7

Search Trees (Part II)

7.1 Introduction

In previous chapter we have learnt two balanced tree structures : binary search trees and AVL trees. These tree structures are suitable for internal memory applications. In this chapter we will discuss one more balanced tree called Red-Black tree which is also suitable for internal memory applications. This chapter involves study of Red-Black tree, Splay tree and B-tree. The splay tree is a data structure which gives the most efficient performance to access the most recently accessed node. The B-tree is a tree structure suitable for external memory applications.

7.2 Red-Black tree

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity. The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. **Root Property** : The root is black
2. **External Property** : Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. **Internal Property** : The children of a red node are black. Hence possible parent of red node is a black node.
4. **Depth Property** : All the leaves have the same black depth
5. **Path property** : Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above mentioned properties is that the Red-Black is tree is roughly balanced.

7.2.1 Representation

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

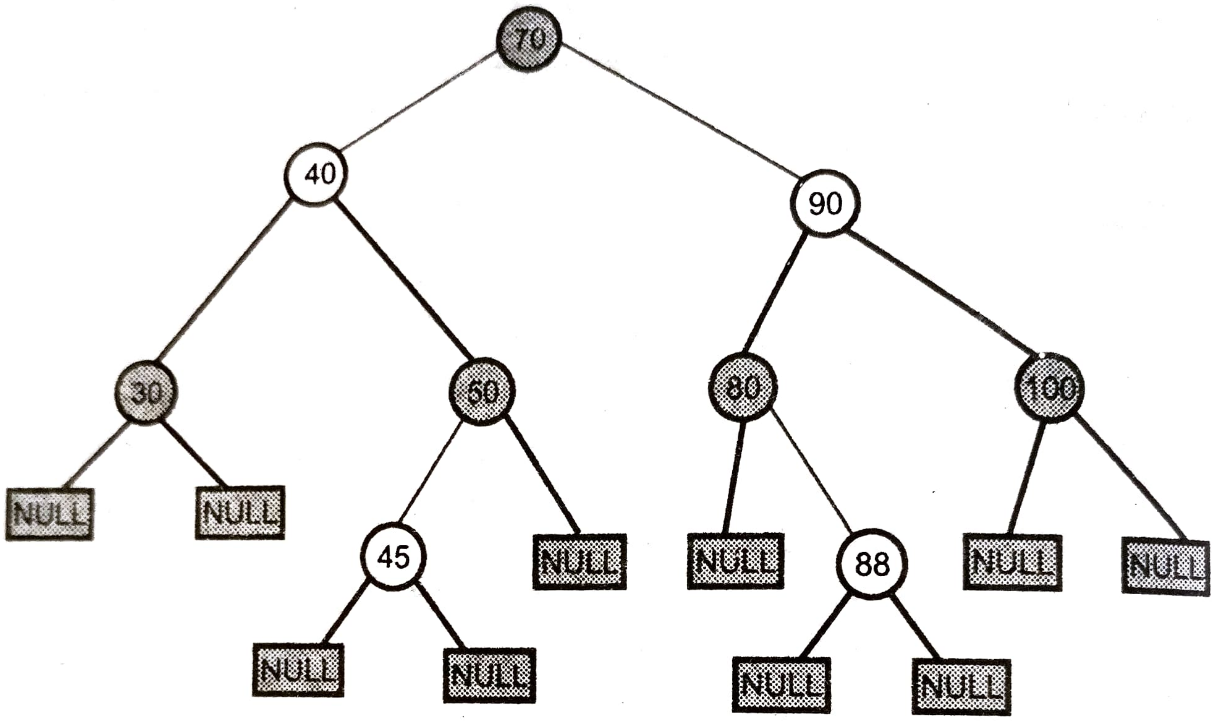


Fig. 7.1 Red-Black tree

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 color of nodes. The above given Red-Black tree follows all the properties of Red-Black tree -

1. It is a binary search tree.
2. The root node is black.
3. The children of red node are black.
4. No root - to external node path has two consecutive red nodes (e.g. 70-90-80-88-NULL).
5. All the root to external node paths contain same number of black nodes (including root and external node).

For e.g. : Consider path 70-40-30-NULL and 70-90-80-88-NULL in both these paths 3 black nodes are there. Similarly other paths can be checked.

7.2.2 Insertion

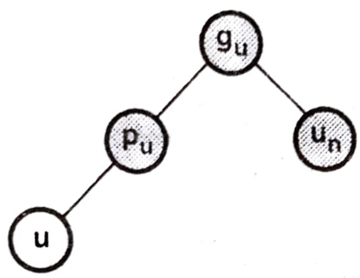
- Every new node which is to be inserted is marked Red.
- Not every insertion causes imbalancing but if imbalancing occurs then that can be removed depending upon the configuration of tree before new insertion made.
- To understand insertion operation, let us understand the configuration of tree by defining 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 p_u.

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.

- When insertion occurs, the new node is inserted in already balanced tree. If this insertion causes any imbalancing then balancing of the tree is to be done at two levels :

  - at grandparent level i.e. g_u
  - at parent level i.e. p_u.

