UNIT-VI

AVL Trees: Definition :

Trees whose height in the worst case turns out to be O(log n) are known as balanced trees or height balanced trees. One such balanced tree viz., AVL trees. AVL trees were proposed by Adelson-Velskiiand Landis in 1962.

An empty binary tree is an AVL tree. If non empty, the binary tree T is an AVL tree if

(i) TL and TR, the left and right subtrees of T are also AVL trees and

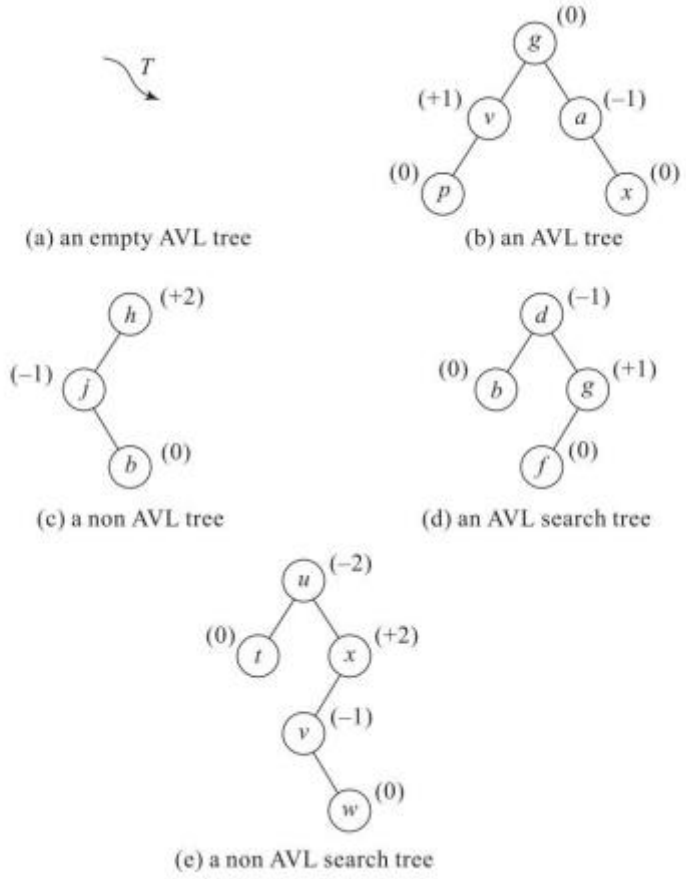
(ii) (ii) $|h(TL) - h(TR)| \leq 1$, where h(TL) and h(TR) are the heights of the left subtree and right subtree of T respectively.

For a node u, bf(u) = (h(uL) – h(uR)) where h(uL) and h(uR) are the heights of the left and right subtrees of the node u respectively, is known as the **balance factor (bf)** of the node u.

In an AVL tree therefore, every node u has a balance factor bf(u) which may be either 0 or +1 or –1.

A binary search tree T which is an AVL tree is referred to as an AVL search tree.

Ex:



AVL trees and AVL search trees just like binary trees or binary search trees may be represented using a linked representation adopting the same node structure.

Operations:

Retrieval from an AVL search tree:

The retrieval of a key from an AVL search tree is in no way different from the retrieval operation on a binary search tree.

However, since the height of the AVL search tree of n elements is O(log n), the time complexity of the find procedure when applied on AVL search trees does not exceed O(log n).

Insertion into an AVL search tree:

The insertion of an element u into an AVL search tree T proceeds exactly as one would to insert u in a binary search tree.

However, if after insertion the balance factor of any of the nodes turns out to be anything other than 0 or +1 or 1, then the tree is said to be unbalanced.

To balance the tree we undertake what are called rotations. Rotations are mechanisms which shift some of the subtrees of the unbalanced tree to obtain a balanced tree.

For the initiation of rotations, it is required that the balance factors of all nodes in the unbalanced tree are limited to 2, 1, 0, 1, and +2.

Also the rotation is initiated with respect to an ancestor node A that is closest to the newly inserted node u and whose balance factor is either +2 or 2.

If a node w after insertion of node u reports a balance factor of bf(w) = +2 or 2 respectively, then its balance factor before insertion should have been +1 or 1 respectively.

The insertion of a node can only change the balance factors of thosenodes on the path from the root to the inserted node.

If the closest ancestor node A of the inserted node u has a balance factor bf(A) = +2 or – 2, then prior to insertion the balance factors of all nodes on the path from A to u must have been 0.

The rotations which are of four different types are listed below. The classification is based on the position of the inserted node u with respect to the ancestor node A which is closest to the node u and reports a balance factor of 2 or +2.

(i) LL rotation - node u is inserted in the left subtree (L) of left subtree (L) of A
(ii) LR rotation - node u is inserted in the right subtree (R) of left subtree (L) of A
(iii) RR rotation - node u is inserted in the right subtree (R) of right subtree (R) of A
(iv) RL rotation - node u is inserted in the left subtree (L) of right subtree (R) of A

Each of the four classes of rotations are illustrated with examples.

LL rotation:

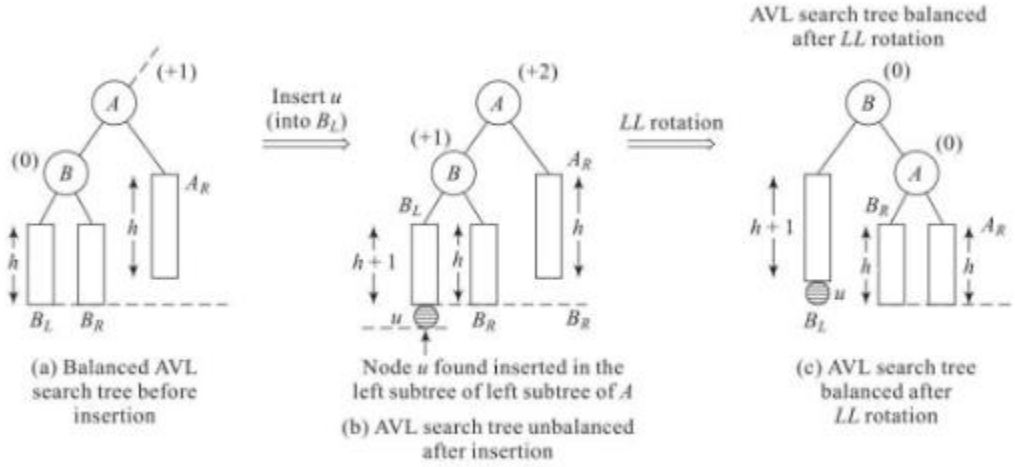
Figure below illustrates a generic representation of LL type imbalance and the corresponding rotation that is undertaken to set right the imbalance.

After insertion of node u, the closest ancestor node of node u, viz., node A, reporting an imbalance (bf(A) = +2) is first found out.

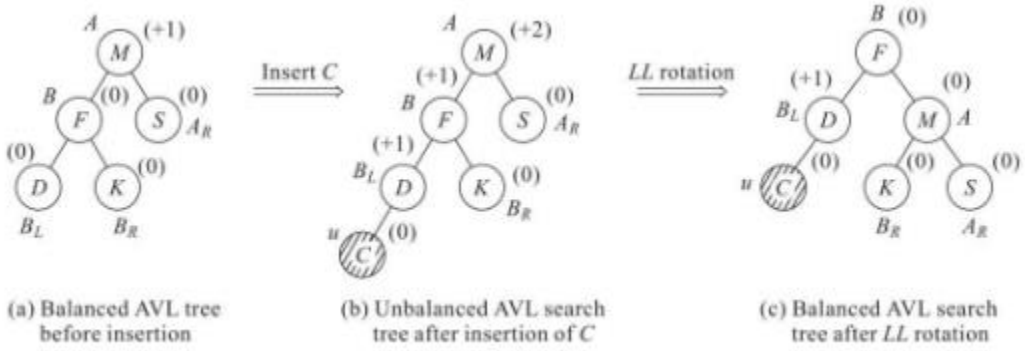
Now with reference to the ancestor node A, we find that the node u has been inserted in the left subtree (L) of left subtree (L) of A.

This implies there is an LL type of imbalance and to balance the tree an LL rotation is to be called for.

The AVL tree before insertion of u (Fig.(a)), the unbalanced tree after insertion of u (Fig.(b)) and the balanced tree after the LL rotation (Fig.(c)) have been illustrated.



Example :



LR rotation:

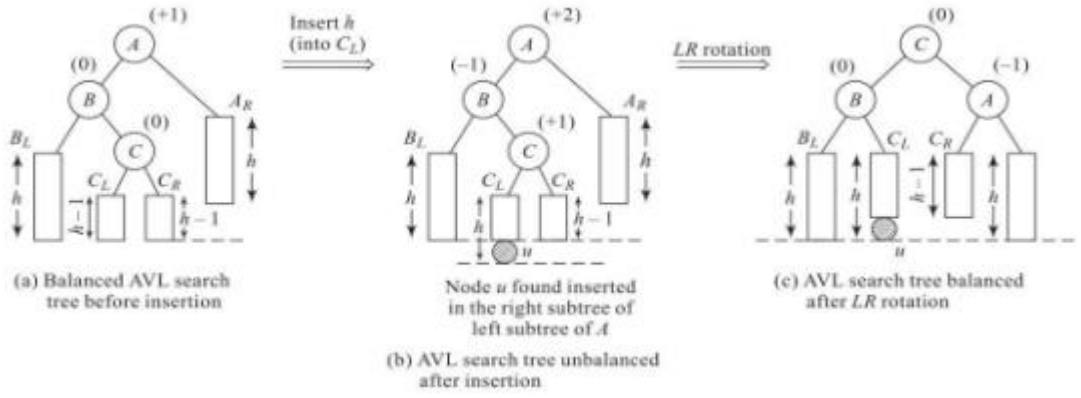
Figure below illustrates the generic representation of an LR type of imbalance and the corresponding rotation that is undertaken to set right the imbalance.

Here the node u on insertion finds A to be its closest ancestor node that is unbalanced and with reference to node A is inserted in the right subtree of left subtree of A.

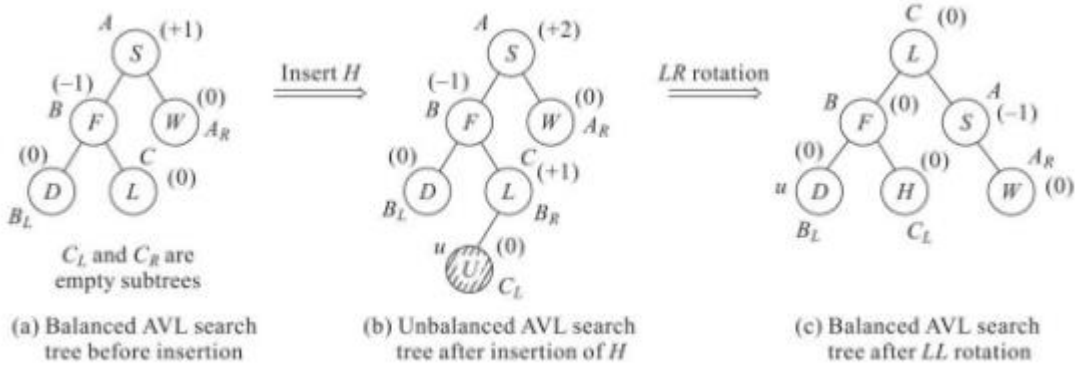
This therefore is an LR type of imbalance and calls for LR rotation to balance the tree. The AVL tree before insertion of u (Fig.(a)), the unbalanced tree after insertion of u (Fig.(b)) and the balanced tree after the LR rotation (Fig. (c)) have been illustrated.

