# UNIT 4

**Syllabus: TREES AND HASHING** : Linear Vs Non-Linear Data Structures -General Tree – Terminologies -Binary Tree – Expression Tree - Traversals - Binary Search Tree – AVL Tree – Red block Tree – Splay Tree – B Tree. - Hashing: Introduction – Hash Function Methods- Collision Resolution.

## Linear Vs Non-Linear Data Structures

**Data structures** are normally **divided into two broad** categories.

(i) Primitive data structures
(ii) Non-primitive data structures

**Primitive Data Structure**

These are **basic structures and are directly operated** upon by **the machine instructions.** In general, they have different representations on different computers. **Integer, floating point numbers, character constants, string constants, pointers** etc.

**Non-Primitive Data Structures**

These are **more sophisticated data structures**. These are **derived from the primitive data structures**. The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. **Arrays, lists and files** are examples.

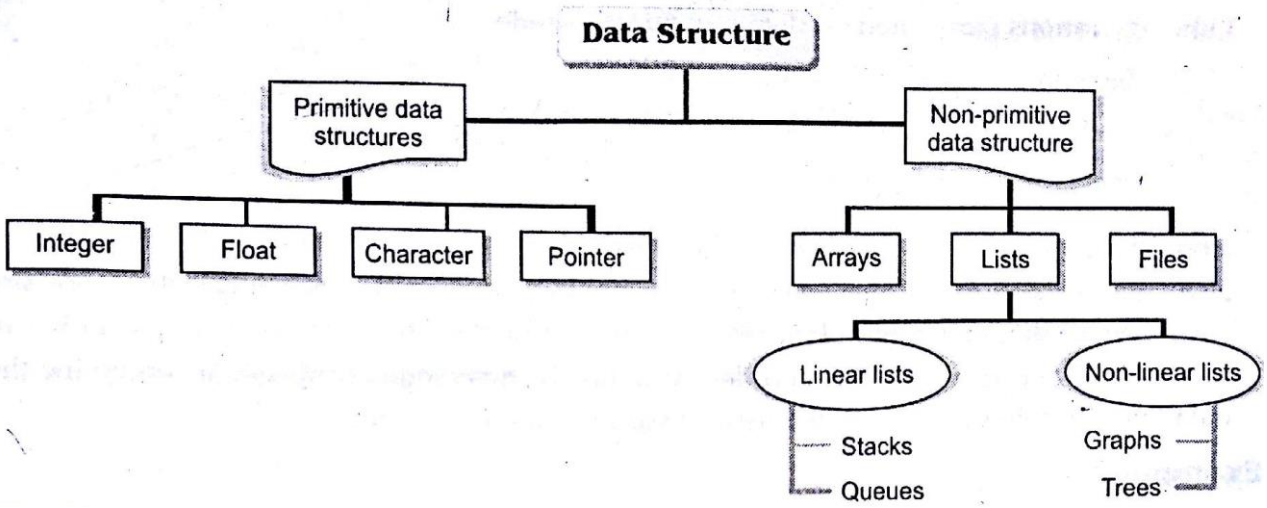



Figure 1 data Structure

1. **Linear data structures –** Those data structure where the data elements are organised in some sequence is called Linear data structures. Here operation on data structure are possible in a sequence. Stack, queue, array are example of linear data structure.

2. **Non Linear data structures –** Those data structure where the data elements are not organised in some sequence, organised in some arbitrary function without any sequence is called Non linear data structures. Graph, Tree are example of linear data structure.

**2.1. Trees** - A **Tree can be defined as finite set of data items (nodes)**. Tree is **non primitive non-linear type of data structures** in which data items are arranged or stored in a sorted sequence. Trees represent the hierarchical relationship between various elements. In trees:

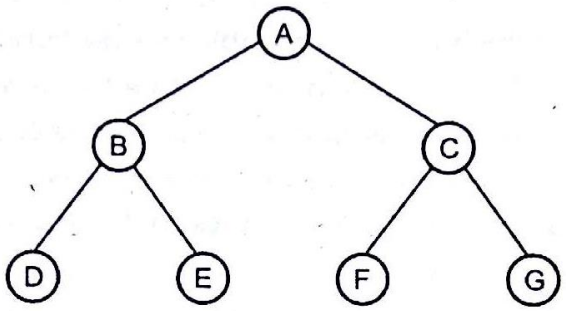



Figure 2 Binary. Tree

1. There is a **special data item** at the top of hierarchy **called the Root** of the tree.
2. The remaining data items are partitioned into number of mutually exclusive subsets, each of which is itself, a tree, which is **called the subtree**.
3. The tree always grows in length towards bottom in data structures, unlike natural trees which grow upwards.

The tree structure organizes the data into branches, which relate the information.

**2.2. Graph** - Graph is a mathematical **non primitive non-linear data structure** capable of representing many kinds of physical structures. A **graph G (V, E) is a set of vertices V and a set of edges E**. An edge connects a pair of vertices and many have weight such as length, cost or another measuring instrument for recording the graph. Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment. Thus an edge can be representing as E (V, W) where V and W are pair of vertices. The vertices V and W lie on E Vertices may be considered as cities and edges, arcs, line segment as roads.

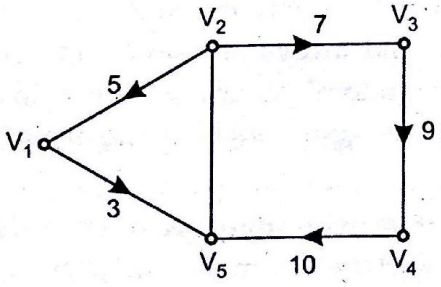
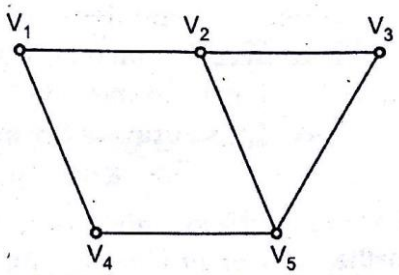



Figure 3. Graph (a)Undirected graph




(b) Directed and Weighted graph.

**Types of Graph** - The graphs are classified in the following categories: -

1. Simple Graph
2. Directed Graph
3. Non-directed Graph
4. Connected Graph
5. Non-connected Graph
6. Multi-Graph

## General Tree – Terminologies

There are many applications in real life situations that make use of non-linear data structures such as graphs and trees. But our main Concern here is tree. Trees are one of the **most important** data structures. Trees are **very flexible, versatile and**

**powerful** data structures that can be used to represent data items possessing **hierarchical relationship** between the grandfather and his descendants and in him their descendants and so on.

**TREE -** A tree is a **non-linear data structure** in which items are arranged in a sorted sequence. It is used to r**epresent hierarchical relationship** existing amongst several data items. The graph theoretic definition of tree is : it is a finite set of one or more **data items (nodes)** such that:

1. There is a **special data item** called the **root** of the tree.

2. And its remaining data items are partitioned into number of mutually exclusive disjoint subsets, each of which is itself a tree and they are **called sub-trees.**

3. Natural trees grow upwards from the ground into the air. But, tree data structure **grow; downwards** from top to bottom. It is an universally practiced convention for trees.

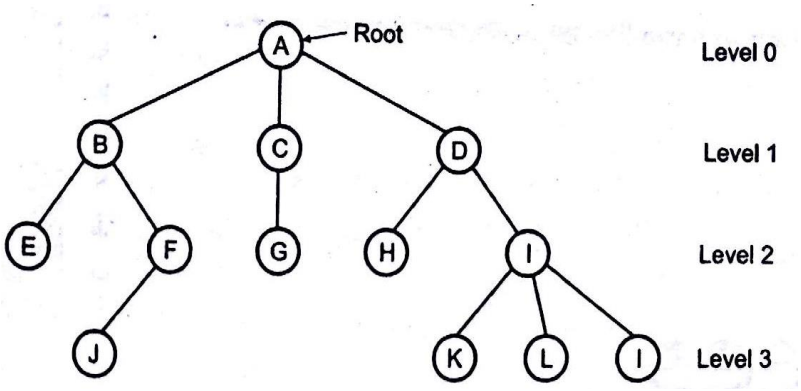



Figure 4. A tree

## Tree Terminology

There are number of terms associated with the. trees which are listed below:

1. **Root** - It is specially designed data item in a tree. It is the first in the hierarchical arrangement of data items. In the above tree **A** is the root item.

2. **Node** - Each data item in a tree is called a node. It is the basic structure in a tree. It specifies the data information and links (branches) to other data items. There are 13 nodes in the above tree.

3. **Degree of a node -** It is the number of subtrees of a node in a given tree. In the above tree. The degree of node A is 3. The degree of node C is 1. The degree of node B is 2. The degree of node H is 0.

4. **Degree of a tree -** It is the maximum degree of nodes in a given tree. In the above tree the node A has degree 3 and another node I is also having its degree 3 In all this value is the maximum. So, the degree of the above tree is 3.

5. **Terminal node (S) -** A node with degree zero is called a terminal node or a leaf. In the above tree, there are 7 terminal nodes. They are E, I, G, H, K, L and M.

6. **Non-terminal node (S) -** Any node (except the root node) whose degree is not zero is called non-terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal nodes (leaves). There are 5 non-terminal nodes.

7. **Siblings** - The children nodes of a given parent node are called siblings. They are also called brothers. In the above tree, E and F are siblings of parent node B and W K, L and M are siblings of parent node I.

8. **Level** - The entire tree structure is levelled in such a way that the root node is always at level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on upto the terminal nodes. In general, if a node is at level n, then its children will be at level n +1. In the 4 levels.

9. **Edge** - It is a connecting line of two nodes. That is, the line drawn from one node to another node is called an edge.

10. **Path** - It is a sequence of consecutive edges from the source node to the destination node. In the above tree, the path between A and I is given by the node pairs, (A, B), (B, F) and (F, J)

11. **Depth** - It is the maximum level of any node in a given tree. In the above tree, the root node A has the maximum level. That is the number of levels one can descend the tree from its root to the terminal nodes (leaves). The term height is also used to denote the depth.

12. **Forest -** It is a set of disjoint trees. In a given tree, it you remove its root node then it becomes a forest. In the above tree, there is forest with three trees.

## Binary tree - definitions and properties, Representation

A binary tree is a finite set of data items which is either empty or consists of a single item **called the root** and two **disjoint binary trees** called the **left subtree** and **right subtree**. A binary tree is a **very important** and the **most commonly used** non-linear data structure. In a binary tree the maximum degree of any node is at most two. That means, there may be a zero-degree node (usually an empty tree) or a one-degree node and two-degree node. Fig. 5 shows a binary tree consisting of 10 elements.

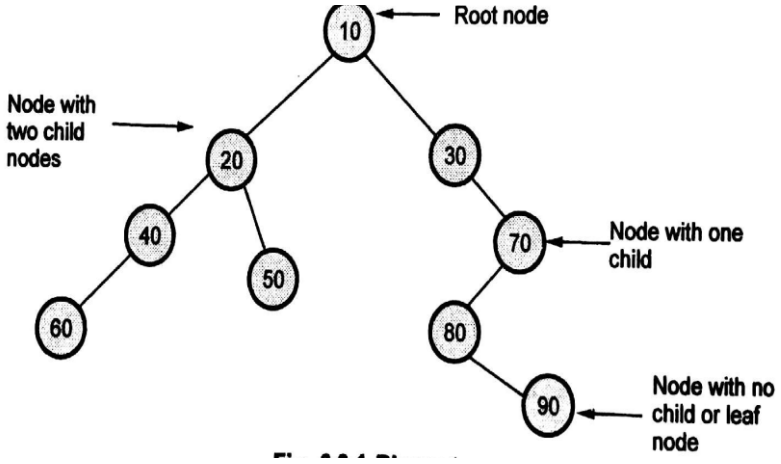


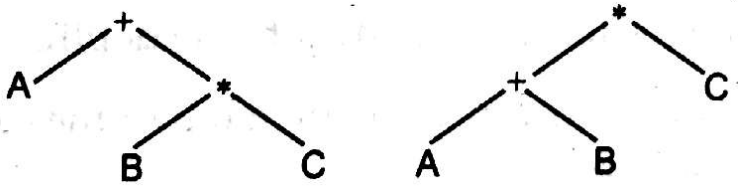

Figure 5 binary tree

In the given binary tree, 10 is the root of the tree. The left subtree consists of the tree with root 20. And the right subtree consists of the tree with root 30. Further 20 has its left subtree with root 40 and right subtree with root 50. Similarly, 30 has its right subtree with root 70 and its left subtree with root 80 In the next level, 90 has an empty left subtree and right subtree. In left side 50 has neither its left subtree nor right subtree. 80 as its root is 40 has an empty left and right subtree.

### Types of Binary Trees

1. **Strictly Binary Trees -** If every non-terminal node in a binary tree consists of **non-empty left subtree and right subtree**, then such a tree is **called strictly binary tree.** In the below binary tree all the non-terminal nodes such as B and E are having non-empty left and right subtrees. Consider the following method of representing an expression containing operands and binary operators by a strictly binary operator by

a strictly binary tree contains an operator that is to be applied to the result of evaluating the expression represented by the left and right subtrees, a node representing an operator is a nonleaf, whereas a node representing an operand is a leaf Fig. 6 & 7 illustrates some expressions and their tree representations.




A+(B*C) (A+B)*C

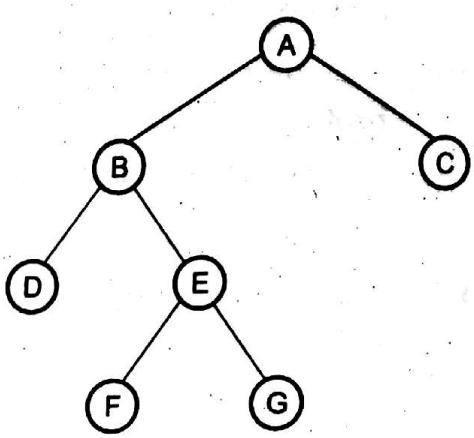
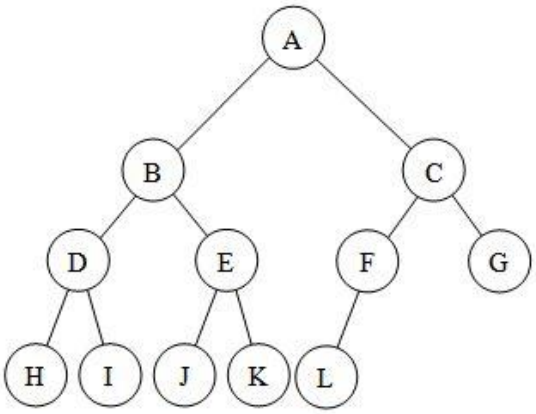



Figure 6. strictly binary tree

Figure 7. Expressions and their binary representation.

**2. Complete Binary Tree / Full & complete Binary Tree -** A complete binary tree is full binary tree in which **all leaves are at the same depth except at the last level at last nodes.** The total number of nodes in complete binary tree are $2^{h+1}$-1 where h is a height of the tree. In above given tree height of tree h is 3. Hence there are **$2^{3+1}$-1= $2^4$-1= 15(maxi. node)** nodes in the tree.



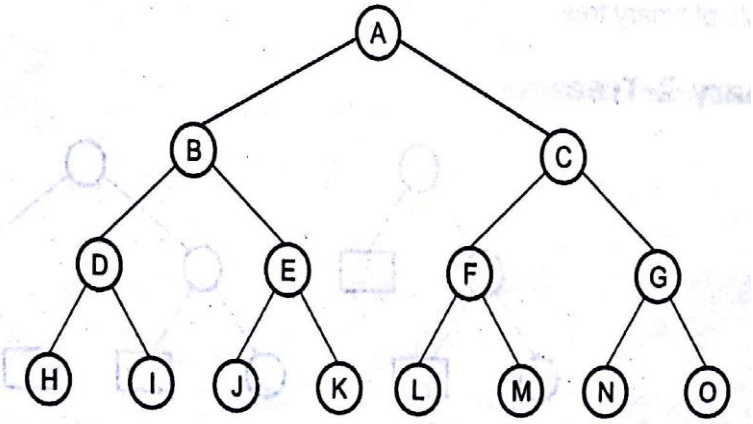




Figure 8 a) Complete binary tree b) Full and complete binary tree

A binary tree with n nodes and of depth d is a strictly binary tree all of whose terminal nodes are at level d. In a complex binary tree, there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. If there are m nodes at level 1 then a binary tree contains at most 2m nodes at level 1+1. A binary tree has exactly one node at the root level and has at most $2^1$ nodes at level l. Taking into consideration of this property, we can show further that a complete binary tree of depth d contains exactly $2^1$ nodes at each level. The value 1 ranges from 0 to d.

**3. Extended Binary Tree –** A tree T is said to be a 2 tree or/and extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with **2 children** are called **internal nodes**, and the nodes **1** with **0 children** are called **external nodes.** Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.

The term "extended binary tree" comes from the following operation. Consider any binary tree T, such as the tree in Fig. 9. Then T may be "converted" into a 2-tree by replacing each empty subtree by a new node, as pictured in Fig. 16. Observe that the

new tree is indeed a 2-tree. New nodes are the external nodes in the external nodes in the extended tree.

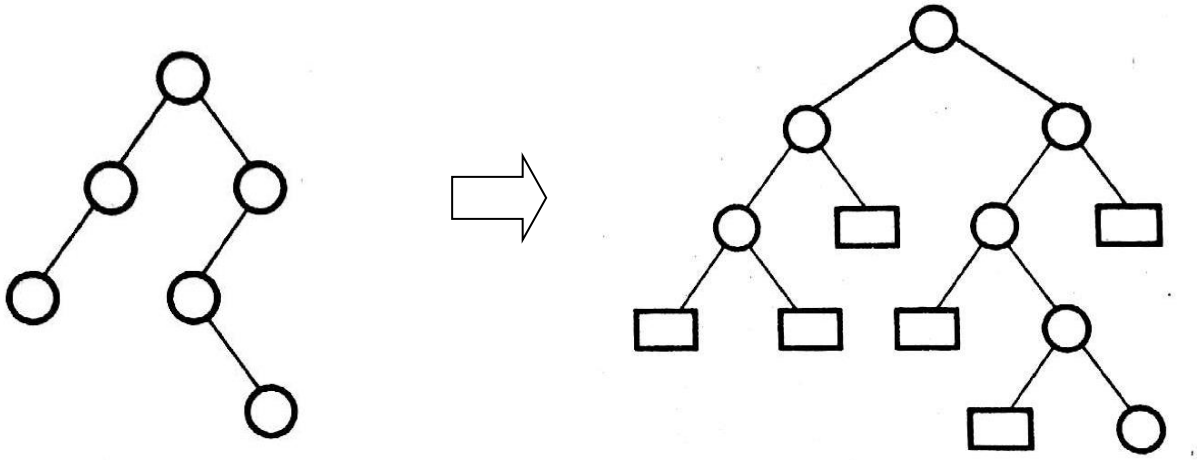


Figure 9 (a) Binary tree T (b) Extended 2 tree

4. **Full Binary tree -** A full binary tree is a tree in which **every node has zero or two children**. Every **full binary tree cannot be complete binary tree** and similarly every complete binary tree cannot be full binary tree.

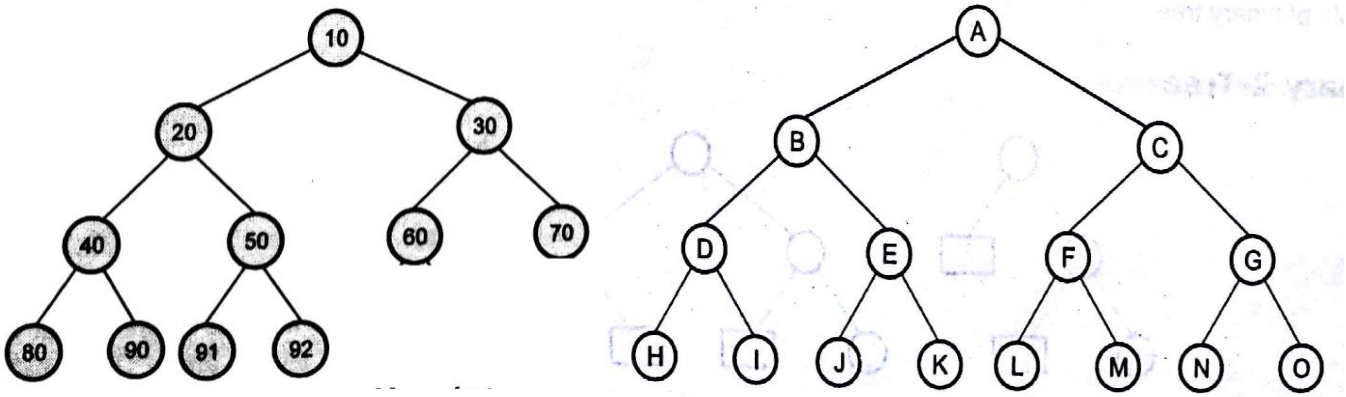



Figure 11. a) Full Binary tree b) Full and complete binary tree

5. **Skewed binary tree -** Skewed binary tree is the tree **having only one child either left or right** of the tree and tree continue grows in same fashion. It is of **two types: Left skewed** binary tree and **Right skewed** binary tree. In a skewed binary tree, only the left subtree is present, this type of binary tree is called left skewed binary tree. You can also have right skewed binary tree. In a skewed binary tree, only the right subtree is present, this type of binary tree is called right skewed binary tree. The array representation of the above tree is given below:

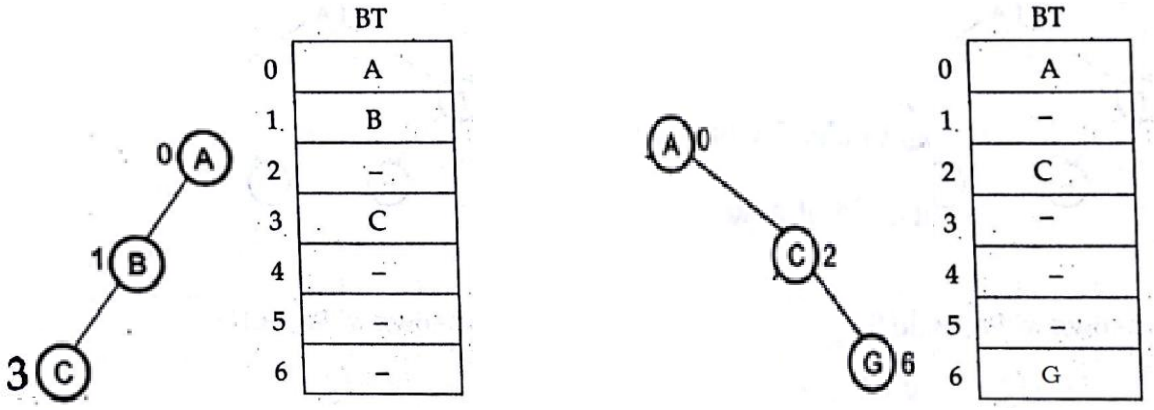
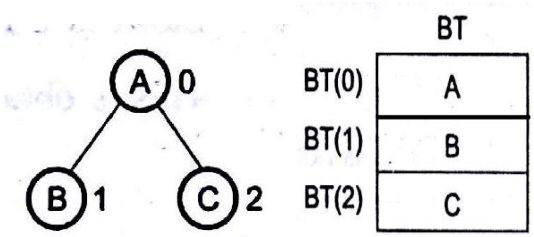



Figure 10 Left & Right Skewed Binary Tree with array representation

Note that the right child of A, is empty, and it's both left child and right child are also empty whose index is 4. Therefore, these indexes in array BT are left unused. This results in wastage of more memory.

## Generalised tree representation - Array Representation of Binary Trees

An array can be used to store the nodes of a binary tree. The nodes stored m an array is accessible sequentially. In C, arrays start with index 0 to (MAXSIZE - 1) Here, numbering of binary tree nodes starts from 0 rather then the maximum number of nodes is specified by MAXSIZE. The **root node** is always at **index 0**. Then, in successive memory locations the left child and right child are stored. Consider a binary tree with only three nodes as shown. Let **BT** denote a binary tree. The array representation of this binary tree is as follows:




Here, A is the father of B and C. B is the left child of A and C is the right child of A. Let us extend the above tree by one more level as shown below and the array representation of this binary tree is as follows:

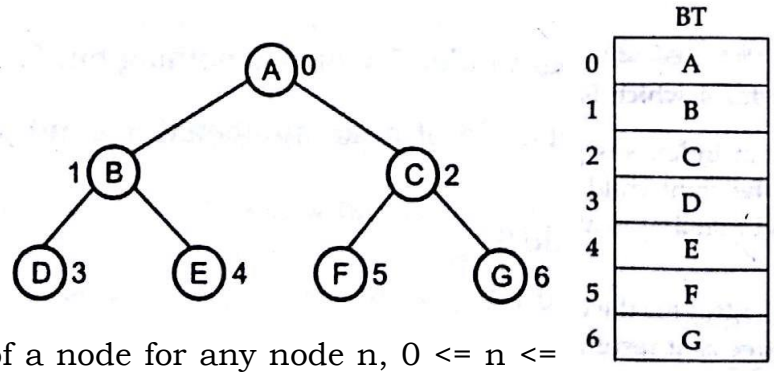



Figure 12 Binary tree and array representation

How to identify the father, the left child and the right child of an arbitrary node in such representation? It is very simple to identify the father and the children of a node for any node n, 0 <= n <= (MAXSIZE - 1), then we have **father (n) -** The father of node having index n is at floor **((n - 1)/ 2)** if n is not equal to 0. If n = 0, then it is the root node and has no father. Example. Consider a node numbered 3 (i.e. D). The father of D, no doubt, is B whose index is 1. This is obtained from **floor ((3 -1)/2) = (2/2) =1**

**lchild (n)** - The left child of node numbered n is at **(2n + 1)**. For example, in the above binary tree in **(1) lchild (A) = lchild (0) = 2 x 0 +1= 1** i.e. the node with index 1 which is nothing but B. **(2) lchild (C) = lchild (2) = 2 x 2 +1 = 5** i.e. the node with index 5 which is nothing but F.

**rchild (n)** - The right child of node numbered n is index **(2n + 2)**. For example, in the above binary tree **(1) rchild (A) = rchild (0) = 2 x 0 + 1 = 2** i.e. the node with index 2 which is nothing but C. **(2) rchild (B) =, rchild (1) = 2 x 1 + 2 = 4** i.e. the node with index 4 which is nothing but E.

**Siblings** - If the left child at index n is given then its right sibling (or brother) is at (n+1), similarly, if the right child at index n is given, then its left sibling is at (n-1). For example, the right sibling is at (n- 1) of node indexed 4 is at index 5 in an array representation.

### Linked List Representation of Binary Tree

Binary trees can be represented either using array representation or using a linked list representation. The basic component to be represented in a binary tree is a node. The node consists of three fields such as:

1. Data
2. Left child»
3. Right child

The data field holds the value to be given. The left child is a link field which contains the address of its left node and the right child contains the address of the right node. The logical representation of the node in C is given below

```
struct node
{
    char ch ;
    struct node *LC;
struct node *RC;
};
typedef stuct node n1;
```

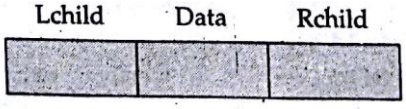



Figure 13. Structure of a node.

Consider the following binary tree and, its linked list representation is shown in Fig. In the binary tree all the data items are of type char.

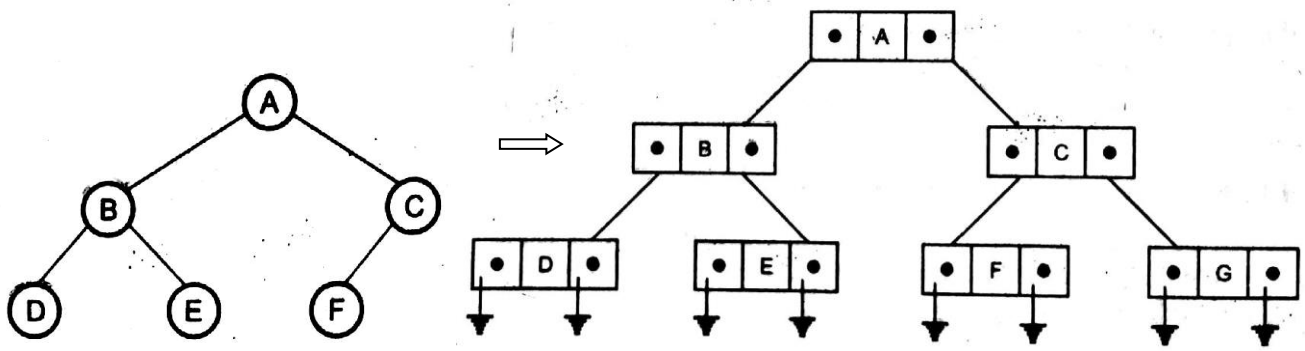



Figure 14. A binary tree in linked representation.

In some applications, it is necessary to include the father (or parent) field In such situation, one mare field to represent the father of a node is inserted into the structure definition of a binary tree node. A binary tree contains one root node and some non-terminal and terminal nodes (leaves). It is clear from the observation of a binary tree that the non-terminal nodes have their left child and right child nodes. But, the terminal nodes have no left child and right child nodes. Their lchild and rchild pointers are set to NULL. Here, the **non-terminal nodes are called internal nodes** and **terminal nodes are called external nodes**.

**OPERATIONS ON BINARY TREES -** The basic Operations to be performed on a binary are listed below.

1. **Create -** It creates an empty binary tree.
2. **Make BT -** It creates a new binary tree having a single node with data field set to some value.
3. **EmptyBT -** It returns true if the binary tree is empty otherwise it returns false.
4. **Lchild -** It returns a pointer to the left child of the node. If the node has no left child it returns the null pointer.
5. **Rchild -** It returns a pointer to the right child of the node. If the node has no right and it returns the null pointer.
6. **Father -** It returns a pointer to the father of the node. Otherwise returns the null pointer

7. **Brother (Siblings) -** It returns a pointer to the brother of the node. Otherwise returns the null

Apart from these primitive operations, other operations that can be applied to the binary tree are:

1. Tree traversal
2. Deletion of nodes
3. Copying the binary tree.
4. Insertion of nodes
5. Searching for the node

## Expression Tree

The expression tree is a tree **used to represent** the various **expressions**. The tree data structure is used to represent the **expressional statements.** In this tree, the **internal node** always denotes the **operators.**

The **leaf nodes always** denote the **operands** and the operations are always performed on these operands. The operator present in the **depth of the tree** is always at the **highest priority** and operator, which is **not much at the depth** in the tree, is always at the **lowest priority** compared to the operators lying at the depth. The operand will always present at a depth of the tree; hence it is considered the highest priority among all the operators.