- The imbalancing is concerned with the color of grandparent's child (i.e. uncle node). If uncle node is red then there are four cases

  1. LR_r
  2. LL_r
  3. RR_r
  4. RL_r

1. LR_r imbalance - The left child of g_u is p_u and u is right child of p_u and u_n node (uncle node) is red.

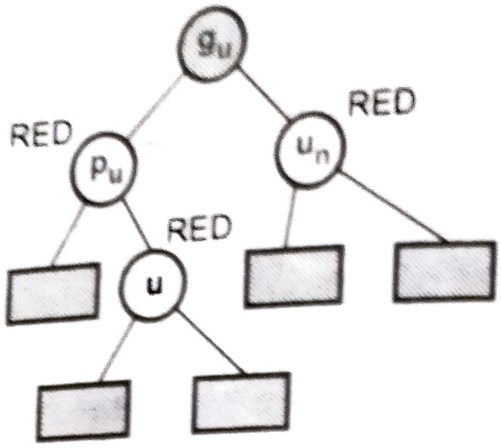


Fig. 7.2 LR_r imbalance

2. LL_r imbalance - The node p_u is a left child of g_u and u is inserted as left child of p_u. Node u_n is red.

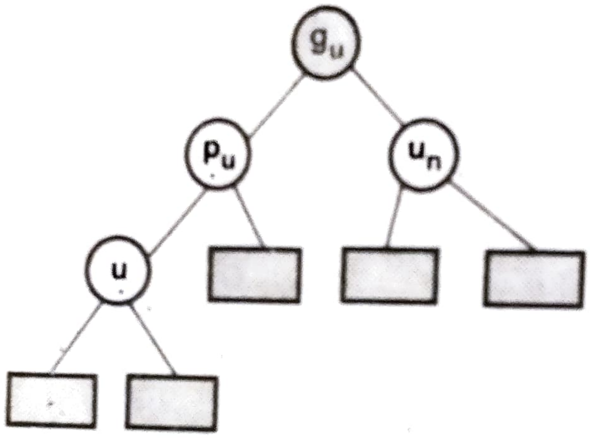


Fig. 7.3 LL_r imbalance

3. RR_r **imbalance** - The right child of node g_u is node p_u and u is inserted as right child of p_u. The u_n node is red.

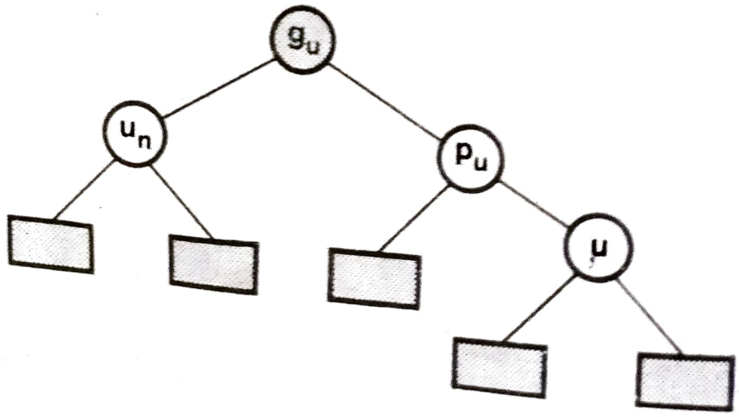


Fig. 7.4 RR_r imbalance

4. RL_r **imbalance** - The node p_u is right child of g_u and u is inserted as a left child of p_u. The uncle node u_n is red.

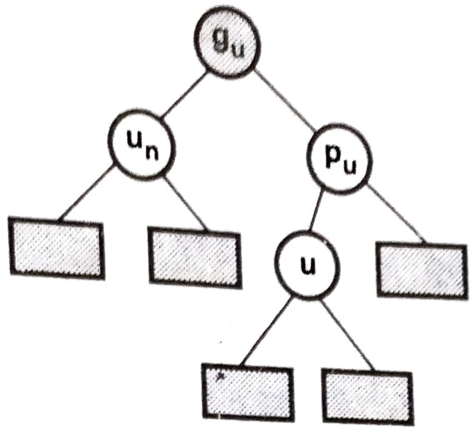
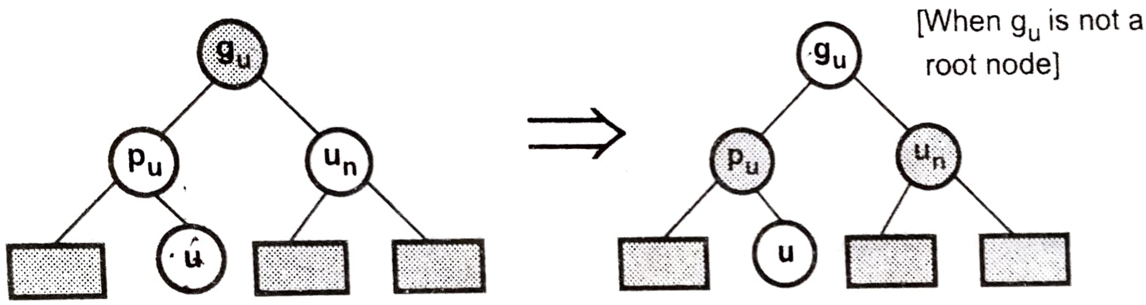


Fig. 7.5 RL_r imbalance

- To remove these imbalancing rotations are not required. Simply by changing the colors required balancing can obtained.

1. Removal of LR_r imbalancing

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



Before After

Fig. 7.6 Removal of LR_r imbalancing

1. Change color of p_u from red to balck.
2. Change color of u_n from red to black.
3. Change color of g_u from black to red provided g_u is not a root node.