In the case of LR rotation, the following observations hold:

i) If BF(C) = 0 after insertion of new node then BF(A)=BF(B)=0 after rotation
ii) If BF(C) = 1 after insertion of new node then BF(A)= 0, BF(B)=+1 after rotation
iii) If BF(C) = +1 after insertion of new node then BF(A)=1, BF(B)=0 after rotation3



Example:



(a) Balanced AVL search tree before insertion

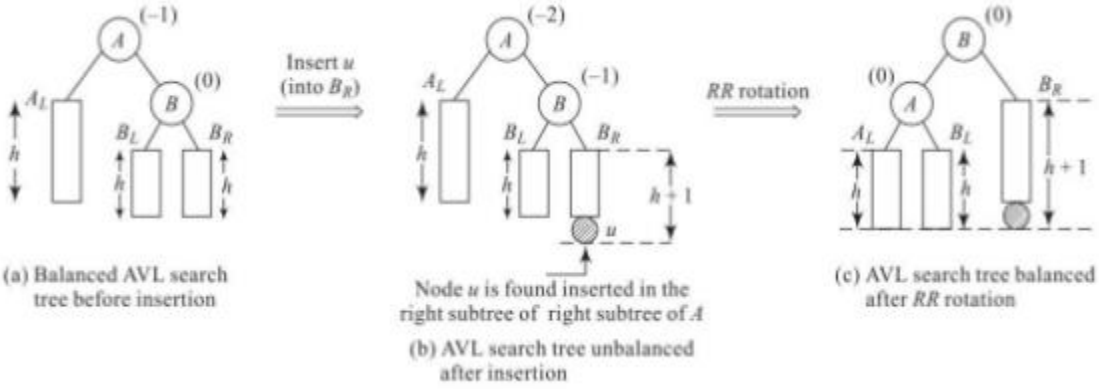
(b) Unbalanced AVL search tree after insertion of *H*

(c) Balanced AVL search tree after *LL* rotation

RR rotation:

The RR rotation is symmetric to the LL rotation. Figure below illustrates the generic representation of the RR rotation scheme.

Observe how node u finds itself inserted in the right subtree of right subtree of A, the closest ancestor node that is unbalanced and the rotation is merely a mirror image of the LL rotation scheme.

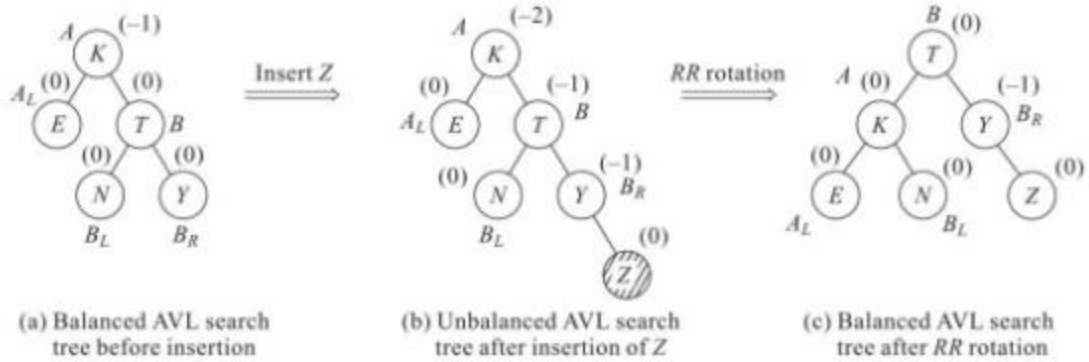


(a) Balanced AVL search tree before insertion

(b) AVL search tree unbalanced after insertion

(c) AVL search tree balanced after *RR* rotation

Example: Consider the AVL search tree shown in Fig. 10.15. The insertion of Z calls for an RR rotation. The unbalanced AVL search tree and the balanced tree after RR rotation have been shown in Fig below.



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion of *Z*

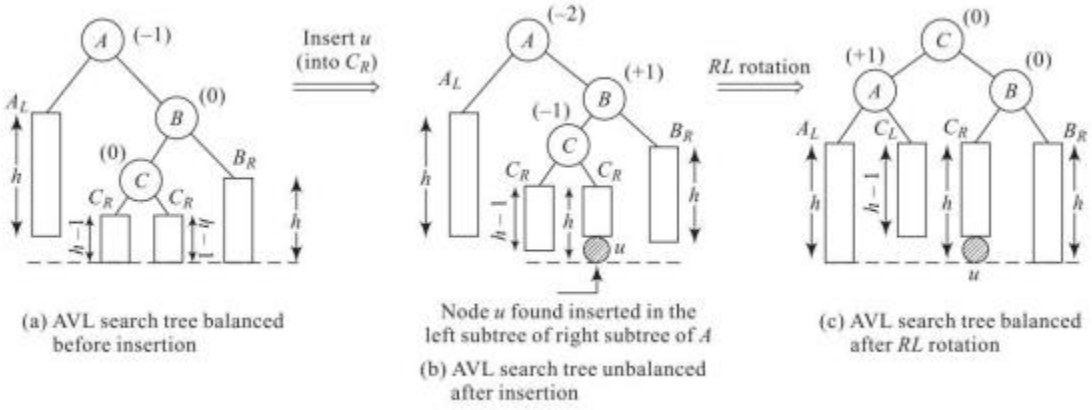
(c) Balanced AVL search tree after *RR* rotation

RL rotation:

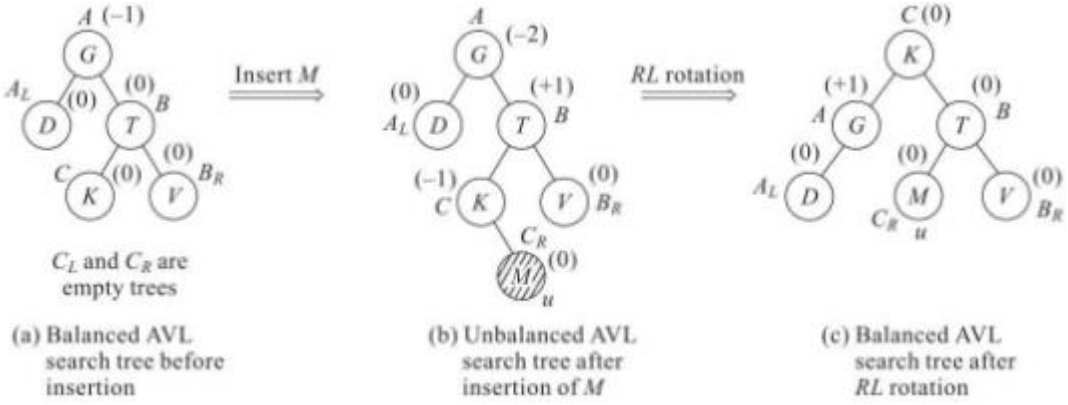
RL rotation is symmetric to LR rotation. Figure below illustrates the generic representation of the RL rotation scheme.

Here node u finds itself inserted in the left subtree of right subtree of node A which is the closest ancestor node that is unbalanced.

As pointed out for the LR rotation scheme, the rotation procedure for RL remains the same irrespective of u being inserted in CL or CR, the left subtree and right subtree of C respectively.



Example: Consider the AVL search tree shown in Fig. (a). The insertion of M calls for an RL rotation. The unbalanced AVL search tree and the balanced tree after RL rotation have been shown in Figs (b) and (c) respectively.



In the above classes of rotations, LL and RR are called as single rotations and LR and RL are called as double rotations.

An LR rotation is a combination of RR rotation followed by an LL rotation and RL rotation is a combination of LL rotation followed by an RR rotation.

Algorithm : Skeletal procedure to insert an ITEM into an AVL search tree T

```
procedure INSERT(T, ITEM)
call GETNODE(X);
DATA(X)=ITEM,
LCHILD(X)=RCHILD(X)=NIL and BF(X)=0;
if (T=NIL) then { Set T to X; exit; }
```

Find node P where ITEM is to be inserted as either the left child or right child of P by following a path from the root onwards. Also, while traversing down the tree in search of the

point of insertion of ITEM, take note of the most recent ancestor node A whose BF(A)= ±1; Insert node X carrying ITEM as the left or right child of node P;

if (node A not found) then
{TEMP = T;
while (TEMP <> X) do
if (DATA(X) > DATA(TEMP))

Deletion from an AVL search tree:

To delete an element from an AVL search tree we discuss the operation based on whether thenode t carrying the element to be deleted is either a leaf node or one with a single non empty subtree or with two non emptysubtrees.

A delete operation just like an insert operation may also imbalance an AVL search tree. Just as LL/ LR/ RL/ RR rotations are called for to rebalance the tree after insertion, a delete operation also calls for rotations categorized as L and R. While the Lcategory is further classified as L0,L1 and L-1 rotations, the R category is further classified as R0, R1 and R-1 rotations.

Update balance factors:

Rule 1: With regard to node p, if node t'sdeletion occurred in its right subtree then bf(p) increases by 1 and if it occurred in its left subtree then bf(p) decreases by 1.

Rule 2: If the new bf(p)=0 then the height of the tree is decreased by1 and therefore this calls for updating the balance factors of its parent node and/or its ancestor nodes.

Rule 3: If the new bf(p) = ±1, then the height of the tree is the sameas it was before deletion and therefore the balance factors of the ancestor nodes remains unchanged.

Rule 4: If the new bf(p)= ±2, then the node p is unbalanced and the appropriate rotations need to be called for.

Classify rotations for deletion:

For the R classification, if bf(A)=+2 then it should have been +1 beforedeletion and A should have a left subtree with root B. Based on bf(B) being either 0 or +1 or –1, classify the R rotations further as R0, R1 and R- 1 respectively.

For the L classification, if bf(A)= -2 then it should have been -1 before deletion and A should have a right subtree with root B. Based on bf(B) being either 0 or +1 or -1, classify the L rotations further as L0, L1 and L-1 respectively.

Rotation free deletion of a leaf node:

In the case of deletion of node t which is a leafnode, we physically delete the node and make the child link of its parent, viz., node p null. Now update the balance factor of node p based on whether the deletion occurred to its right or left.

If it had occurred in the right then we increase bf(node p) by 1 or else decrease bf(node p) by 1. The new updated value of bf(node p) is now tested against Rules 1- 4 for updating the balance factors of its ancestor nodes.

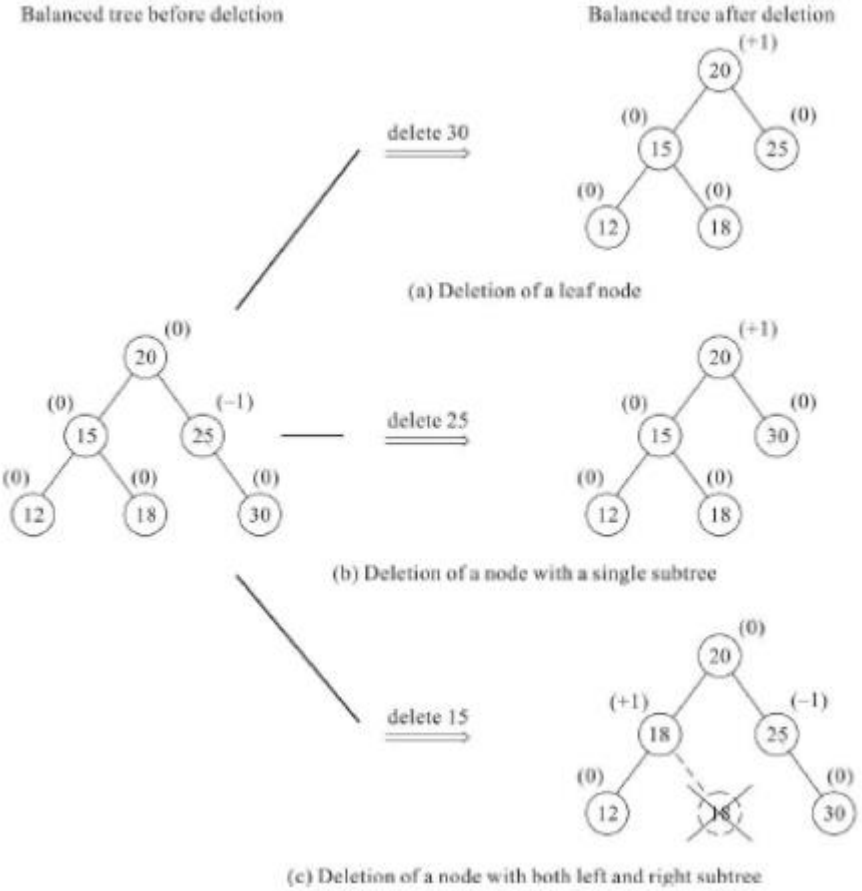
Rotation free deletion of a node having a single subtree:

In the case of deletion ofnode t with a single subtree, just as before, we reset the child link of the parent node, node p, to point to the child node of node t. The balance factors of node p and / or its ancestor nodes are updated using Rules 1-4.

Rotation free deletion of a node having both the subtrees:

In the case of deletionof node t which has both its subtrees to be non empty, the deletion is a little more involved. We first replace DATA (node t) with the smallest key of the right subtreeofnodet or with the largest key of the left subtree of node t.

The smallest key of the right subtree of node t can be obtained by moving right and then moving deep left until the left child link is NIL. Similarly, moving left and then moving deep right until an empty right child link is seen will yield the largest element in the left subtree of node t.



