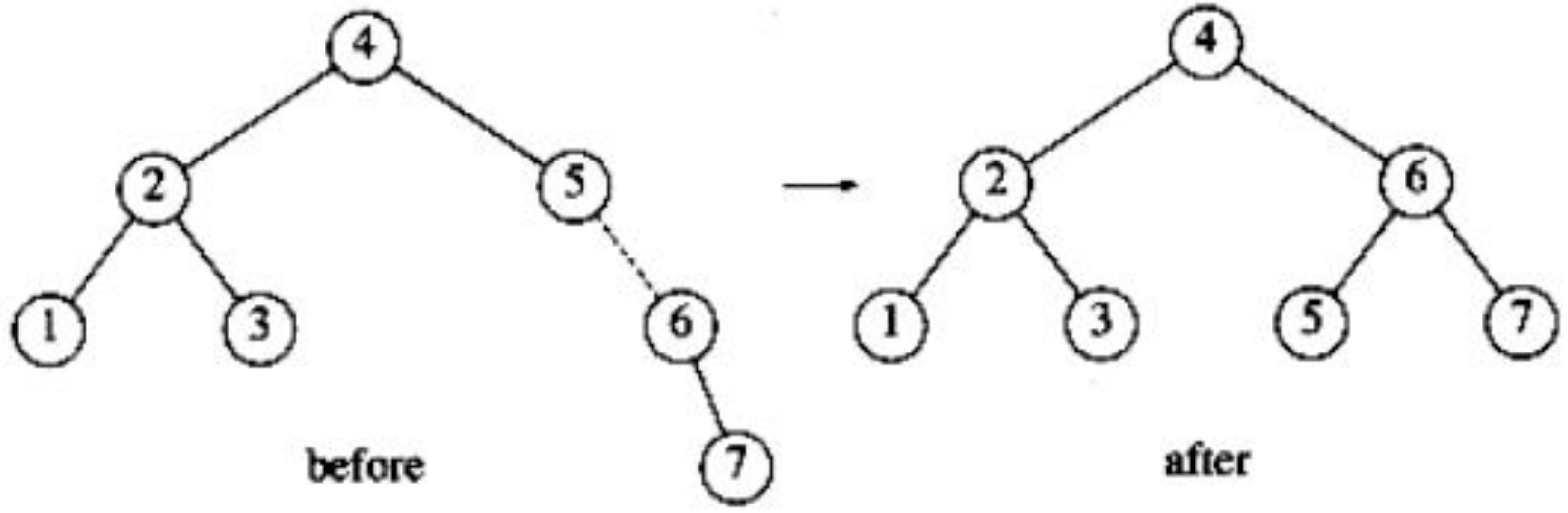
Next, we insert the key 4, which causes no problems, but the insertion of 5 creates a violation at node 3, which is fixed by a left rotation.



- Next, we insert 6. This causes a balance problem for the root.
- The rotation is performed by making 2 a child of 4 and making 4's original left subtree the new right subtree of 2.



- The next key we insert is 7, which causes another rotation.

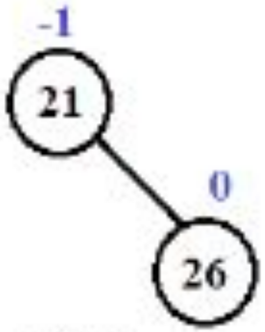


Construction of the AVL Tree for the given Sequence 21, 26, 30, 9, 4, 14, 28, 18,15,10, 2, 3, 7