2. Removal of LL_r imbalancing

Removal of LL_r imbalance

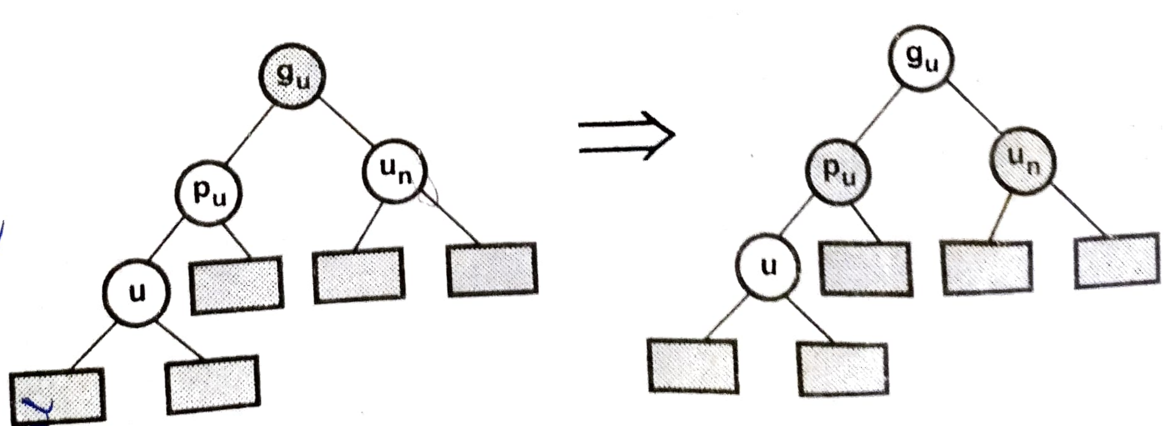


Fig. 7.7 Removal of LL_r imbalancing

1. Change the color of p_u from red to black.

2. Change color of u_n from red to black.

3. Change color of g_u from black to red when g_u is not a root node.

3. Removal of RR_r imbalancing

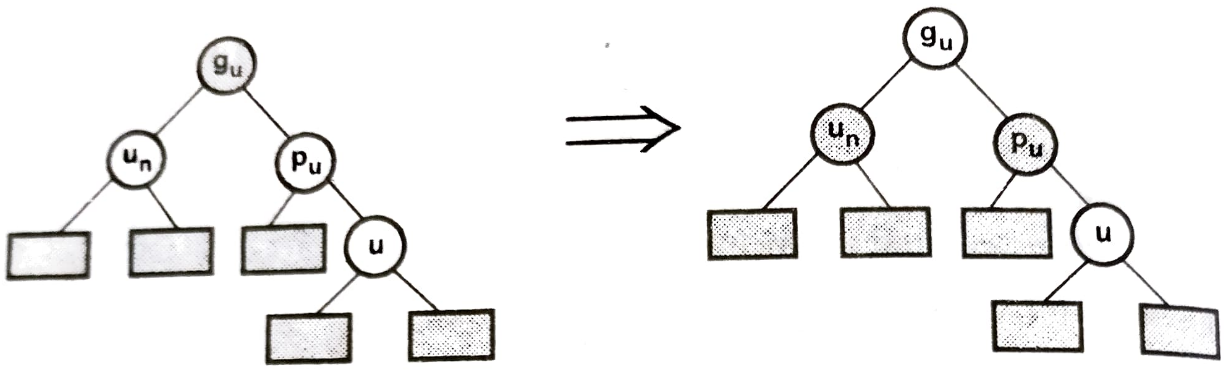


Fig. 7.8 Removal of RRr imbalancing

1. Change color of p_u from red to black.

2. Change color of u_n from red to black.

3. Change the color of g_u from black to red provided that g_u is not the root of the tree.

4. Removal of RL_r imbalancing

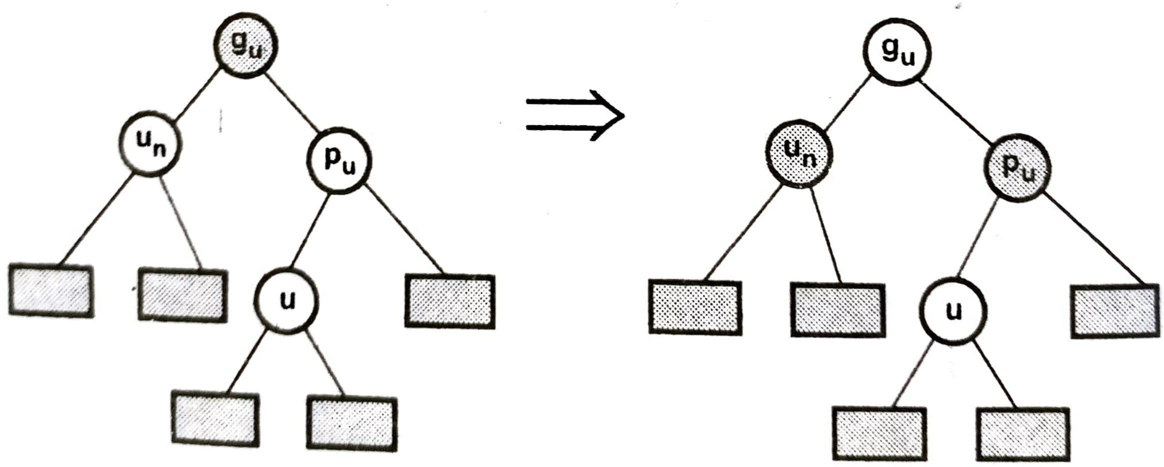


Fig. 7.9 Removal of RL_r imbalancing

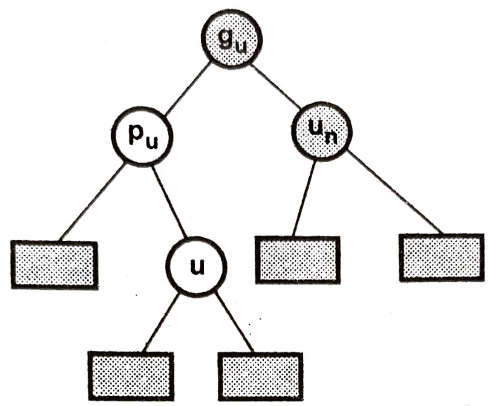
1. Change the color of p_u from Red to black.