R category rotations associated with the delete operation:

R0 rotation:

Figure below illustrates the generic representation of an R0 rotation. Node t is to be deleted from a balanced tree with A shown as the root (for simplicity) and with A_R as its right subtree.

B is the root of A's left subtree and has two subtrees B_L and B_R. The heights of the subtrees are as shown in the figure. Now the deletion of node t results in an imbalance with bf(A)=+2.

Since deletion of node t occurred to the right of A and since A is the first node on the path to the root, the situation calls for an R rotation with respect to A. Again since bf(B)=0, the rebalancing needs to be brought about using an R0 rotation only.

Here, B pushes itself up to occupy A's position pushing node A to its right along with A_R. While B retains B_L as its left subtree, B_R is handed over to node A to function as its left subtree.

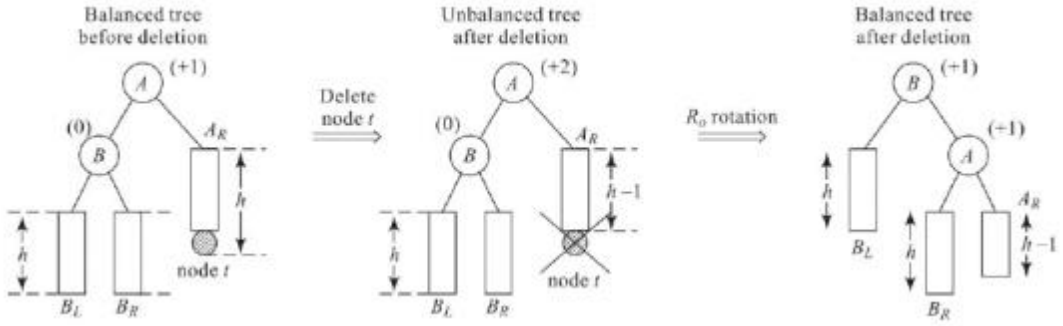


Fig. Generic representation of an R0 rotation

Example:

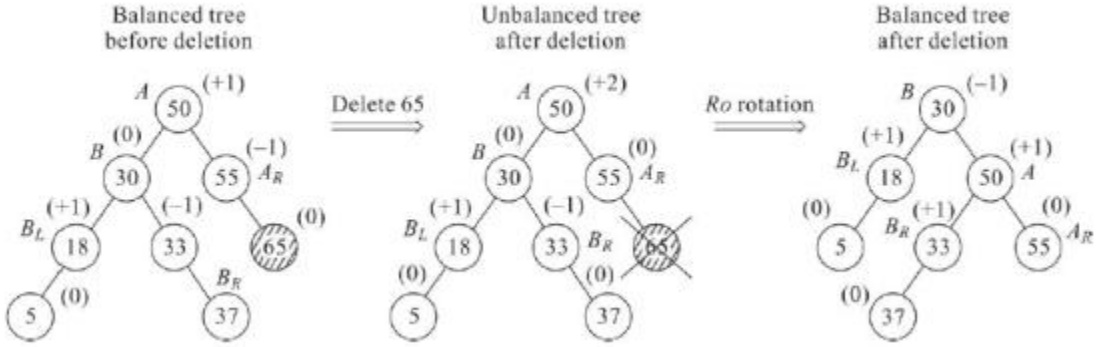


Fig. Deletion of a node calling for R0 rotation

R1 Rotation:

Figure below illustrates the generic representation of an R1 rotation. Deletion of node t occurs to the right of A the first ancestor node whose bf(A)=+2. But bf(B)=+1 classifies it further as R1 rotation. The rotation is similar to the R0 rotation and yields a balanced tree.

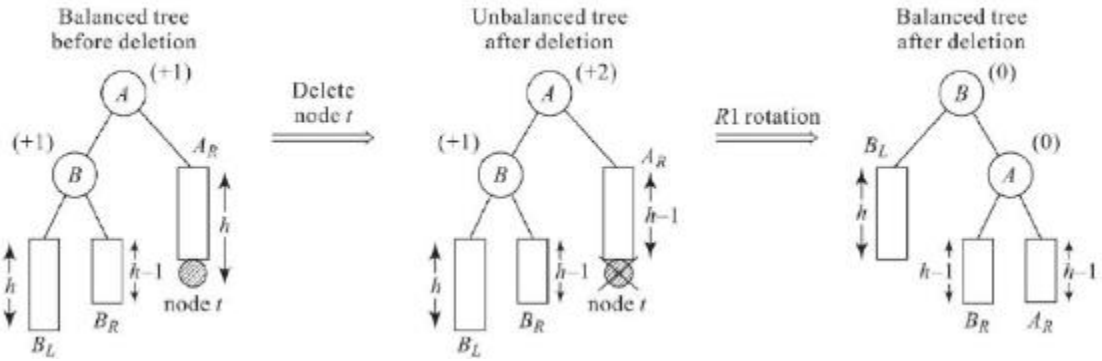


Fig. Generic representation of an R1 rotation

Example:

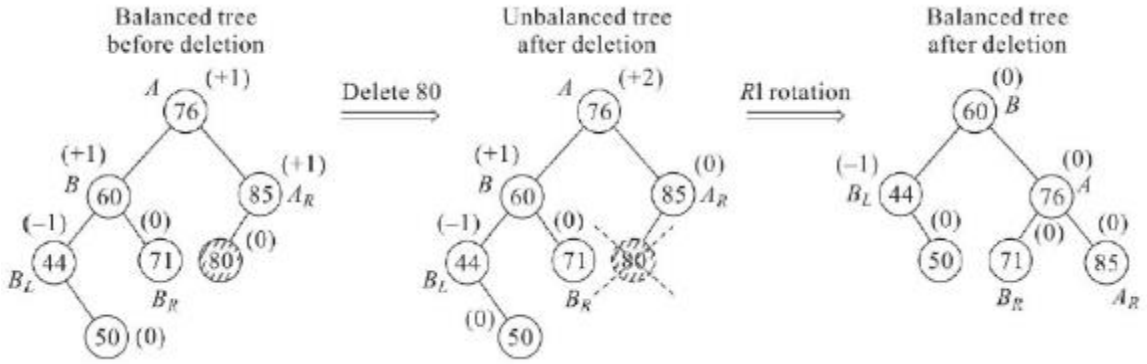


Fig. Deletion of a node calling for R1 rotation

R-1 rotation:

The generic representation of an R-1 rotation is shown in Fig. below. As in the other rotations deletion of node t results in the imbalance of the tree with regard to A and also leaves bf(B)= -1 calling for R-1 rotation.

Here let C be the root of the right subtree of B and C_L and C_R its left and right subtrees respectively. During the rotation C elevates itself to become the root pushing A along with its right subtree A_R to its right.

The tree is now rearranged with C_L as the right subtree of B and C_R as the left subtree of A. The tree automatically gets balanced.

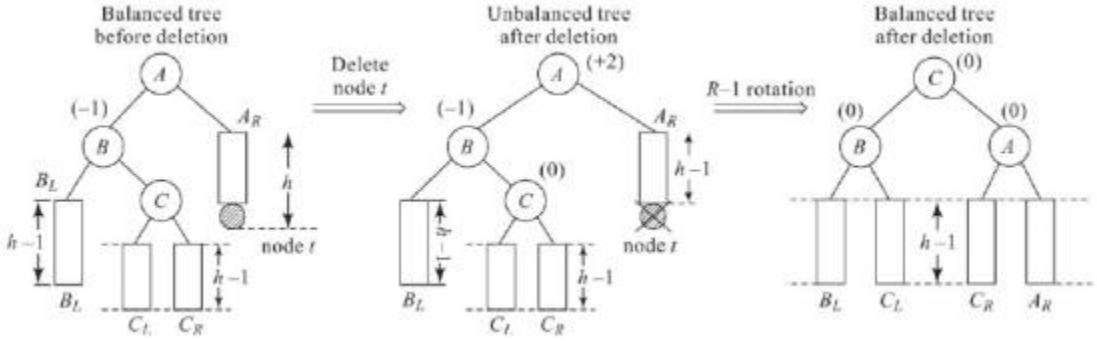


Fig. Generic representation of an R–1 rotation

Example:

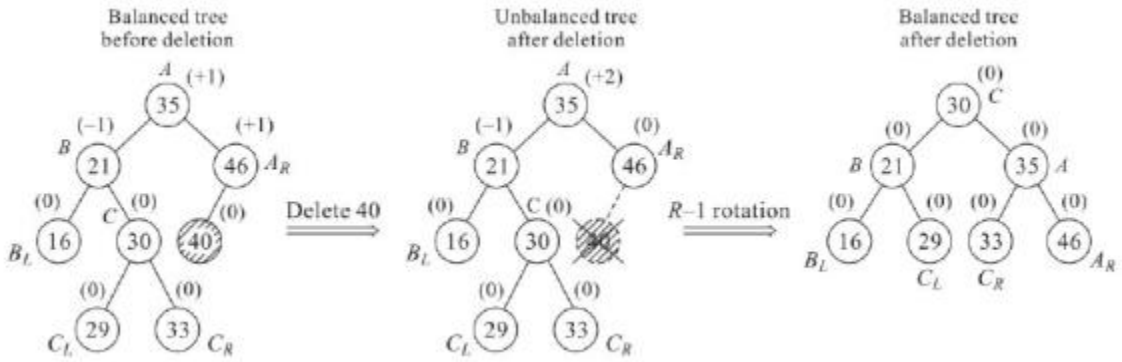


Fig. Deletion of a node calling for R–1 rotation

<u>L category rotations associated with the delete operation:</u>

If the deletion of node t occurs to the left of A, the first ancestor node on the path to the root reporting bf(A)= -2, then the category of rotation to be applied is L.

As in R rotations, based on bf(B)= +1, -1, or 0 the L rotation is further classified as L1, L-1 and L0 respectively.

The generic representations of the L0, L1 and L-1 rotations are shown in Fig.below.

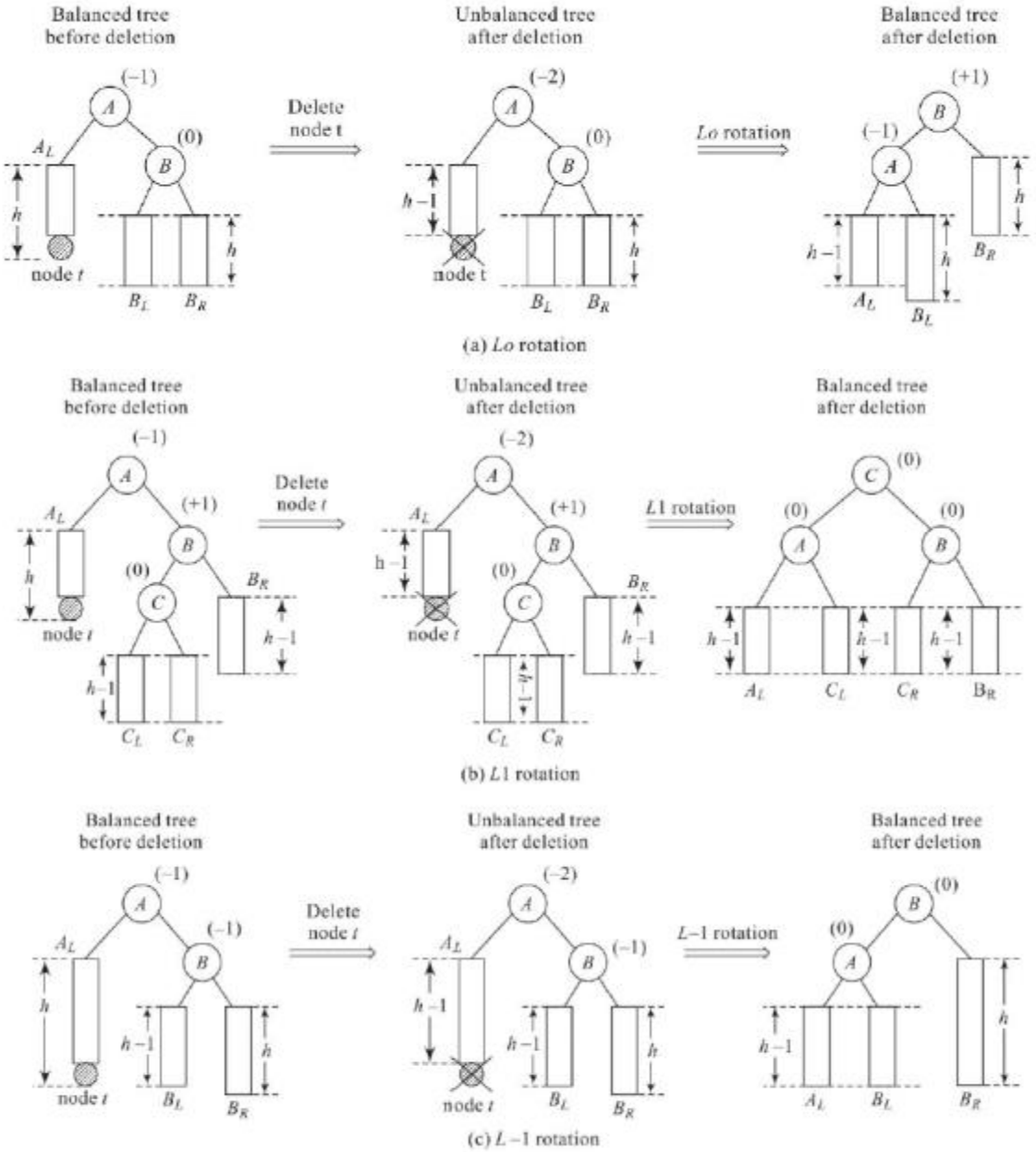


Fig. Generic representations of L0 L1 and L–1 rotations

RED-BLACK TREES:

Definition

A red-black tree is an extended binary search tree in which the nodes and the edges from which these nodes emanate are either red or black and satisfy the following properties:

(i) The root node and the external nodes are always black nodes.