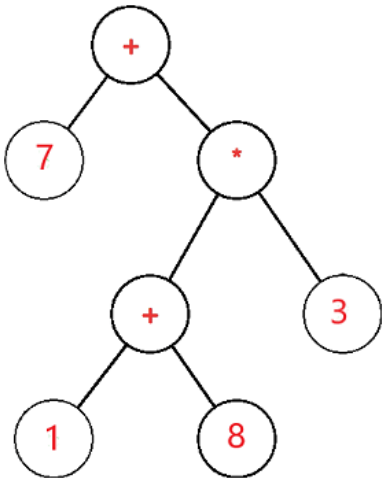
In short, we can summarize it as the value present at a depth of the tree is at the highest priority compared with the other operators present at the top of the tree. The **main use** of these expression trees is that it is used to **evaluate, analyze** and **modify** the various expressions. It is also used to **find out the associativity** of each operator in the expression. **For example,** the + operator is the left-associative and / is the right-associative. The dilemma of this associativity has been cleared by using the expression trees.

- These expression trees are **formed** by using a **context-free grammar.**
- We have **associated a rule** in context-free grammars in front of each grammar production.
- These **rules are also known as semantic rules**, and by using these semantic rules, we can be easily able to construct the expression trees.

Semantic rules is one of the **major parts** of compiler design and belongs to the **semantic analysis phase**. In this semantic analysis, we will use the **syntax-directed translations**, and in the form of output, we will **produce the annotated parse tree**. An annotated parse tree is nothing but the **simple parse tree** associated with the type **attribute and each production rule**. The main objective of using the expression trees is to make **complex expressions** and can be **easily be evaluated** using these expression trees. It is **immutable**, and once we have created an expression tree, we can not change it or modify it further. To make more modifications, it is required to construct the new expression tree wholly. It is also **used to solve the postfix, prefix,** and **infix expression** evaluation.

Expression trees play a very important role in representing the **language-level** code in the form of the data, which is mainly stored in the tree-like structure. It is also used in the memory representation of the **lambda** expression. Using the tree data structure, we can express the lambda expression more transparently and explicitly. It is first

created to convert the code segment onto the data segment so that the expression can easily be evaluated. The expression tree is a binary tree in which each external or leaf node corresponds to the operand and each internal or parent node corresponds to the operators so for example expression tree for 7 + ((1+8)*3) would be:




**Let S be the expression tree** Figure 15

If S is not null, then if S.value is an operand, then
Return S.value
x = solve(S.left)
y = solve(S.right)
Return calculate(x, y, S.value)

Here in the above example, the expression tree used context-free grammar.

**Use of Expression tree**

1. The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees.
2. It is also used to find out the associativity of each operator in the expression.
3. It is also used to solve the postfix, prefix, and infix expression evaluation.

**Implementation of an Expression tree**

To **implement** the expression tree and write its program, we will be required to use a **stack data structure**. As we know that the stack is **based on the last in first out LIFO** principle, the data element pushed recently into the stack has been popped out whenever required. For its implementation, the main **two operations** of the stack, **push and pop**, are used. Using the push operation, we will push the data element into the stack, and by using the pop operation, we will remove the data element from the stack.

**Main functions of the stack in the expression tree implementation:**

First of all, we will do scanning of the given expression into left to the right manner, then one by one check the identified character,

1. If a scanned character is an operand, we will apply the push operation and push it into the stack.
2. If a scanned character is an operator, we will apply the pop operation into it to remove the two values from the stack to make them its child, and after then we will push back the current parent node into the stack.
3. In the end, the only element of the stack will be the root of an expression tree.

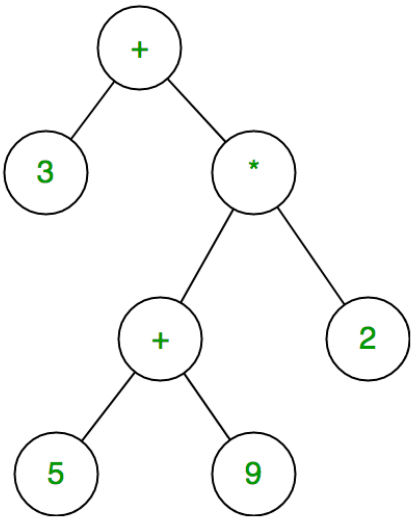



Figure 16

Example: Expression tree for 3 + ((5+9)*2) would be:
Input: A B C*+ D/
Output: A + B * C / D

# Binary Tree Traversals

Tree traversal is one of the **most common operations** performed on tree data structures. It is a way in which **each node** in the tree is **visited exactly once** in a systematic manner. There are many applications that essentially require traversal of binary trees. The full binary tree traversal would **produce a linear order** for the nodes ins a binary tree, there are three popular ways of binary tree traversal. They are:

**1. Preorder traversal  2. Inorder traversal  3. Postorder traversal**

The three ways are defined recursively. This helps in simply **visiting the root node** and **traversing the left and right subtree** of a binary tree. Note that we have nothing to do with empty binary tree. So, we consider non-empty binary tree for the purpose of traversal.

**1. Preorder Traversal -** The preorder traversal of a non-empty binary tree is defined as :

i. Visit the root node RN
ii. Traverse the left subtree in inorder L
iii. Traverse the right subtree in inorder R

That is in a preorder traversal the root node is visited (or processed) before traversing its left and right subtrees. The preorder notion is recursive in nature, so even within the left subtree and right subtree the above three steps are followed. The preorder traversal of binary tree shown in Fig.

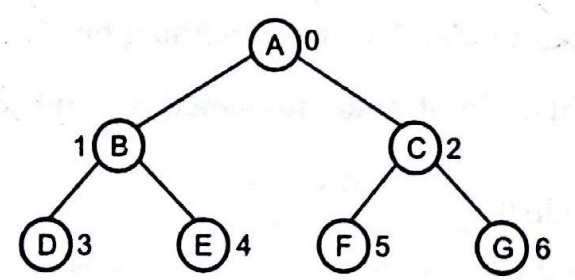


Figure 17 Preorder traverse of tree

Preorder is: **A  B  D  E  C  F  G**
Root  Left  Right

**2. Inorder Traversal -** The inorder traversal of a non-empty binary tree is defined as follows:

i. Traverse the left subtree in inorder L
ii. Visit the root node RN
iii. Traverse the right subtree in inorder R

That is, in an inorder traverse, the left subtree is traversed recursively in inorder before visiting the root node. After visiting the root node, the right subtree is taken up and it is traversed recursively again in inorder. The inorder traversal of the binary tree shown in Fig.

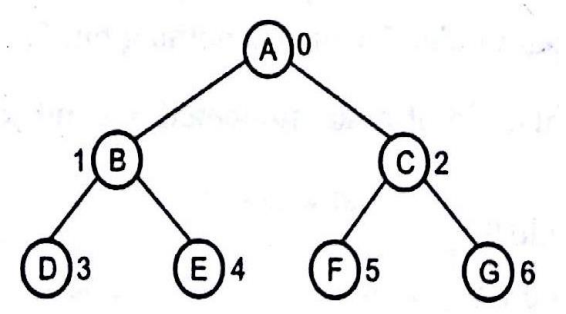


Figure 18. Inorder traverse of tree

Inorder is: **D  B  E  A  F  C  G**
Left  Root  Right

**3. Postorder Traversal -** The postorder traversal of non-empty binary tree is defined as follow:

i. Traverse the left subtree in inorder L
ii. Traverse the right subtree in inorder R
iii. Visit the root node RN

That is, in a postorder traversal the left and right subtree are recursively processed before visiting the root. The left subtree is taken up first and is traversed in postorder. Then the right subtree is taken up and is traversed in postorder. Finally, the data of the root node is displayed. The postorder traversal of the binary tree shown in Fig.

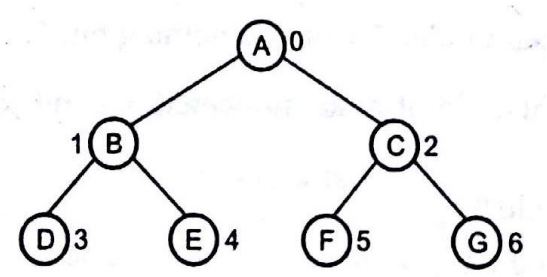



Figure 19. Preorder traverse of tree

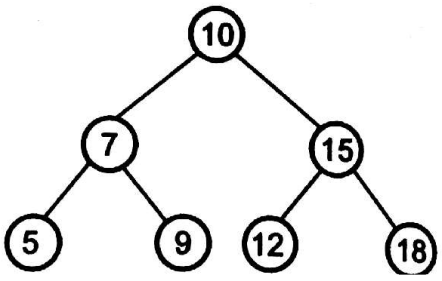
Preorder is: **D E B** (Left) **F G C** (Right) **A** (Root)

## Binary Search Tree

In the simple binary tree the **nodes are arranged in any fashion**. Depending on user's desire the new nodes can be **attached as a left or right child** of any desired node. In such a case **finding for any node** is a **long procedure,** because in that case we have to search the entire tree. And thus the **searching time complexity** will get **increased** unnecessarily. So to make the **searching algorithm faster** in a binary tree we will go for building the **binary search tree**. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at **left subtree < root node value < Right subtree values**.

If you observe the Fig. carefully, you will find that the left value < parent value < right value is followed throughout the tree. Various operations that can be performed on binary search tree are:

Figure 20 Binary search tree




1. **Insertion** of a node in a binary tree
2. **Deletion** of element from the binary search tree.
3. **Searching** of an element in the binary tree.
4. **Display** of binary tree.

**1. Insertion of a node in a binary tree**

**Algorithm:**

1. Read the value for the node which is to be created, and store it in a node called New.
2. Initially if (root==NULL) then root=New.
3. Again read the next value of node created in New.
4. If (New→value < root→value) then attach New node as a left Child of at otherwise attach New node as a right child of root.
5. Repeat step 3 and 4 for constructing required binary search tree completely-

**2. Deletion of an element from the binary tree**

For deletion of any node from binary search tree there are three cases which are possible.

i. Deletion of leaf node having zero child..
ii. Deletion of a node having one child.
iii. Deletion of a node having two children.

**Deletion of leaf node having zero child** - This is the simplest deletion, in which we set the left or right pointer of parent node as NULL. From the below tree, we want to delete the node having value 6 then we will set left pointer of its parent node as NULL. That is left pointer of node having values is set NULL.

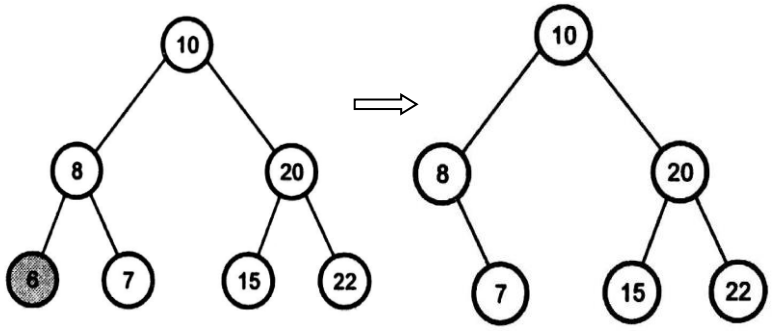



Figure 21 Before and after deletion of leaf node

**i. Deletion of a node having one child** - To explain this kind of deletion, consider a tree as given below.

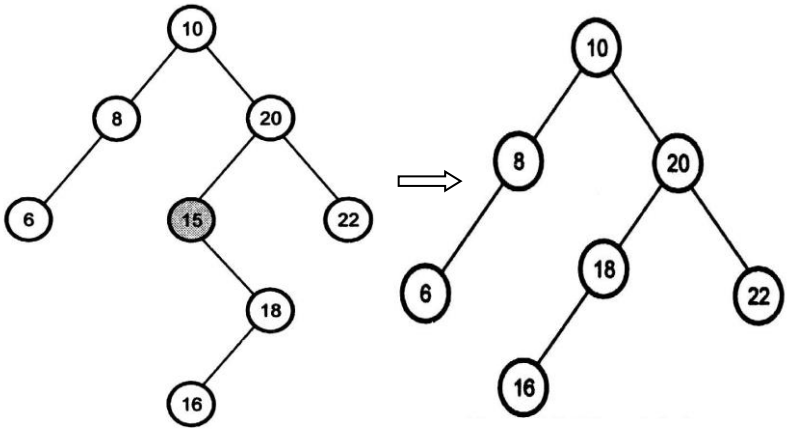



Figure 22. Before and after deletion of non leaf node

If we want to delete the node 15, then we will simply copy node (left or right node available at the time of deletion) here in our case 18 at place of 15 And then set the node free.

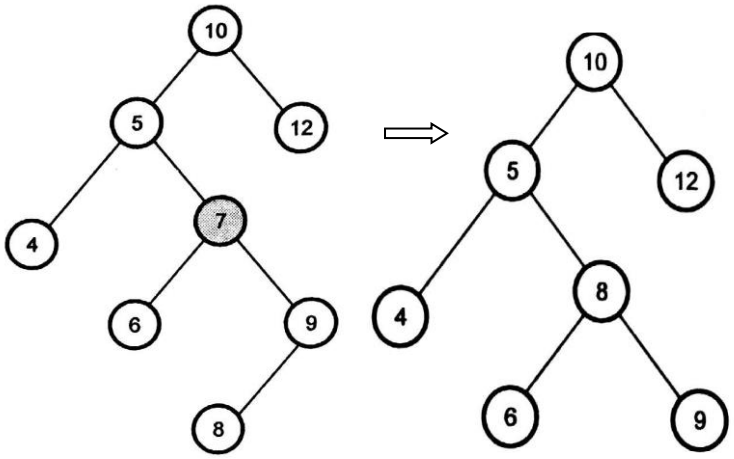



Figure 23. Before and after deletion of non leaf node

**ii. Deletion of a node having two children** - Again, let us take some example for discussing this kind of deletion. Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7. That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.

### 3. Searching a node from Binary search tree

In searching, the node which we want to search is **called a key node**. The key node will be **compared with each node** starting from root node if value of key node is greater than current node then we search for it on right sub-branch otherwise on left sub-branch. If we reach to leaf node and still we do not get the value of key node then we declare "**node is not present in the tree".**

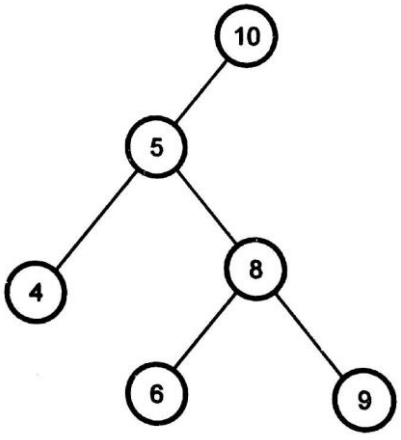



Figure 24. Binary search tree

In the above tree, if we want to search for value 9. Then we will compare 9 with root node10. As 9 is less than 10 we will search on left sub-branch. Now compare 9 with 5

but 9 is greater than 5. So we will move on right sub-branch. Now compare 9 with 8 but as 9 is greater than 8 we will move on right sub-branch as the node we will get holds the value 9. Thus the desired node can be searched.

## AVL Tree

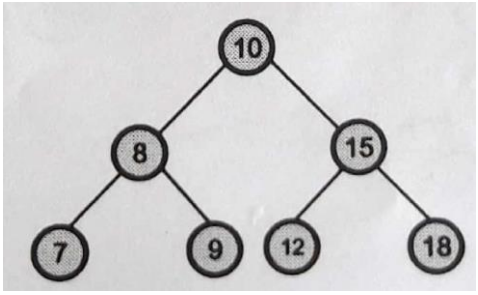



Figure 25. AVL tree

**Adelsion, Velski and Lendis in 1962 introduced** binary tree structure that is balanced with respect to height of sub tree. Then tree can be made balanced and because of this retrieval of any node can be done in O(log n) times, where **n** is total number of nodes. From the name of the scientist the tree is **called AVL tree**.

**Definition of AVL Tree –** An tree is height balanced if T is a non-empty binary tree with $T_L$ and $T_R$ as its left and right subtrees. The tree T is height balanced if and only if

i. $T_L$ and $T_R$ are height balanced.
ii. $H_L$ - $H_R$ <=1 where $H_L$ and $H_R$ are heights of $T_L$ and $T_R$.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

**Balance Factor –** The balance factor BF(T) of a node in binary tree is defined to be $H_L$ - $H_R$ when $H_L$ and $H_R$ are heights of left and right sub trees of T. For any node in AVL tree the balance factor i.e. **BF(T) is -1, 0 or +1**

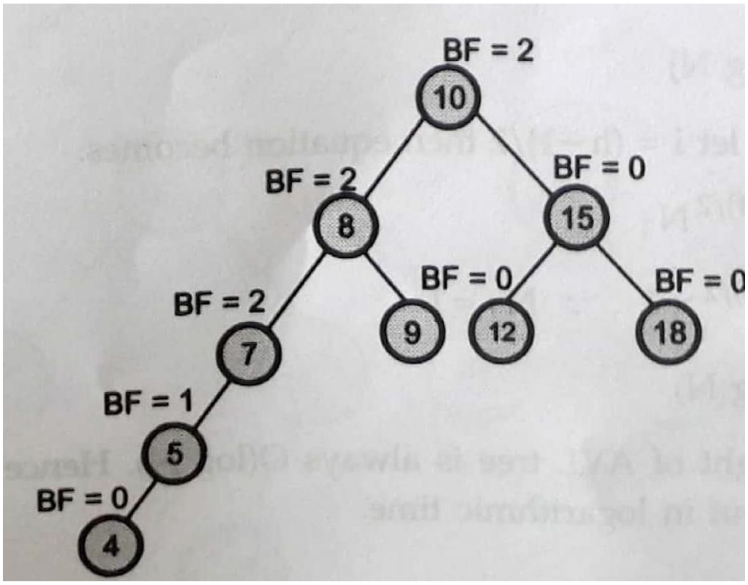



Figure 26 Not a AVL tree

**Representation of AVL Tree -** The AVL tree follows the property of binary search tree. Infact AVL trees are basically binary search trees with **balance factor as -1, 0 or +1**.

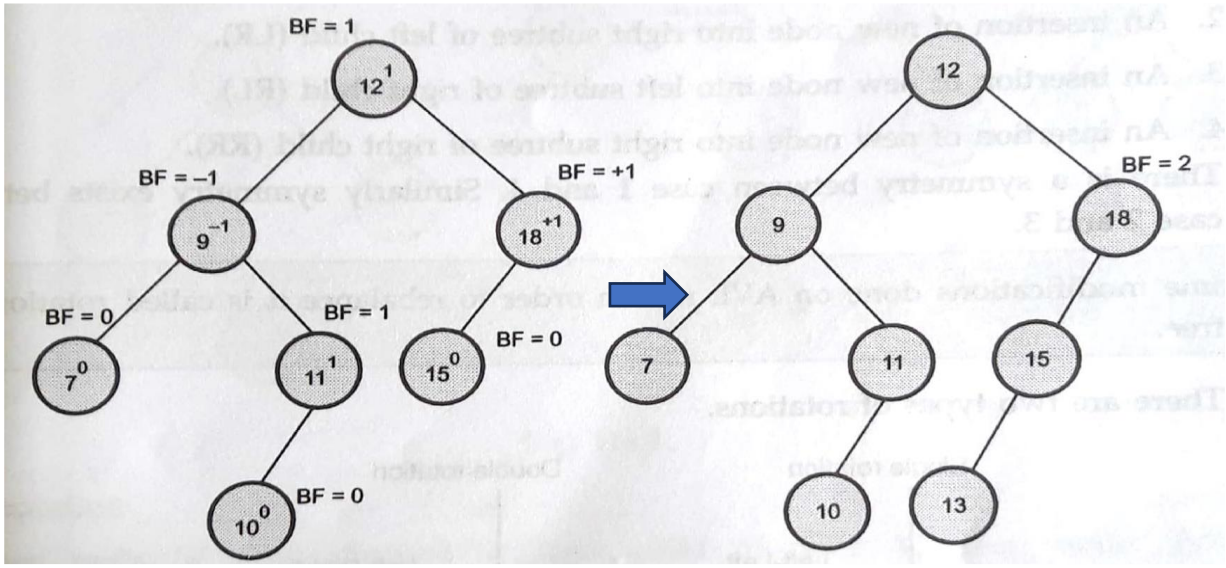



Figure 27 Original AVL tree

Tree after insertion of 13

**After insertion of any node** in an AVL tree if the **balance factor** of any node **becomes other than -1, 0 or +1** then it is said that **AVL property is violated**. Then we have to restore the destroyed balance condition. The balance fsactor is denoted at right top comer inside the node. **After insertion** of a new node **if balance condition gets destroyed** then the nodes on that path (new node insertion point to root) **needs to be readjusted**. That means only the affected subtree is to be rebalanced. The **rebalancing should** be such that **entire tree should satisfy AVL property**. For example shown in figure 27 and 28:

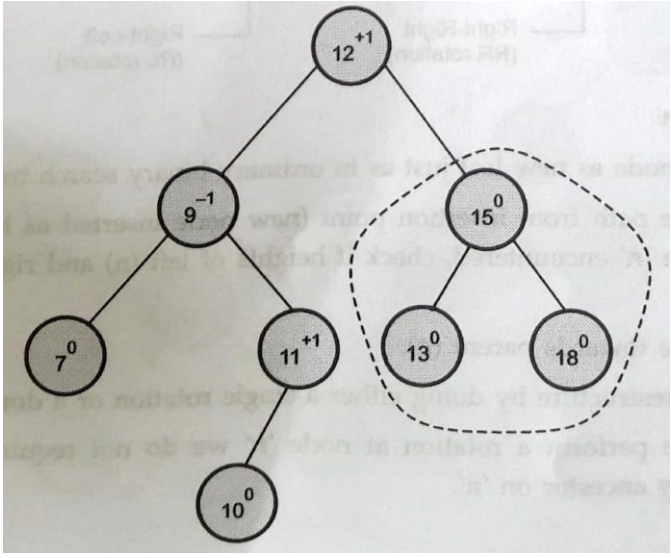



Figure 28 AVL tree after balancing

## Insertion in AVL tree

There are four different cases when rebalancing is required after insertion of node.

1. An insertion of new node into left subtree of left child (LL).
2. An insertion of new node into right subtree of left child (LR).
3. An insertion of new node into left subtree of right child (RL).
4. An insertion of new node into right subtree of right child (RR).

There is asymmetry between case1 and 4. Similarly symmetry exist between case2 and 3. There are two types of rotations.

1. **Single rotation**

   1.1. Left - Left (LL rotation)
   1.2. Right - Right (RR rotation)

2. **Double rotation**

   2.1. Left – Right (LR rotation)
   2.2. Right - Left (RL rotation)

**Algorithm of Insertion in AVL**

1. Insert a new node as new leaf just as in ordinary binary search tree.

2. Now trace the path from insertion point (new node inserted as leaf) towards root. For each node **n** encountered, check if heights of **left (n)** and **right (n)** differ by at most **1.**

   2.1. If yes, move towards parent **(n)**.
   2.2. Otherwise restructure by doing either a single rotation or a double rotation.

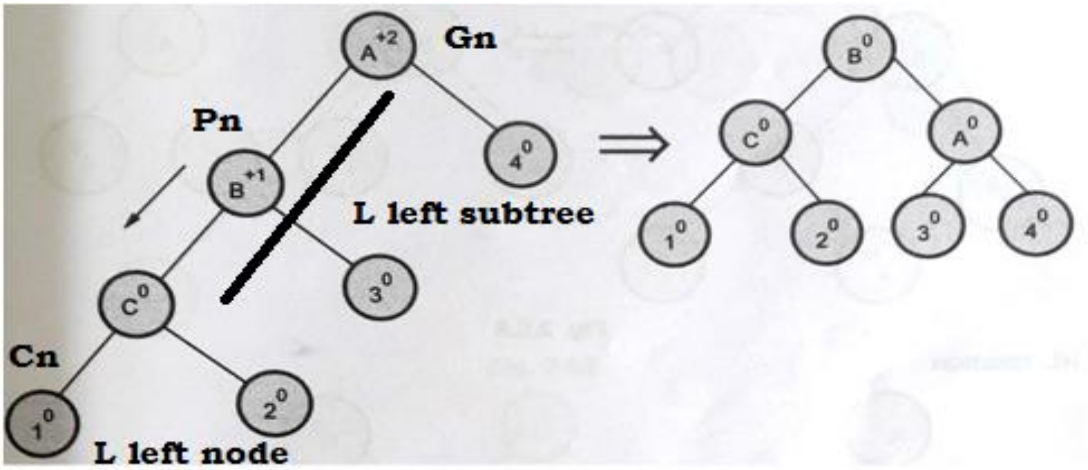
Thus once we perform a rotation at node **'n'** we do not require to perform an rotation at any ancestor on **'n'**.

**Different rotations In AVL tree**

1. **LL rotation -** When node 1 or /and 2 gets inserted as a left / right child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2. Tree becomes imbalance so we need rebalance then tree. For rebalance we need to apply rotation. As **node is added to left of left subtree** so this is LL rotation. The LL rotation has to be applied to rebalance the nodes.

In LL rotation **3 nodes** are important grant parent node **Gn**, parent node **Pn** and child node **Cn**. When rotation is applied at imbalance node (i.e. when new child node is added in existing tree and tree get imbalance, newly added not become grant child node **Cn**) immediate child of grant parent node **Gn** i.e **Pn** become root of the tree.

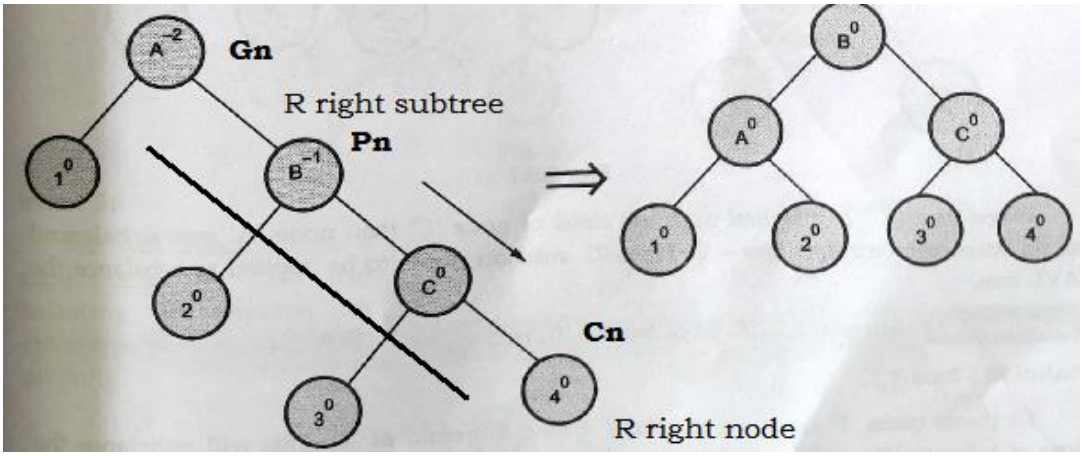
Figure 29




Node **A** balancing factor is +2, so LL rotation is performed. In LL rotation immediate child of imbalance node i.e. **B** become root node and tree is restructure by moving node **A** in Right subtree following the rule of **B**inary **S**earch **T**ree**.**

2. **RR rotation -** When node 3 or /and 4 gets inserted as a left / right child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2. Tree becomes imbalance so we need rebalance then tree. For rebalance we need to apply rotation. As **node is added to right of right subtree** so this is RR rotation. The RR rotation has to be applied to rebalance the nodes.

Figure 30



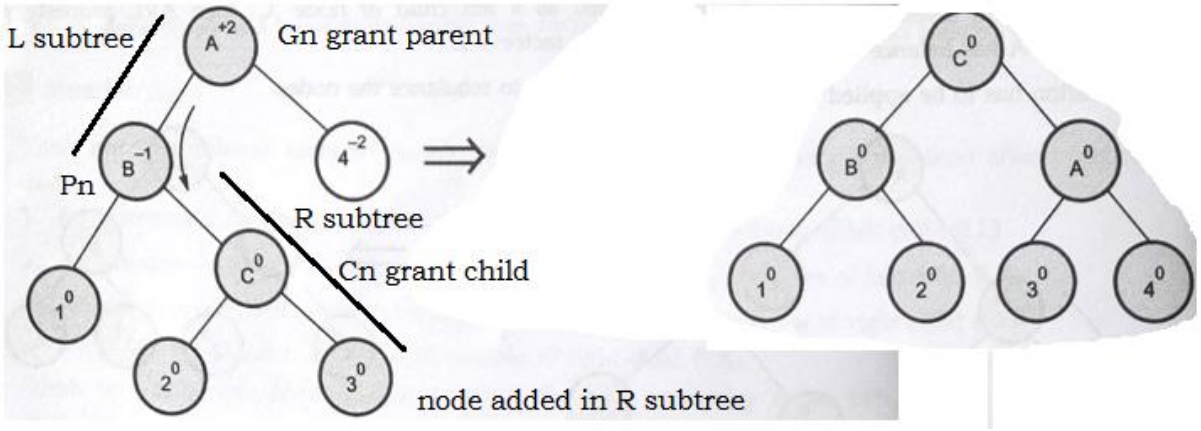

Similarly like LL rotation in RR rotation **3 nodes** are important grant parent node **Gn**, parent node **Pn** and child node **Cn**. When rotation is applied at imbalance node (i.e. when new child node is added in existing tree and tree get imbalance, newly added not become grant child node **Cn**) immediate child of grant parent node **Gn** i.e **Pn** become root of the tree.

Node **A** balancing factor is +2, so RR rotation is performed. In RR rotation immediate child of imbalance node i.e. **B** become root node and tree is restructure by moving node **A** in left subtree following the rule of **B**inary **S**earch **T**ree**.**

3. **LR rotation -** When node 3 gets inserted as a right child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2. Tree becomes imbalance so we need rebalance then tree. For rebalance we need to apply rotation. As **node is added to left subtree at right side** so this is LR rotation. The LR rotation has to be applied to rebalance the nodes.