Step 1 - Insert 21

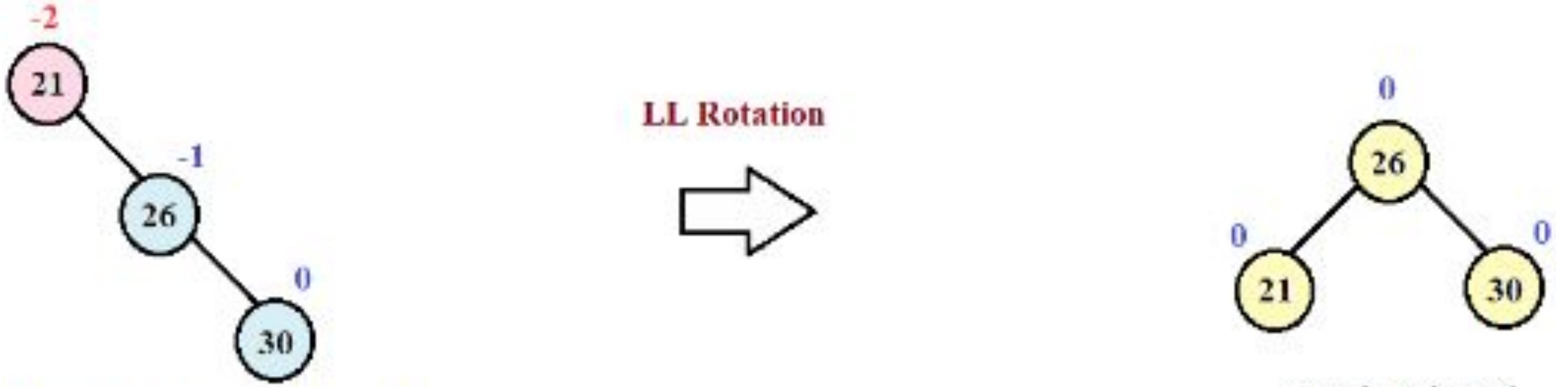
Tree is Balanced

Step 2 - Insert 26



Tree is Balanced

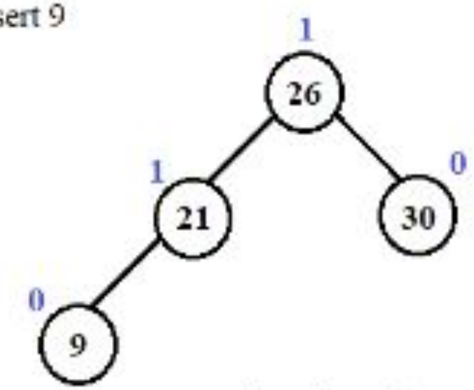
Step 3 - Insert 30



Tree is Not Balanced, Need a Rotation

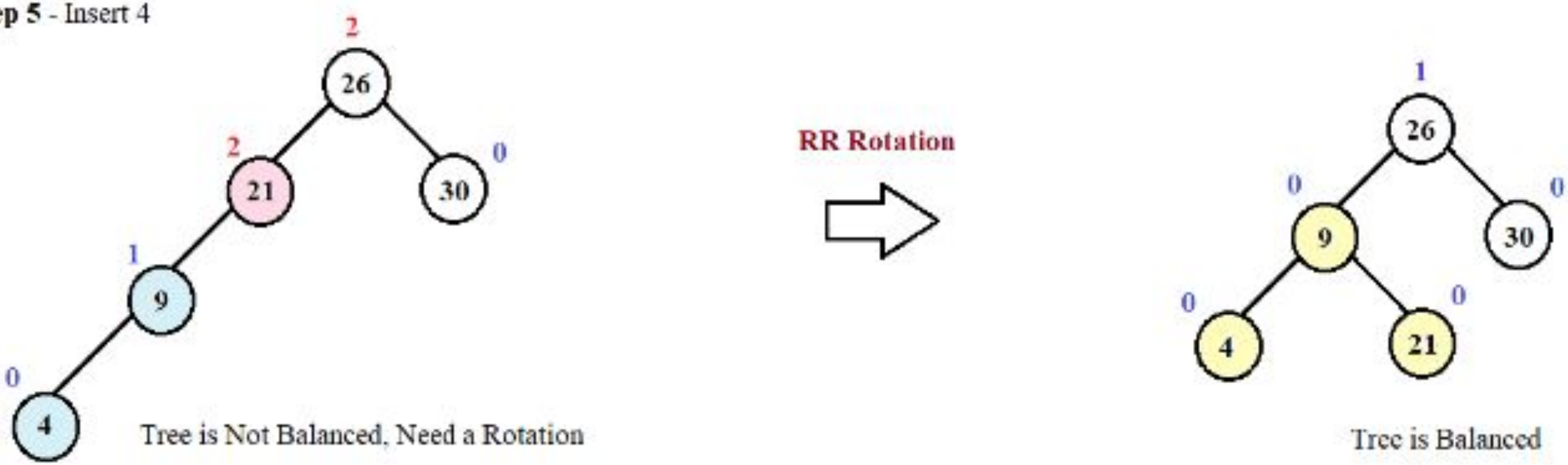
Tree is Balanced

Step 4 - Insert 9



Tree is Balanced

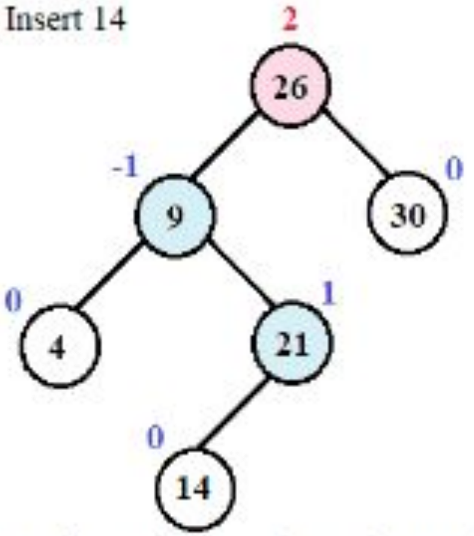