(ii) [Red Condition] No two red nodes can occur consecutively on the path from the root node to an external node.

(iii) [Black Condition] The number of black nodes on the path from the root node to an external node must be the same for all external nodes.

Example

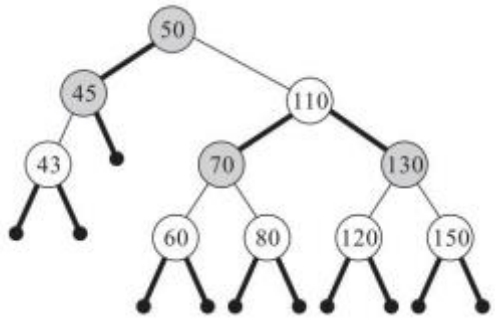


Fig. An example red-black tree

Representation of a red-black tree :

Since a red-black tree is an extended binary search tree, the kind of node representation used for a binary search tree may be employed for the tree as well.

However since the color of a node plays a dominant role in the definition of the red-black tree it is essential that the color is also recorded in the node structure as a field (COLOUR).

Searching a red-black tree :

Searching a red-black tree for a key is in no way different from the procedure used to search for a key in a binary search tree.

Inserting into a red-black tree:

Inserting a key K into a red-black tree follows a procedure exactly similar to the one employed for binary search trees. The only concern now is to determine the colour to which the node must be set to.

If the node is set to black, then the path from the root node to the external node passing through the node would have one more black node. This results in the violation of the Black Condition of a red-black tree.

Hence the other alternative is to set the node to red. Now, if doing so leads to the violation of the Red Condition, then the red-black tree is said to be unbalanced.

To set right the imbalance we need to undertake rotations. Let us suppose u is the newly inserted red node and parent_u its consecutive red node which is also the parent of node u.

Now, u must have a grand parent, grandparent_u which is a black node. Based on the position of node u in relation to parent_u and grandparent_u, and the colour of the other child of grandparent_u, the imbalances are classified as LLb, LLr, RLb, RLr, LRb, LRr, RRb and RRr.

Thus if u is inserted as the Left child of parent_u(L) which in turn is the Left child of grandparent_u (L) and the other child of grandparent_u is black (b) -the child may in fact be an external node which is black- then the rotation undertaken is LLb.

Imbalances of the type LLr, RLr, LRr, and RRr with □r' as its suffix only call for a colour change of the nodes to set right the imbalance. On the other hand imbalances of the type LLb, LRb, RRb and RLb, with □b' as its suffix, call for rotations to set right the imbalance.

LLr, LRr, RRr, and RLr imbalances:

Figure below illustrates a generic representation of the LLr, LRr, RRr, and RLr imbalances and the colour changes that need to be undertaken to set right the imbalance.

The notations L, R and r inscribed on the edges of the red-black trees illustrate the classification of the imbalance.

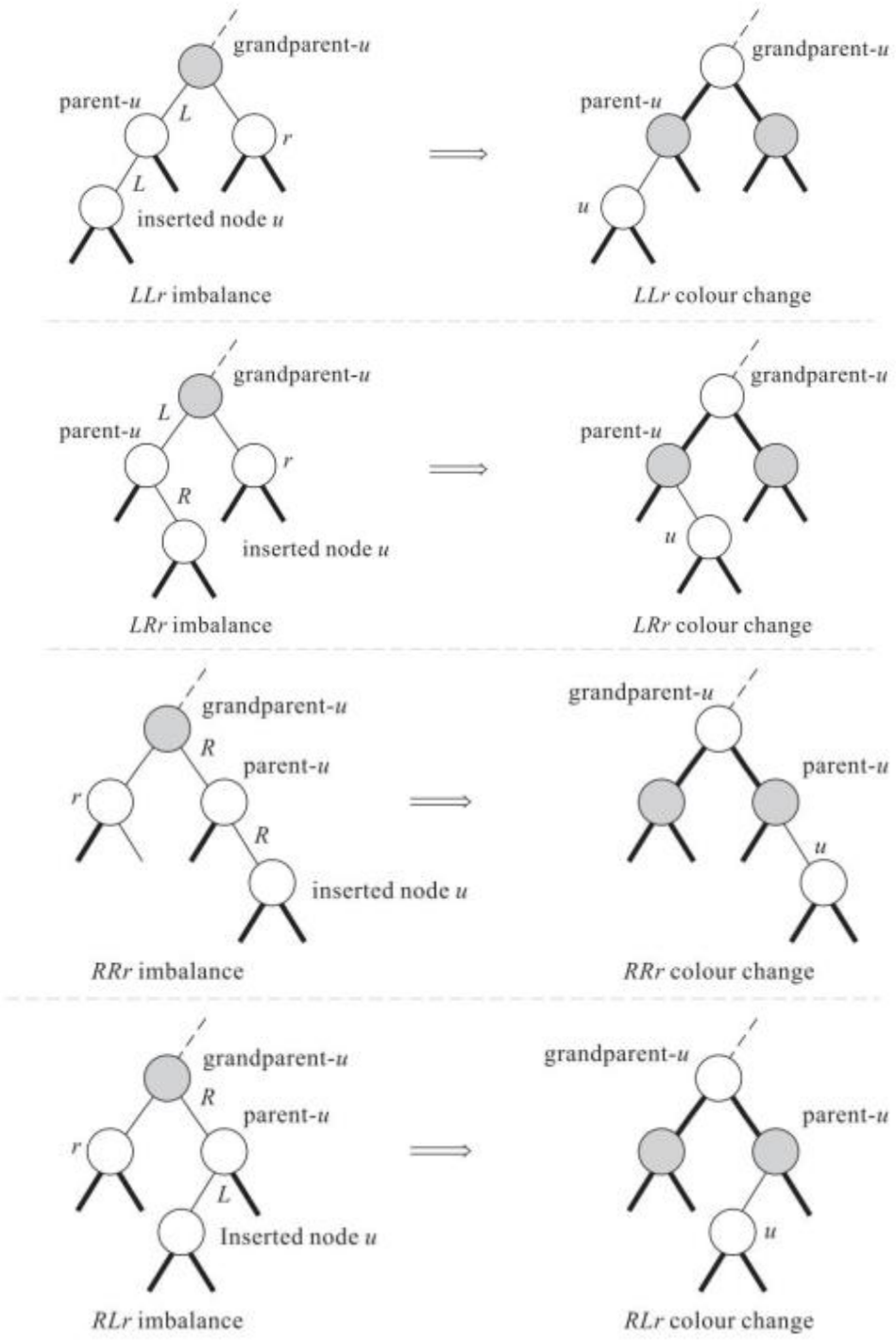


Fig. Generic representations of LLr, LRr, RRr, and RLr imbalances and their colour change

Example:

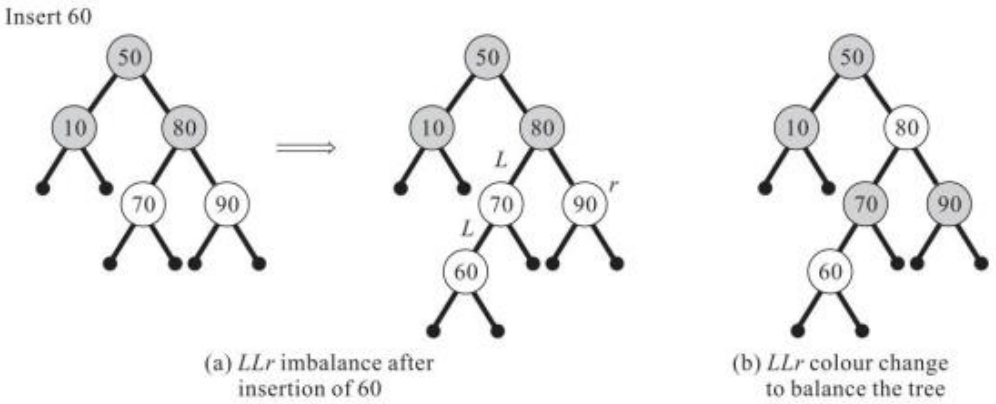


Fig. 12.6 *An example LLr imbalance and its colour change*

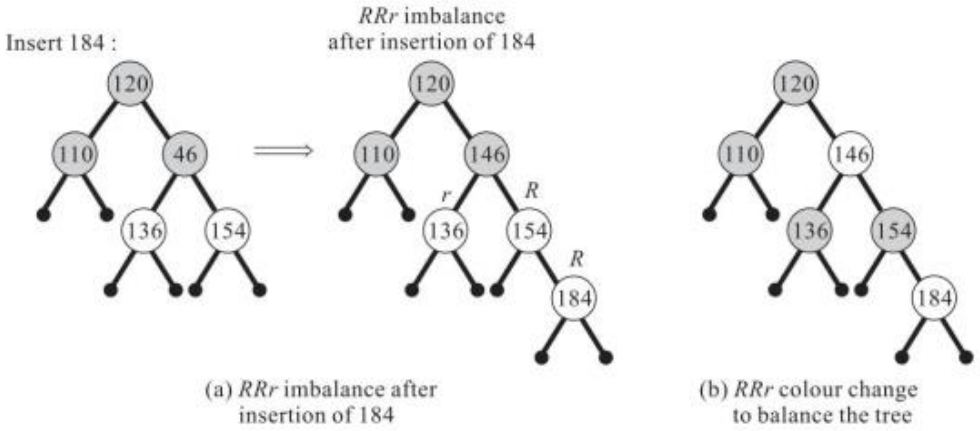
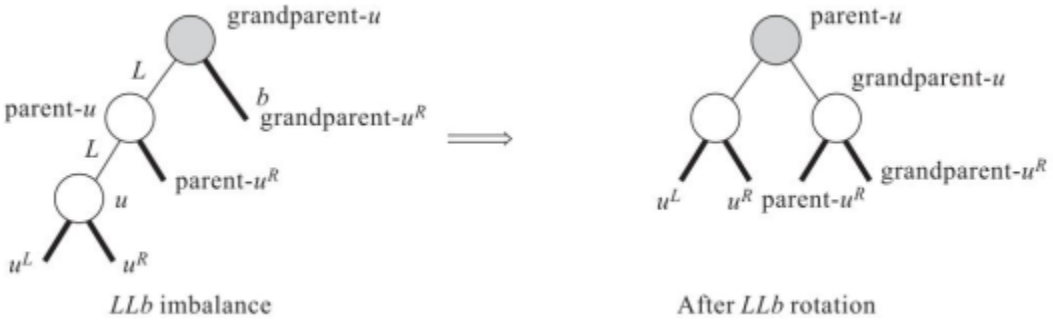


Fig. An example RRr imbalance and its colour change

LLb, LRb, RRb, and RLb imbalances:

Figure illustrates the generic representations of the LLb, LRb, RRb, and RLb imbalances and the respective rotations to rebalance the red-black tree. The notations L, R and b inscribed on the edges of the red-black trees illustrate the classification of the imbalance.