In LR rotation **3 nodes** are important grant parent node **Gn**, parent node **Pn** and child node **Cn**. When rotation is applied at imbalance node (i.e. when new child node is added in existing tree and tree get imbalance, newly added not become grant child node **Cn**) immediate grant child of grant parent node **Gn** i.e **Cn** become root of the tree.

Figure 31




Node **A** balancing factor is +2, so LR rotation is performed. In LR rotation immediate grandchild of imbalance node i.e. **C** become root node and tree is restructure by moving node **A** in right subtree following the rule of **B**inary **S**earch **T**ree**.**

4. **RL rotation -** When node 2 gets inserted as a right child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2. Tree becomes imbalance so we need rebalance then tree. For rebalance we need to apply rotation. As **node is added to right subtree at left side** so this is RL rotation. The RL rotation has to be applied to rebalance the nodes.

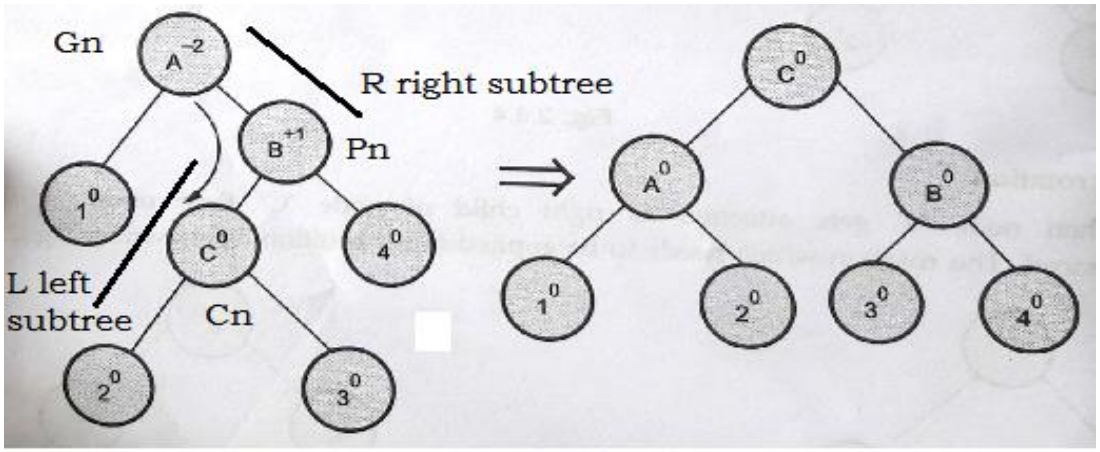



Figure 32

In RL rotation **3 nodes** are important grant parent node **Gn**, parent node **Pn** and child node **Cn**. When rotation is applied at imbalance node (i.e. when new child node is added in existing tree and tree get imbalance, newly added not become grant child node **Cn**) immediate grant child of grant parent node **Gn** i.e **Cn** become root of the tree.

Node **A** balancing factor is +2, so LR rotation is performed. In RL rotation immediate grandchild of imbalance node i.e. **C** become root node and tree is restructure by moving node **A** in right subtree following the rule of **B**inary **S**earch **T**ree**.**

## Deletion in AVL tree

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to applied.

**Algorithm of Deletion in AVL –**

The deletion algorithm is more complex than insertion algorithm.

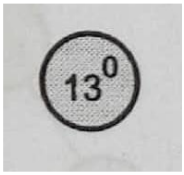
1. Search the node which is to be deleted.
2. **a)** If the node to be deleted is a leaf node then simply make it NULL to remove.
   **b)** If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some subtree then balance that subtree using appropriate single or double rotation.

The deletion algorithm takes **O (log n)** time to delete any node.

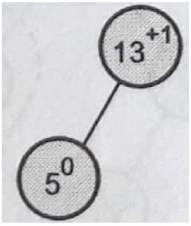
**Example** Construct the AVL tree for the following set of elements **13, 5, 1, 7, 8, 98, 67, 26, 33, 12, 6, 7, 8**

**Solution:**

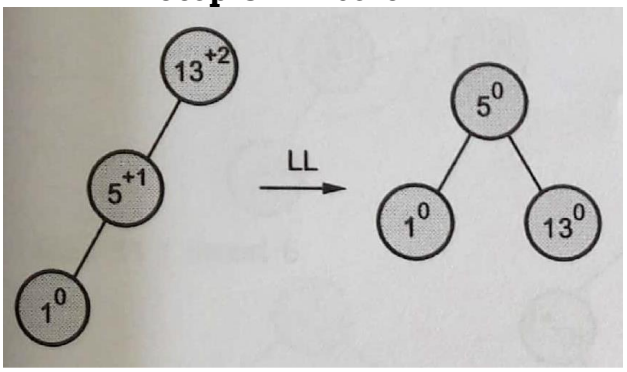
**Step 1 – Insert 1**



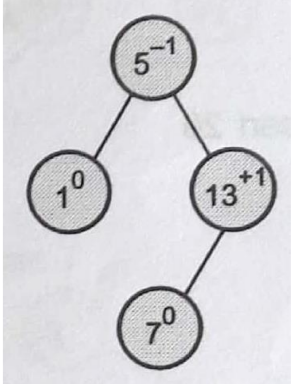

**Step 2 – Insert 5**



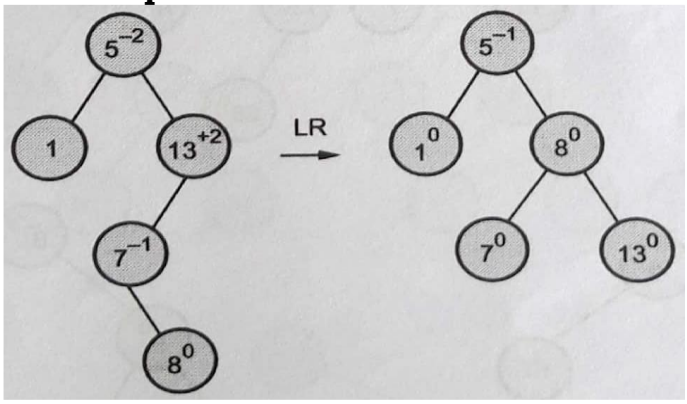

**Step 3 – Insert 1**



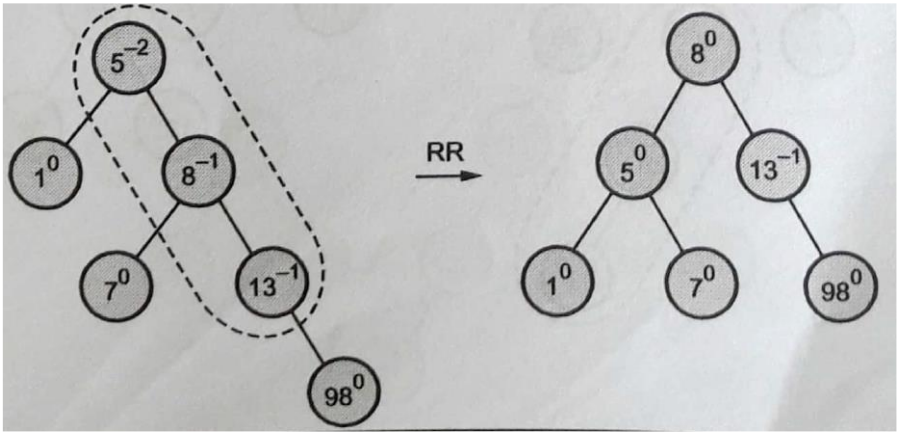

**Step 4 – Insert 7**



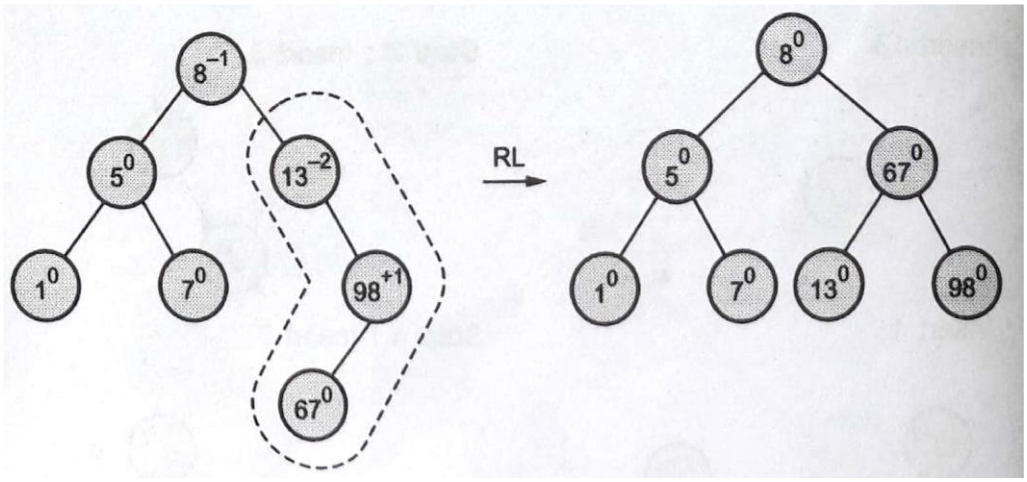

**Step 5 – Insert 8**



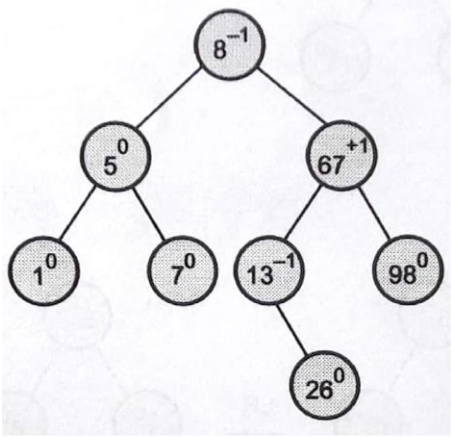
**Step 6 – Insert 98**



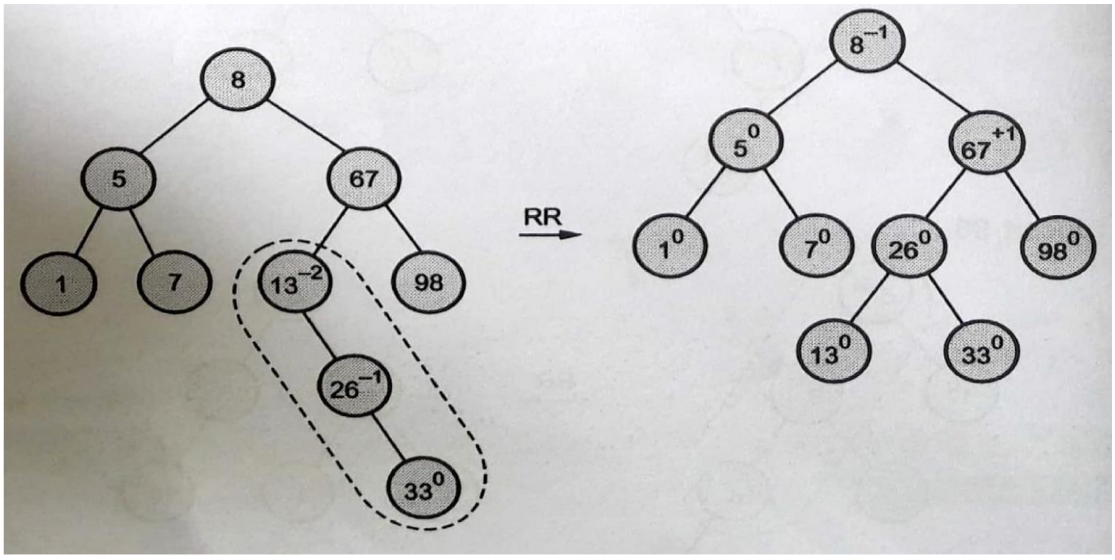

**Step 7 – Insert 67**



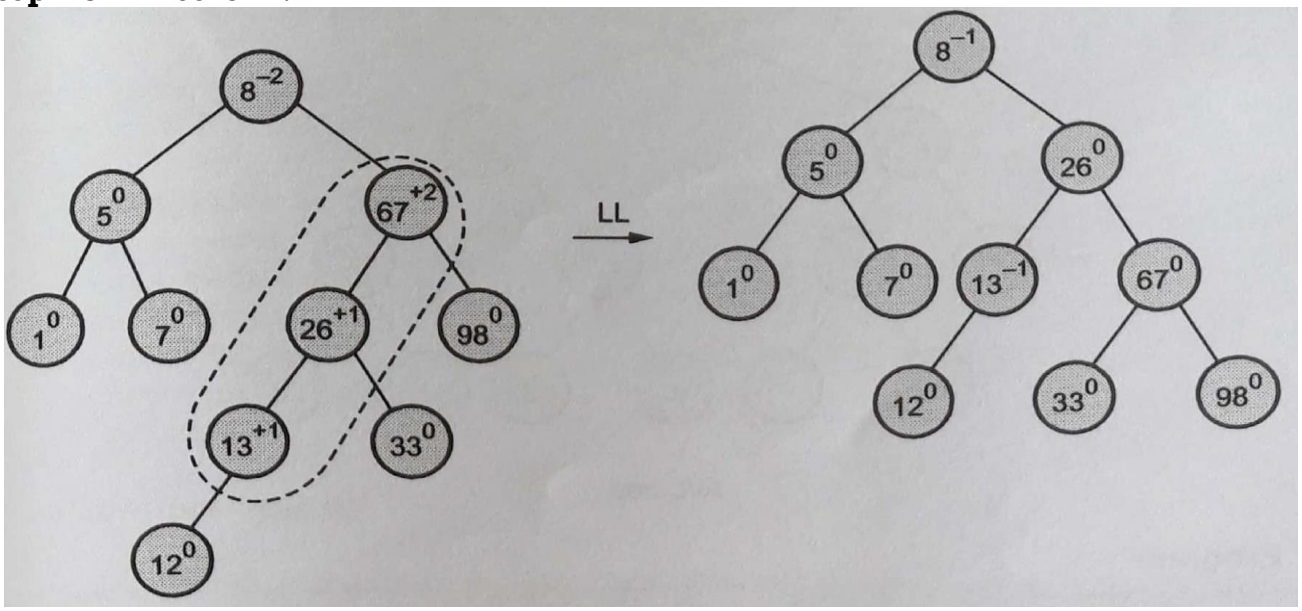

**Step 8 – Insert 26**



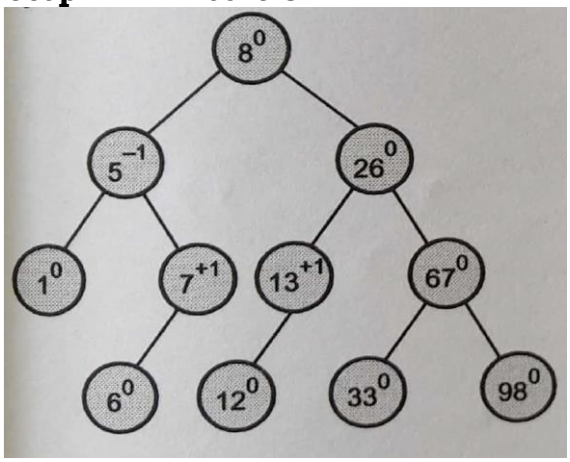

**Step 9 – Insert 33**



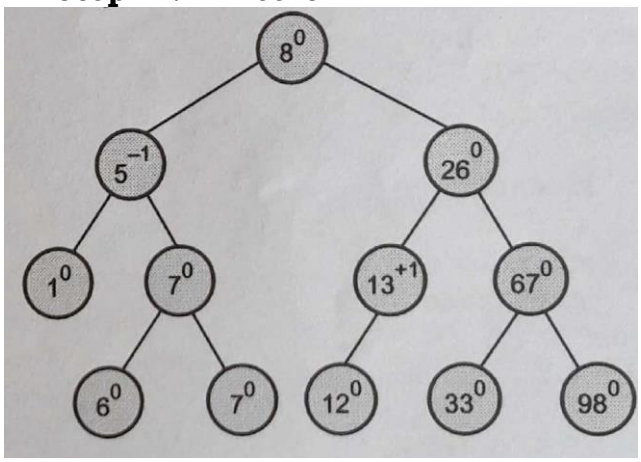
**Step 10 – Insert 12**



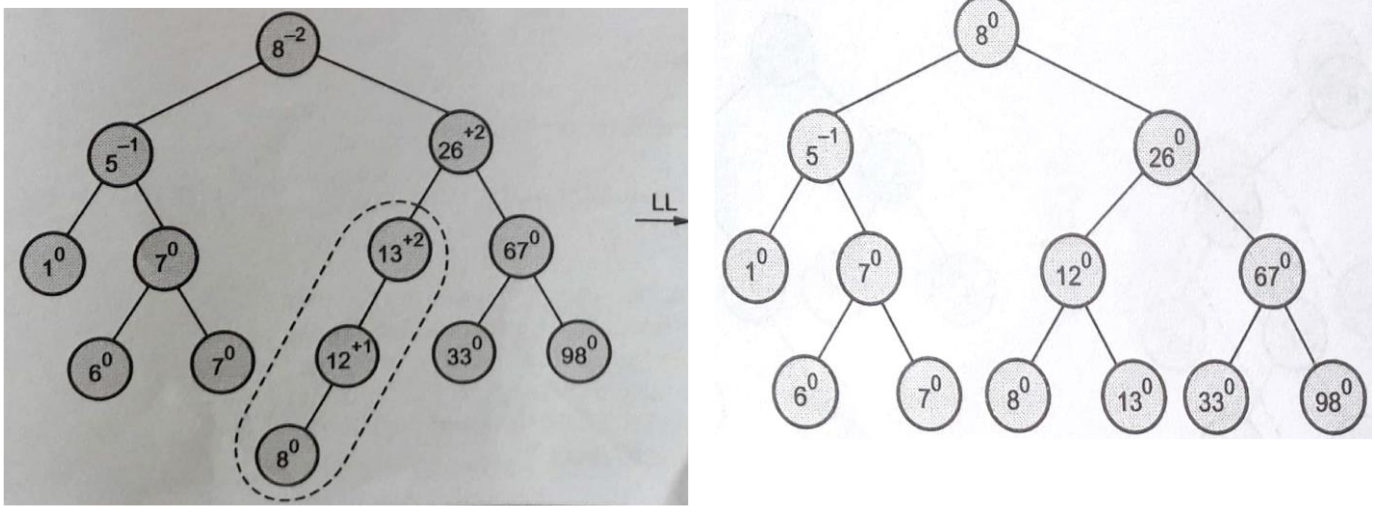

**Step 11 – Insert 6**



**Step 12 – Insert 7**



**Step 13 – Insert 8**




## Red black Tree

A **red-black tree is a binary search tree** with **one extra bit of storage** per node: **its colour**, which can be either **RED or BLACK**. By constraining the way nodes can be coloured on any path from the root to a leaf, red-black trees ensure that no such path

is more than twice as long as any other, so that the tree is approximately balanced. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

Each node of the tree now **contains the fields colour, key, left, right**. If a **child or the parent** of a node does not exist, the corresponding pointer fields of the node **contains the value NIL**. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

**Properties of Red Black trees**

A binary search tree is a red-black tree if it satisfies the following red-black properties:

1. Every node is either **red or black.**
2. **Depth property:** All the leaves have the same black depth.
3. **Root property:** The root is black.
4. **External property:** Every leaf (NIL) is black.
5. **Internal property:** If a node is red, then both its children are black.
6. **Path property:** For each node, all paths from the node to descendant leaves contain the same number of black nodes.

**Representation**

While representing a Red-Black tree colour of every node and pointer colours are shown. The leaf nodes are simply NULL nodes and not any physical node containing data. As an example a Red-Black tree is as shown below.

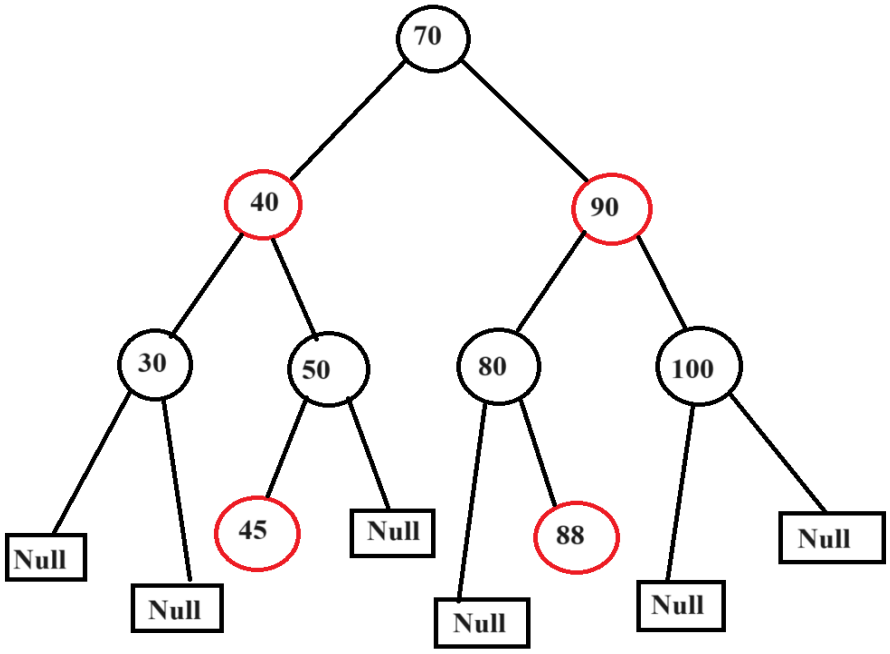



Fig. 33 Red-Black tree

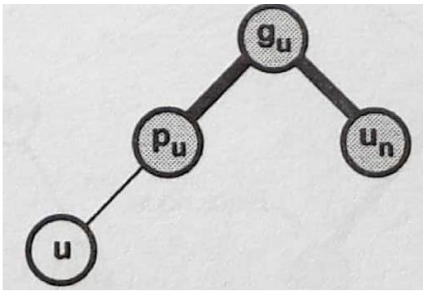
The Red-Black tree shown in above given figure has black nodes that are shaded in black and unshaded nodes are red nodes. Similarly the leaves are black and all are NULL pointers only. The pointers to black node are black pointers which are shown by thick lines remaining are red pointers. But in practice, we explicitly mention the colour of nodes.

**Insertion in Red Black tree**

1. Every **new node** which is to be inserted is **marked red.**
2. **Not every insertion** causes **imbalancing** but **if imbalancing occurs** then that can be removed depending upon the **configuration of tree** before new insertion made.
3. To understand insertion operation, let us understand the configuration of tree by defining following roles.

Let **U** is newly inserted node.

**$P_u$** is the parent node of **U**.
**$G_u$** is grandparent of **U** and parent node of .
**$U_n$** is an uncle node of **U** i.e. its a right child of **$G_u$**.




The tree is said to be imbalanced if properties of Red-Black tree are violated.

Fig. 34

When **insertion occurs**, the new node is inserted in already balanced tree lf this **insertion causes** any **unbalancing** then balancing of the tree is to be done at **two levels:**

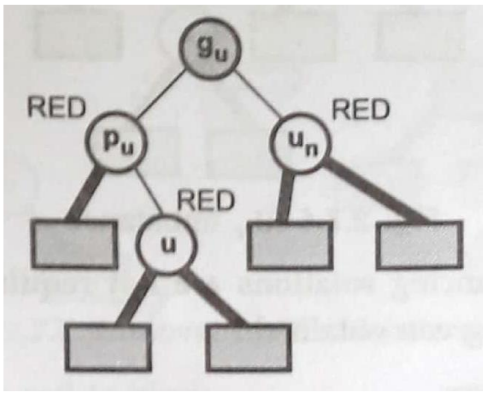
i) At grandparent level i.e. **$G_u$** ii) At parent level i.e. **$P_u$**.

The **unbalancing is concerned** with the colour of **grandparent's child (i.e uncle node)** is red. In these types of unbalancing **only colours of nodes were change not the position of node.**

**1. LRr** **2. LLr** **3. RRr** **4 LRr**

In red black tree when **red unbalancing occur**. To remove these unbalancing **rotations are not required**. Simply by **changing the colours required** balancing can obtained. Let us see how the **red unbalancing occurs** in tree and how we solve this **red unbalancing by** changing colour. Note it

1) While **inserting any node** its colour is by **default red**.
2) If **uncle node** is a **NULL node** then it is **always black**.




1. **LRr unbalancing:** The left child of **$g_u$** is **$p_u$** and **u** is inserted as right child of **$p_u$**. Colour of **$p_u$** and **$u_n$** node (uncle node) is red and newly inserted node **u** is also red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in left subtree** as a child in **right side**, so it is **LR** unbalancing. Now as its uncle node **$u_n$** is red so it is **LRr unbalancing** inserted node always take colour reference from uncle node **$u_n$**).

Fig. 35 LRr unbalancing

**Removal of LRr unbalancing -** Before colour change note that if **$g_u$** in given figures is root then there should not be any colour change of **$g_u$** (Because root is always black). But if **$g_u$** happens to be red that rebalancing can be done as-

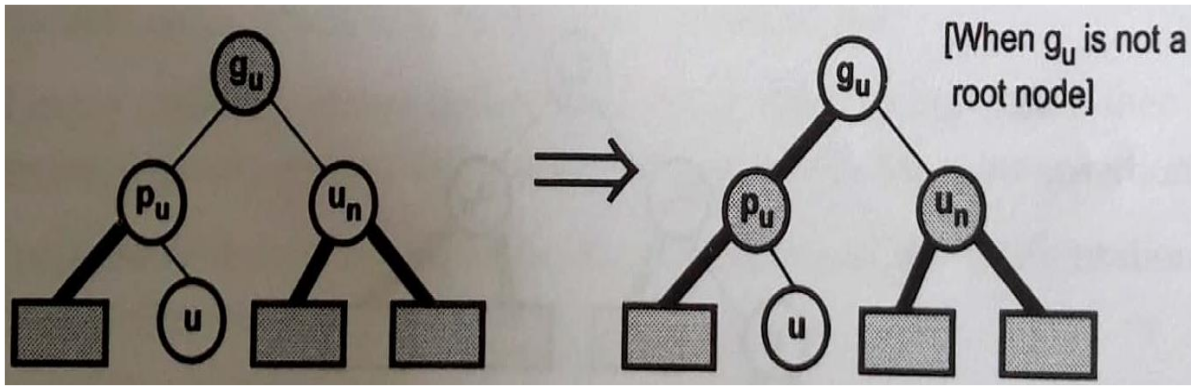



Fig. 45 Removal of LRr unbalancing

1. Change the colour of $\mathbf{p_u}$ from red to black.
2. Change the colour of $\mathbf{u_n}$ from red to black.
3. Change the colour of $\mathbf{g_u}$ from black to red provided gu is not a root node.

2. **LLr unbalancing:** The node $\mathbf{p_u}$ is a left child of $\mathbf{g_u}$ and **u** is inserted as left child of $\mathbf{p_u}$. Colour of $\mathbf{p_u}$ and $\mathbf{u_n}$ node (uncle node) is red and newly inserted node **u** is also red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in left subtree** as a child in **left side**, so it is **LL** unbalancing. Now as its uncle node $\mathbf{u_n}$ is red so it is **LLr unbalancing** (inserted node always take colour reference from uncle node $\mathbf{u_n}$).

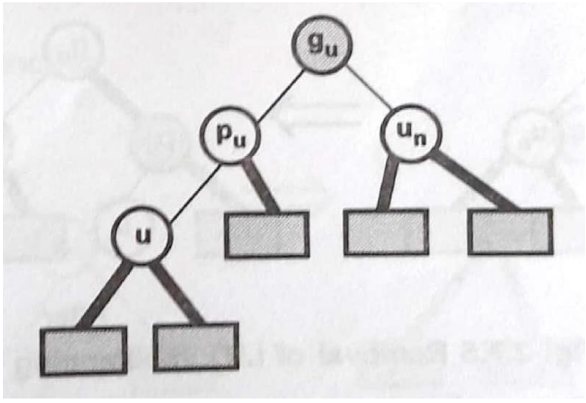



Fig. 36 LLr unbalancing

**Removal of LLr unbalancing -** Before colour change note that if $\mathbf{g_u}$ in given figures is root then there should not be any colour change of $\mathbf{g_u}$ (Because root is always black). But if $\mathbf{g_u}$ happens to be red that rebalancing can be done as-

1. Change the colour of $\mathbf{p_u}$ from red to black.
2. Change the colour of $\mathbf{u_n}$ from red to black.
3. Change the colour of $\mathbf{g_u}$ from black to red provided gu is not a root node.

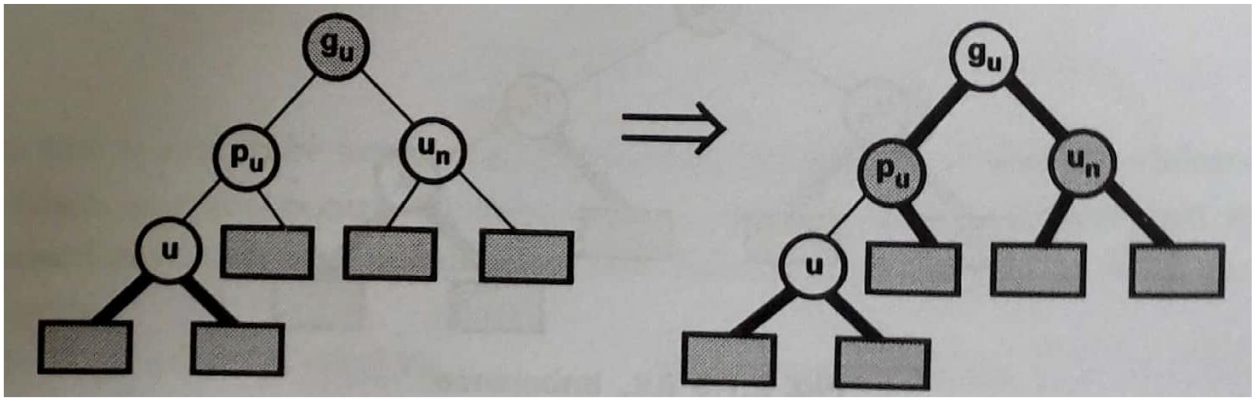



Fig. 37 Removal of LLr unbalancing

3. **RRr unbalancing:** The right child of node $\mathbf{g_u}$ is node $\mathbf{p_u}$ and **u** is inserted as right Child of $\mathbf{p_u}$. Colour of $\mathbf{p_u}$ and $\mathbf{u_n}$ node (uncle node) is red and newly inserted node **u** is also red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in right subtree** as a child in **right side**, so it is **RR** unbalancing. Now as its uncle node $\mathbf{u_n}$ is red so it is **RRr unbalancing** (inserted node always take colour reference from uncle node $\mathbf{u_n}$).