2. Change the color of u_n from red to black.

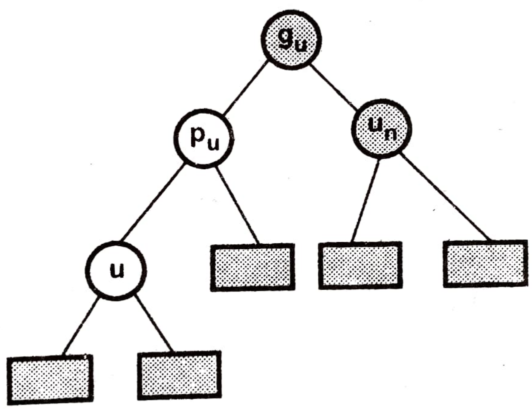
3. Change the color of g_u from black to red provided that g_u is not the root of the tree.

- Now when other child of g_u i.e. uncle node u_n is black then there arises four cases.

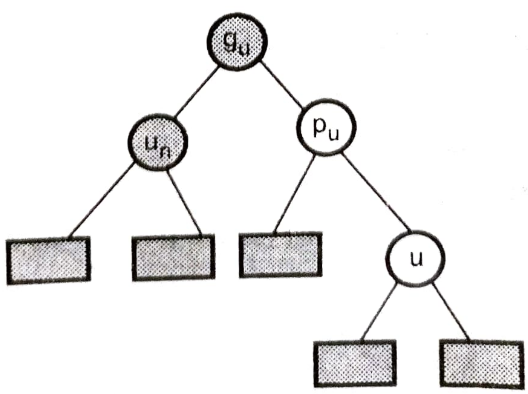
1. **LR_b imbalancing :** The p_u node is attached as a left child of g_u and u is inserted as a right child of p_u. The node u_n is black.



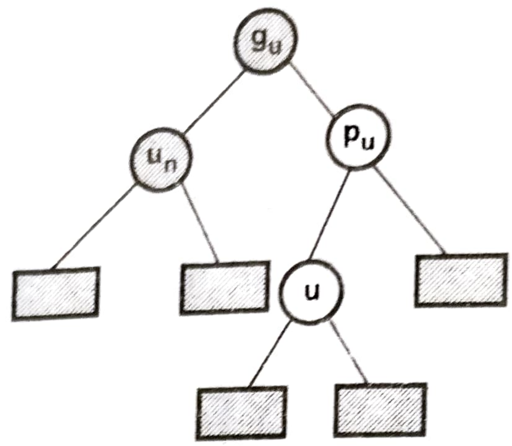
2. **LL_b imbalancing :** The node p_u is a left child of g_u and u node is a left child of p_u. The node u_n is black.



3. **RR_b imbalancing :** The node p_u is a right child of node g_u and node u is a right child of p_u. The node u_n is black.



4. **RL_b imbalancing :** The node p_u is a right child of g_u and u node is attached as a left child of p_u. The uncle node u_n is black.



As u node gets inserted rebalancing must be performed.

- LL_b and RR_b cases require single rotation followed by recoloring.
- LR_b and RL_b cases require double rotation followed by recoloring.