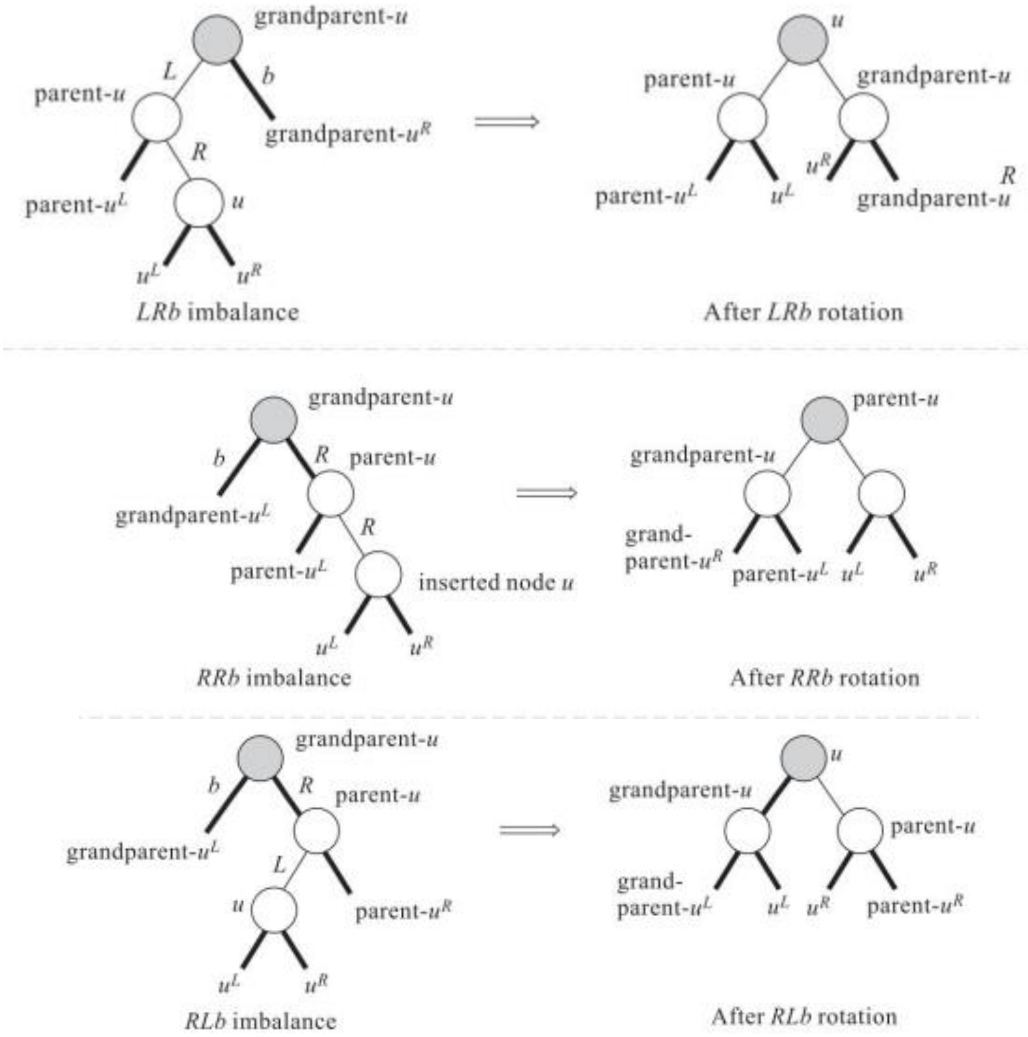
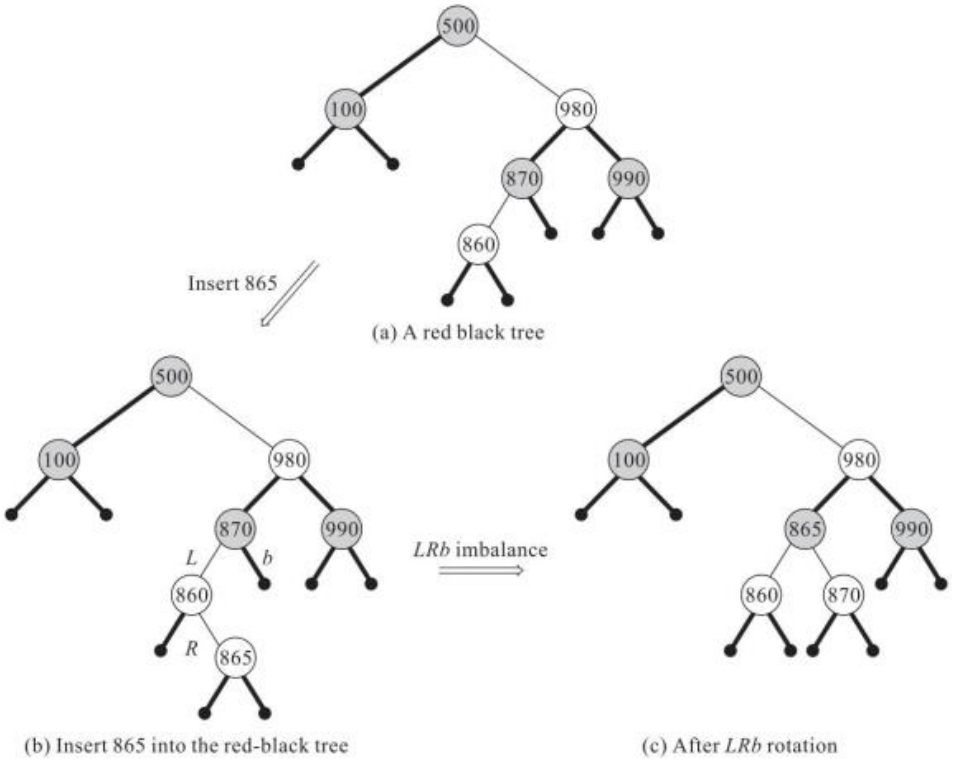


Fig. Generic representations of LLb, LRb, RRb and RLb imbalances and their rotations

Example:



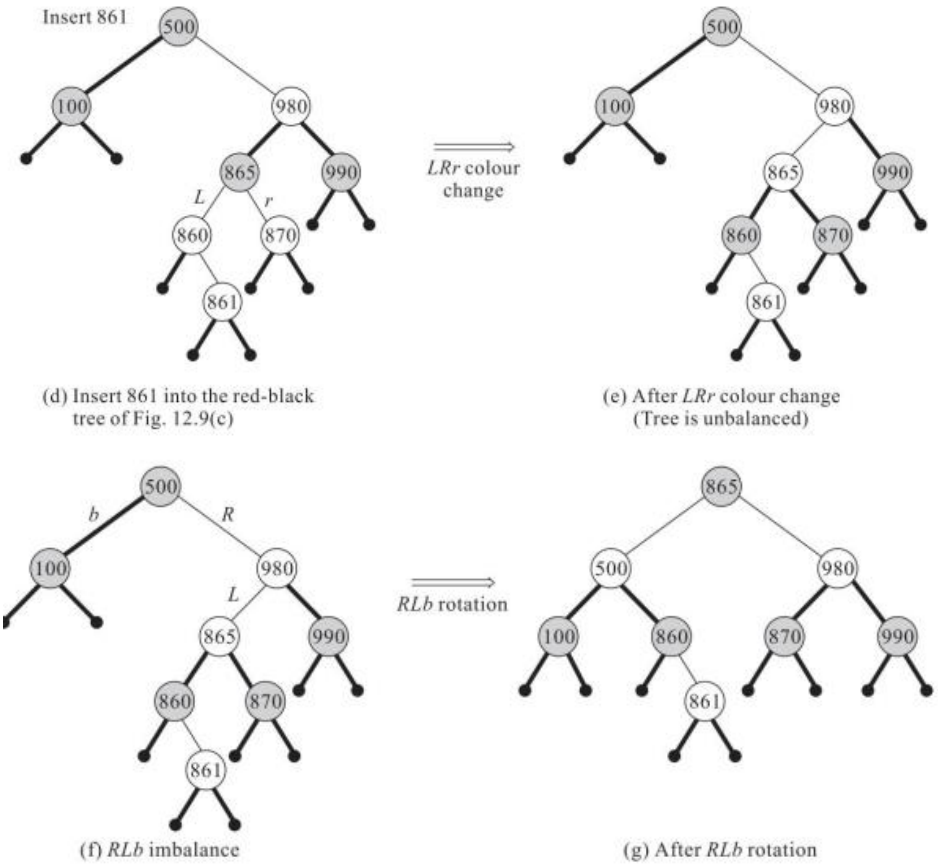


Fig. An example LRb, RLb imbalance

Deleting from a red-black tree:

Deleting a key K from a red-black tree proceeds as one would to delete the same from a binary search tree, when K is a leaf node or K has a lone subtree (left subtree or right subtree only) or K has both left subtree and right subtree.

However, if the deletion results in an imbalance in the tree then this may call for a colour change or a rotation if necessary. If the deleted node were red, then there is no way that the Black Condition would be violated and hence no imbalance in possible.

On the other hand if the deleted node were to be black then there is every possibility of violation of the Black Condition due to the shortage of a black node in a specific root-to-external node path.

In such a case the tree is said to be unbalanced. The imbalance is classified as Left (L) or Right (R) based on whether the deleted node v, occurs to the right or left of its parent node, parent_v.

Again if the sibling of node v, sibling_v is a black node then the imbalance is further classified as Lb or Rb. If sibling_v is a red node, then the imbalance is classified as Lr or Rr. Based on whether sibling_v has 0 or 1 or 2 red children the Lb, Rb imbalances are further sub classified as Lb0, Lb1 and Lb2, and Rb0, Rb1 and Rb2 respectively.

Similarly, the Lr, Rr imbalances are also sub classified as Lr0, Lr1 and Lr2, and Rr0, Rr1 and Rr2 respectively. During rebalancing, v denotes the node that was deleted but physically replaced by another node which takes its place as called for by the delete process.

Rb0, Rb1 and Rb2 imbalances:

Figure below illustrates the generic representations of Rb0, Rb1 and Rb2 imbalances. The

notations R, b and 0/1/2 inscribed on the edges of the red-black trees illustrate the classification of the imbalance. In the case of Rb0 imbalance the rebalancing only calls for a colour change of nodes.

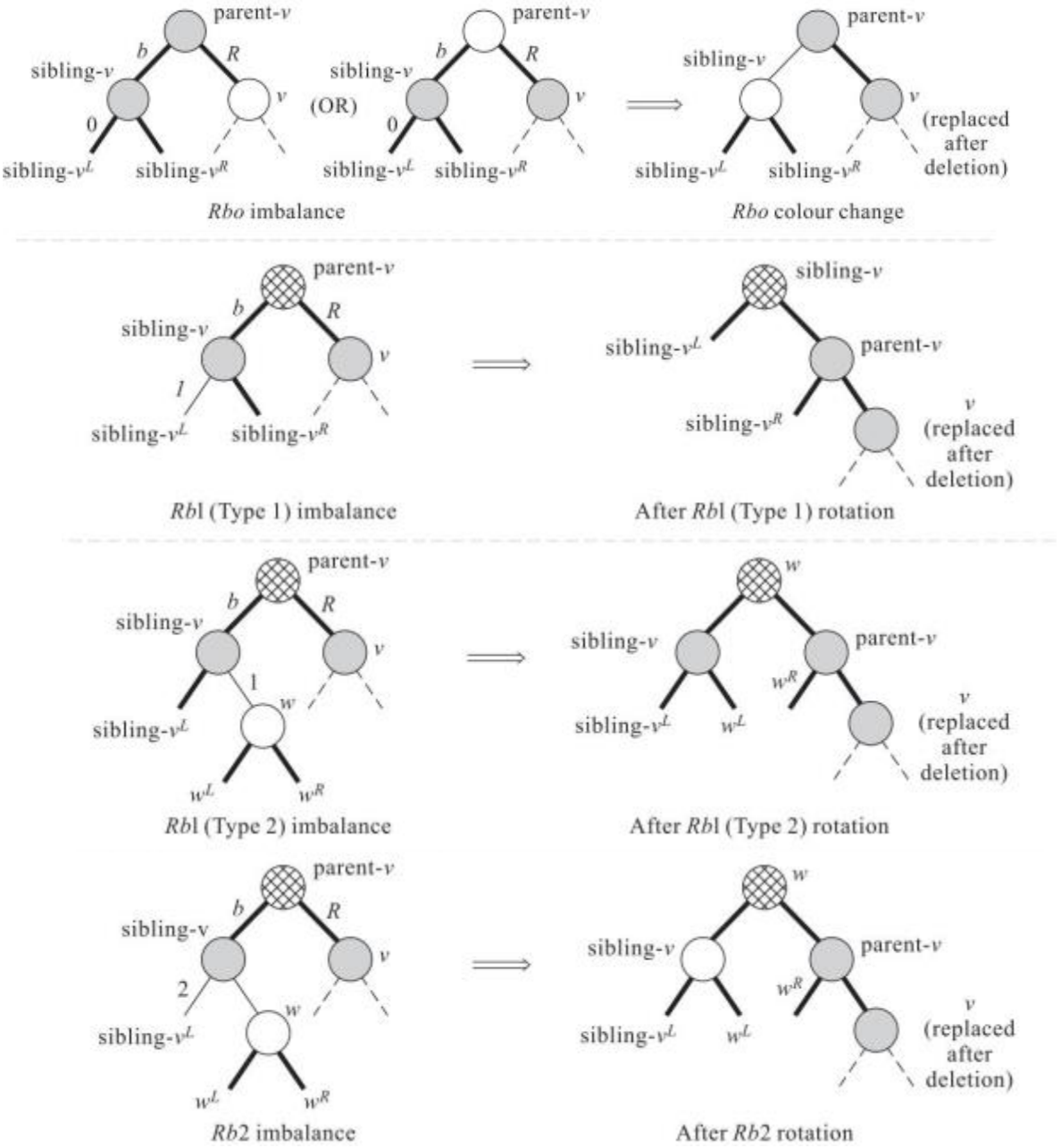


Fig. Generic representations of Rb0, Rb1 and Rb2 imbalances and their rebalancing mechanisms

Rr0, Rr1 and Rr2 imbalances:

Figure below illustrates the generic representations of Rr0, Rr1 and Rr2 imbalances. The notations R, r and 0/1/2 inscribed on the edges of the red-black tree illustrate the classification of imbalance. Rotations are undertaken in all the three cases to rebalance the trees. Rr1 imbalance is of two types indicated as Rr1(type 1) and Rr1(type 2).