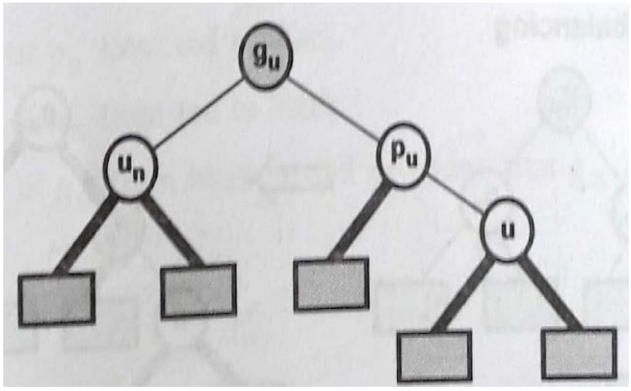



Fig. 38 RRr unbalancing

**Removal of RRr unbalancing -** Before colour change note that if $\mathbf{g_u}$ in given figures is root then there should not be any colour change of $\mathbf{g_u}$ (Because root is always black). But if $\mathbf{g_u}$ happens to be red that rebalancing can be done as-

1. Change the colour of $\mathbf{p_u}$ from red to black.
2. Change the colour of $\mathbf{u_n}$ from red to black.
3. Change the colour of $\mathbf{g_u}$ from black to red provided gu is not a root node.

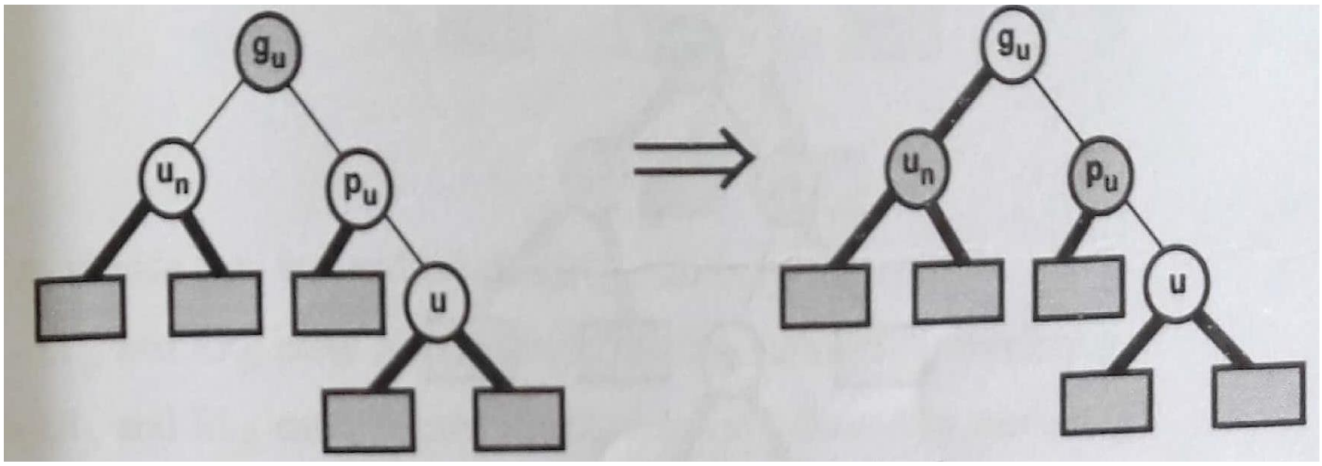



Fig. 39 Removal of RRr unbalancing

4. **RLr Imbalance:** The node $\mathbf{p_u}$ is right child of $\mathbf{g_u}$ and **u** is inserted as a left child of $\mathbf{p_u}$. Colour of $\mathbf{p_u}$ and $\mathbf{u_n}$ node (uncle node) is red and newly inserted node **u** is also red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in right subtree** as a child in **left side**, so it is **RL** unbalancing. Now as its uncle node $\mathbf{u_n}$ is red so it is **RLr unbalancing** (inserted node always take colour reference from unsscle node $\mathbf{u_n}$).

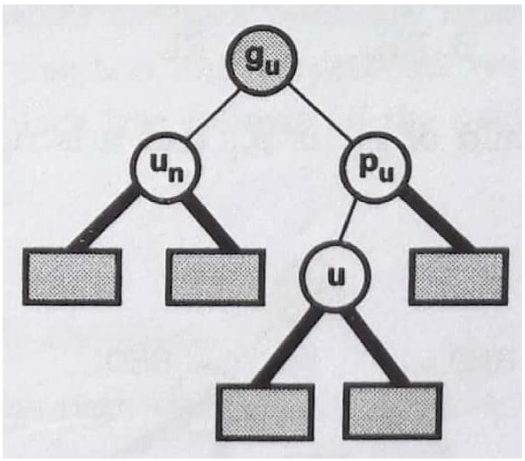



Fig. 40 RLr unbalancing

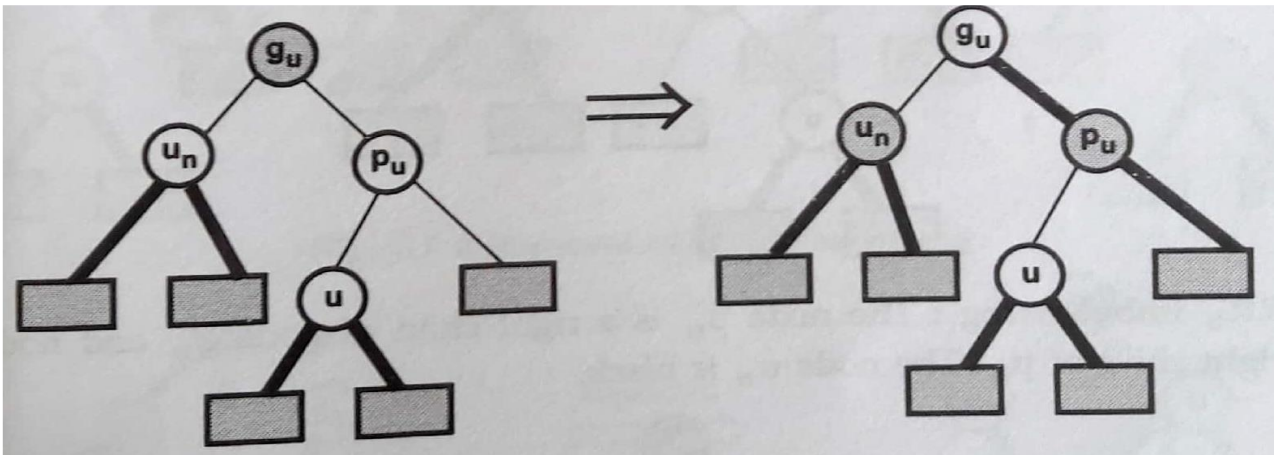



Fig. 41 Removal of RLr unbalancing

**Removal of RLr unbalancing** = Before colour change note that if $\mathbf{g_u}$ in given figures is root then there should not be any colour change of $\mathbf{g_u}$ (Because root is always black). But if $\mathbf{g_u}$ happens to be red that rebalancing can be done as-

1. Change the colour of $\mathbf{p_u}$ from red to black.
2. Change the colour of $\mathbf{u_n}$ from red to black.
3. Change the colour of $\mathbf{g_u}$ from black to red provided gu is not a root node.

The **black unbalancing** is concerned with the **colour of grandparent's child (i.e uncle node) is black.** In these types of unbalancing **first rotation** is **applied** (position of node is changed) and **second colour** of node **is changed.**

**1. LRb** **2. LLb** **3. RRb** **4 LRb**

In red black tree when **black unbalancing occur**. To remove these unbalancing **rotations are required** and then by **changing the colour required** balancing can obtained. Let us see how the black **unbalancing occurs** in tree and how we solve this black **unbalancing** by **rotation** and **changing colour**.

1. **LRb unbalancing:** The $p_u$ node is attached as a left child of $g_u$ and **u** is inserted as a right child of $p_u$. Colour of $u_n$ node (uncle node) is black and newly inserted node **u** and parent node $p_u$ is red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in left subtree** as a child in **right side**, so it is **LR** unbalancing. Now as its uncle node $u_n$ is black so it is **LRb unbalancing** (inserted node always take colour reference from uncle node $u_n$).

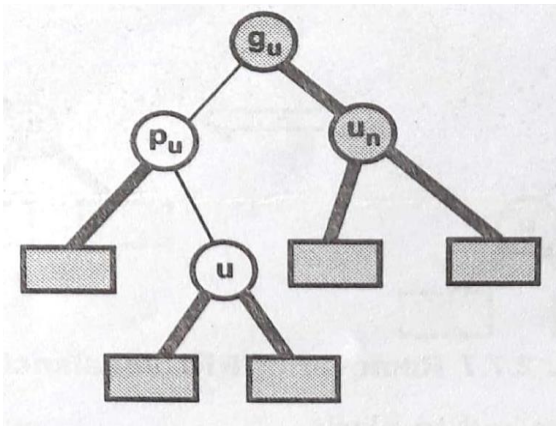


Fig 42 LRb unbalancing

**Removal of LRb unbalancing –** As **u** node gets inserted **rebalancing** must be **performed**. In **LRb case** require **Double rotation** followed by **recolouring.**

1. Apply double rotation of **u** about $p_u$ followed by **u** about $g_u$.
2. For **LRb** recolour **u** to black and recolour $p_u$ and $g_u$ to red.
3. For **RLb** recolour $p_u$ to black.

In **Double rotation** Grandchild of $g_u$ **i.e. u (**newly inserted node**) replace** $g_n$ **become root node** and tree bend toward right side as shown in fig. Change the colour of **u from red** to **black** as red root always have both black child otherwise it violate RED BLACK property and make tree unbalance. Colour of node $g_n$ **will change black to red** but colour of node $p_u$ and uncle node $u_n$ will **not change.**

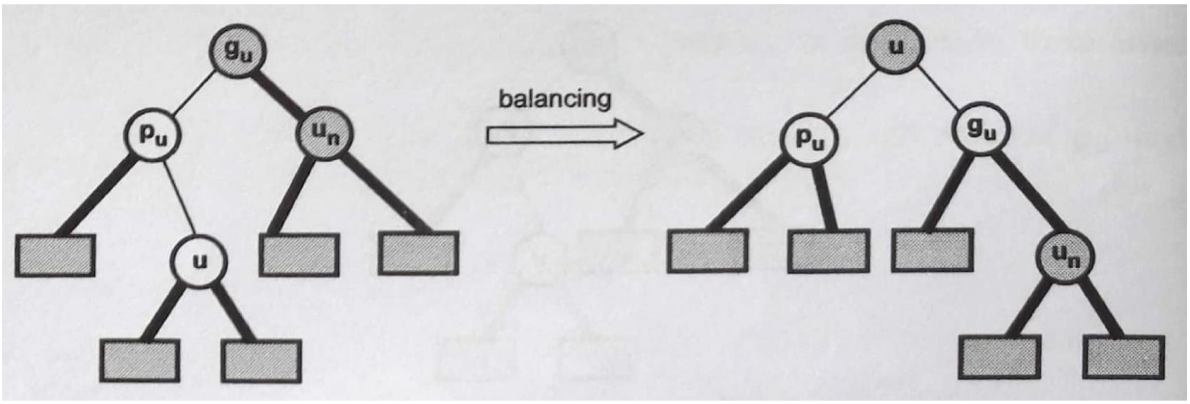


Fig 43 Removal of LRb unbalancing

If node **u** is red then it is not a root node and it's both child will be black. We can **recolour node u to black** if needed in case to make node **u** as root node, as root of red black tree is always black

2. **LLb unbalancing:** The $p_u$ node is attached as a left child of $g_u$ and **u** is inserted as a left child of $p_u$. Colour of $u_n$ node (uncle node) is black and newly inserted node **u** and parent node $p_u$ is red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in left subtree** as a child in **left side**, so it is **LL** unbalancing. Now as its uncle node $u_n$ is black so it is **LLb unbalancing** (inserted node always take colour reference from uncle node $u_n$).

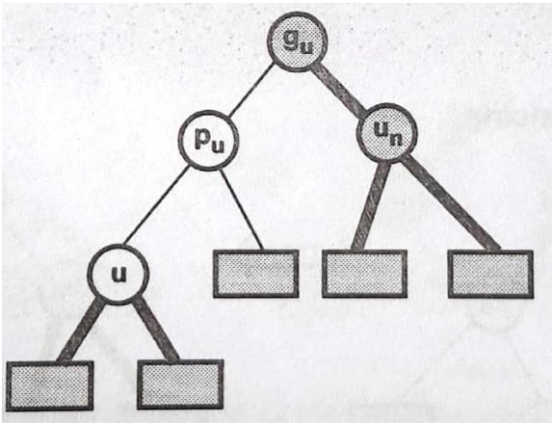



Fig 55 LLb unbalancing

**Removal of LLb unbalancing –** As **u** node gets inserted **rebalancing** must be **performed**. In **LLb case** require **Single rotation** followed by **recolouring.**

1. Apply single rotation of $p_u$ about $g_u$.
2. Recolour $p_u$ to black and $g_u$ to red.

In **single rotation** immediate child of $g_u$ **i.e.** $p_u$ **become root node**, where new node **u** is connected and tree bend toward right side as shown in fig. Then **recolour** $p_u$ **to black** as it is root node and root of red black tree is always black. Change the colour of $g_u$ **from black to red** as in a path two continuous black non leaf node violate RED BLACK property and make tree unbalance. Colour of uncle node $u_n$ and node **u** colour will **not change.**

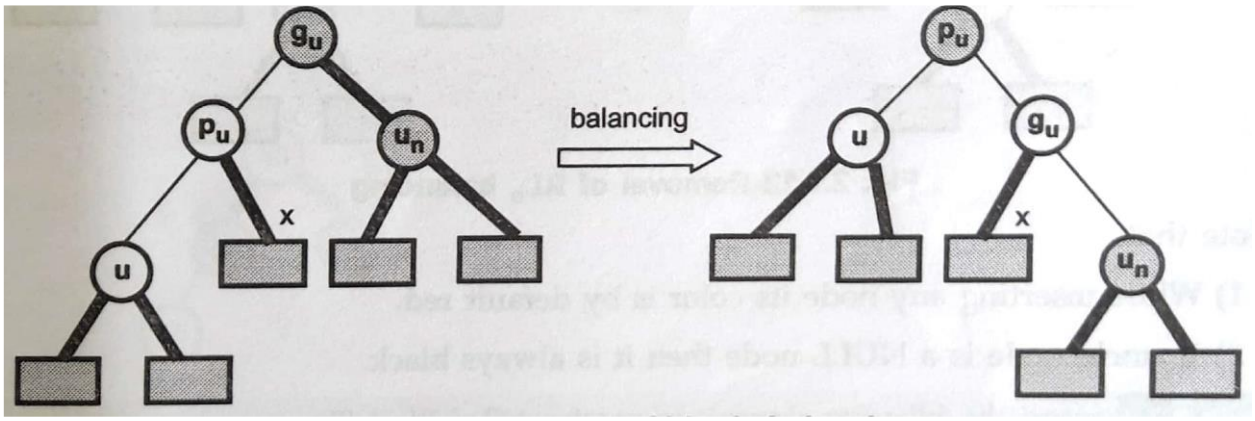



**Fig 56 Removal of LLb unbalancing**

3. **RRb unbalancing:** The $p_u$ node is attached as a right child of $g_u$ and **u** is inserted as a right child of $p_u$. Colour of $u_n$ node (uncle node) is black and newly inserted node **u** and parent node $p_u$ is red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in right subtree** as a child in **right side**, so it is **RR** unbalancing. Now as its uncle node $u_n$ is black so it is **RRb unbalancing** (inserted node always take colour reference from uncle node $u_n$).

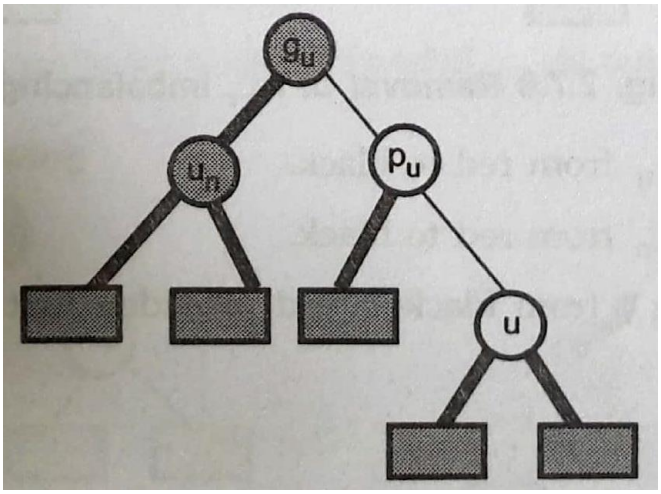



Fig 57 RRb unbalancing

**Removal of RRb unbalancing –** As **u** node gets inserted **rebalancing** must be **performed**. In **RRb case** require **Single rotation** followed by **recolouring.**

1. Apply single rotation of $p_u$ about $g_u$.
2. Recolour $p_u$ to black and $g_u$ to red.

In **single rotation** immediate child of $g_u$ **i.e.** $p_u$ **become root node**, where new node **u** is connected and tree bend toward left side as shown in fig. Then **recolour** $p_u$ **to black** as it is root node and root of red black tree is always black. Change the colour of $g_u$ **from black to red** as in a path two continuous black non leaf node violate RED BLACK property and make tree unbalance. Colour of uncle node $u_n$ and node **u** colour will **not change.**

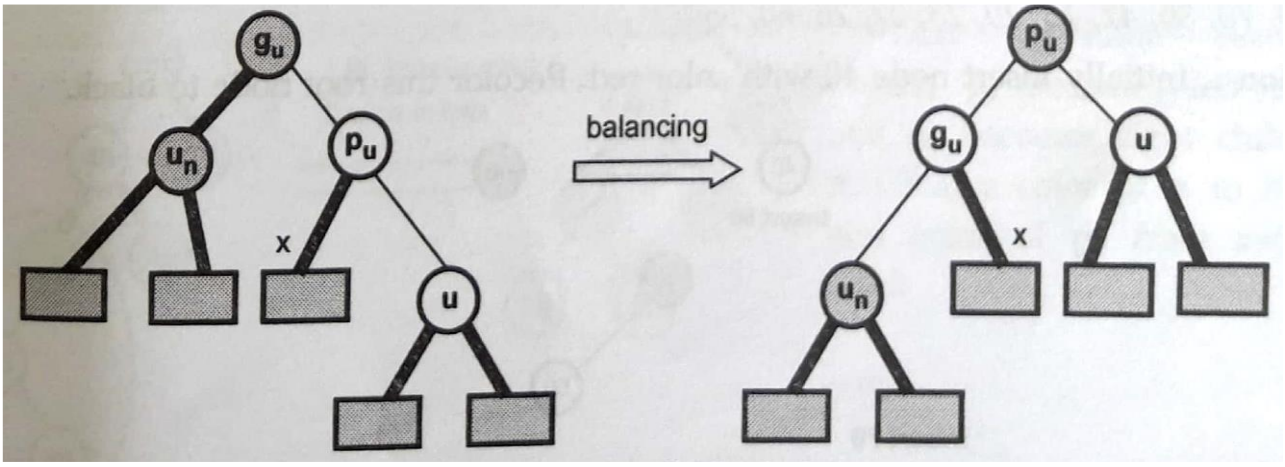



Fig 58 Removal of RRb unbalancing

4. **RLb unbalancing:** The $p_u$ node is attached as a right child of $g_u$ and **u** is inserted as a left child of $p_u$. Colour of $u_n$ node (uncle node) is black and newly inserted node **u** and parent node $p_u$ is red. There are two continuous red nodes, which violate RED BLACK tree property. **u** is **inserted in right subtree** as a child in **left side**, so it is **RL** unbalancing. Now as its uncle node $u_n$ is black so it is **RLb unbalancing** (inserted node always take colour reference from uncle node $u_n$).

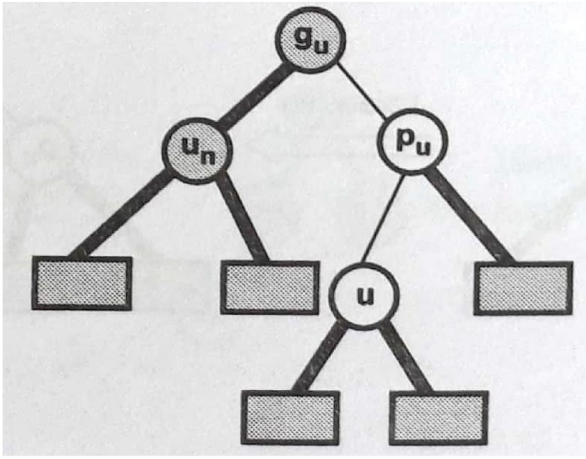



Fig 59 RLb unbalancing

**Removal of RLb unbalancing –** As **u** node gets inserted **rebalancing** must be **performed**. In **RLb case** require **Double rotation** followed by **recolouring.**

1. Apply double rotation of **u** about $p_u$ followed by **u** about $g_u$.
2. For **LRb** recolour **u** to black and recolour $p_u$ and $g_u$ to red.
3. For **RLb** recolour $p_u$ to black.

In **Double rotation** Grandchild of $g_u$ **i.e. u (**newly inserted node**) replace** $g_n$ **become root node** and tree bend toward left side as shown in fig. Change the colour of **u from red** to **black** as red root always have both black child otherwise it violate RED BLACK property and make tree unbalance. Colour of node $g_n$ **will change black to red** but colour of node $p_u$ and uncle node $u_n$ will **not change.**

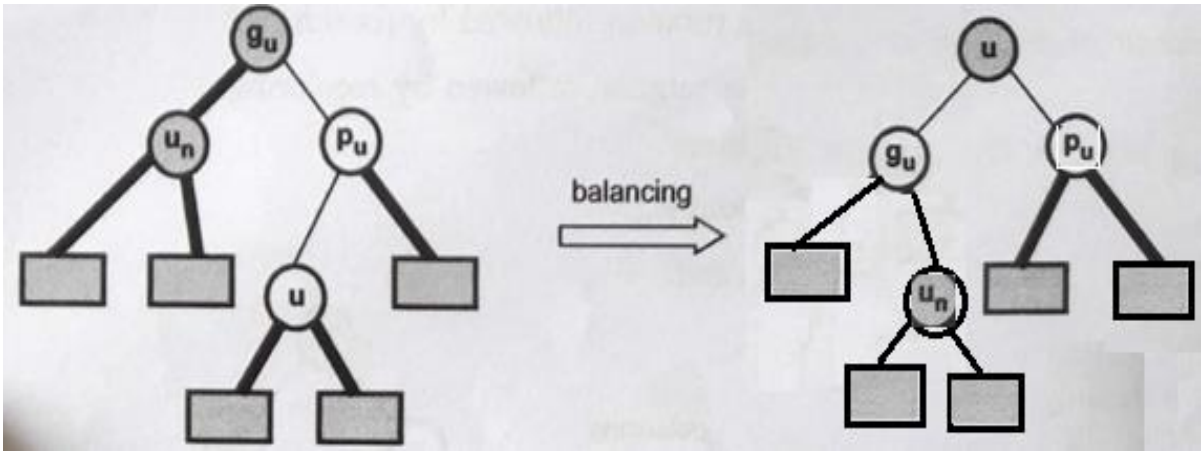


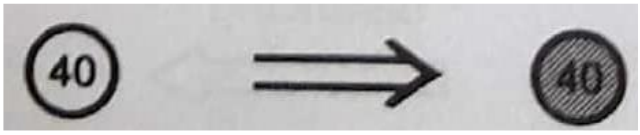

**Fig 60 Removal of RLb unbalancing**

If node **u** is red then it is not a root node and it's both child will be black. We can **recolour node u to black** if needed in case to make node **u** as root node, as root of red black tree is always black

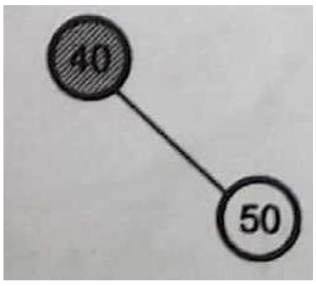
**Example:** Insert the following element to construct RED BLACK tree
40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55

**Solution:** Initially insert node 40 with colour red. Recolour this root node to black.

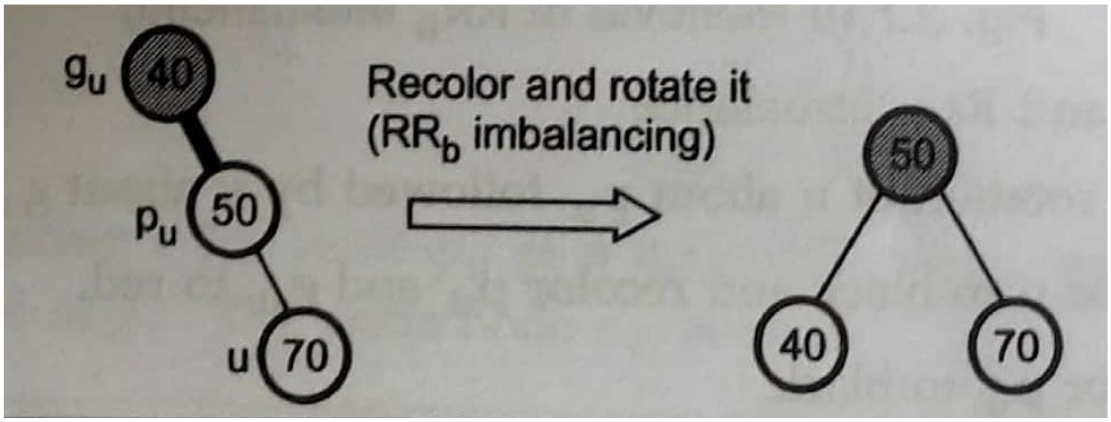
**Step 1: INSERT 40**
Inserted node 40 is red. As it is root node so recolouring it in black
Recolour node 40 **red→black**