Step 5 - Insert 4



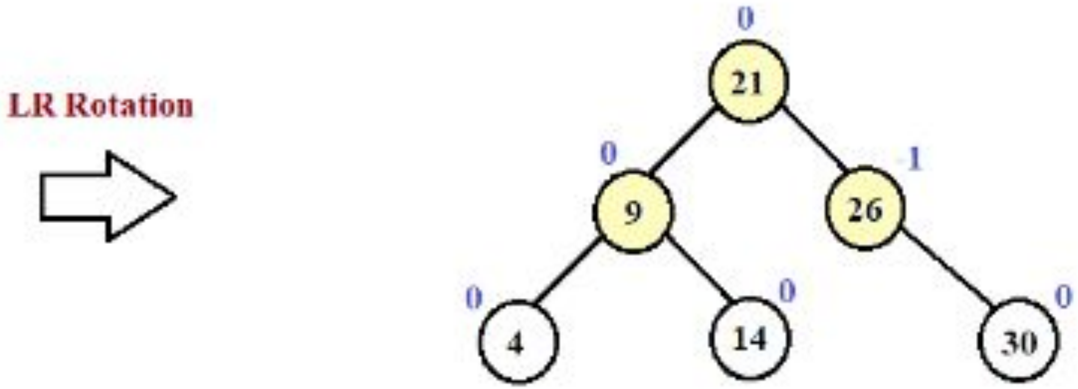
Tree is Not Balanced, Need a Rotation

Tree is Balanced

Step 6 - Insert 14

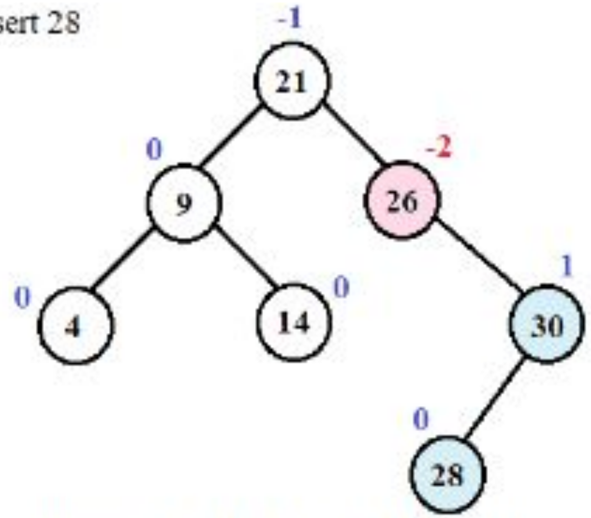


Tree is Not Balanced, Need a Rotation

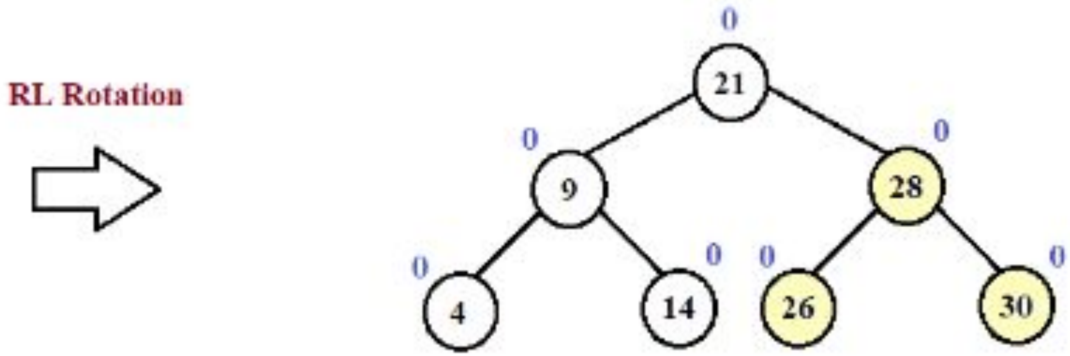


Tree is Balanced

Step 7 - Insert 28

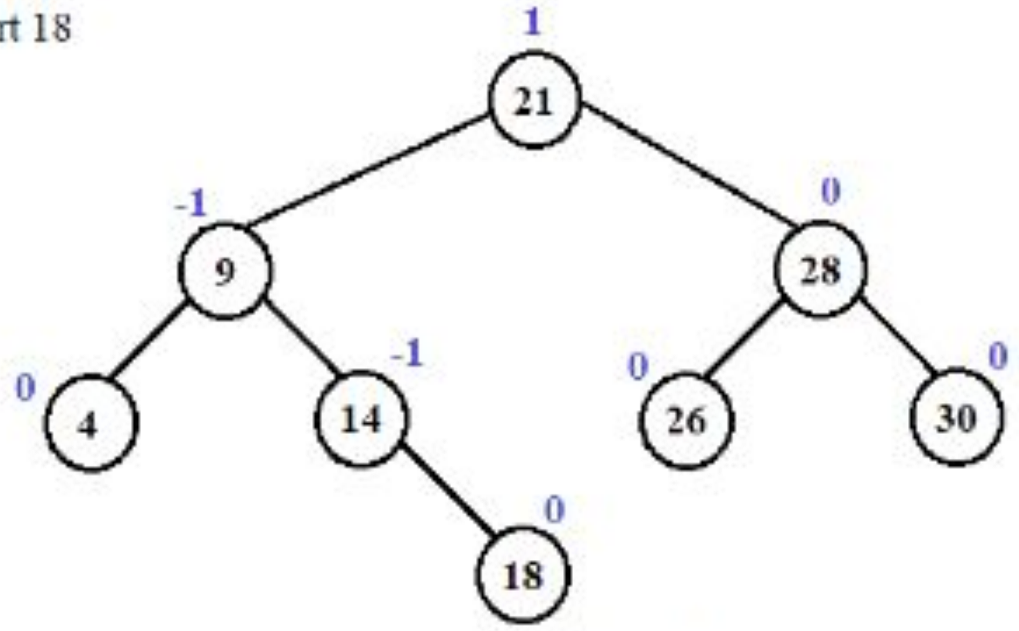


Tree is Not Balanced, Need a Rotation