- **Removing LL_b and RR_b imbalances**

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

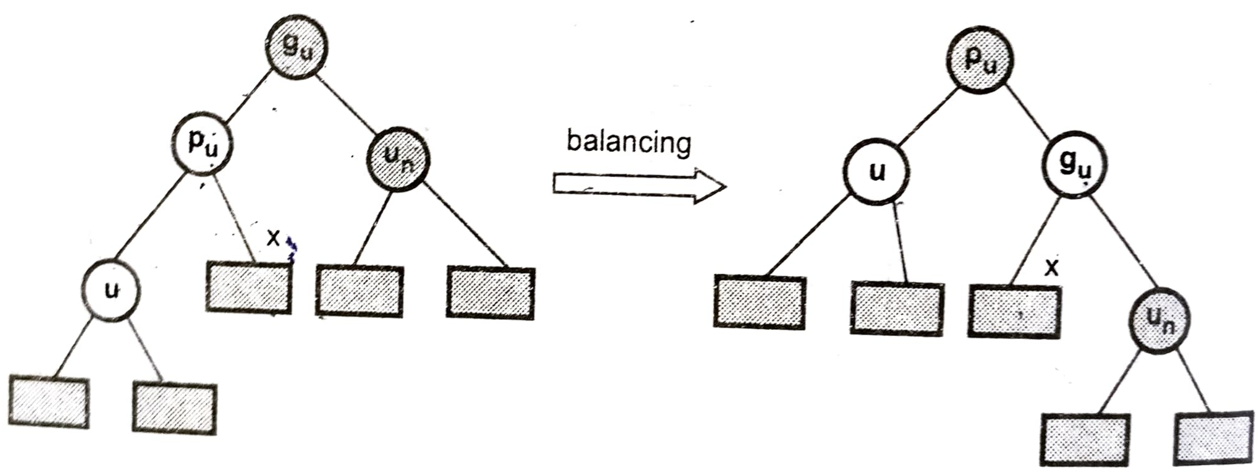


Fig. 7.10 Removal of LL_b imbalancing

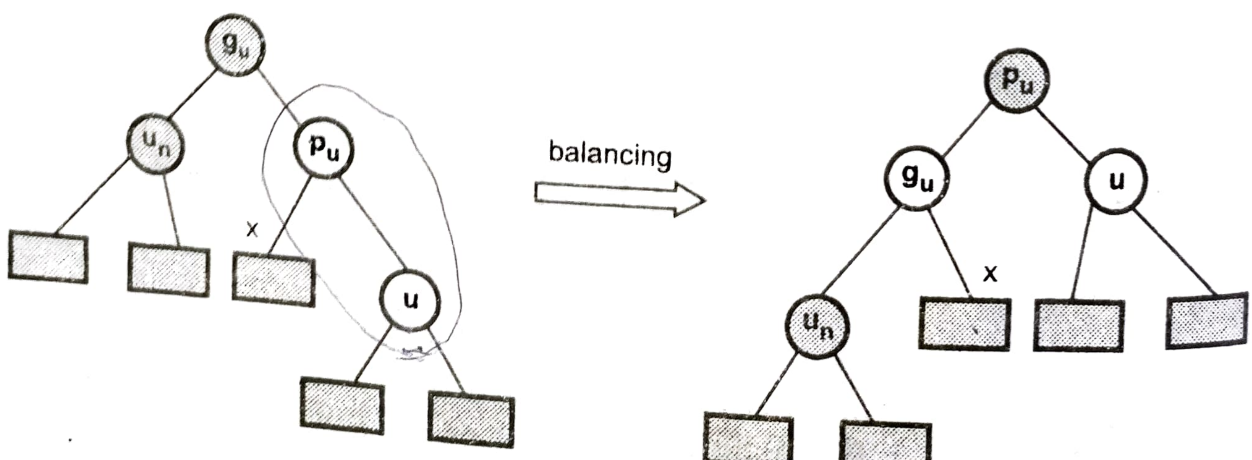


Fig. 7.11 Removal of RR_b imbalancing

- **Removing LR_b and RL_b imbalance**

1. Apply double rotation of u about p_u followed by u about g_u.
2. For LR_b recolor u to black and recolor p_u and g_u to red.
3. For RL_b recolor p_u to black.

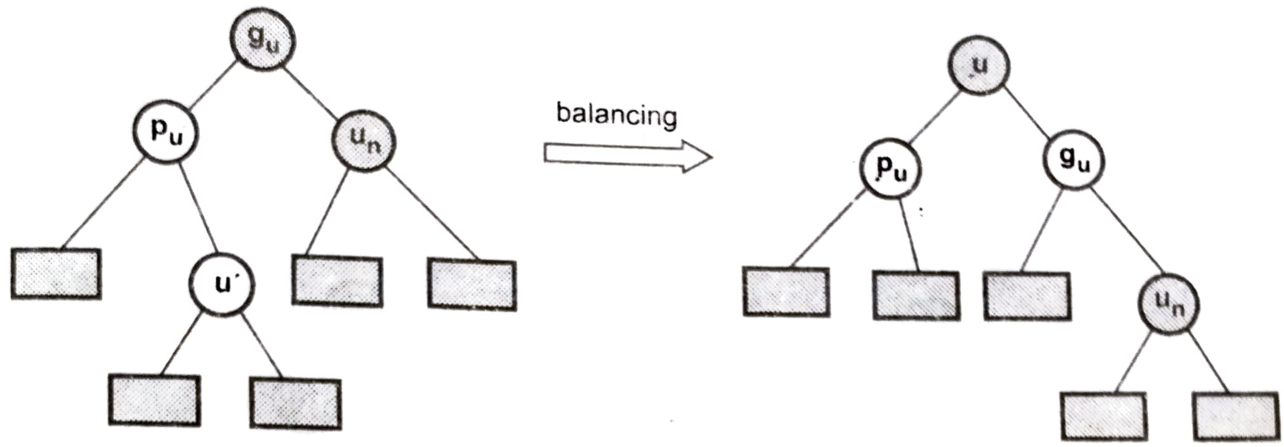


Fig. 7.12 Removal of LR_b balancing

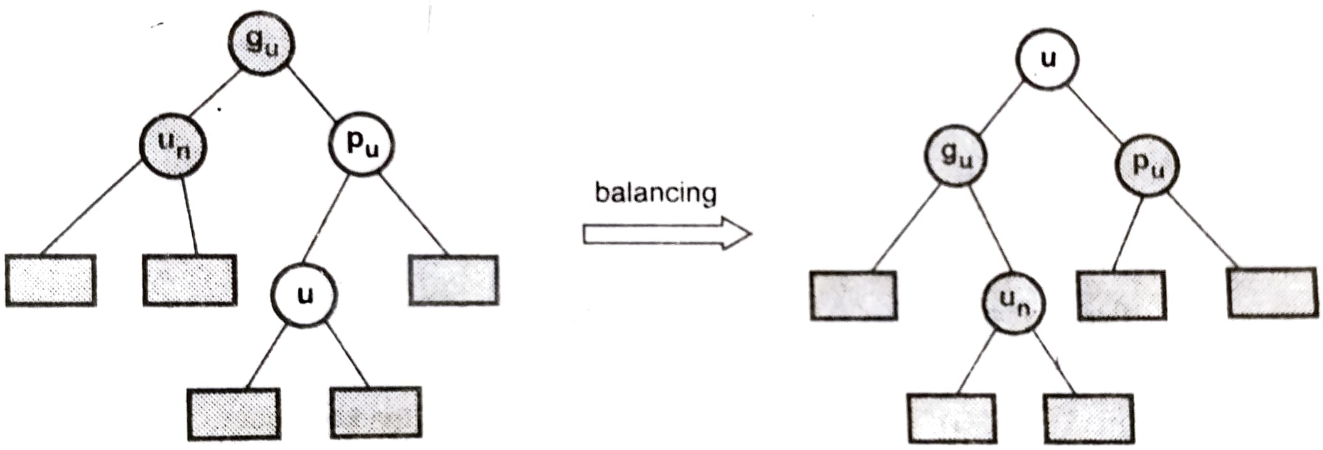


Fig. 7.13 Removal of RL_b balancing

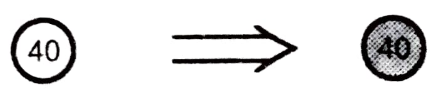
Example for insertion of elements in Red-Black tree

Insert key sequence is as given below

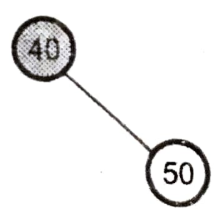
40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55

Construct Red-Black tree

Initially insert node 40 with color red. Recolor this root node to black.