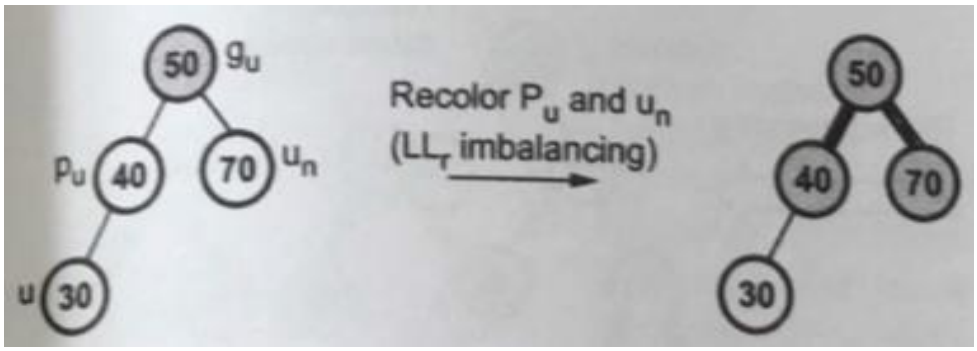

**Step 2: INSERT 50**



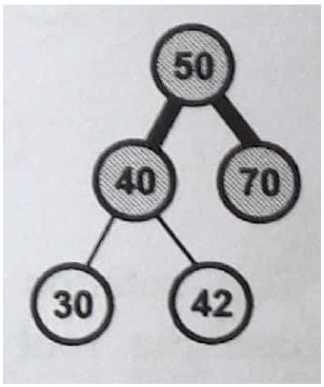

**Step 3: INSERT 70**



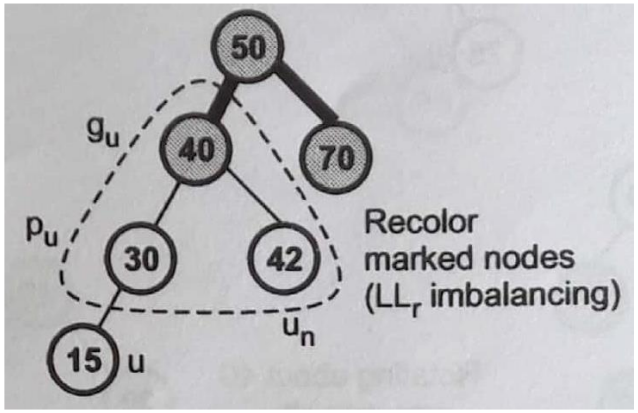
**Step 4: INSERT 30**

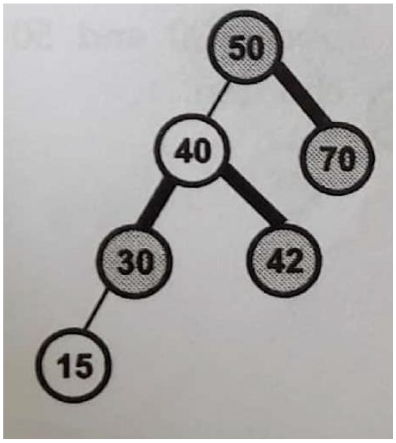



**Step 5: INSERT 42**

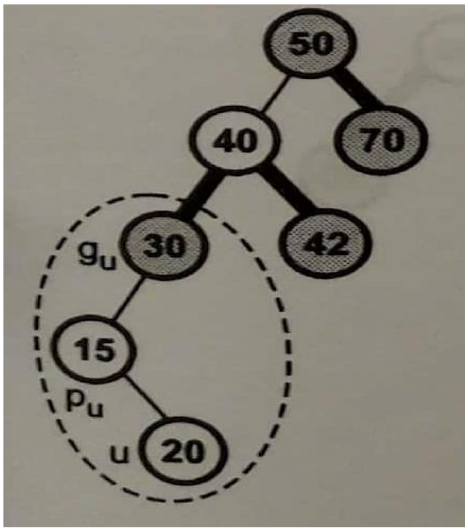


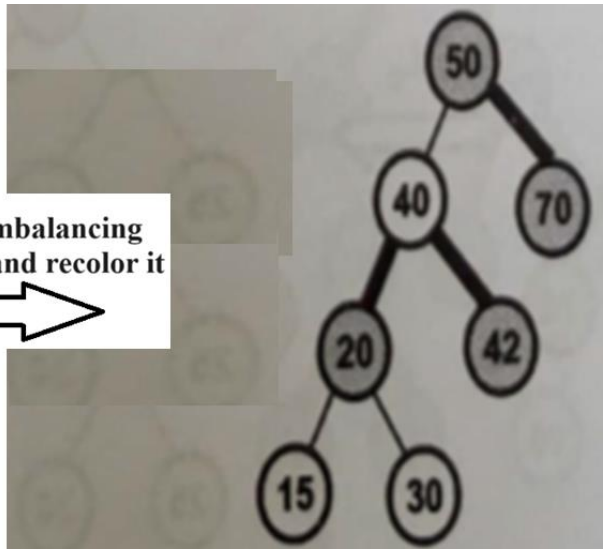

**Step 6: INSERT 15**



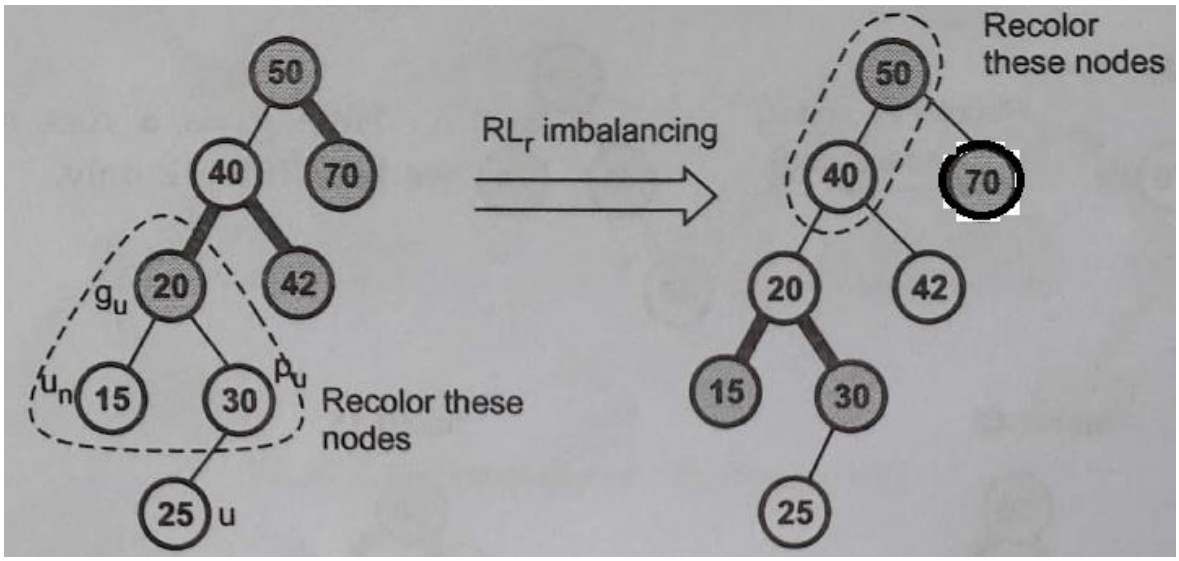



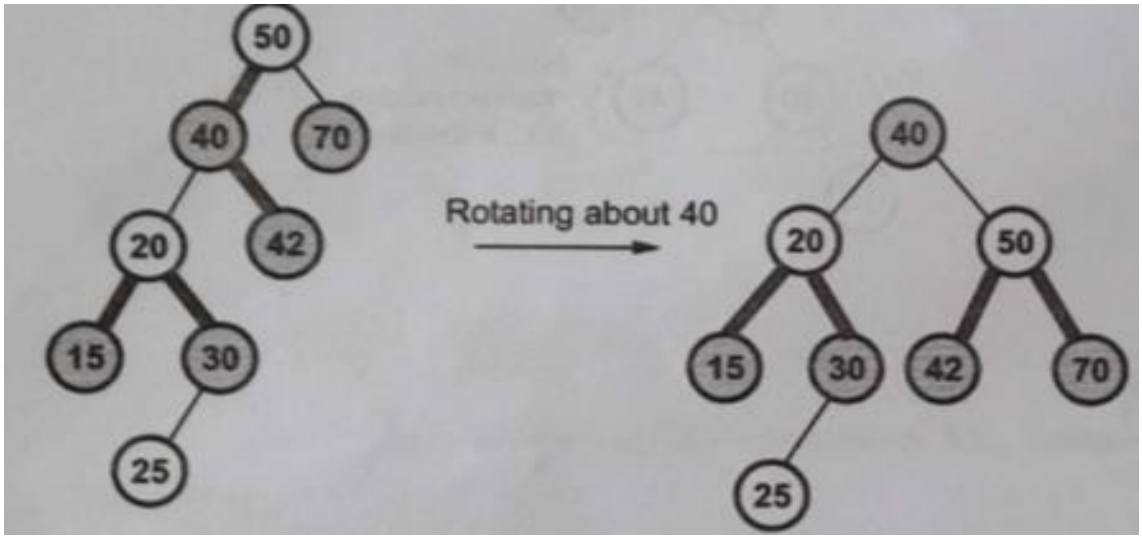

**Step 7: INSERT 20**



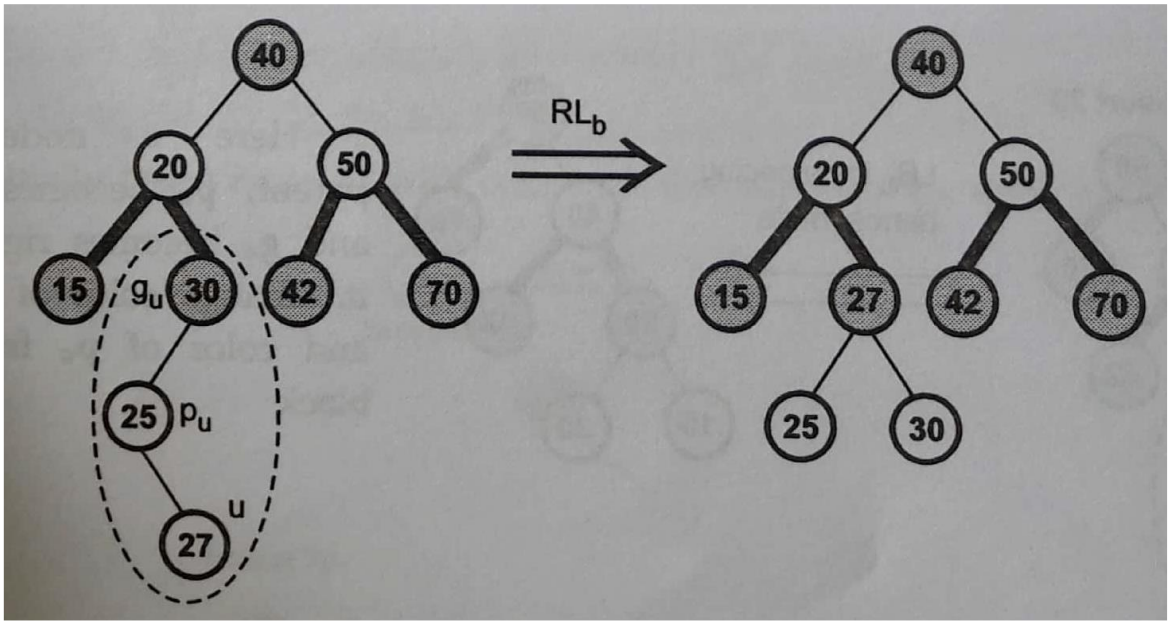



**Step 8: INSERT 25**

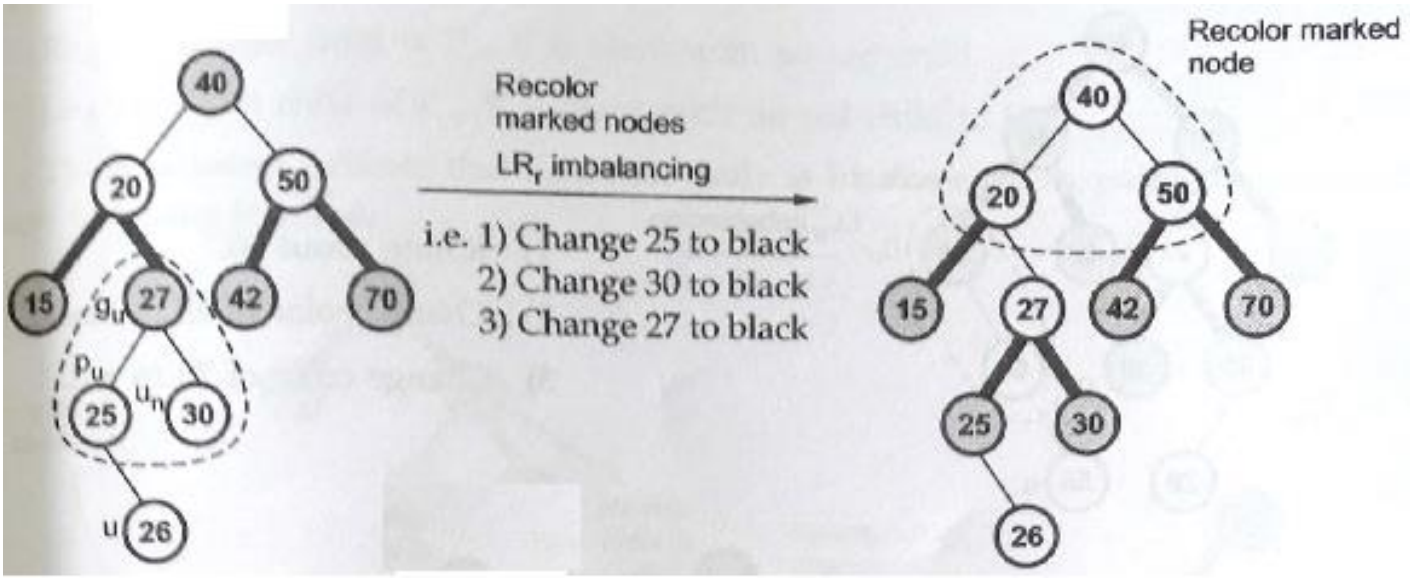


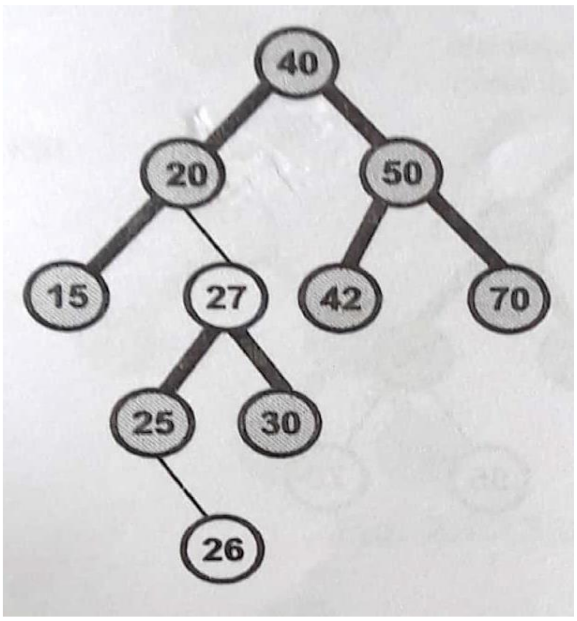




**Step 9: INSERT 27**

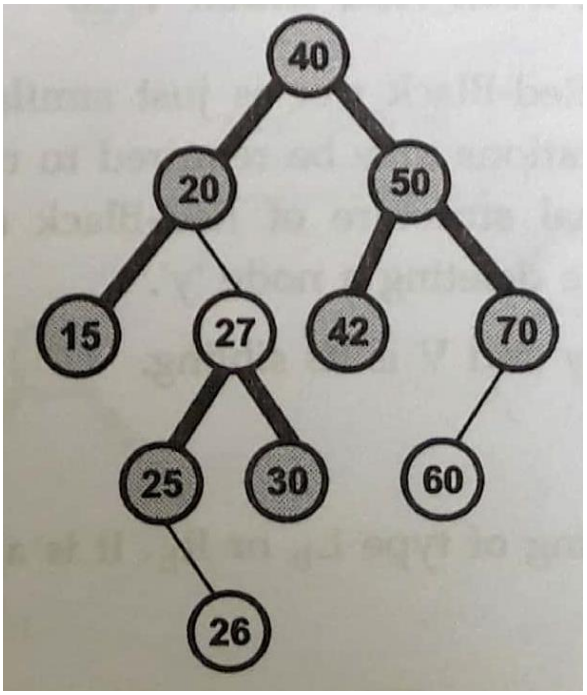



**Step 10: INSERT 26**




**Step 11: INSERT 60**



**Step 12: INSERT 55**

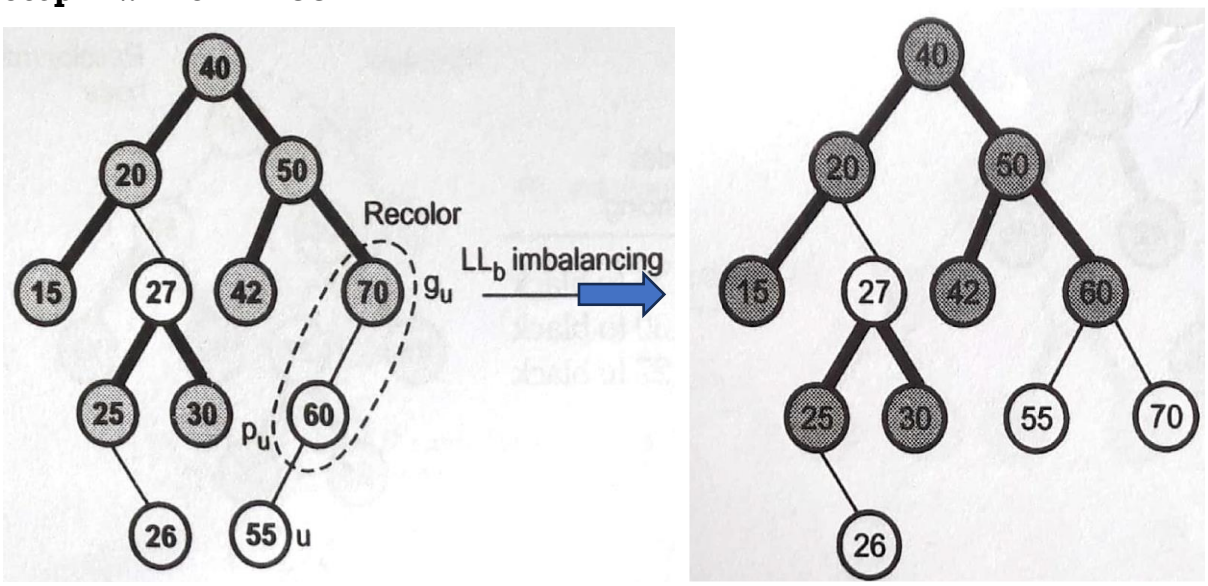



Fig. 61 Led Black Tree

1) Rotate about 60.
2) Change colour of 60 to black. Is final red-black tree
3) Change colour of 70 to red.

**Deletion of Element from Red Black Tree**

Deletion of a node from Red-Black tree is just similar to deletion of a node from BST. But colour changes and rotations may be required to rebalance the tree after deletion occurs depending upon original structure of Red-Black tree. There are different cases that should be considered while deleting a node 'y'.

Let $\mathbf{P_y}$ a parent of node **y** and **V** is its sibling.

1. **$X_b$ Imbalancing:** If **V** is black then imbalancing of type **Lb** or **Rb.** It is also known as Xb imbalancing. Here X is either 'L' or 'R'.
2. **$X_r$ Imbalancing:** If **V** is red then imbalancing of type **Lr** or **Rr.** It is also known as Xb imbalancing. Here X is either 'L' or 'R'.

**Removing Xb imbalancing** (i.e. either Lb or Rb imbalancing) - The Xb imbalancing has three cases, based on number of children of V node.

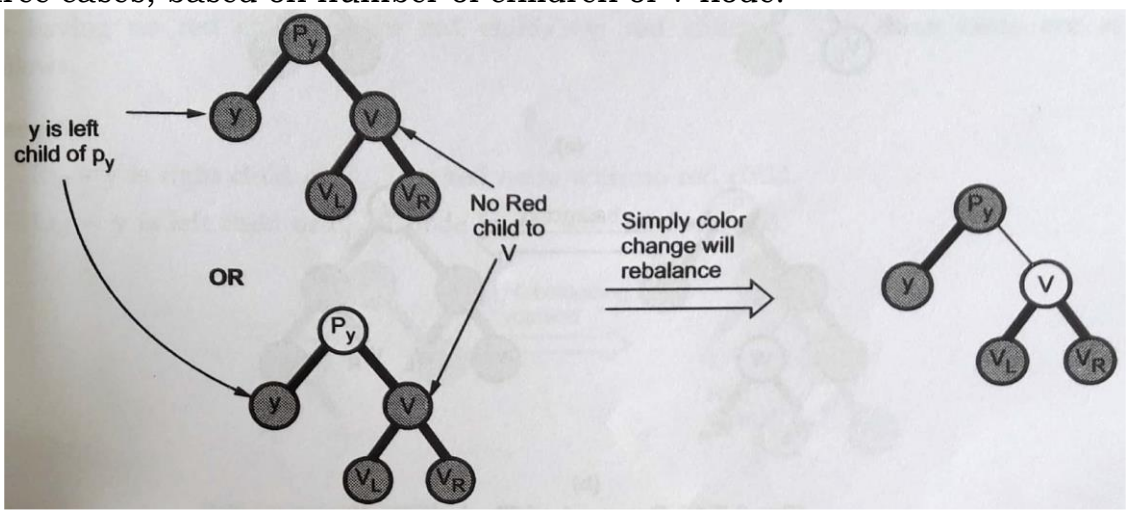



Fig. 62 Lbo unbalancing

**Case 1: - Rbo –** y is right child only, V is black with no red child.
**Lbo –** y is left child only, V is black with no red child to it.

Thick pointers indicate that successor node is black and thin pointers indicate that node following it is red.

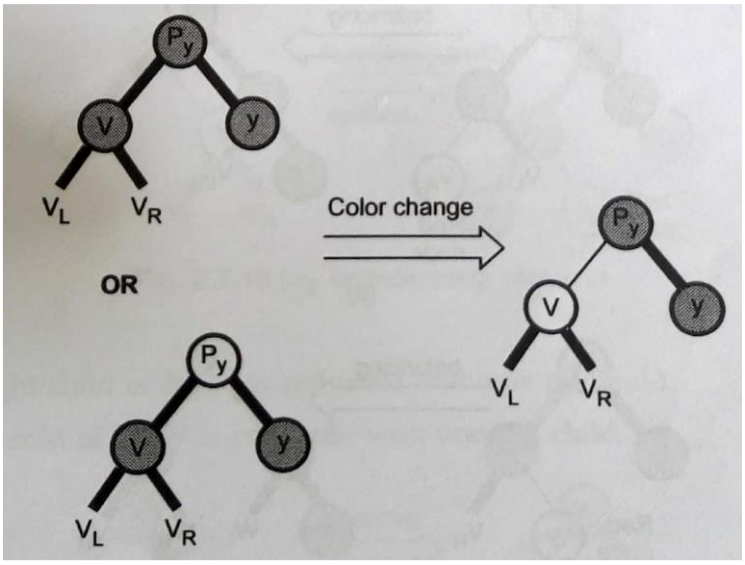



Fig. 63 Removal of Rbo unbalancing

**Case 2: - Rb1-** y is right child of Py, V is black node having one red child.
**Lb1-** y is left child of Py, V is black node having one red child.

The dotted node indicates that node may be red or black and is unchanged by rotation.

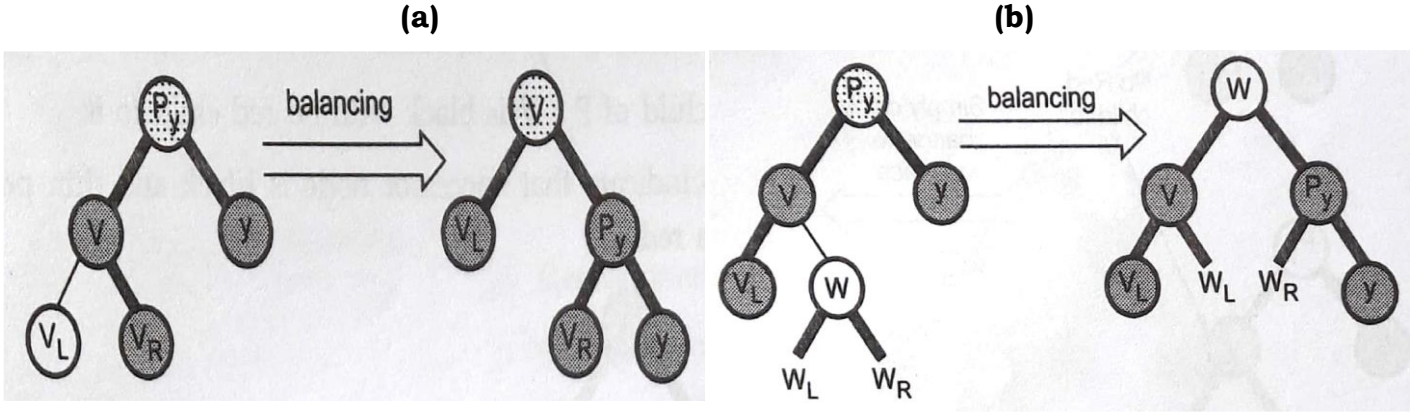



Fig. 64 Removal of Rb1 imbalancing

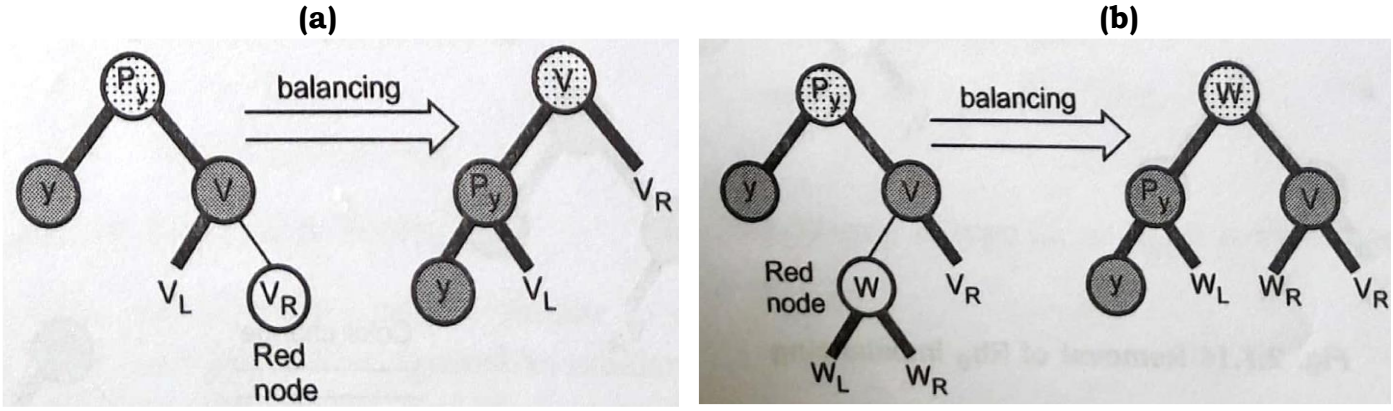



Fig. 65 Removal of Lb1 unbalancing

**Case 3: - Rb2** - y is right Child Of Py. V is black node with two red children.
**Lb2 -** y is left child of Py, V is black node with two red children.

**Removing Xr imbalancing –** In Xr imbalancing again there are three cases. The V (uncle node) is a red node may be having no red children/one red child/two red children. The three cases are as follows.

**Case 1:- Rbo –** y is right child only, V is red with no black child.
**Lbo –** y is left child only, V is red with no black child to it.

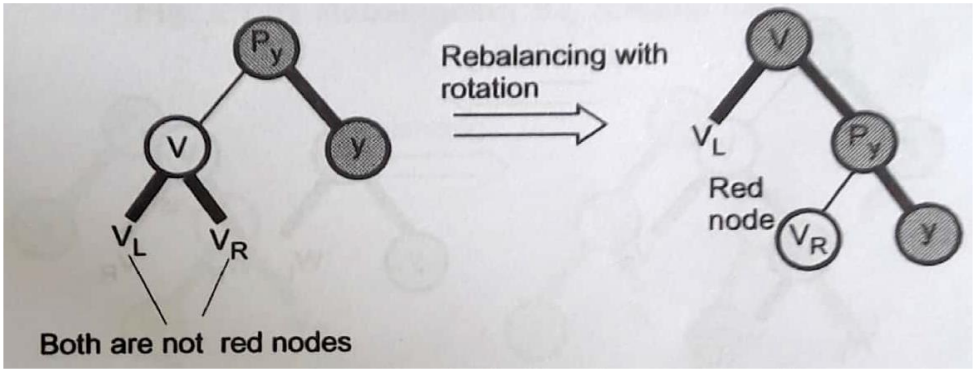



Fig. 66 Rro imbalancing removal

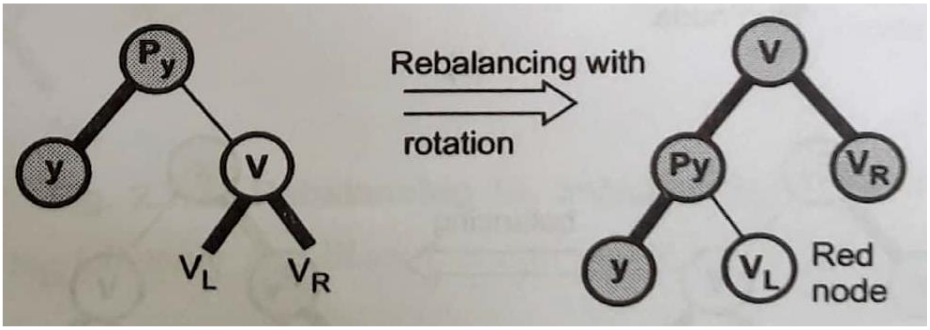
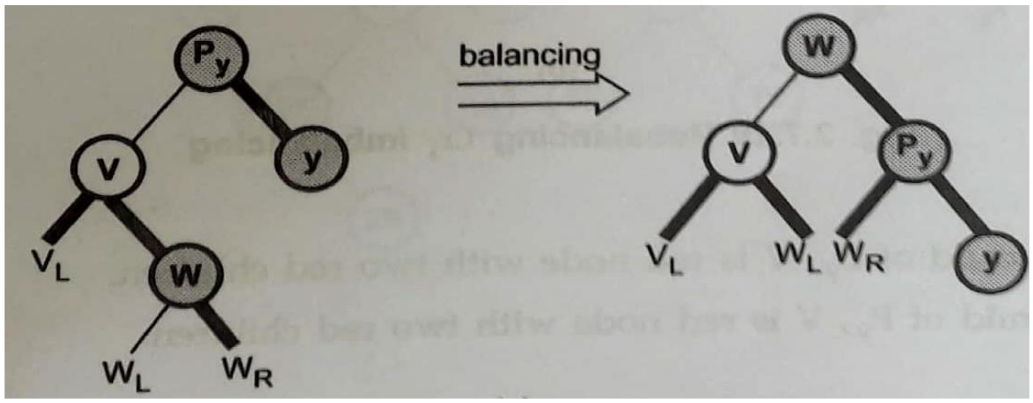



Fig. 67 Lro Imbalancing removal

**Case 2: - Rb1-** y is right child of Py, V is red node having one black child.
**Lb1-** y is left child of Py, V is red node having one black child.

**(a)**




**(b)**

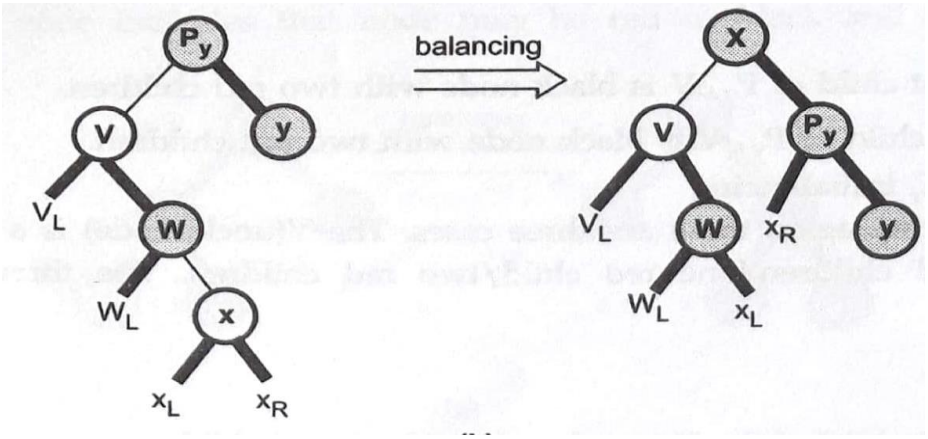
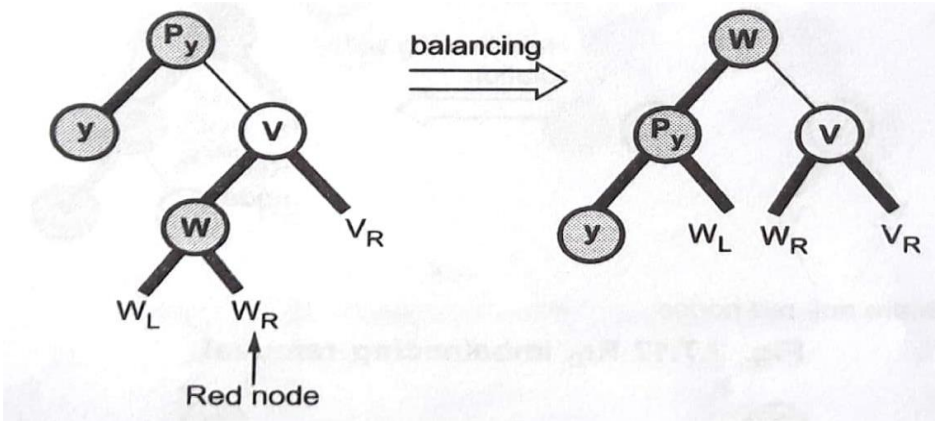



Fig. 68 Rr1 imbalancing removal

(a)




(b)

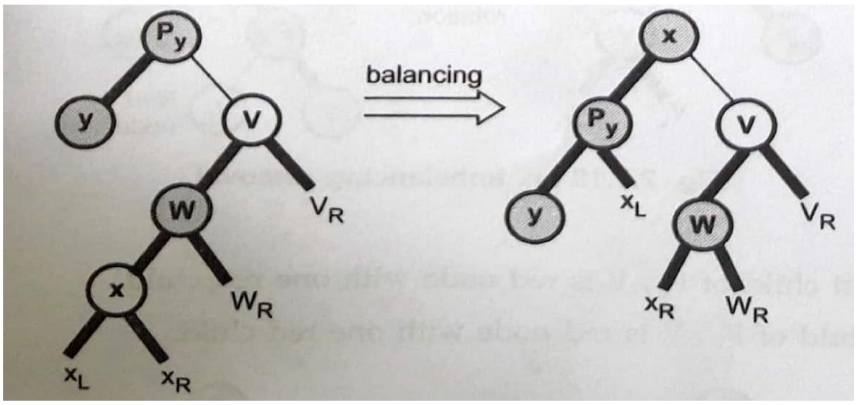



Fig. 69 Lr1 Imbalancing removal

**Case 3: - Rb2** - y is right Child Of Py. V is red node with two black children.
**Lb2 -** y is left child of Py, V is red node with two black children.