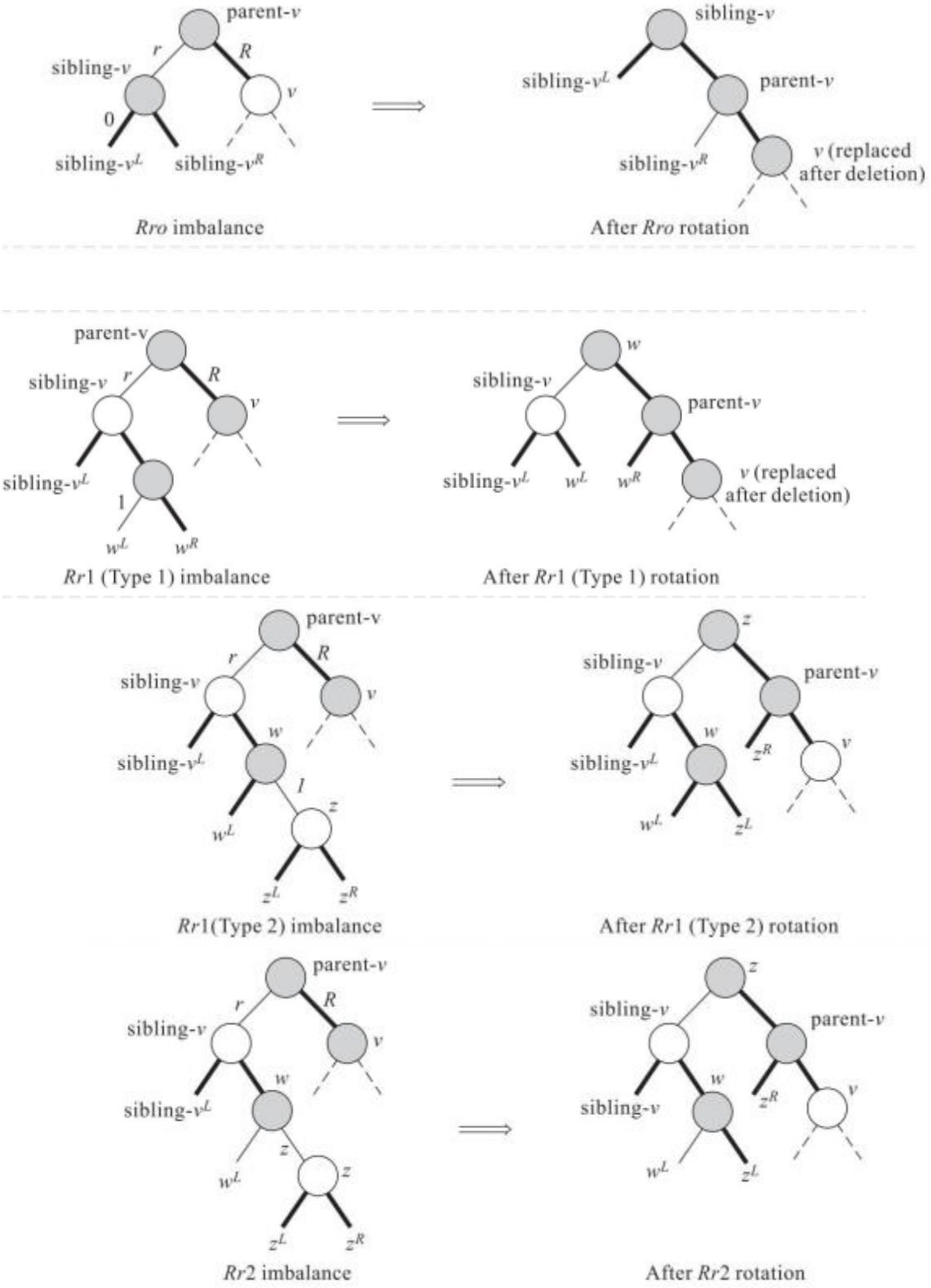


Fig. Generic representations of Rr0, Rr1 and Rr2 imbalances and their rotations

Example:

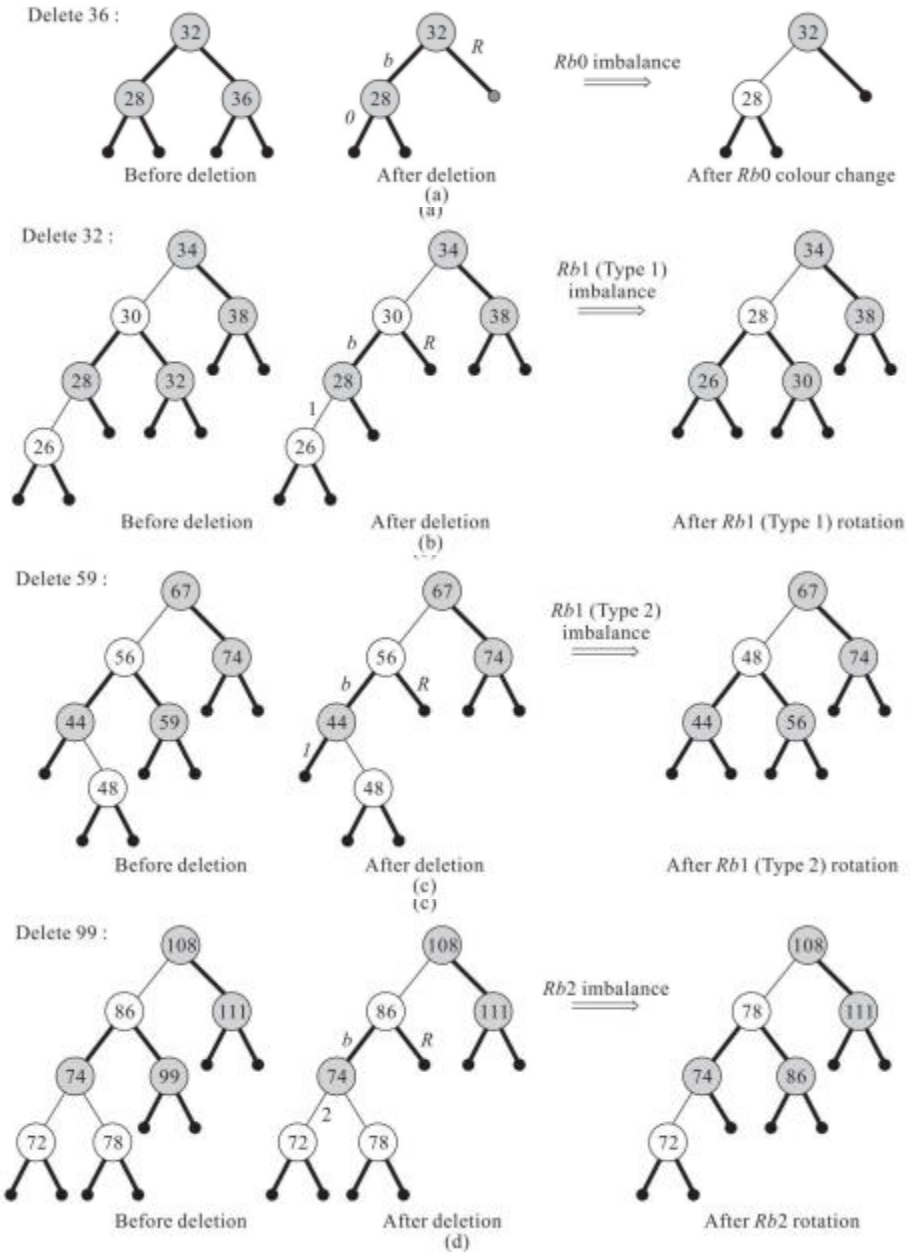


Fig. Example Rb0, Rb1 and Rb2 imbalances

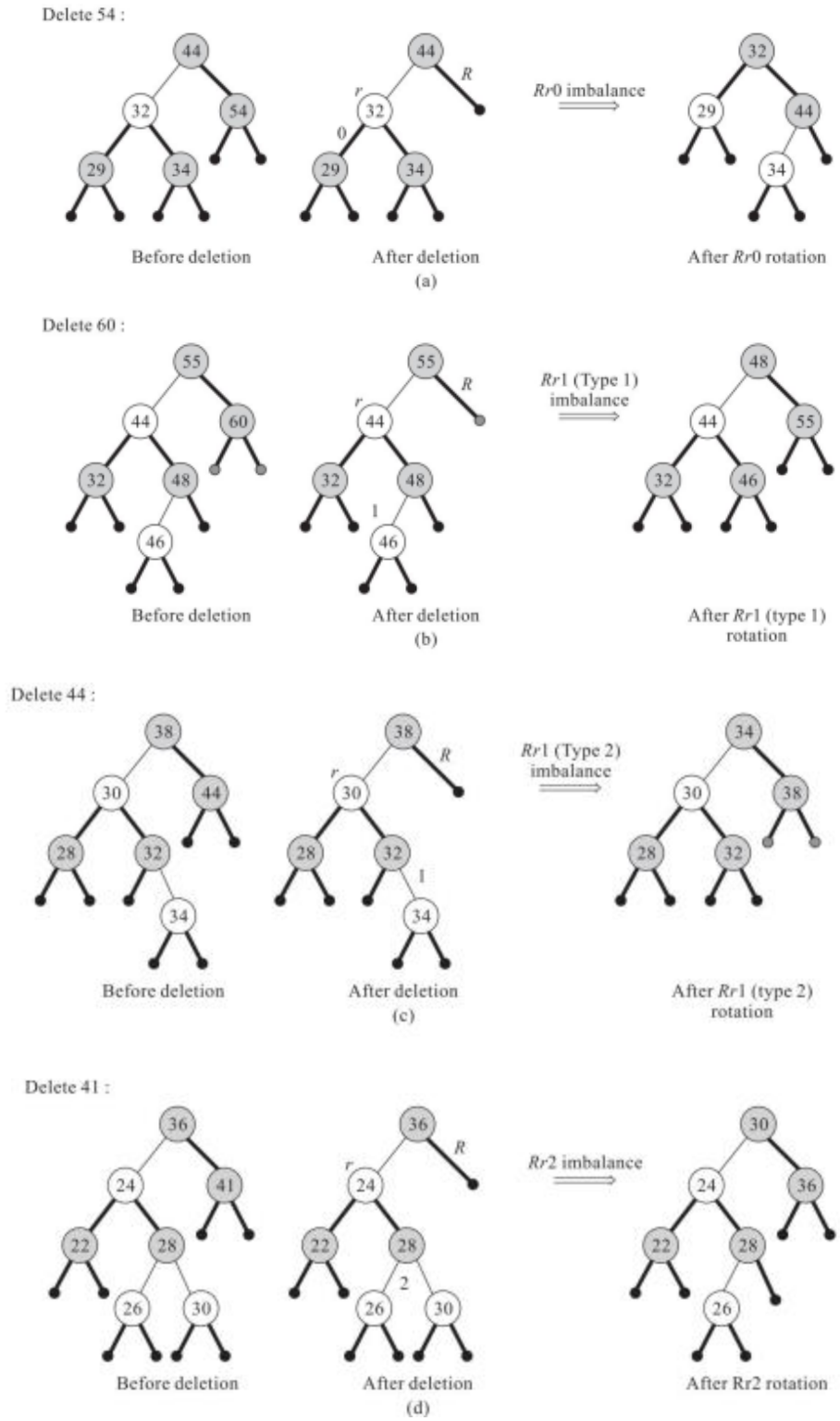


Fig. Example Rr0, Rr1 and Rr2 rotations

Lb0, Lb1 and Lb2 imbalances:

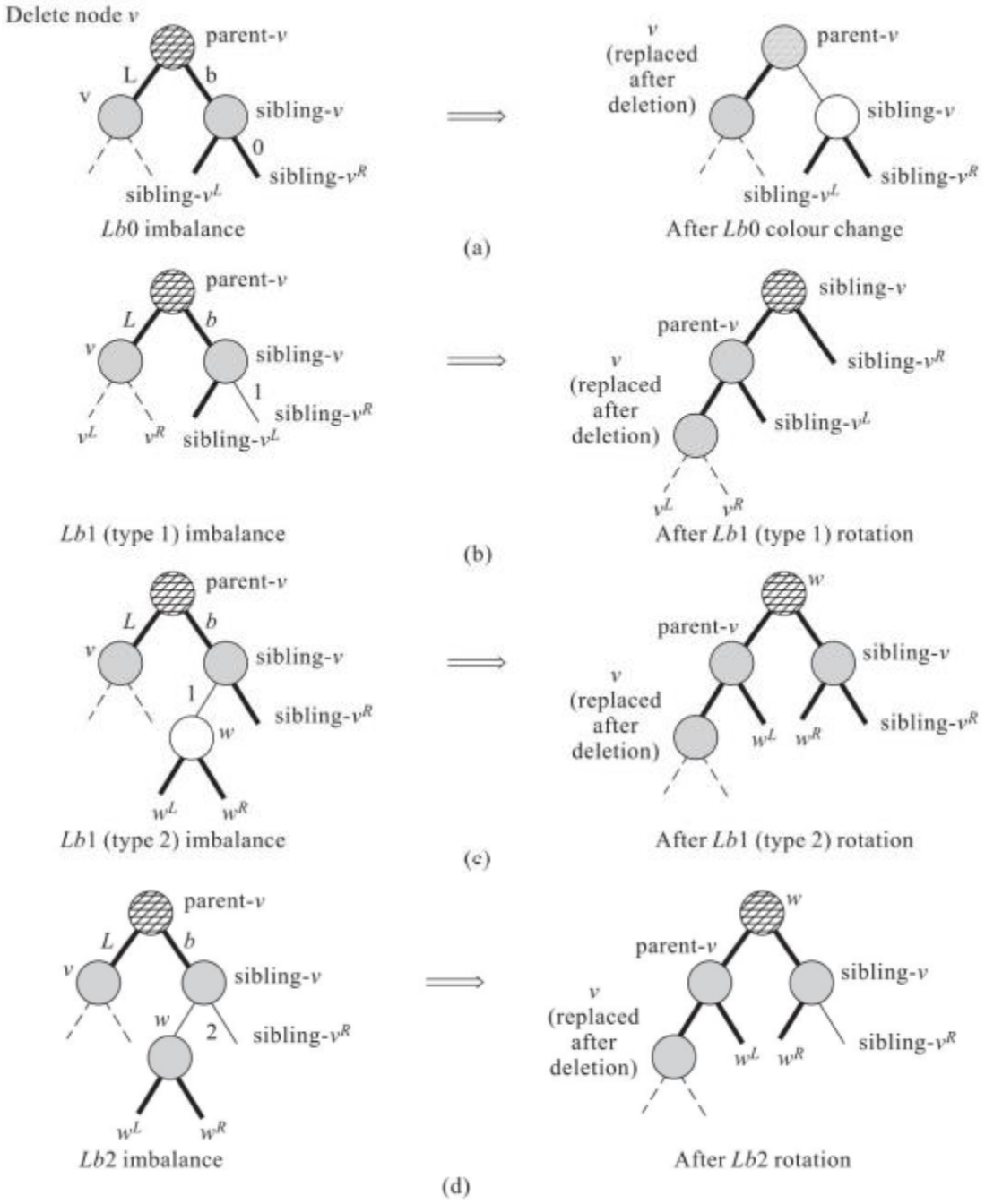
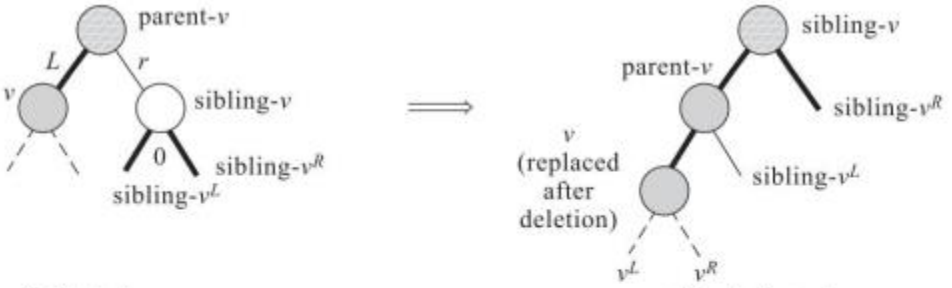


Fig. Generic representations of Lb0, Lb1 and Lb2 imbalances and their rebalancing mechanisms

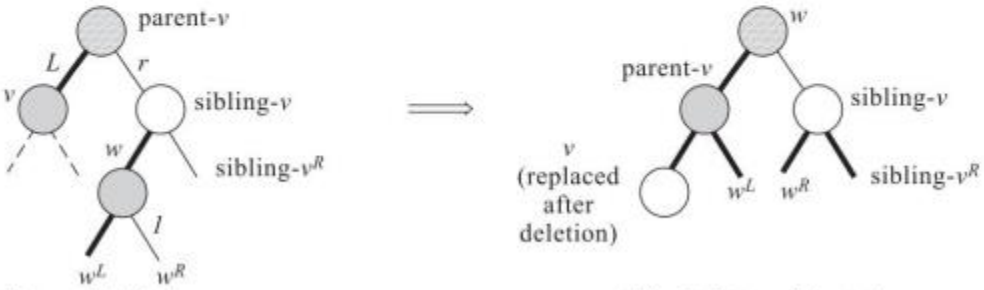
Lr0, Lr1 and Lr2 imbalances:



*Lr*0 imbalance

After *Lr*0 rotation

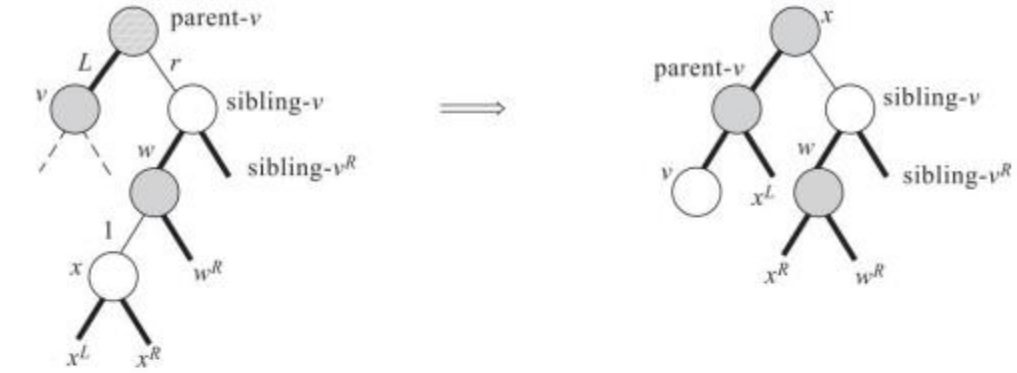
(a)



*Lr*1 (type 1) imbalance

After *Lr*1 (type 1) rotation

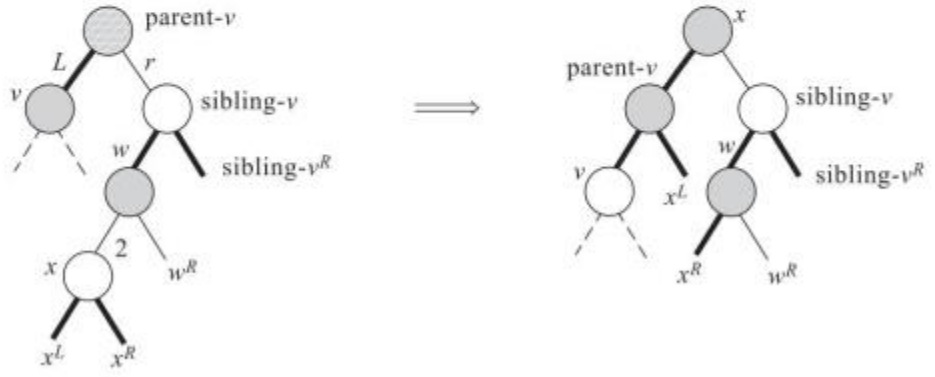
(b)



*Lr*1 (type 2) imbalance

After *Lr*1 (type 2) rotation

(c)



*Lr*2 imbalance

After *Lr*2 rotation

(d)

Fig. Generic representations of Lr0, Lr1 and Lr2 imbalances and their rotations

SPLAY TREES:

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.

The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

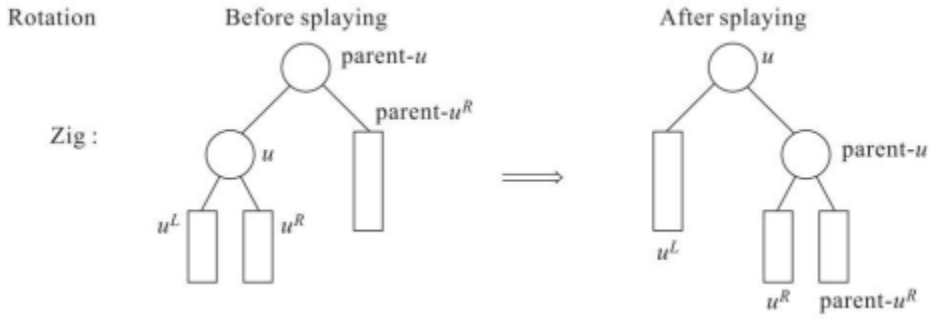
In splay tree, to splay any element we use the following rotation operations...

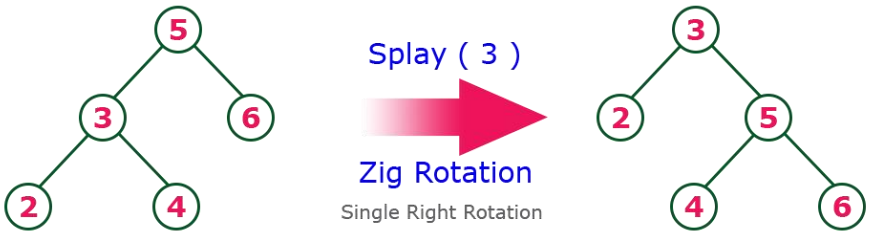
Rotations in Splay Tree

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

Zig Rotation

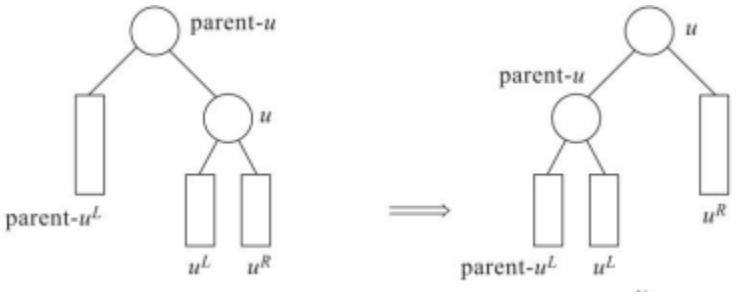
The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...