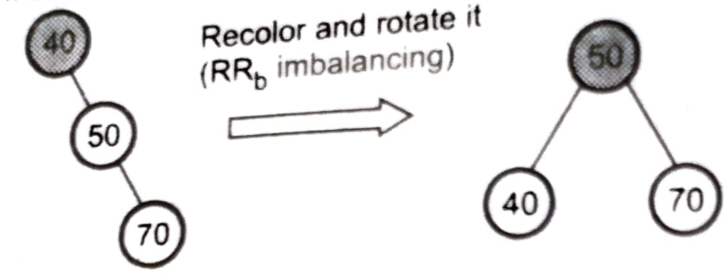


Insert 50

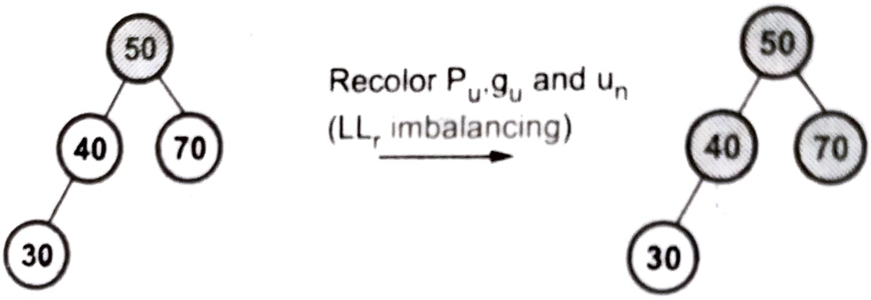


Insert 70

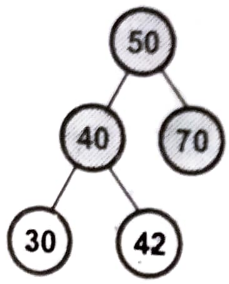
Insert 70



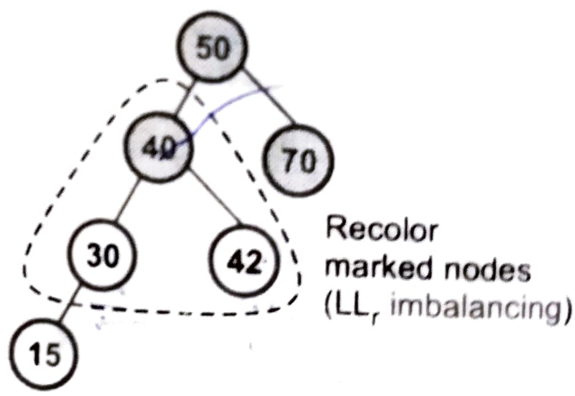
Insert 30

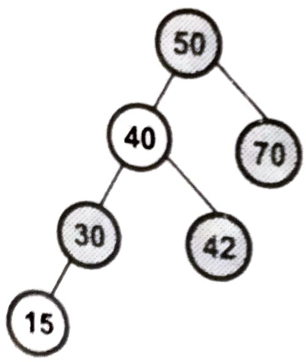