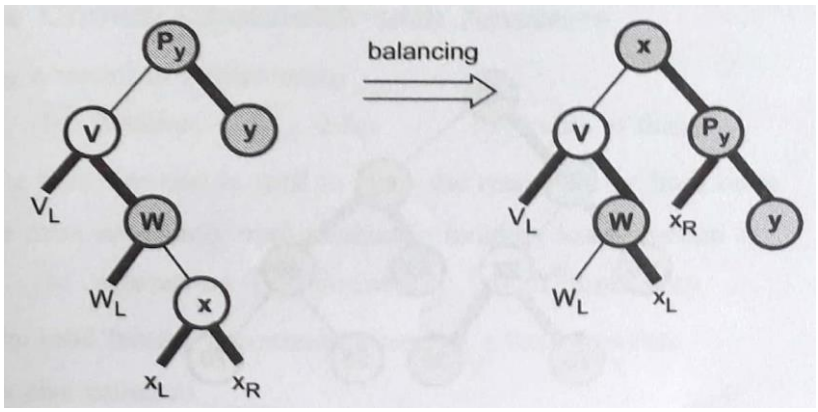



Fig. 70 Rr2 imbalancing removal

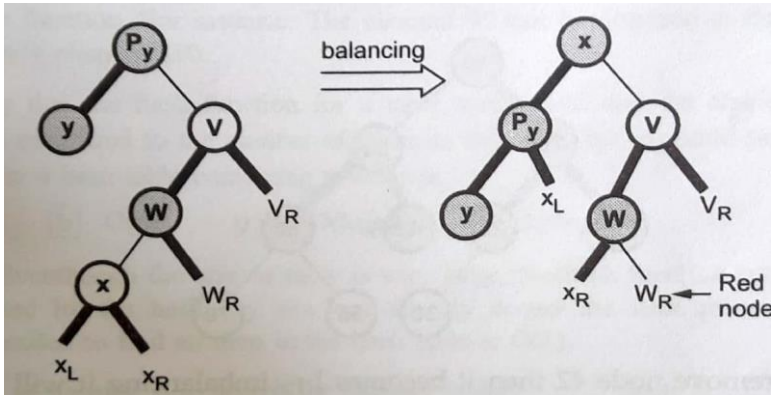
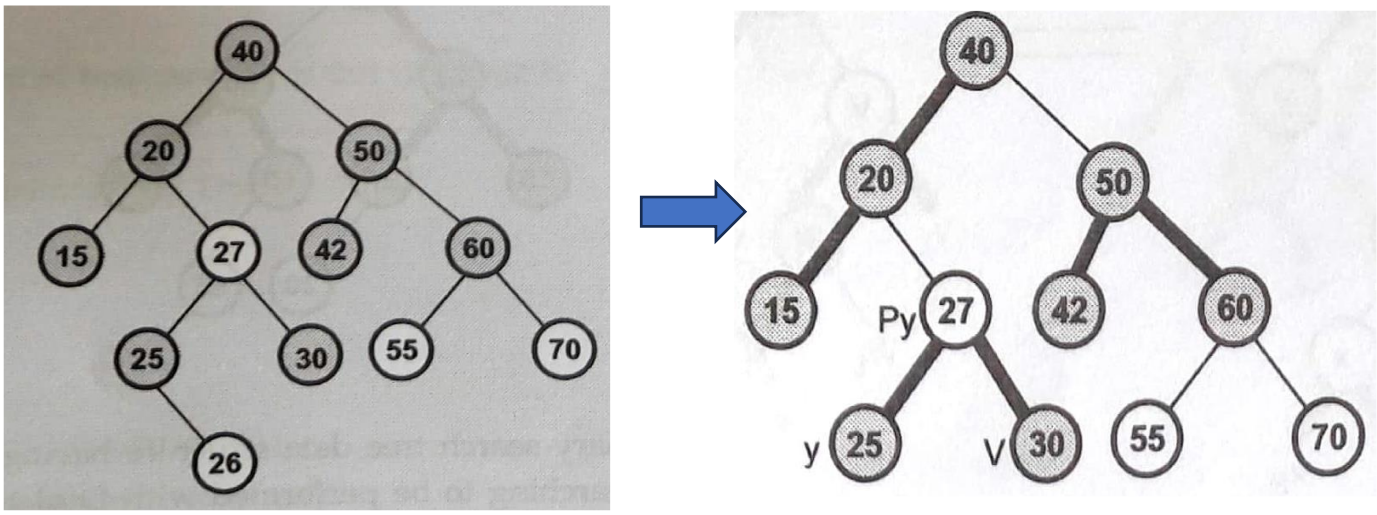


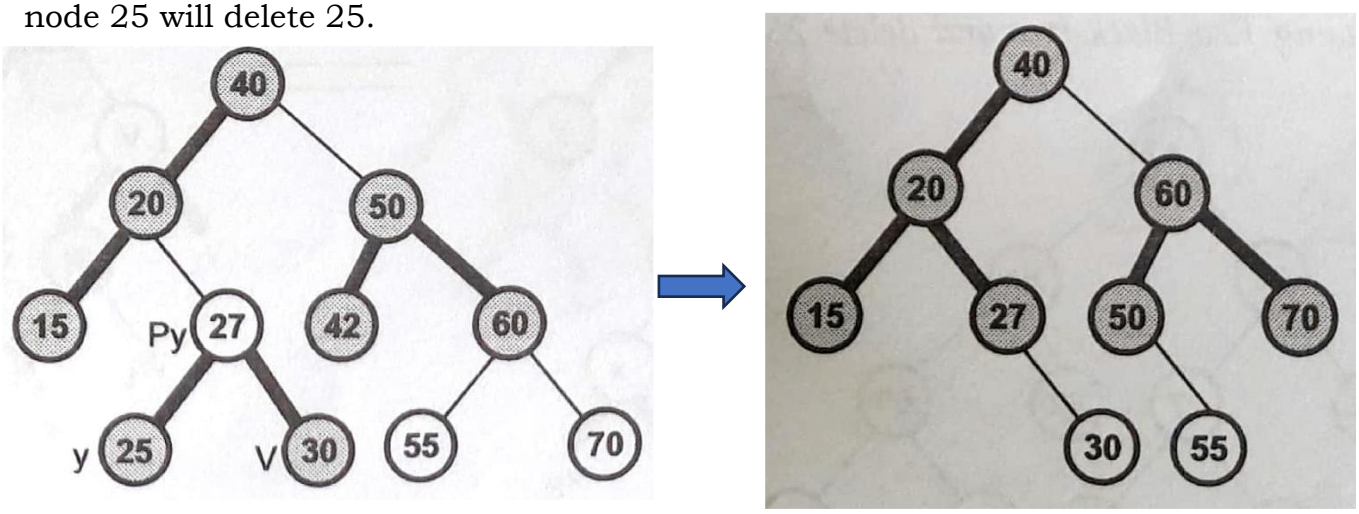

Fig. 71 Lr2 Imbalancing removal

**Example** - Consider the following Red Black tree and delete 26, 25 and 42.

**Solution: Step 1 -** If we want to **delete node 26** then simply node 26 will be removed. The tree then becomes.




**Step 2 -** Now if we want to **delete node 25** then the rebalancing will be of Lbo type. Hence simply colour change of V node to red and recolour 27 to black and removal of node 25 will delete 25.




**Step 3 -** If we want to **remove node 42** then it becomes Lb imbalancing it will be rebalanced by rotations and recolouring. Hence the tree becomes –

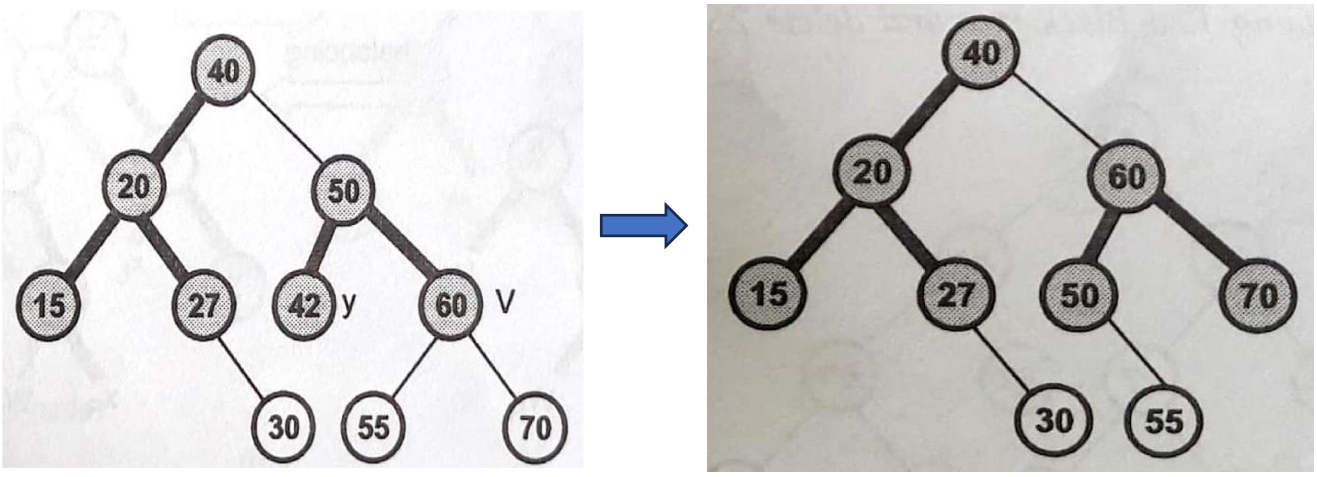



Fig 73 Deletion in Red Black tree

# Splay Tree

Splay tree is a **self-adjusting binary search tree** data structure, which means that the tree structure is **adjusted dynamically** based on the **accessed or inserted elements**. In other words, the tree **automatically reorganizes itself** so that frequently accessed or inserted elements become closer to the root node.

The splay tree was **first introduced** by **Daniel Dominic** Sleator and **Robert Endre** Tarjan **in 1985**. It has a **simple and efficient** implementation that allows it to perform search, insertion, and deletion operations in **O(log n)** amortized **time complexity,** where **n** is the **number of elements** in the tree. The **basic idea** behind splay trees is to bring the most **recently accessed or inserted element** to the root of the tree by performing a **sequence of tree rotations**, called **splaying**. Splaying is a process of **restructuring the tree** by making the most recently accessed or inserted element the new root and gradually moving the remaining nodes closer to the root.

Splay trees are **highly efficient** in practice due to their self-adjusting nature, which **reduces** the **overall access time** for frequently accessed elements. This makes them a good choice for applications that **require fast and dynamic** data structures, such as caching systems, data compression, and network routing algorithms. However, the main **disadvantage** of splay trees is that they **do not guarantee a balanced tree** structure, which may lead to **performance degradation** in worst-case scenarios. Also, splay trees are not suitable for applications that require guaranteed worst-case performance, such as real-time systems or safety-critical systems. Overall, splay trees are a **powerful and versatile** data structure that offers **fast and efficient access** to frequently accessed or inserted elements. They are widely used in various applications and provide an excellent tradeoff between performance and simplicity.

A splay tree is a self-balancing binary search tree, designed for efficient access to data elements based on their key values. The **key feature** of a splay tree is that each time an **element is accessed,** it is **moved to the root** of the tree, creating a more balanced structure for subsequent accesses. Splay trees are **characterized by their use** of rotations, which are local transformations of the tree that change its shape but preserve the order of the elements. **Rotations are used to bring the accessed element** to the root of the tree, and also to **rebalance the tree** if it becomes unbalanced after multiple accesses.

**Operations in a splay tree:**

- **Insertion:** To insert a new element into the tree, start by performing a regular binary search tree insertion. Then, apply rotations to bring the newly inserted element to the root of the tree.
- **Deletion**: To delete an element from the tree, first locate it using a binary search tree search. Then, if the element has no children, simply remove it. If it has one child, promote that child to its position in the tree. If it has two children, find the successor of the element (the smallest element in its right subtree), swap its key with the element to be deleted, and delete the successor instead.
- **Search**: To search for an element in the tree, start by performing a binary search tree search. If the element is found, apply rotations to bring it to the root of the tree. If it is not found, apply rotations to the last node visited in the search, which becomes the new root.

- **Rotation**: The rotations used in a splay tree are either a Zig or a Zig-Zig rotation. A Zig rotation is used to bring a node to the root, while a Zig-Zig rotation is used to balance the tree after multiple accesses to elements in the same subtree.

**Rotations in Splay Tree**

1. Zig rotation
2. Zag rotation
3. Zig-Zig rotation
4. Zag-Zag rotation
5. Zig-Zag rotation
6. Zag-Zig rotation

**1) Zig Rotation:** The Zig Rotation in splay trees operates in a manner similar to the **single right rotation** (clock wise) **in AVL Tree** rotations (used in LL rotation). This rotation results in nodes moving one position to the right from their current location. For example, consider the following scenario:

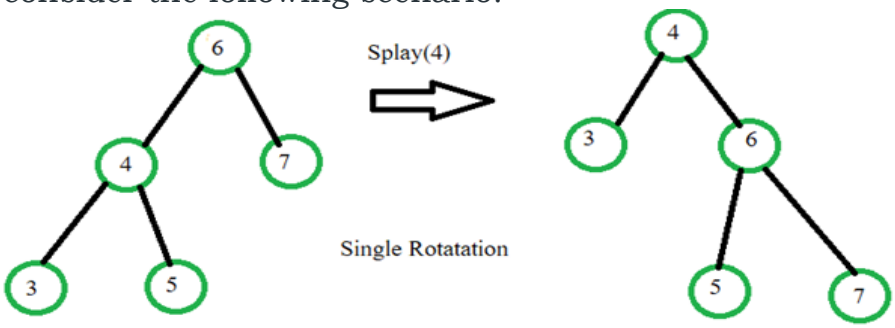



Fig. 74 Zig Rotation (Single Rotation)

**2) Zag Rotation:** The Zag Rotation in splay trees operates in a similar fashion to the **single left rotation** (anti clock wise) **in AVL** Tree rotations (used in RR rotation). During this rotation, nodes shift one position to the left from their current location. For instance, consider the following illustration:

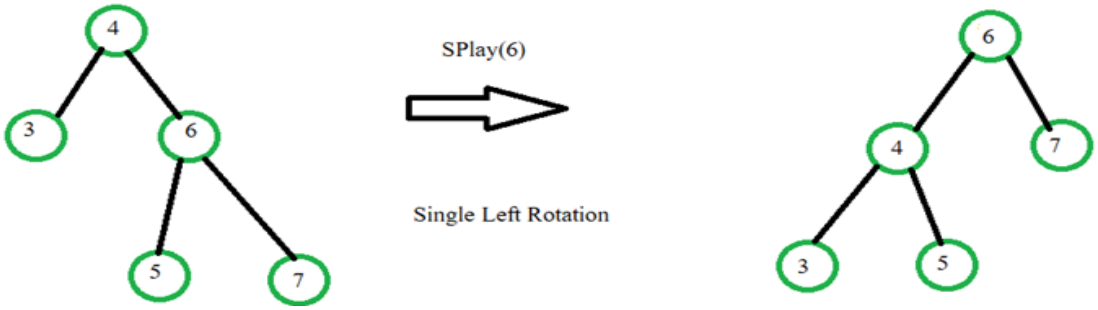



Fig. 75 Zag Rotation (Single left Rotation)

**3) Zig-Zig Rotation:** The Zig-Zig Rotation in splay trees is a **double zig rotation** (clock wise & clock wise)**.** This rotation results in nodes shifting two positions to the right from their current location. Take a look at the following example for a better understanding:

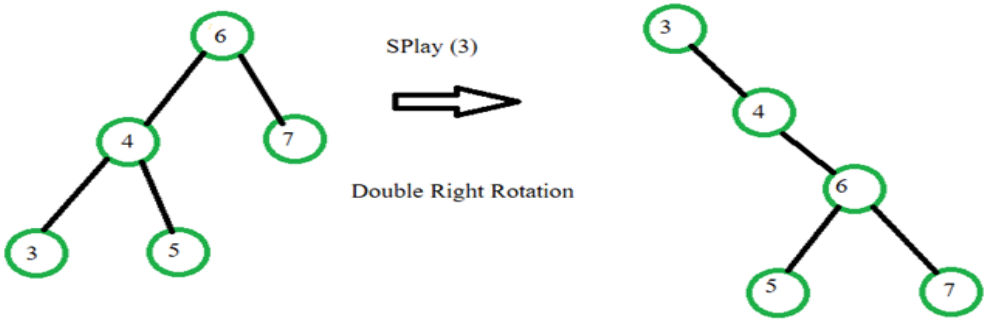



Fig. 76 Zig-Zig Rotation (Double Right Rotation)

**4) Zag-Zag Rotation:** In splay trees, the Zag-Zag Rotation is a **double zag rotation** (anti clock wise & anti clock wise)**.** This rotation causes nodes to move two positions to the left from their present position. For example:

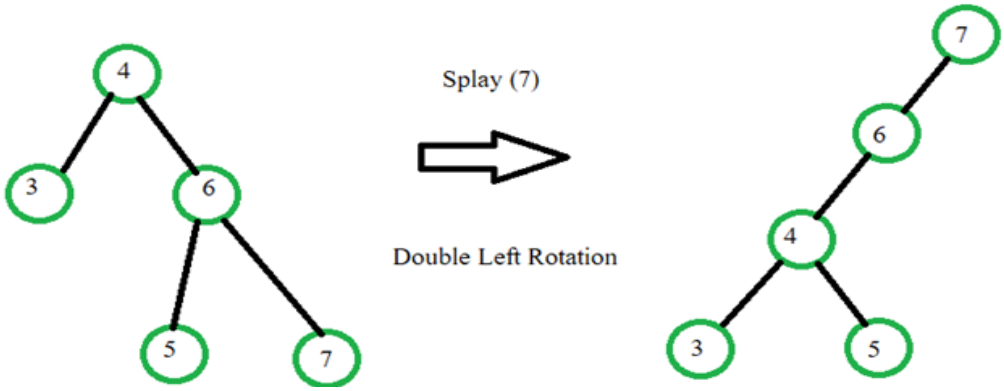



Fig. 77 Zag-Zag Rotation (Double left rotation)

**5) Zig-Zag Rotation:** The Zig-Zag Rotation in splay trees is a c**ombination of a zig rotation followed by a zag rotation** (clock wise & anti clock wise). As a result of this rotation, nodes shift one position to the right and then one position to the left from their current location. The following illustration provides a visual representation of this concept:

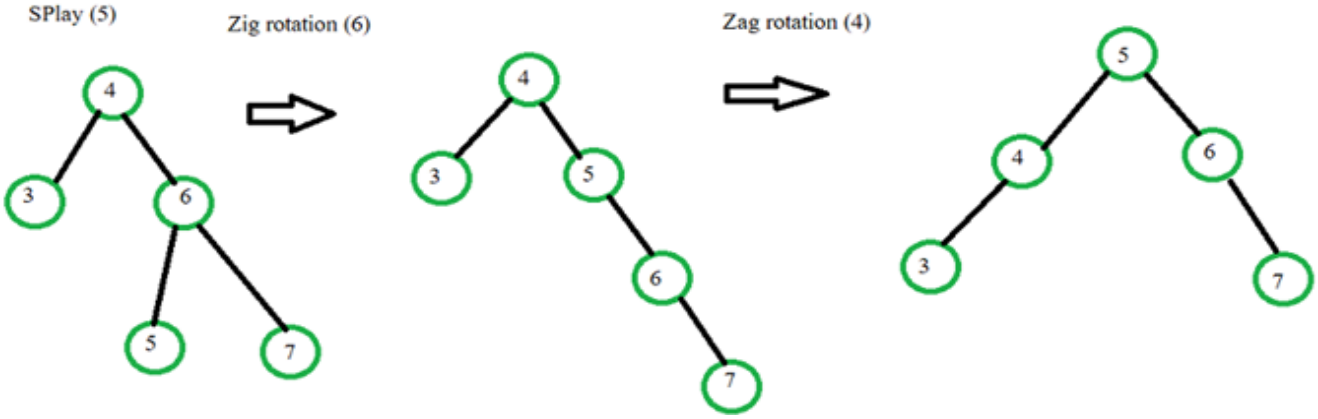



Fig. 78 Zig- Zag rotation

**6) Zag-Zig Rotation:** The Zag-Zig Rotation in splay trees is a **series of zag rotations followed by a zig rotation** (anti clock wise & clock wise). This results in nodes moving one position to the left, followed by a shift one position to the right from their current location. The following illustration offers a visual representation of this concept:

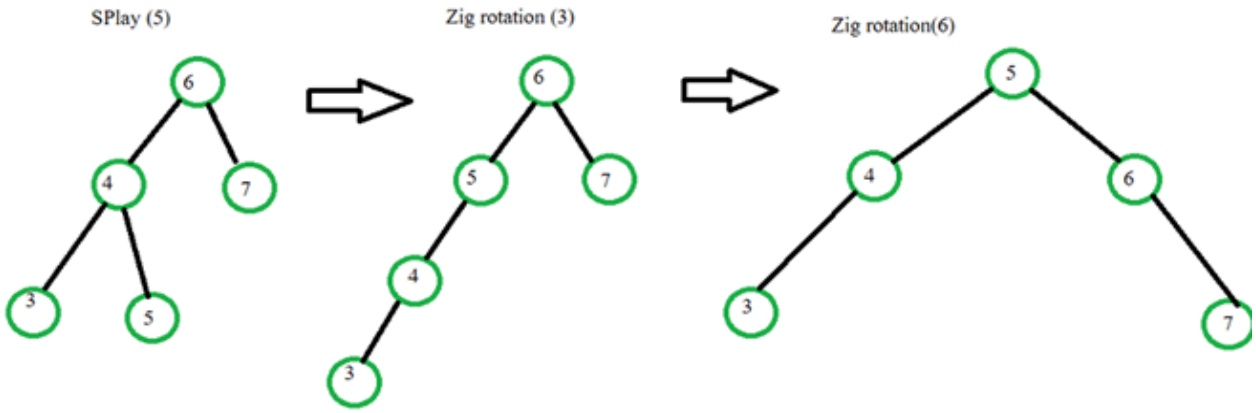
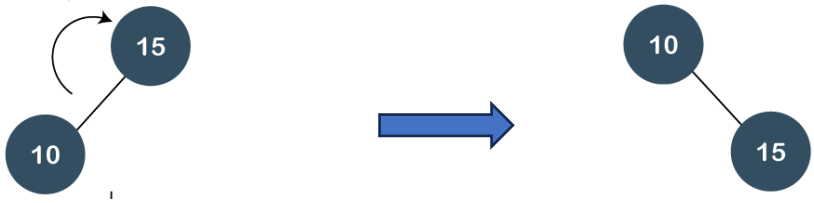



Fig. 79 Zag-Zig Rotation

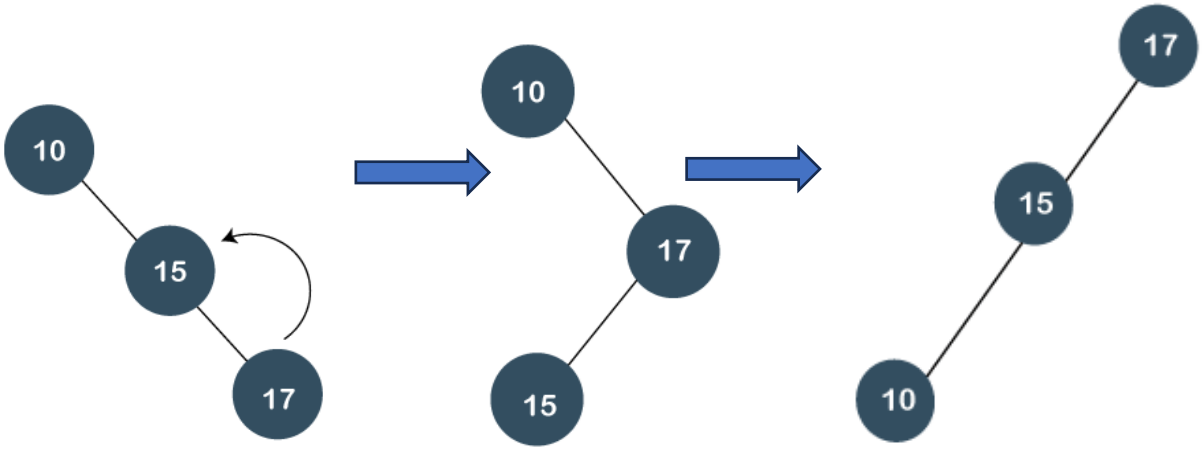
**Insertion operation in Splay tree** - In the ***insertion*** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.
**15, 10, 17, 7**

**Step 1:** First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.

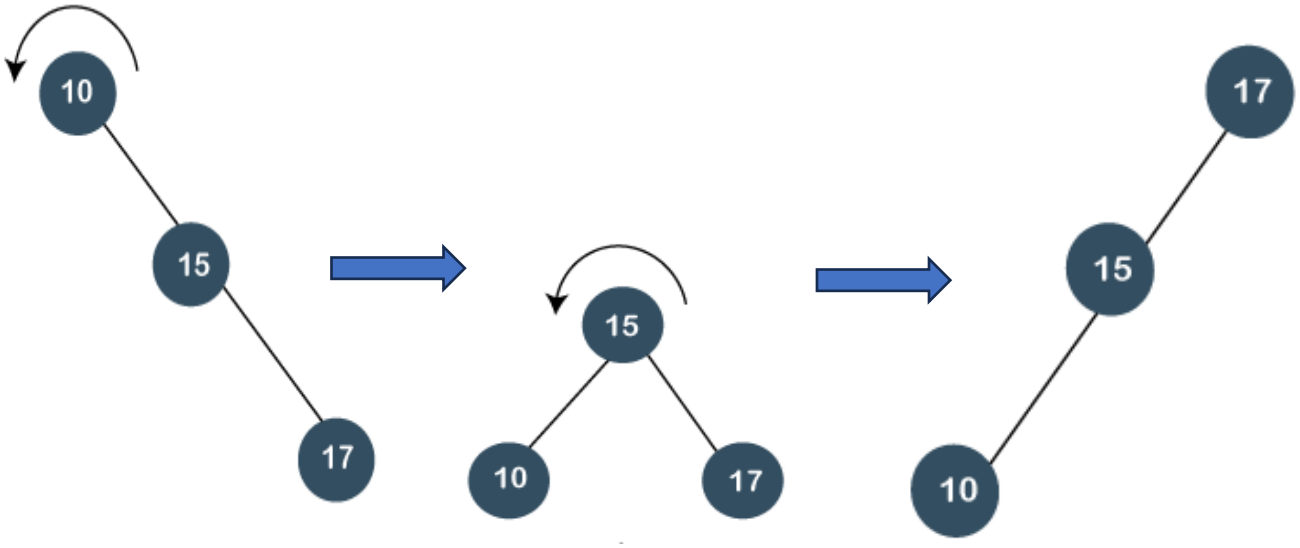

**Step 2:** The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15. Now, we perform ***splaying***. To make 10 as a root node, we will perform the right rotation, as shown below:



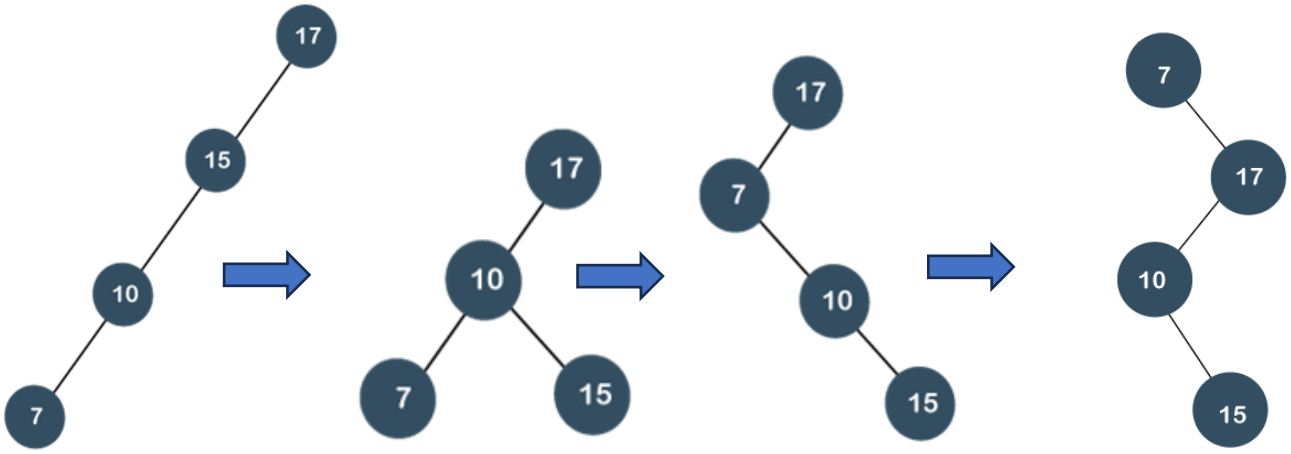

**Step 3:** The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15. Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zag and zag rotations.