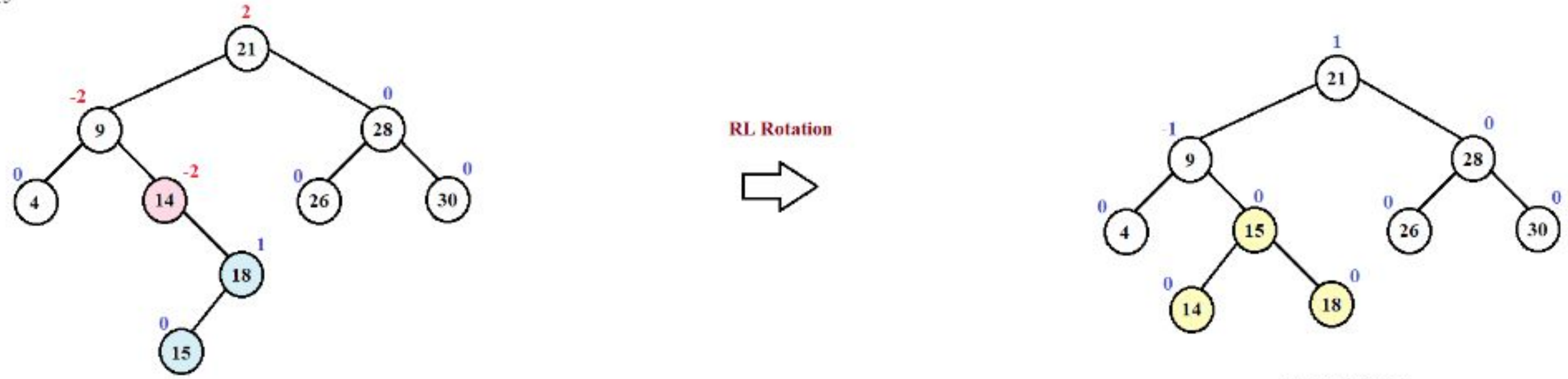
Tree is Balanced

Step 8 - Insert 18



Tree is Balanced

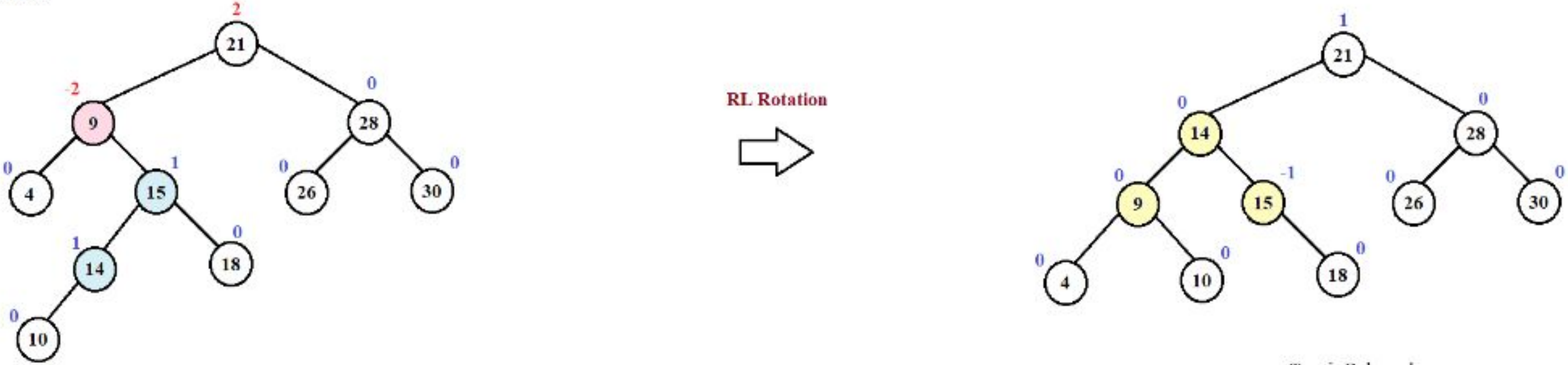
Step 9 - Insert 15



Tree is Not Balanced, Need a Rotation

Tree is Balanced

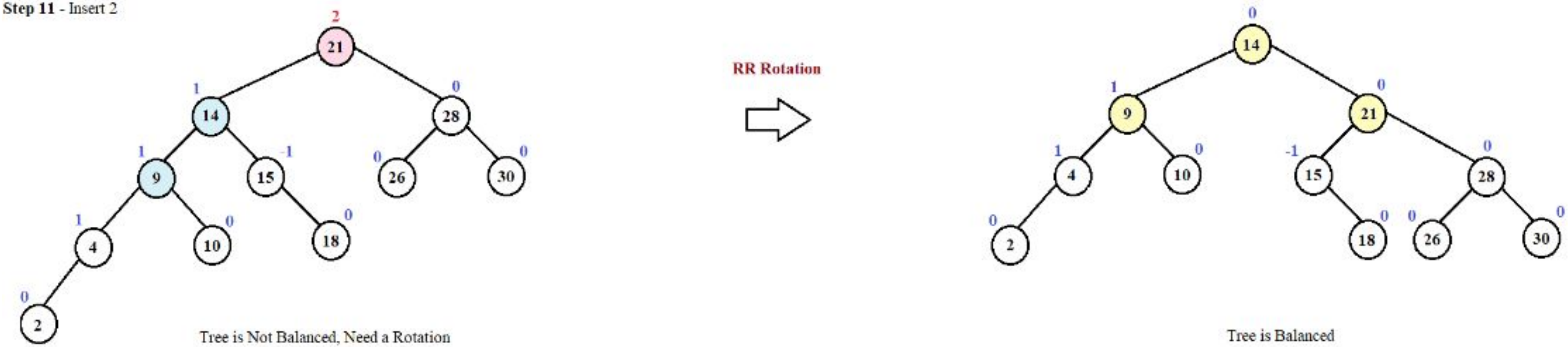
Step 10 - Insert 10



Tree is Not Balanced, Need a Rotation

Tree is Balanced