Insert 42

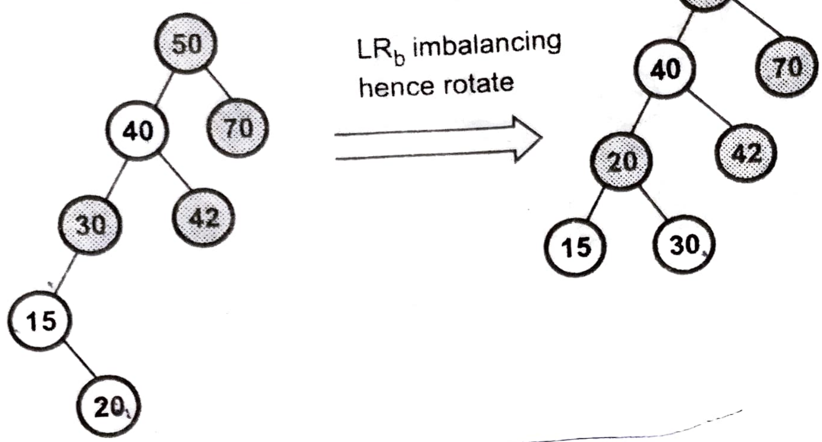


Insert 15

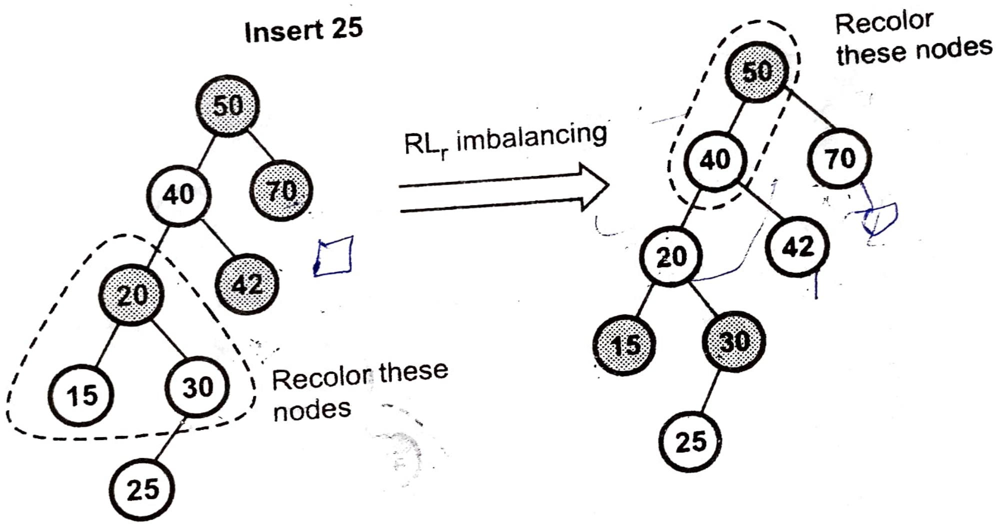


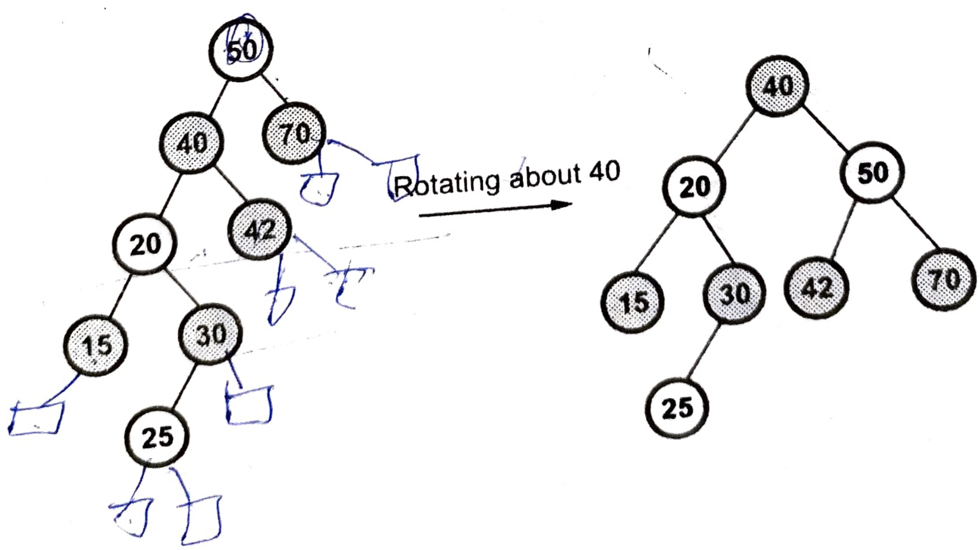


Insert 20

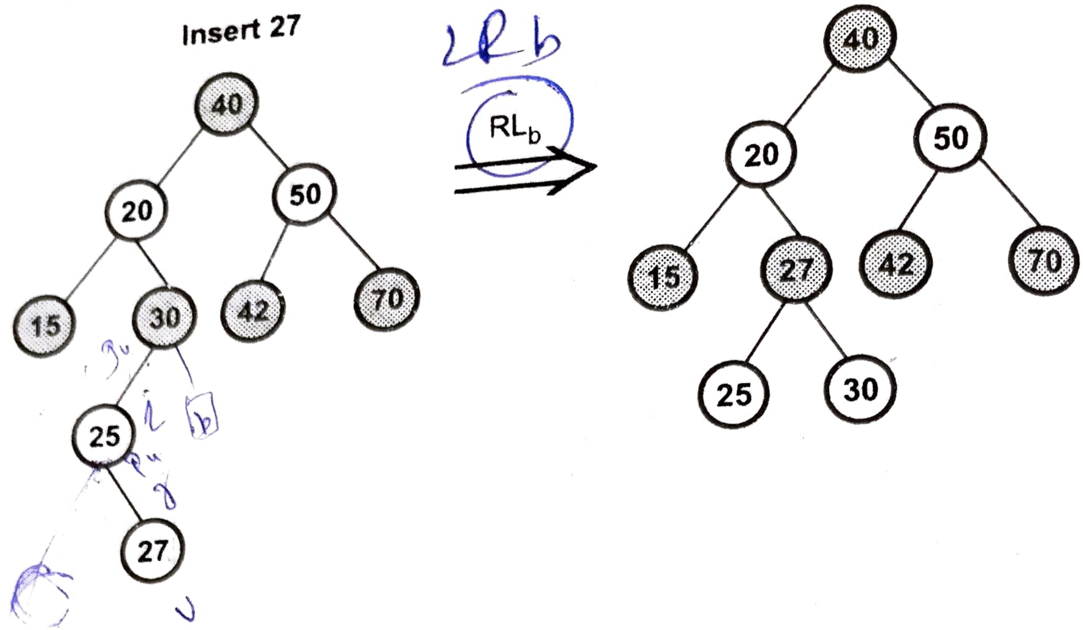


Insert 25