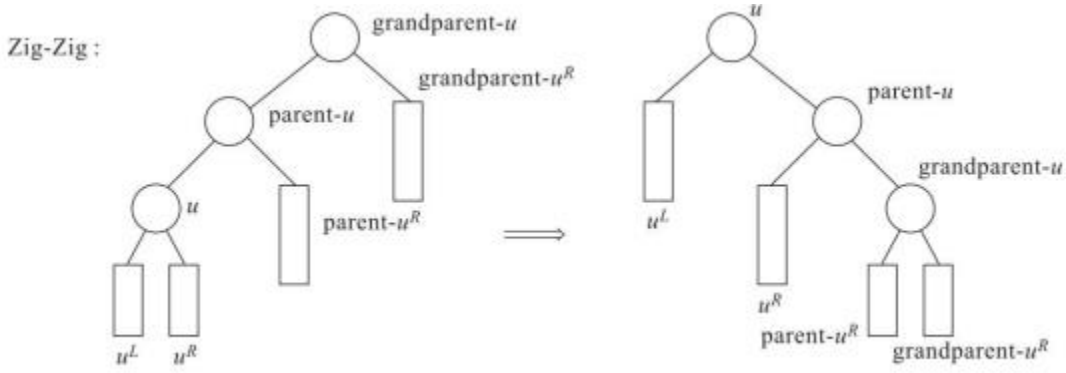
Zag Rotation

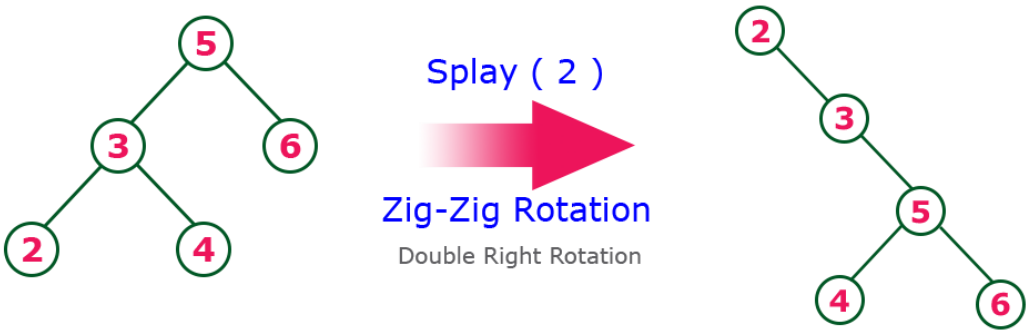
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



Zig-Zig Rotation

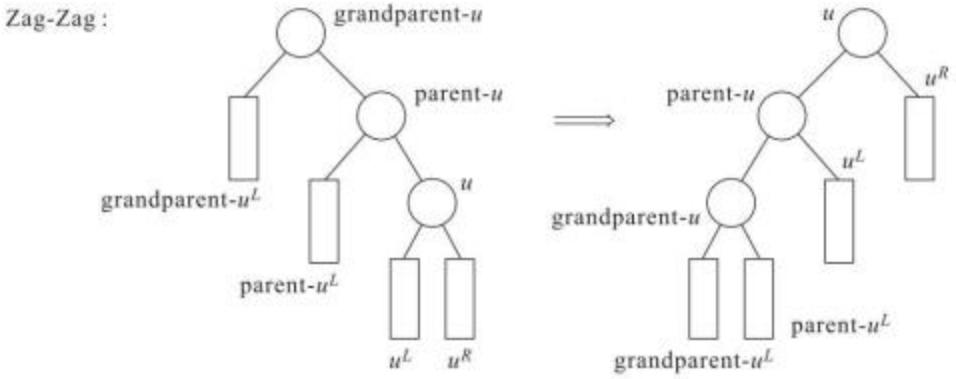
The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

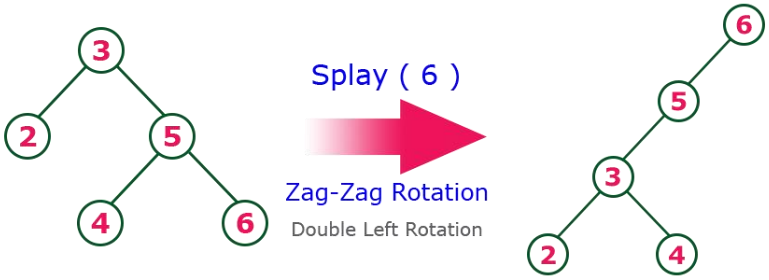




Zag-Zag Rotation

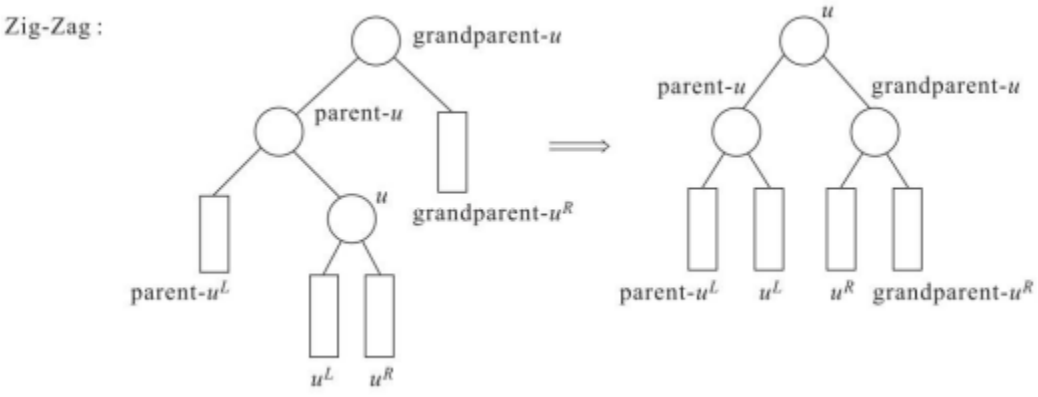
The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

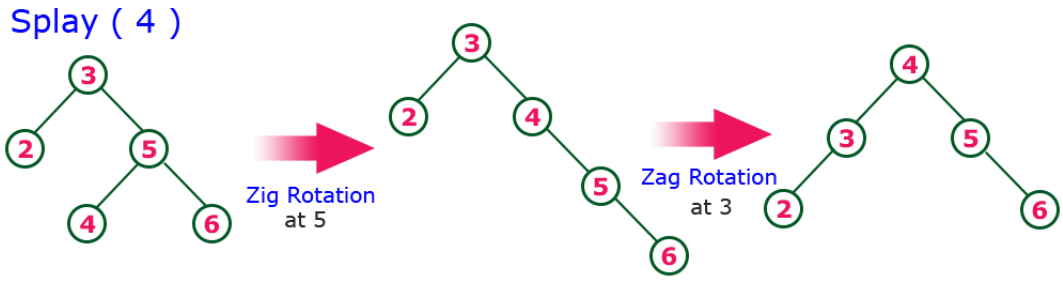




Zig-Zag Rotation

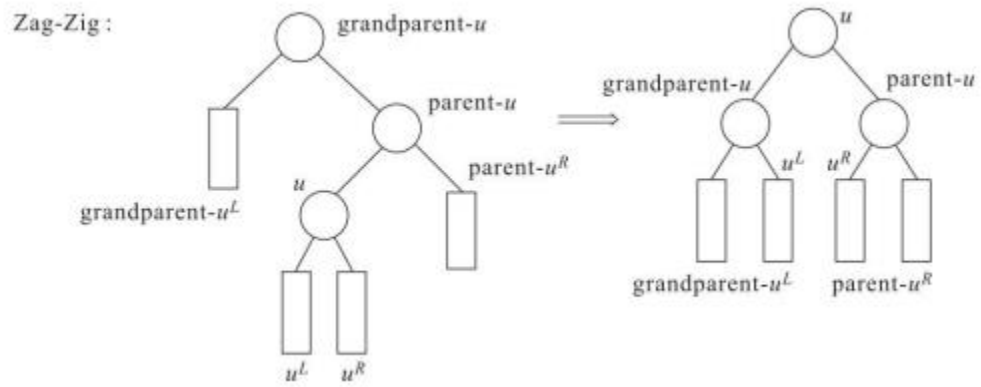
The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...

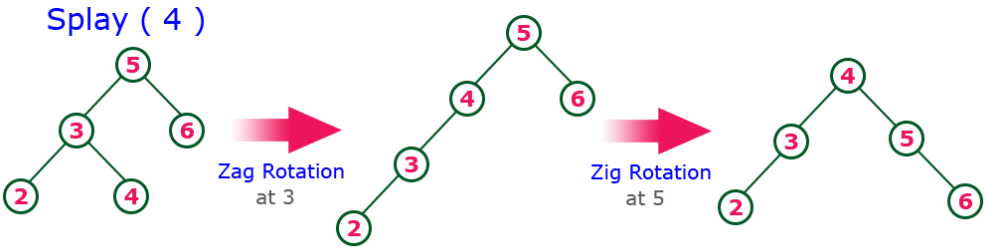




Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...





- **Every Splay tree must be a binary search tree but it is need not to be balanced tree.**

Insertion Operation in Splay Tree

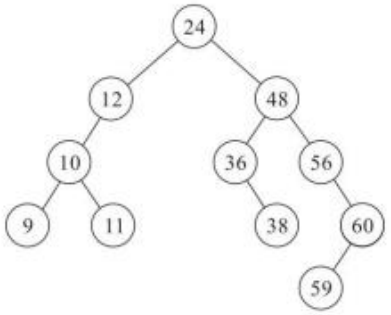
The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

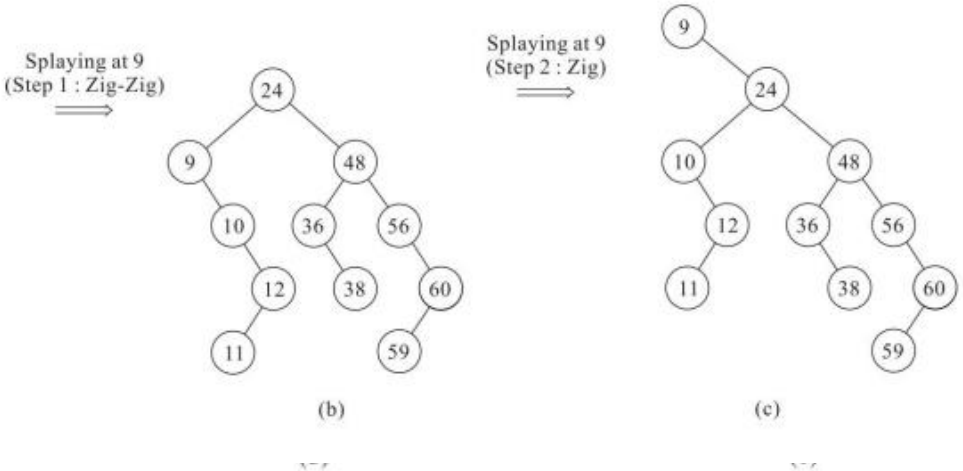
Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

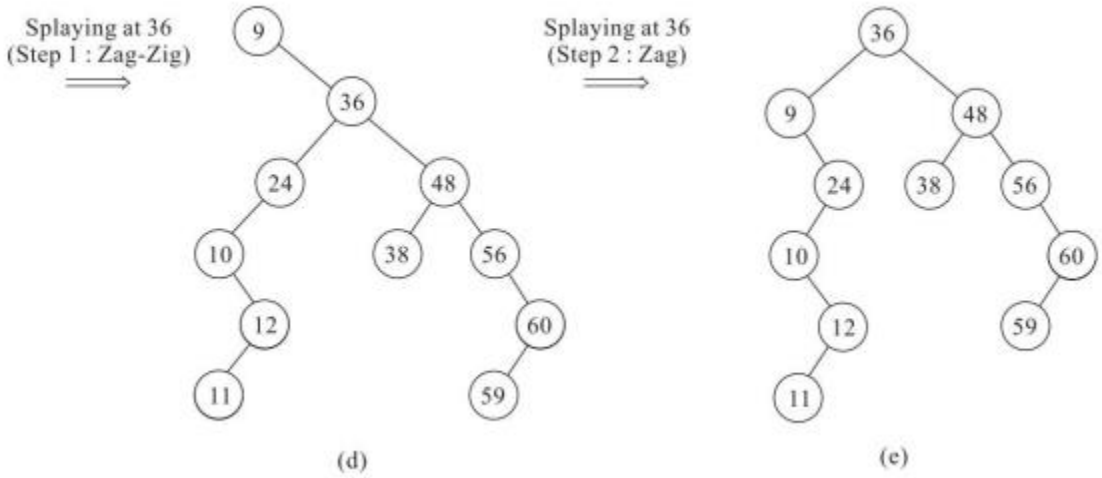
Example: Consider the binary search tree shown in Fig. 12.19(a). Let us attempt splaying the tree at node 9.



(a) a binary search tree



(b) (c)



(d) (e)

Example: Build a splay tree inserting the following elements in the sequence shown: H, Q, A, N, P, O.