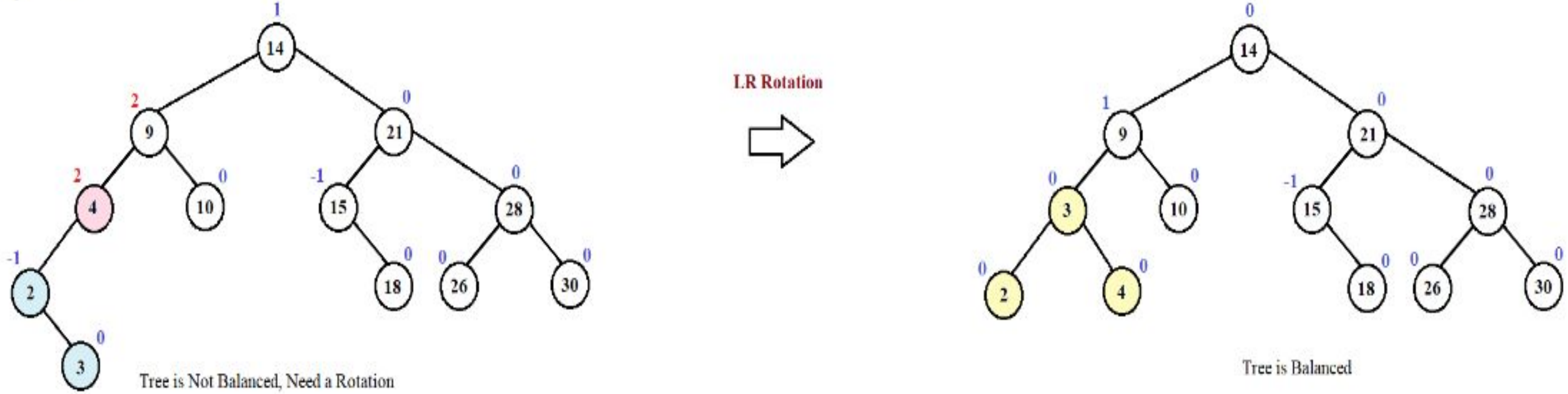
Step 11 - Insert 2



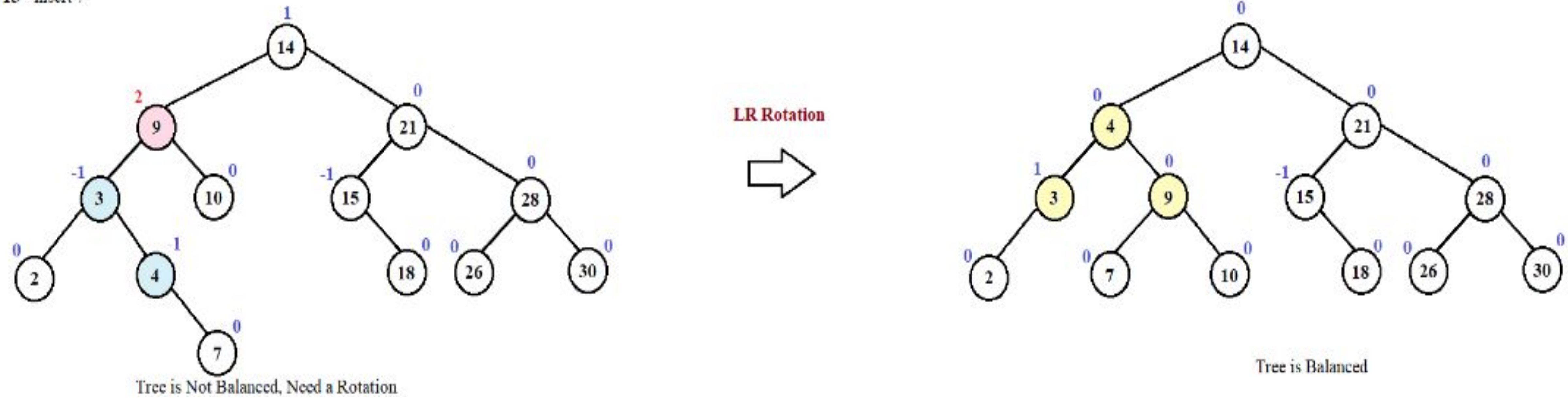
Tree is Not Balanced, Need a Rotation

Tree is Balanced

Step 12 - Insert 3



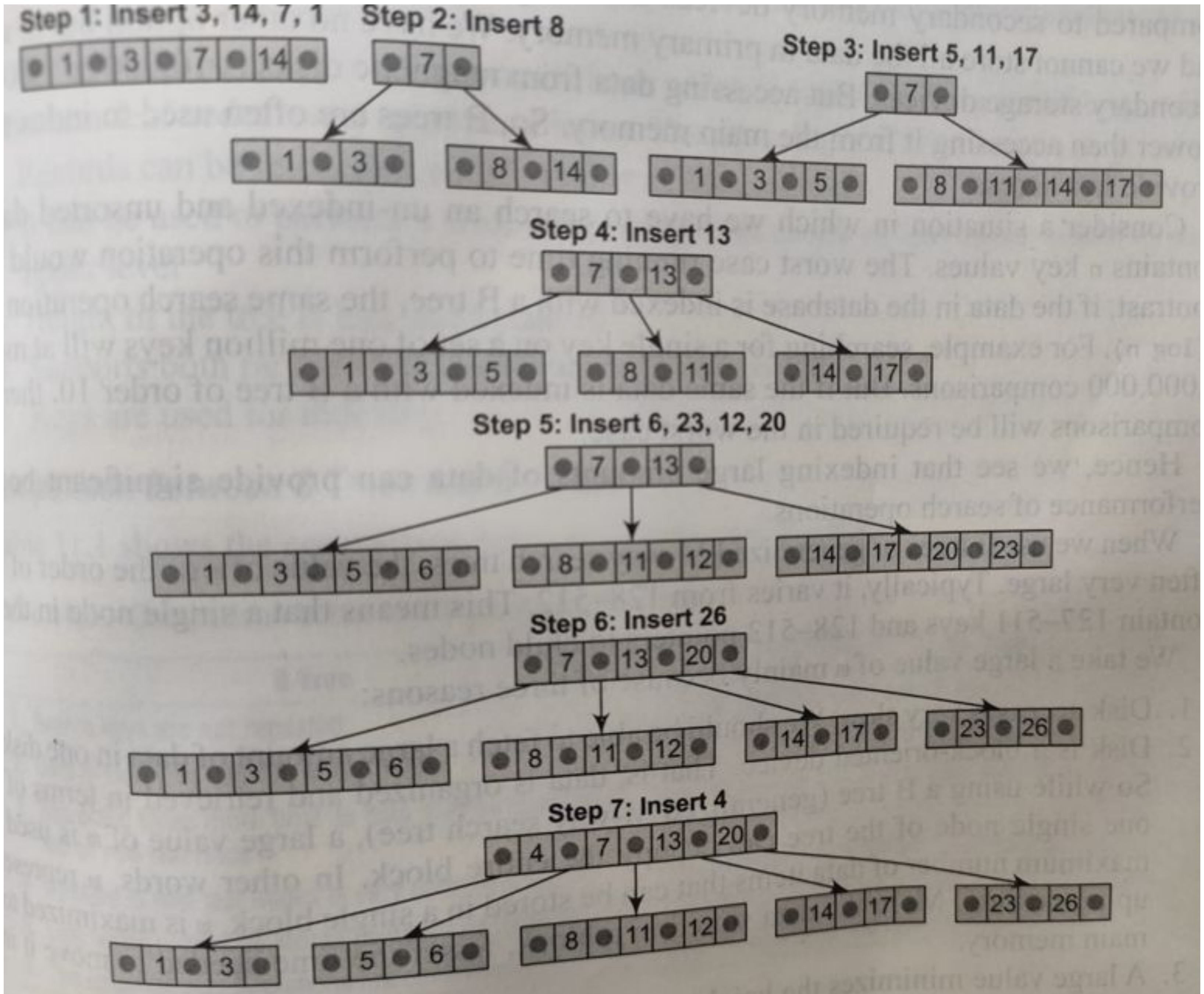
Step 13 - Insert 7



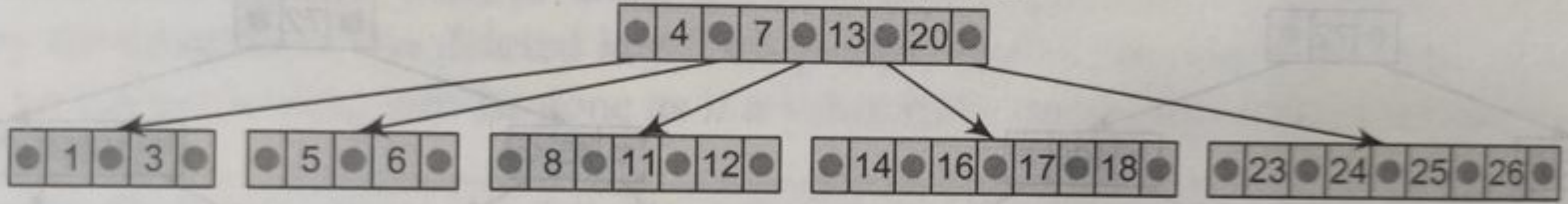
B Tree

- Specialized m-way tree that can be widely used for disk access.
- A B-Tree of order m can have at most m-1 keys and m children.
- A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties:
  - Every node in a B-Tree contains at most m children.
  - Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
  - The root nodes must have at least 2 nodes.
  - All leaf nodes must be at the same level.
- It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