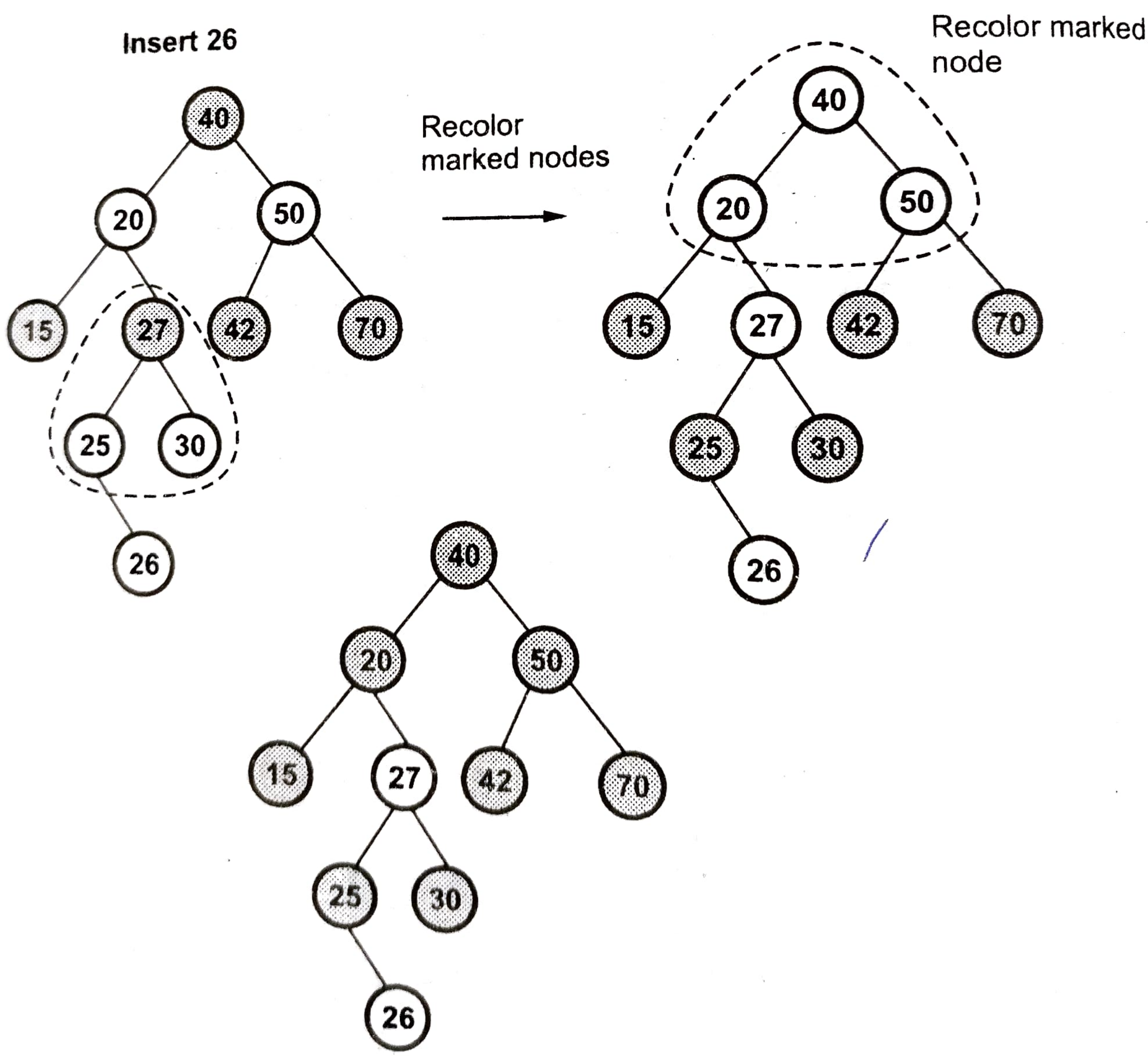




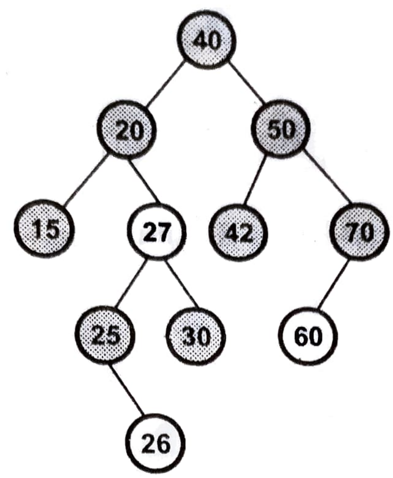
Insert 27



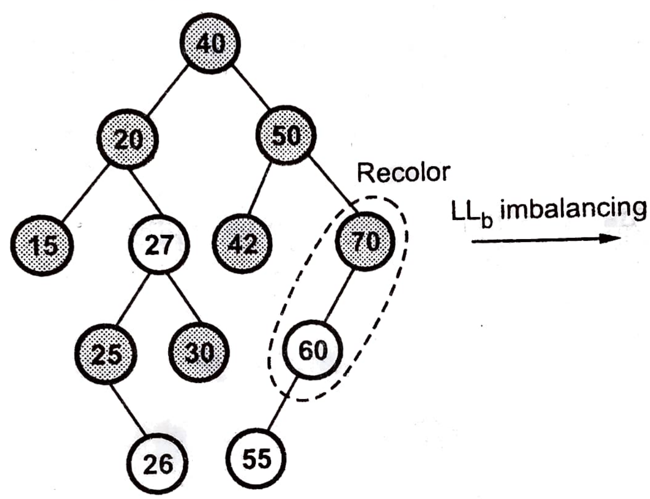
Insert 26

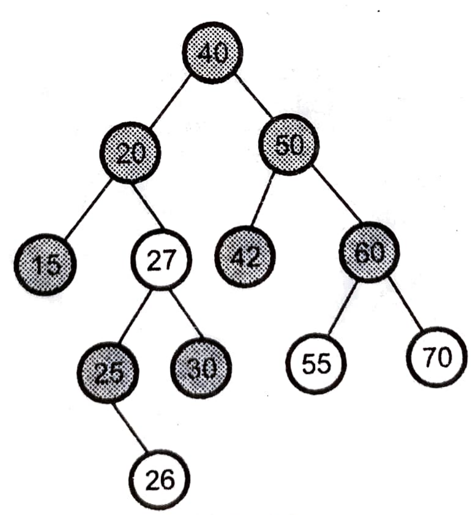


Insert 60



Insert 55





is final red-black tree