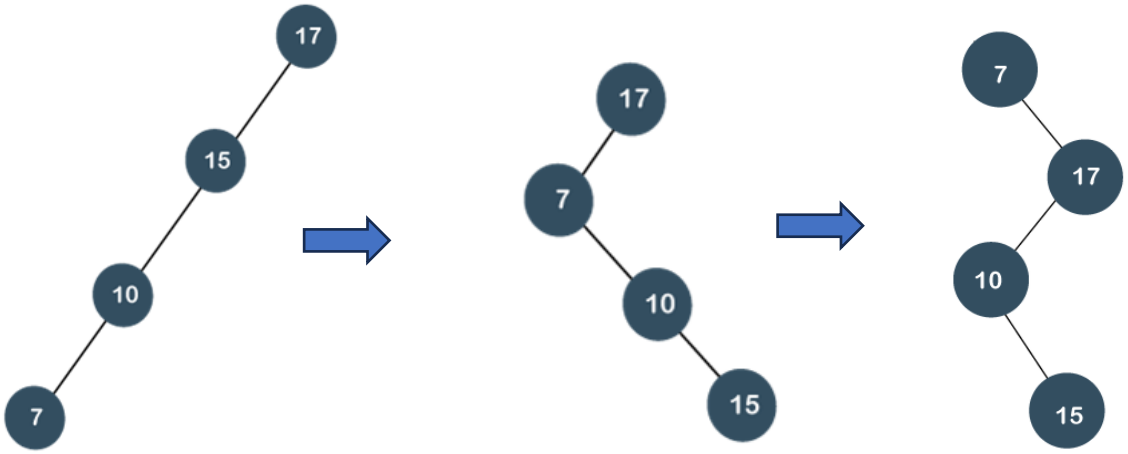

**OR**



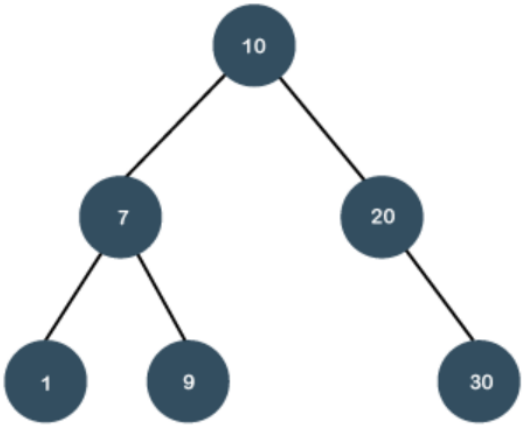

**Step 4:** The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10. Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:




OR

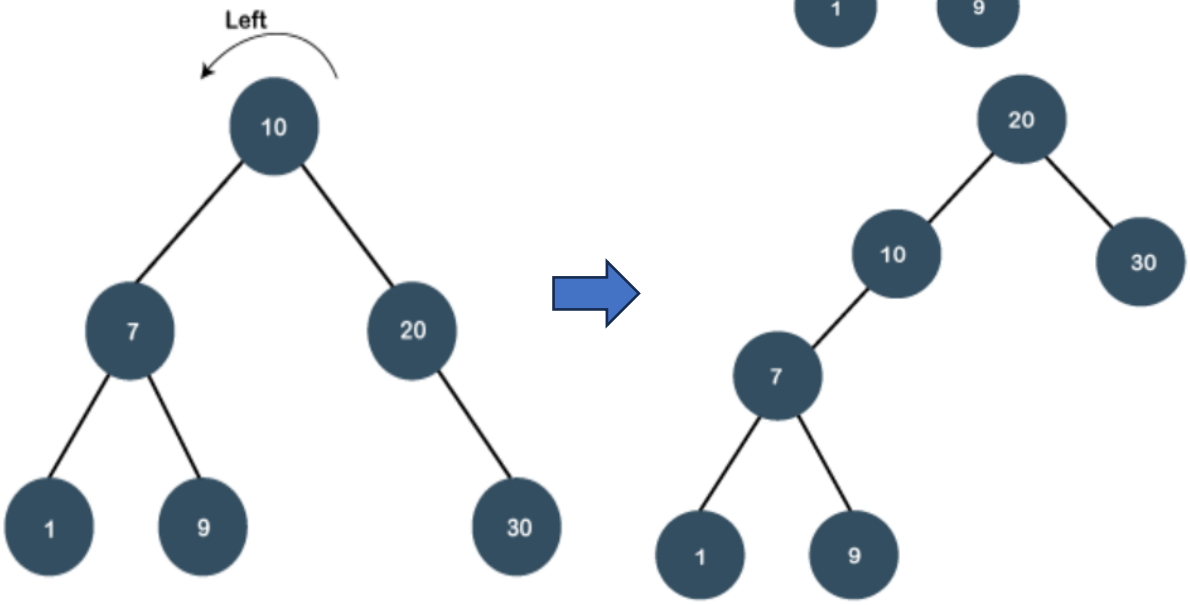



**Let's consider example -** We have to search 20 element in the tree. We will follow the below steps:




**Step 1:** First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.

**Step 2:** Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



# B Tree

The **limitations** of traditional **binary search trees** can be frustrating. Meet the B-Tree, the **multi-talented** data structure that can **handle massive amounts** of data with ease. When it comes to **storing and searching large amounts** of data, traditional binary search trees can become **impractical** due to their **poor performance** and **high memory usage**. B-Trees, also known as **B-Tree or Balanced Tree**, are a type of **self-balancing** tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the **large number of keys** that they can **store in a single node**, which is why they are also known as **"large key" trees.** Each node in a B-Tree can contain **multiple keys**, which allows the tree to have a **larger branching** factor and thus a **shallower height**. This shallow height leads to **less disk I/O**, which results in **faster search and insertion** operations. B-Trees are particularly well **suited for storage** systems that have slow, **bulky data access** such as hard drives, flash memory, and CD-ROMs.

B Tree is a 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**. One of the main reason of using B tree is its capability to store large number of **keys in a single node** and **large key values** by keeping the height of the tree relatively small.

B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the **tree is always balanced**. This balance guarantees that the **time complexity** for operations such as insertion, deletion, and searching is always **O(log n)**, regardless of the initial shape of the tree. A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every **node** in a B-Tree contains **at most m children.**
2. Every **node** in a B-Tree **except the root node** and the **leaf node** contain **at least m/2** children.
3. The **root nodes** must have **at least 2 nodes.**
4. 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.

**Time Complexity of B-Tree:**

| Sr. No. | Algorithm | Time Complexity |
|---|---|---|
| 1. | Search | O(log n) |
| 2. | Insert | O(log n) |
| 3. | Delete | O(log n) |

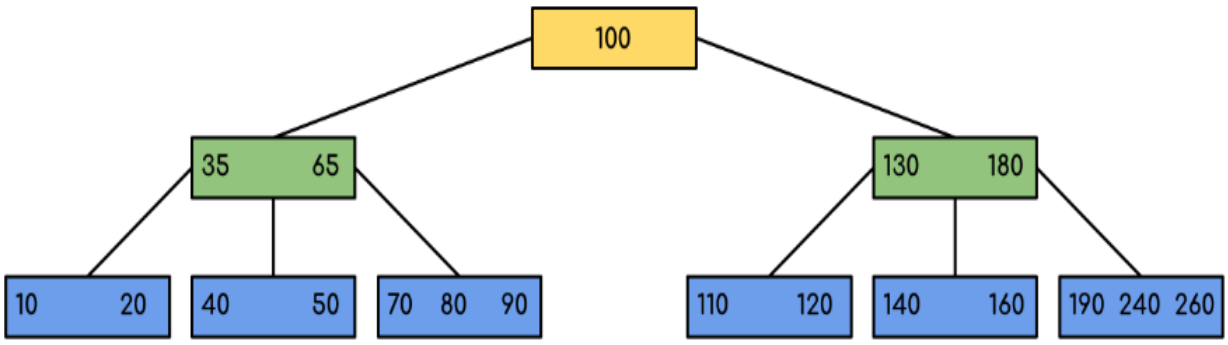
"n" is the total number of elements in the B-tree

**Properties of B-Tree:**

- All leaves are at the **same level.**
- B-Tree is defined by the **term minimum degree** '**m**'. The value of '**m**' depends upon disk block size.

- Every node **except the root** must contain **at least m-1 keys.** The root may contain a minimum of **1** key.
- **All nodes (including root)** may contain **at most (2*m – 1) keys.**
- Number of **children of a node** is equal to the **number of keys** in it **plus 1.**
- All keys of a node are **sorted in increasing order**. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree **grows and shrinks from the root** which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the **time complexity** to search, insert and delete is **O(log n).**
- **Insertion of a Node** in B-Tree happens **only at Leaf Node.**

**Example** of a B-Tree of minimum order 5




**Interesting Facts about B-Trees:**

- The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:

$$h_{min} = \lceil \log_m(n+1) \rceil - 1$$

- The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$$

*and*

$$t = \lceil \frac{m}{2} \rceil$$

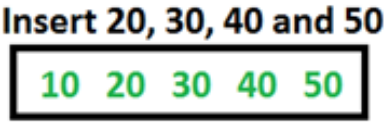
**Inserting data in B-Tree -** Insertions are **done at the leaf node** level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. **Traverse** the B Tree in order to **find the appropriate leaf node** at which the node can be inserted.
2. If the **leaf node contains less than m-1 keys** then **insert the element** in the increasing order.
3. Else, if the **leaf node contains m-1 keys,** then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

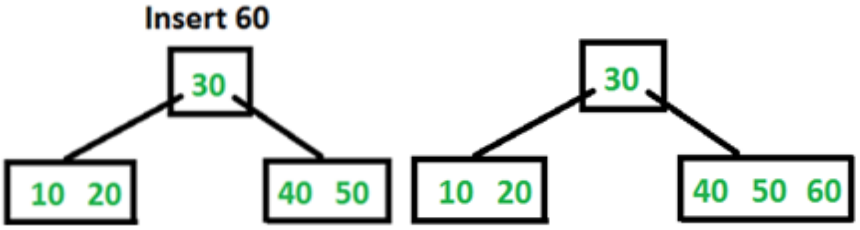
**Example 1 (May have maximum 2m-1 approach) –** Create B tree of minimum degree 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

**Step 1:** Initially root is NULL. Let us first insert 10.

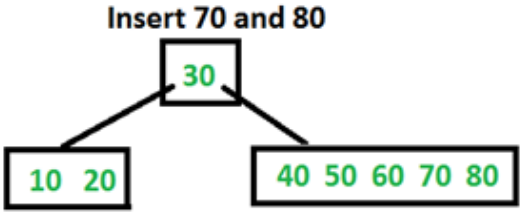

**Step 2:** Now insert 20, 30, 40 and 50. They all will be **inserted in root** because the maximum number of keys a node can accommodate is **2*t–1** which is 5.



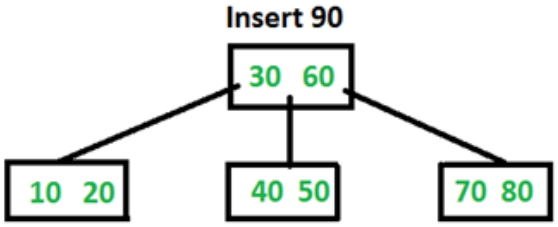

**Step 3:** Now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.




**Step 4:** Now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.




**Step 5:** Now insert 90. This insertion will cause a split. The middle key will go up to the parent.




**Example 2 (Must have maximum m-1 approach) –** Create B tree of minimum degree 4 and a sequence of integers 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 in an initially empty B-Tree.

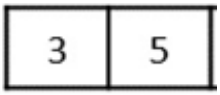
**Step 1 –** Calculate the following

Order M = 4 Maximum keys M–1= 3 Minimum key ([M/2])–1 = 1.
Maximum children = 4 Minimum children ([M/2]) = 2

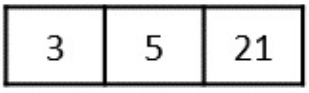
**Step 2 –** The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children

**Inserted 3**

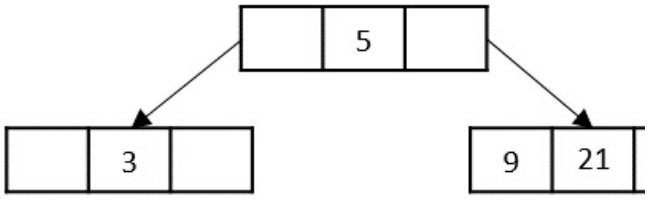

**Inserted 5**



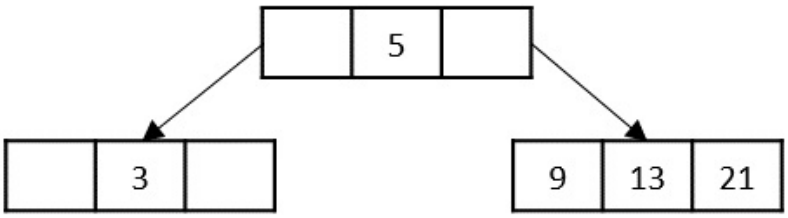

**Inserted 21**



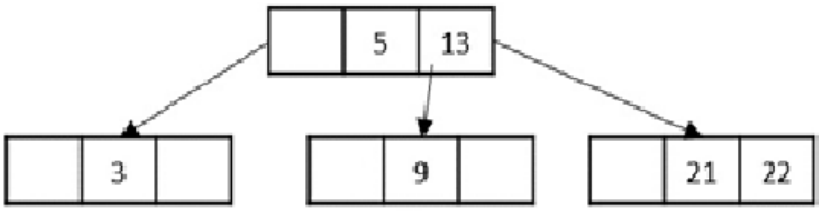

**Inserted 9,** Now adding 9 will cause overflow in the node. Hence, it must be split, because it will violate B tree property.



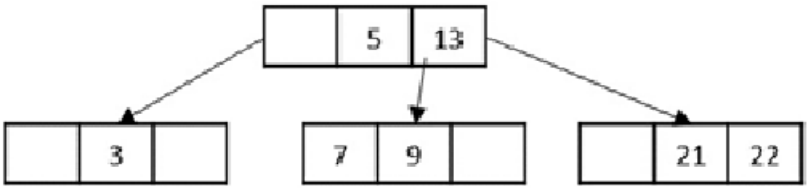
**Inserted 13**



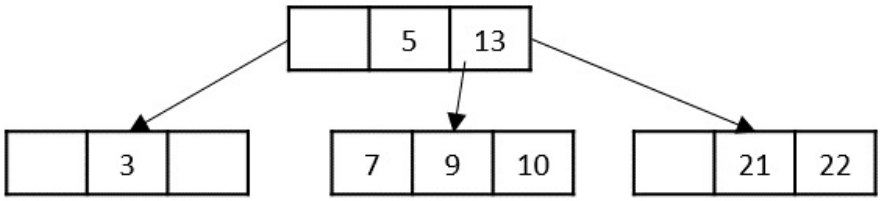

**Inserted 22,** it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.



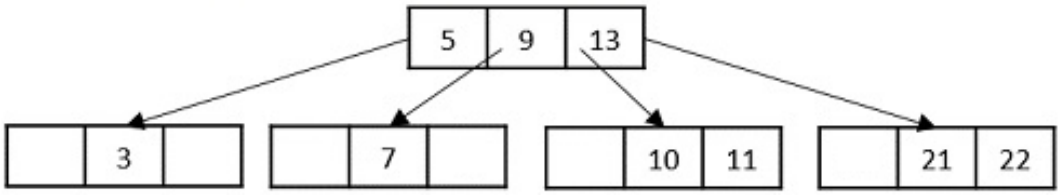

**Inserted 7**



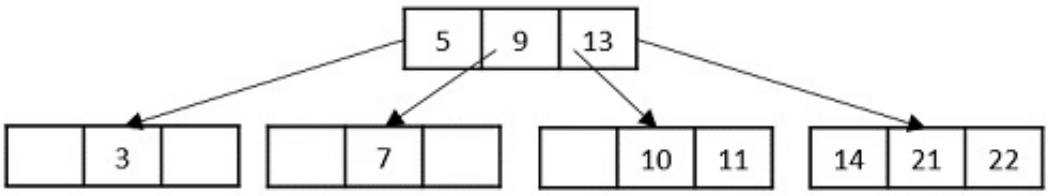

**Inserted 10**



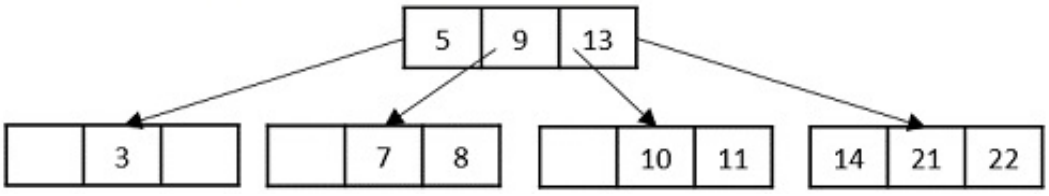

**Inserted 11,** it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.



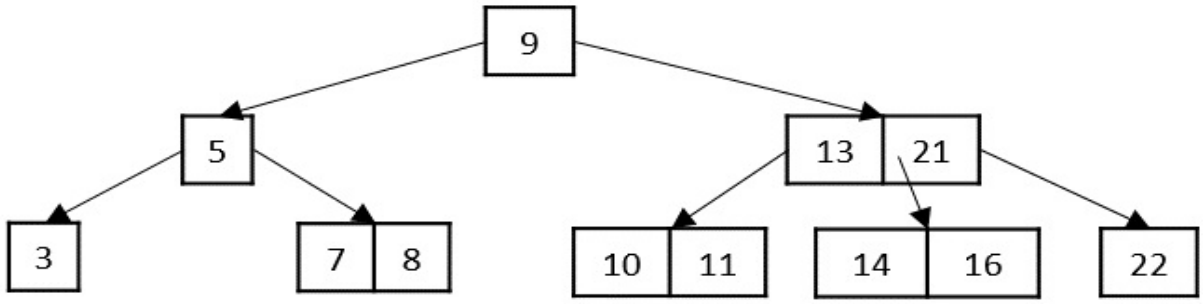

**Inserted 14**




**Inserted 8**



**Inserting 16**, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.



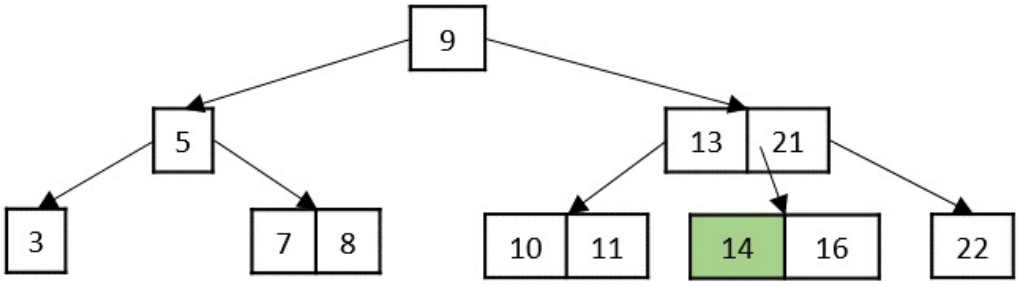

**Deletion data in B-Tree -** Deletion is also **performed at the leaf nodes**. The node which is to be deleted can either be a leaf node or an internal node. Following **algorithm needs** to be followed in order to delete a node from a B tree.

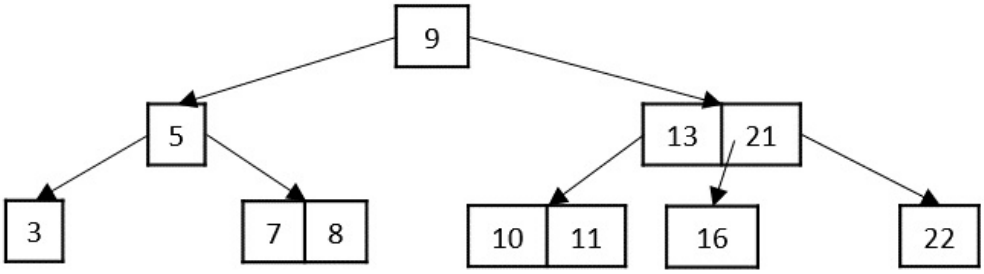
1. Locate the leaf node.
2. If there are more than **m/2 keys** in the leaf node then **delete the desired key** from the node.
3. If the leaf node **doesn't contain m/2 keys** then **complete the keys** by taking the element from right or left sibling.
   - If the **left** sibling contains **more than m/2 elements** then **push its largest** element **up to its parent** and move the intervening element down to the node where the key is deleted.
   - If the **right** sibling contains **more than m/2 elements** then **push its smallest** element **up to the parent** and move intervening element down to the node where the key is deleted.
4. If **neither** of the sibling contain **more than m/2 elements** then **create a new leaf node** by joining two leaf nodes and the intervening element of the parent node.
5. If **parent** is left with **less than m/2 nodes** then, **apply the above process** on the parent too.

If the node which is to be **deleted is an internal node**, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

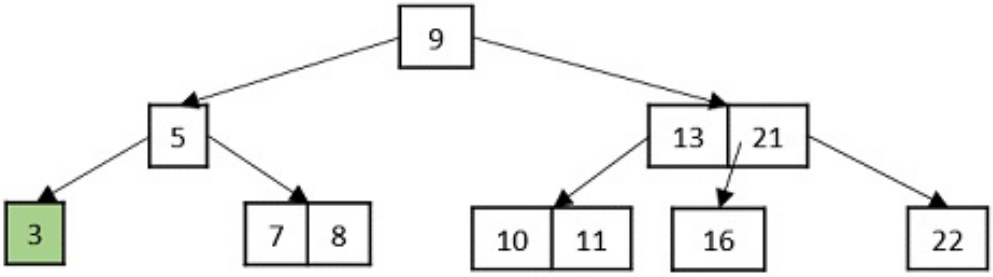
The procedure to delete a node from a B tree is as follows −

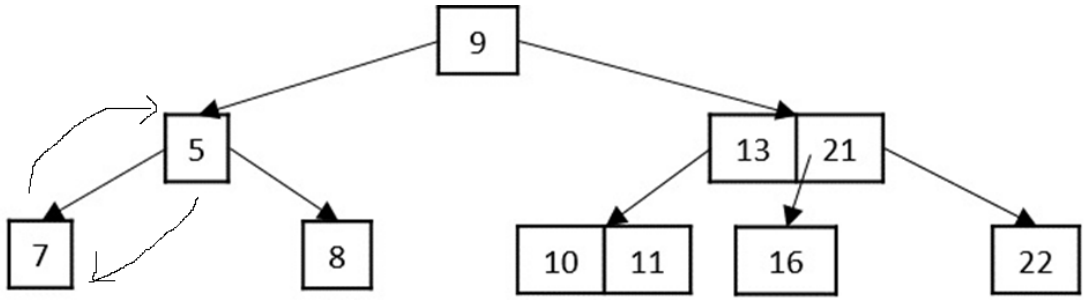
**Case 1** − If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node. **Delete 14 Where m=3**