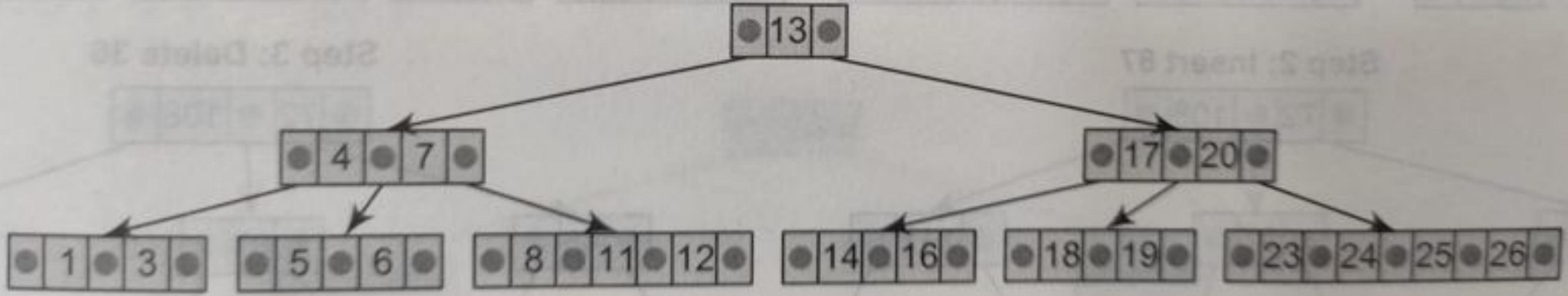
Create a B tree of order 5 by inserting following element
3 ,14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20 ,26, 4, 16, 18, 24, 25 ,19


Step 1: Insert 3, 14, 7, 1
1 3 7 14
Step 2: Insert 8
7
1 3
8 14
Step 3: Insert 5, 11, 17
7
1 3 5
8 11 14 17
Step 4: Insert 13
7 13
1 3 5
8 11
14 17
Step 5: Insert 6, 23, 12, 20
7 13
1 3 5 6
8 11 12
14 17 20 23
Step 6: Insert 26
7 13 20
1 3 5 6
8 11 12
14 17
23 26
Step 7: Insert 4
4 7 13 20
1 3
5 6
8 11 12
14 17
23 26