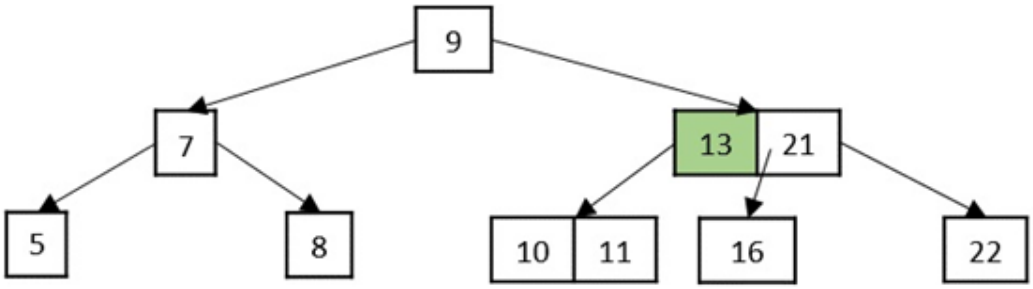


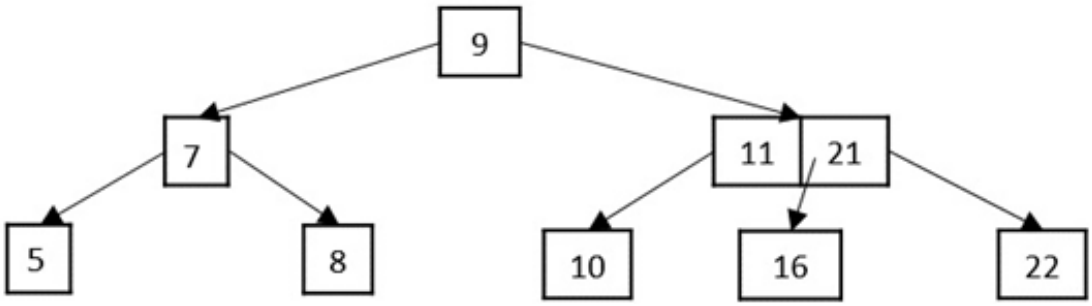

**Case 2** – If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them. **Delete 3 Where m=3**



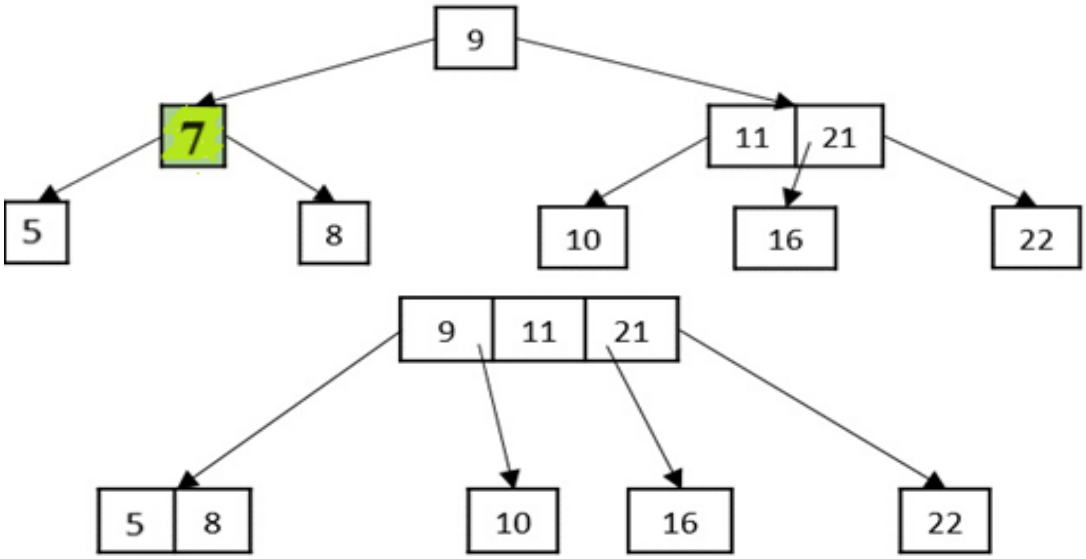




**Case 3** – If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they're merged together. **Delete 13 Where m=3**



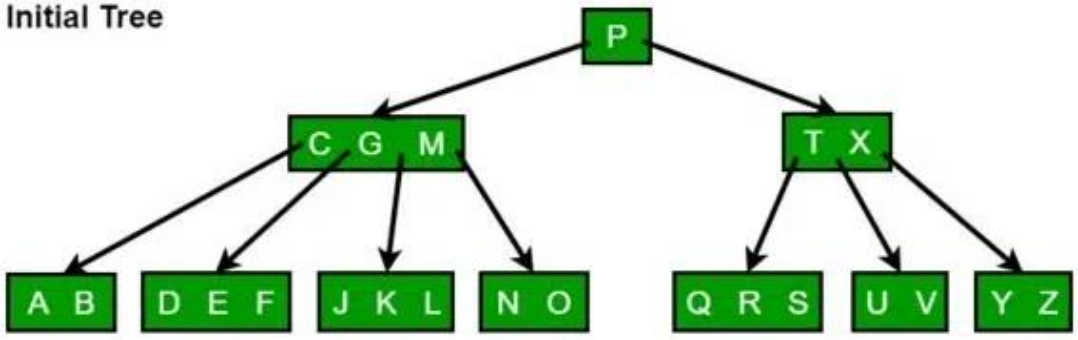



**Case 4** – If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent. **Delete 5**

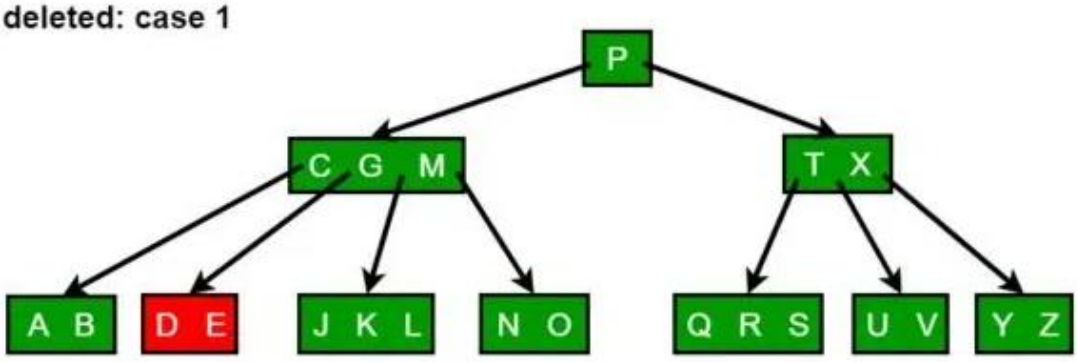



**Example:** The following figures explain the deletion process.

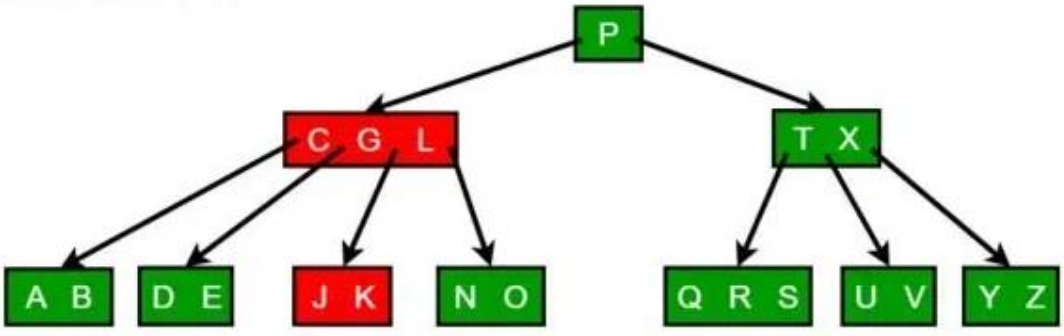
(a) Initial Tree

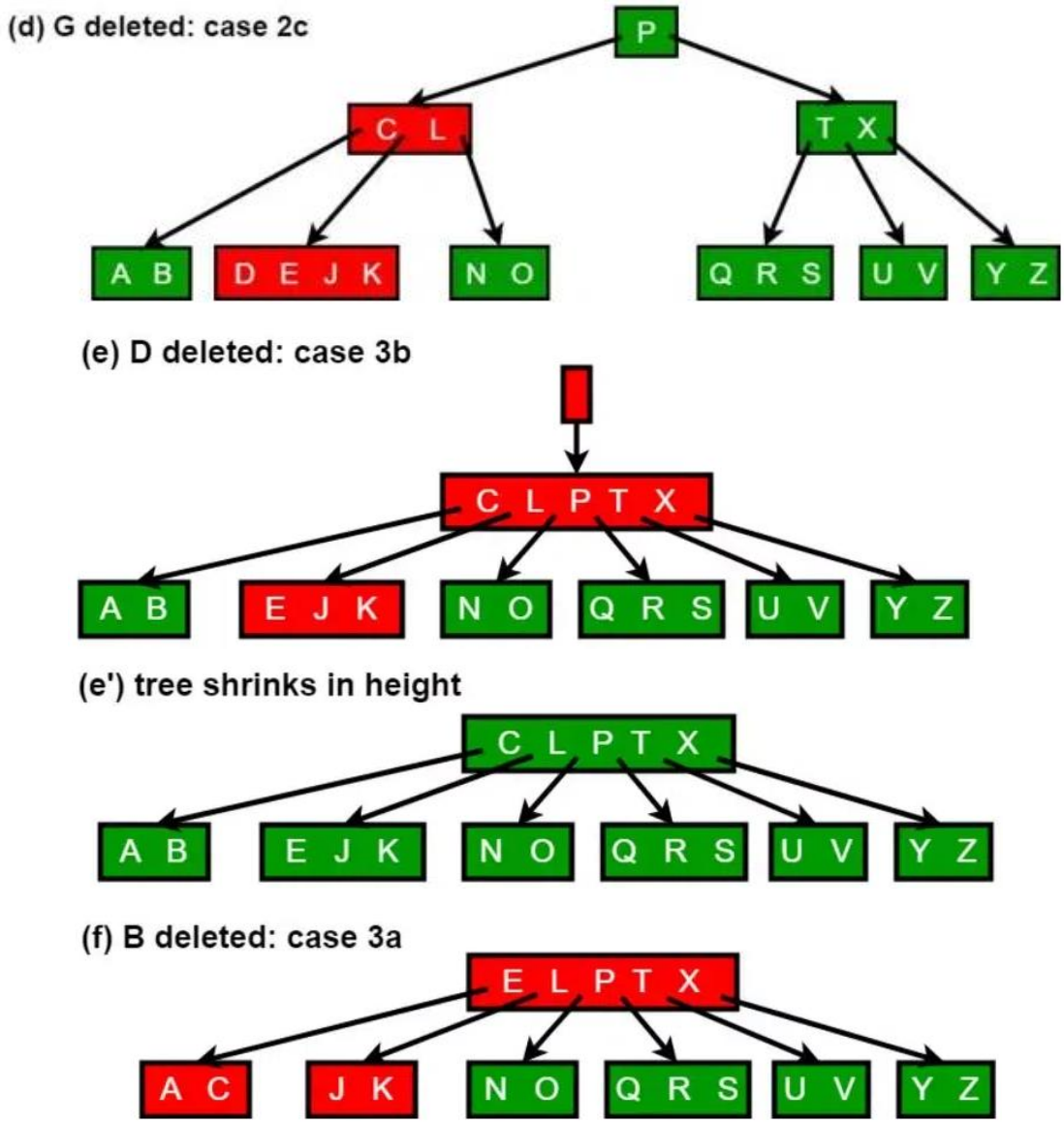



(b) F deleted: case 1




(c) M deleted: case 2a





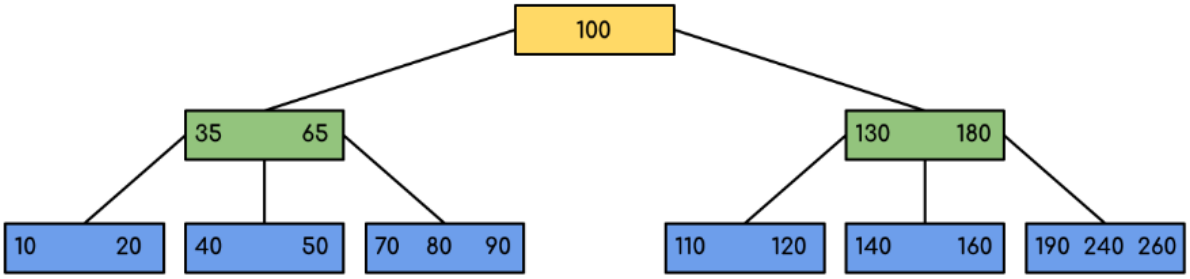

**Traversal in B-Tree:** Traversal is also similar to **inorder traversal of Binary Tree.** We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

**Search Operation in B-Tree:** Search is **similar** to the **search in Binary Search Tree.** Let the key to be searched is k.

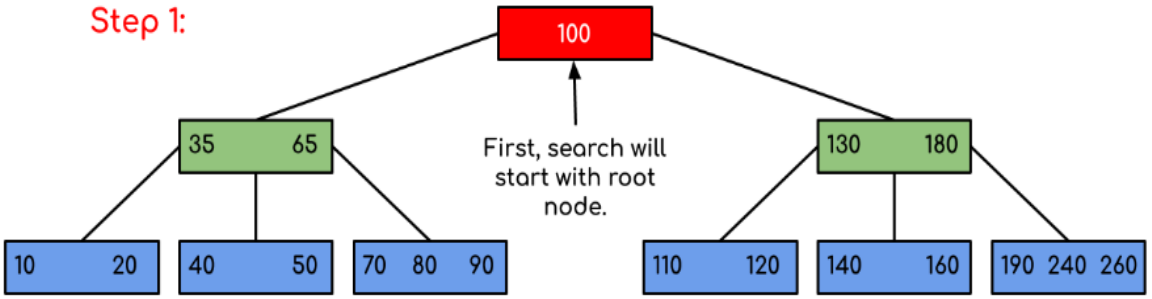
- Start from the root and recursively traverse down.
- For every visited non-leaf node,
- If the node has the key, we simply return the node.
- Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

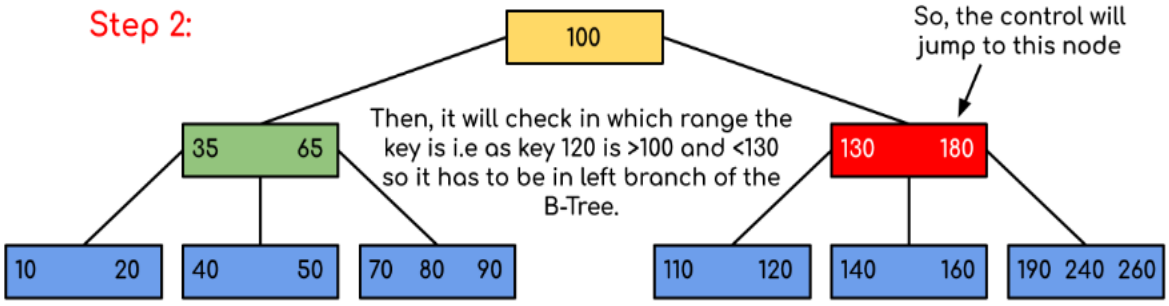
Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

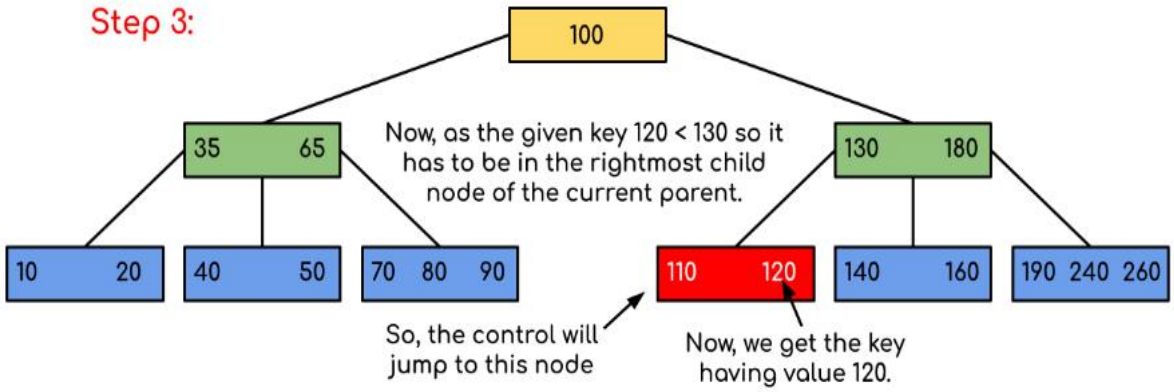
**Example:** Search 120 in B-Tree
**Input:** Search 120 in the given B-Tree.




**Solution:**




**Applications of B-Trees:**

- It is used in **large databases** to access data stored on the disk
- Searching for **data in a data set** can be achieved in significantly less time using the B-Tree
- With the **indexing feature**, multilevel indexing can be achieved.
- Most of the **servers** also use the B-tree approach.
- B-Trees are used in **CAD systems** to organize and search geometric data.

- B-Trees are also used in other areas such as **natural language processing, computer networks, and cryptography.**

**Advantages of B-Trees:**

- B-Trees have a guaranteed **time complexity of O(log n)** for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and **real-time applications.**
- B-Trees are **self-balancing.**
- **High-concurrency** and high-**throughput.**
- **Efficient** storage utilization.

**Disadvantages of B-Trees:**

- B-Trees are based on disk-based data structures and can have a **high disk usage.**
- **Not the best** for all cases.
- **Slow in** comparison to other data structures.

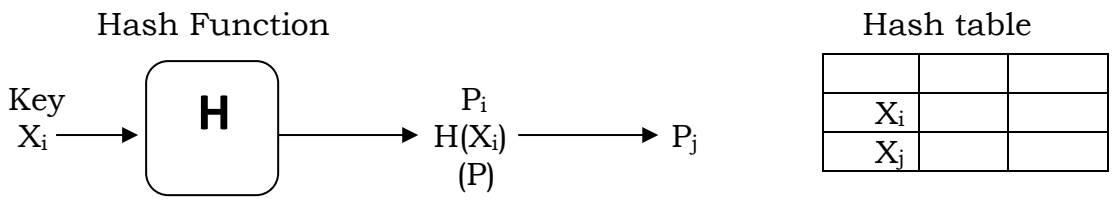
# Hashing: Introduction

Hash Table is a data structure used for **storing and retrieving** data very **quickly. Insertion of data** in file hash table is based on the **key value.** Hence every entry in the hash table is associated with some key. Hash tables are ideal data structures for **dictionaries**.

**Dictionary is a collection of data elements** uniquely identified by a field **called key**. A dictionary supports the operations of search, insert and delete. The ADT of a dictionary is defined as a set of elements with distinct keys supporting the operations of search, insert, delete and create. While most dictionaries deal with distinct keyed elements, it is not uncommon to find applications calling for dictionaries with duplicate or repeated keys. A **dictionary supports** both **sequential and random access.** A sequential access is one in which the data elements of the dictionary are ordered and accessed according to the order of the keys (ascending or descending, for example). A random access is one in which the data elements of the dictionary are not accessed according to a particular order.

A **hash function H(X)** is a mathematical function which given **a key X** of the **dictionary D maps** it to **position P in a storage table termed as Hash Table.** The **process of mapping** the keys to their respective position in the hash table is **called Hashing**. A hash function is denoted as

**Figure 1 Hash Key**




**For example** for storing an employee record in the hash table the employee ID will work as a key. Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table. When the data element of the dictionary are to be stored in the hash table, each key $X_i$ is mapped to a position $P_i$ in the hash table as determined by the value $H(X_i)$, i.e. $P_i = H(X_i)$. To search for a key X in the hash table all that one is to

be determine the position P by computing $P_i = H(X_i)$ and access the appropriate data element. In case of insertion of key X or in deletion, the position P in the hash table where the data element needs to be inserted or from where it is to be deleted, respectively is determined by computing P=H(X). The hash table accommodating the data elements appears as shown in fig. 2. The hash function yields distinct values for the individual keys. If this were to be followed as a criterion, then the situation may turn out of control since in the case of dictionaries with very large set of data elements, the hash table size can be **too huge to be handled efficiently.**

| | | |
|---|---|---|
| 1 | AB12 | ..... |
| 2 | ... | |
| 3 | CG45 | |
| 4 | ... | |
| 5 | EX44 | ..... |
| ... | ... | |
| 11 | KL78 | |
| ... | | |
| 15 | OW31 | ..... |
| ... | | |
| 18 | RK32 | |
| 19 | ST65 | ..... |
| ... | | |
| 22 | VP99 | ..... |
| ... | ... | ..... |

**Figure 2 Hash Table**

Therefore it is **convenient to choose hash functions** which yield values lying within a limited range so as to restrict the length of the table. This would consequently imply that the hash functions may yield identical values for a set of keys, in other words, a set of keys could be mapped to the same position in the hash table. Let X1, X2, ..... Xn, be the n keys which are mapped to the same position P in the hash table. Then H(X1) = H(X2) = H (Xn) = P. In such a case, X1, X2, .....Xn are **called as synonyms**. The act of two or more synonyms vying for the same position in the hash table is **known as collision**.

The **choice of the hash function** plays a **significant role** in the structure and performance of the hash table. It is therefore essential that a **hash function satisfies** the following characteristics:

I. Easy and quick to compute
II. Even distribution of keys across the hash table. In other words, a hash function must minimize collisions.

## Hash Function Methods

**Hash function is a function** which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table. The integer returned by the hash function is **called hash key**.

**For example**: Consider that we want place some employee records in the hash table. The record of employee is placed with the help of key : employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record, the key 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at $0^{th}$ position. The second key is 8421002, the record of this key is placed at $2^{nd}$ position in the array. Hence the hash function will be

**H (key) = key % 1000.**

Where **key % 1000** is a hash function and key obtained by hash function is **called hash key**. The hash table will be –

| | Employee ID | Record |
|---|---|---|
| 0 | 4968000 | |
| 1 | | |
| 2 | 7421002 | |
| | | |

**Figure 3 Hash Table**

**Bucket and Home bucket**: The hash function H(key) is used to map several dictionary entries in the hash table. **Each position** of the **hash table** is **called bucket** and the **current position** where we are pointing the value is **home bucket.** The function H(key) is home bucket for the dictionary with pair whose value is key.

**Types of Hash Function**

There are various types of hash functions that are used to place the record in the hash table —

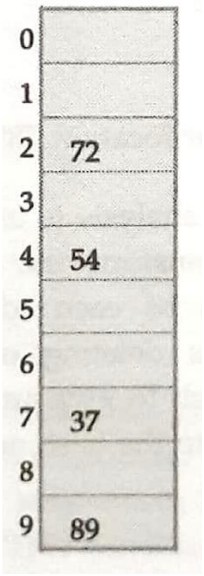



**Figure 4**

**1. Division Method:** The hash function **depends upon the remainder** of division. Typically the **divisor is table length.** For example :- If the record 54, 72, 89, 37 is to be placed in the hash table and if the table size is 10 then

H(key) = record % table size
= 54 % 10
= 4
Similarly for all values hash function is

H(key) = record % table size
=72%10 =89%10 =37%10
=2 =9 =7

**2. Mid Square:** In the mid square method the key is **squared and the middle or mid part** of the result is **used as the index.** If the key is a string it has to be pre-processed to produce a number. Consider that if we want to place a record 3111 then

$(3111)^2 = 9678321$
For the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

**3. Multiplicative hash function:** The given record is **multiplied by some constant** value. The formula for computing the hash key is -

**H(key)=Floor (P * (fractional part of key * A))**

where p is integer constant and A is constant real number. Donald Knuth suggested to use constant A =0.61803398987

If key 107 and p=50 then
H(key) = floor (50*(107*0.61803398987)
= floor (3306.4818458045)
= 3306

At 3306 location in the hash table the record 107 will be placed.

**4. Digit folding:** The **key is divided** into **separate parts** and using some simple operation these **parts are combined** to produce the hash key. For example, consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12
= 789

The record will be placed at location 789 in the hash table.

6. **Digit Analysis:** The digit analysis is **used in a situation** when all the **identifiers are known in advance**. We first **transform** the identifiers into numbers **using some radix**. Then we examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Thus these digits are used to calculate the hash address.

**Example -** If our inputs were 1938m, 3391i, 3091b, 4903a, 4930a, 6573b, and 4891c,

**Solution:**
**Analyze the digits :** There are four 1s, six 9s, seven 3s, two 8s, one m, one i, three 0s, two bs, three 4s, two as, one 6, one 5, one 7, and one c.

**Eliminate some data :** 1, 8, m, i, 0, b, 4, a, 6, 5, 7 and c from the inputs to produce 93, 339, 39, 93, 93, 3, 9.

**We only want a one-character hash :** So, we take only the first character: 9, 3, 3, 9, 9, 3, 9. We now have a hash function that hashes these seven identifiers more-or-less evenly into one of two buckets.

**Characteristics of Good Hashing function -**

I. The hash function should be **simple to compute.**
II. Number of **collisions should be less** while placing the record in the hash table.
III. Hash functions should produce such a keys (buckets) which will get **distributed uniformly** over an array.
IV. The hash function should depend upon **every bit of the key**. Thus then hash function that simply extract the position of a key is not suitable

## Collision Resolution

**Collision -** The situation in which the hashing returns the same hash key for more then -one record is **called collision**. The hash function is a function that **returns the key** value using which the **record can be placed** in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from the location. This function needs to be **designed very carefully** and it should not return the hash key address for two different records. This is undesirable situation in hashing.

Similarly when there is **no room for a new pair** in the hash table then such a situation is called **overflow**. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions. For example : Consider a hash function.

H(key) = record key %10 having the hash table of size 10. The record keys to be placed are **131, 44, 43, 78, 19, 36, 57 and 77**

Now if we try to place 77 in the hash table than we get the hash key to be 7 and at index 7 already the record key 57 is place. **This situation is called collision**. From the index 7 if we look for next vacant passion at subsequent indices 8,9 then we find that there

is no room to place 77 in the hash table. **This situation is called overflow**. If **collision occurs** then it should be **handled by applying** some techniques. Such a technique is called **collision handling technique.** There are some methods for detecting collisions and overflows in the hash table. These are

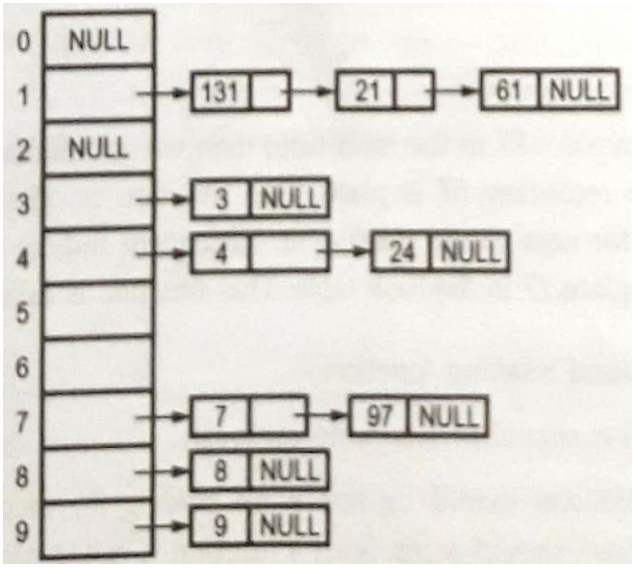



**Figure 2 Hash Table**

**1. Chaining**
**2. Open addressing (linear probing)**
**3. Quadratic probing.**
**4. Double hashing.**

**1. Chaining -** In collision **handling method** chaining is a concept which **introduces an additional field with data** i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list (chain) is maintained at the home bucket. For example:

Consider the keys to be placed in their home buckets are **131, 3, 4, 21, 61, 24, 7, 97, 8, 9**

Then we will apply a hash function as **H(key) = key % D** where D is the size of table. The hash table will be - **Here D = 10.**

A chain is maintained for colliding elements. For instance 131 has a home bucket (key) 1. Similarly keys 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1. Simi1arly the chain at index 4 and 7 is maintained.

**2. Open Addressing or Linear Probing-** This is the easiest method of handling collision. When collision occurs i.e. **when two records demand for the same home bucket** in the hash table then **collision can be solved by placing the second record linearly down** whenever the **empty bucket is found.** When use linear probing (open addressing), the hash table is represented as a one dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we insert elements into the table. Then using some suitable hash function the element can be inserted into the hash table. For example :

Consider that following keys are to be inserted in the hash table. **131, 4, 8, 7, 21, 5, 31, 61, 9, 29.**

We will use Division hash function, That means the keys are placed using the formula

**H(key) = key % table size**
**H(key) = key % 10** here size of array is 10

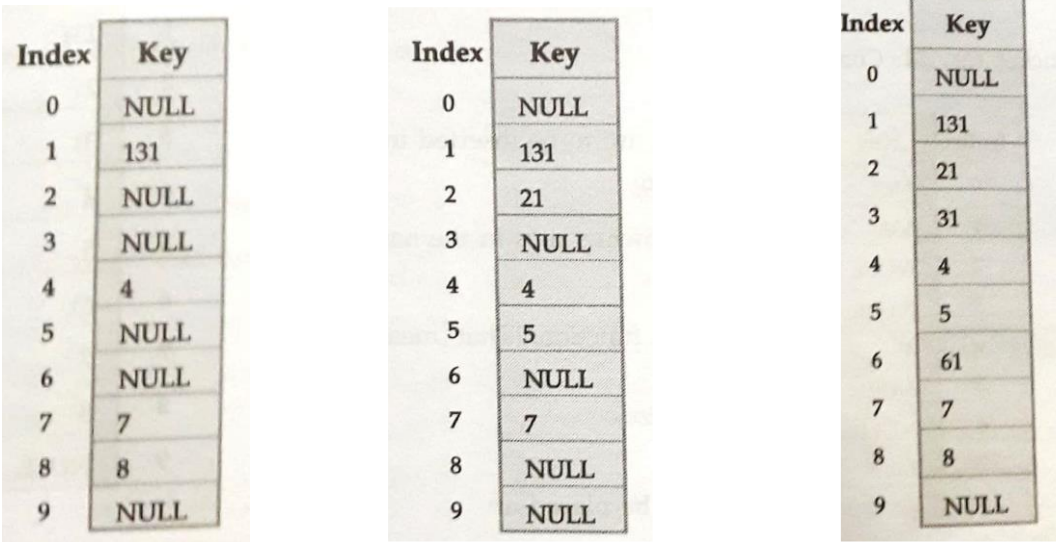
For instance the element 131 can be placed at **H (K) = 131%10 = 1**

Index 1 will be the home bucket for 131. Continuing in this fashion we will place Initially, the following keys in the hash table. **131, 4, 8, 7** (shown in fig. a)**.**

Now the next key to be inserted is 21. According to the hash function

H(key) = 21%10
H(Key) = 1.

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2 (shown in fig. b). If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5 (shown in fig. b).



| Index | Key |
|---|---|
| 0 | NULL |
| 1 | 131 |
| 2 | NULL |
| 3 | NULL |
| 4 | 4 |
| 5 | NULL |
| 6 | NULL |
| 7 | 7 |
| 8 | 8 |
| 9 | NULL |

a.

| Index | Key |
|---|---|
| 0 | NULL |
| 1 | 131 |
| 2 | 21 |
| 3 | NULL |
| 4 | 4 |
| 5 | 5 |
| 6 | NULL |
| 7 | 7 |
| 8 | NULL |
| 9 | NULL |

b.

| Index | Key |
|---|---|
| 0 | NULL |
| 1 | 131 |
| 2 | 21 |
| 3 | 31 |
| 4 | 4 |
| 5 | 5 |
| 6 | 61 |
| 7 | 7 |
| 8 | 8 |
| 9 | NULL |

c.

**Figure 6**

After placing record keys 31, 61 the hash table will be like, shown in fig. c. The next record key that comes is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key is 29 audit hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0 index.

**Problem with linear probing - One major problem** with linear probing **is primary clustering**. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved. For example :

19%10 = 9
18%10 = 8
39%10 = 9
39%10 = 9 and 8% 10 = 8

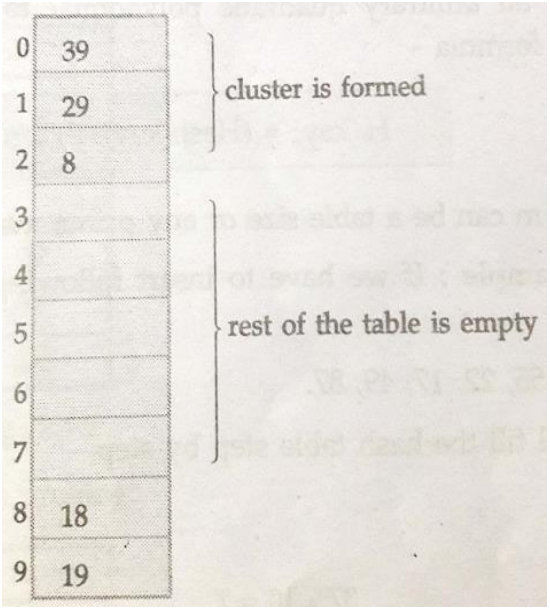



**Figure 7**

This **clustering problem** can be **solved by quadratic probing**.

Performance Analysis of Linear Probing Let,
**n** be the total number element in the hash table.
**b** is the total number of buckets.
**D** is the divisor, used as hash function such as b = D.

Then the worst-case time complexity of insert and find operations is θ(n). For average case analysis, consider successful and unsuccessful search.

Let $\mathbf{S_n}$ be the successful search and $\mathbf{U_n}$ an be the unsuccessful search, for linear probing

$$S_n \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

$$U_n \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

Where α(alpha) = n/b be the loading factor

**3. Quadratic probing -** Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula $H_i(key) = (Hash(key) + i^2) \% m$

Where m can be a table size or any prime number. **For example:** If we have to insert following elements in the hash table with **table size 10**: 37, 90. 55, 22. 17, 49, 87. We will fill the hash table step by step

**37%10=7**
**90%10=0**
**55%10=5**
**22%10=2**
**11%10=1**

Now if we want to place 17 a collision will occur as 17%10=7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

H(K) = (Hash (K) + $i^2$) % m  Consider i = 0 then
**$(17 + 0^2) \% 10 = 7$**
**$(17+1^2) \% 10 = 8$ , When i = 1**

The bucket 8 is empty hence we will place the element at index 8. Then comes 49 which will be placed at index 9 -> **49 % 10 = 9**

Now to place 87 we will use quadratic probing.

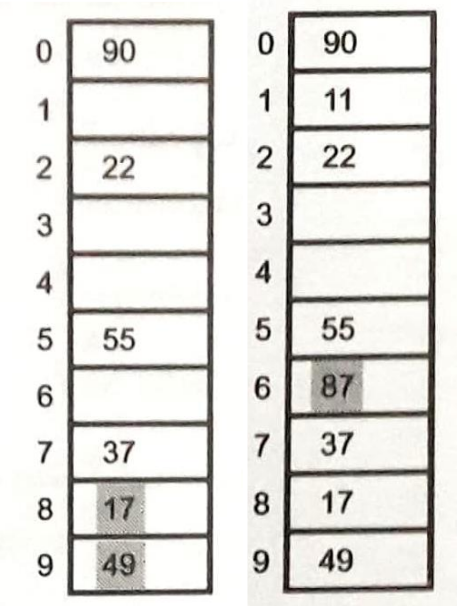



**Figure 8** **a)** **b)**

**$(87 + 0) \% 10 = 7$**
**$(87 + 1^2) \% 10 = 8$ .....but already occupied**
**$(87 + 2^2) \% 10 = 1$ ..... already occupied**
**$(87 + 3^2) \% 10 = 6$**

It is observed that if we want to place all the necessary elements in the hash table the size of divisor (m) should be - twice as large as total number of elements.

**4. Double Hashing -** Double hashing is technique in which a s**econd hash function** is applied to the key **when a collision occurs**. By applying the second hash function we will get the **number of positions from the point of collision** to insert. There are two important rules to be followed for the second function:

1. It must never evaluate to zero.
2. Must make sure that all cells can be probed.

The formula to be used for double hashing is

**$H_1(key)$ = key mod table size**

**$H_2$(key) = M - (key mod M)**

Where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10: **37, 90, 45, 22, 17, 49, 55**

Initially insert the elements using the formula for $H_1$(key). Insert 37, 90, 45, 22.

**$H_1$(37) = 37 % 10 = 7**
**$H_1$(90) = 90 % 10 = 0**
**$H_1$(45) = 45 % 10 = 5**
**$H_1$(22) = 22 % 10 = 2**
**$H_1$(49) = 49 % 10 = 9**

Now if 17 is be inserted then
$H_1$(17) = 17%10 = 7
$H_2$(key) = M - (key mod M)

Here M is a prime number smaller than the size of the table Prime number smaller than table size 10 is 7

Hence M = 7
$H_1$(17) = 7 - (17 % 7) = 7 – 3 = 4

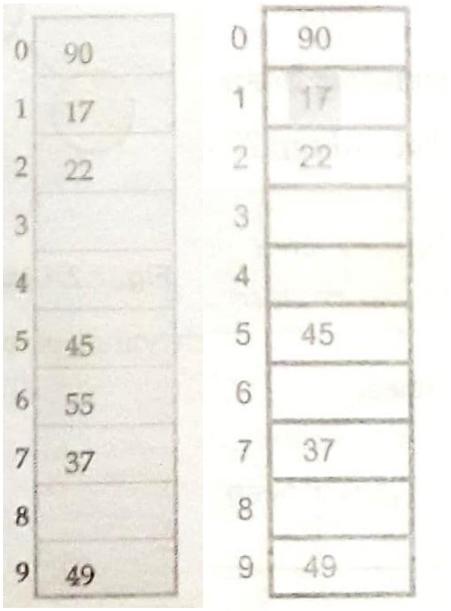



**Figure 9** **a.** **b.**

That means we have to insert the element 17 at 4 places from 37. In short we have to take 4 jumps. Therefore the 17 will be placed at index 1 (shown in fig. a).

Now to insert number 55

$H_1$(55) = 55 % 10 = 5 i.e. collision
$H_1$(55) = 7- (55 % 7) = 7 – 6 = 1

That means we have to take one jump from index 5 to place 55. Finally the hash table will be (shown in fig. b)