Step 8: Insert 16, 18, 24, 25



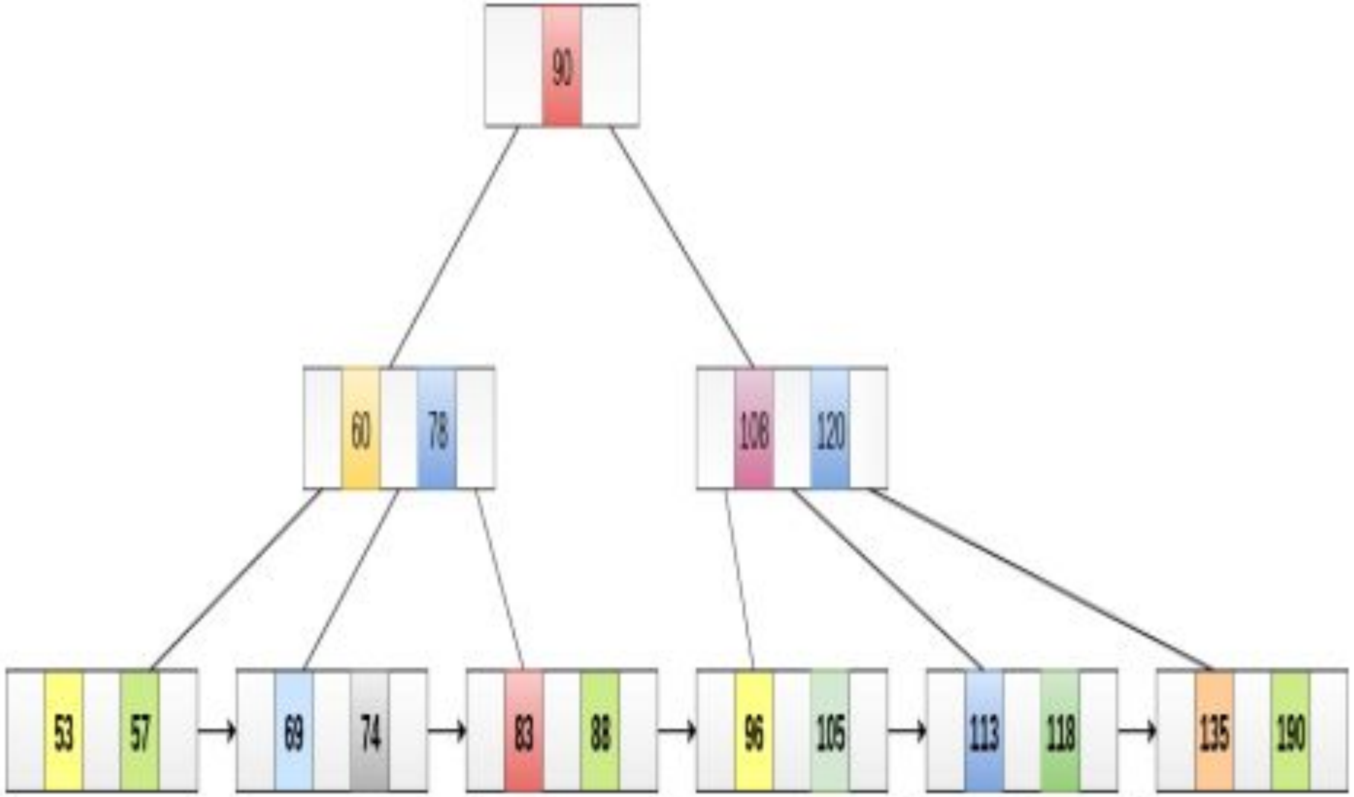
Step 9: Insert 19



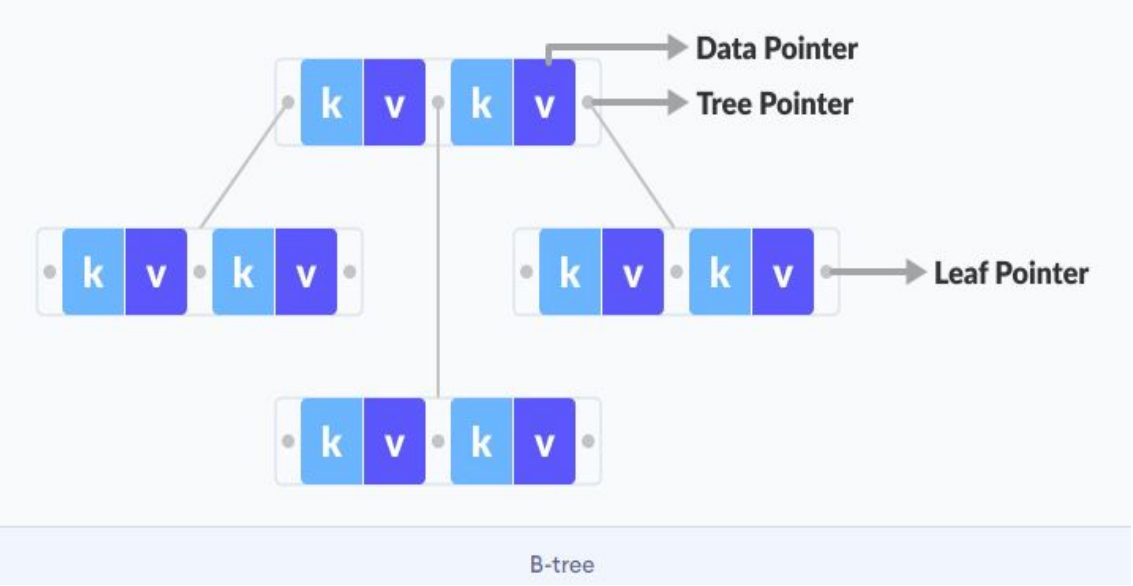
B+ Tree

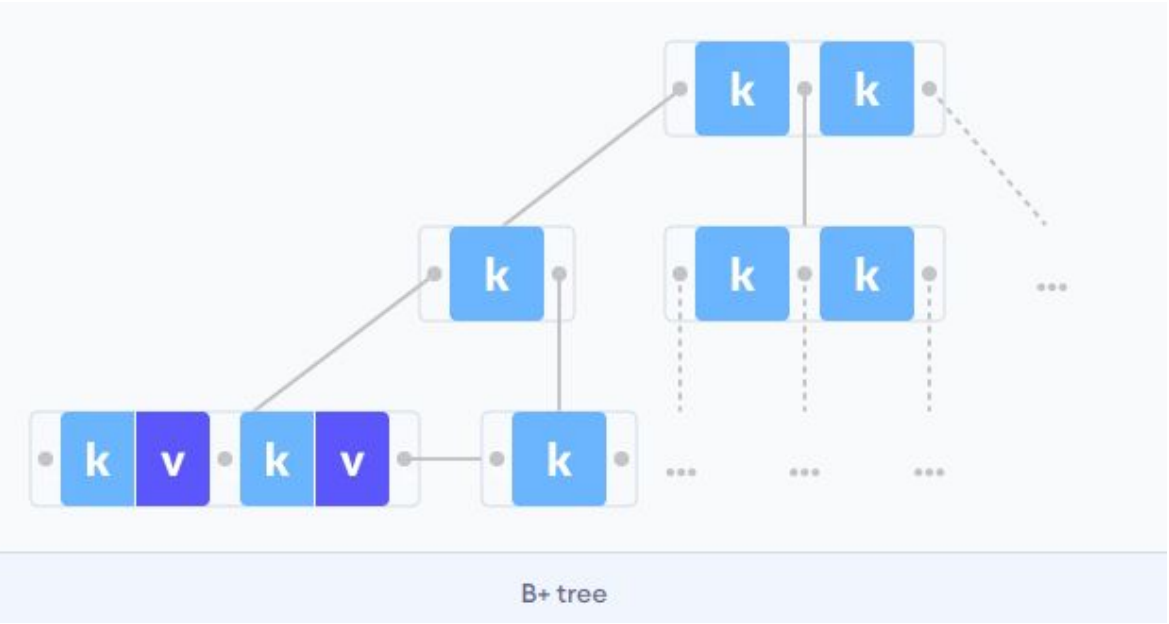
- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Trees are used to store the large amount of data which cannot be stored in main memory. The internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

B+ Tree of order 3



Comparison between a B-tree and a B+ Tree





- The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.
- The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.