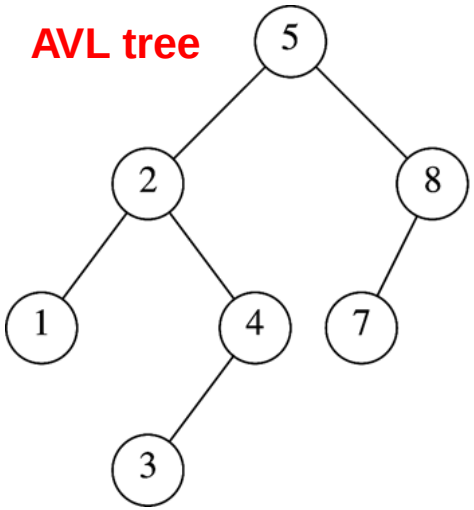
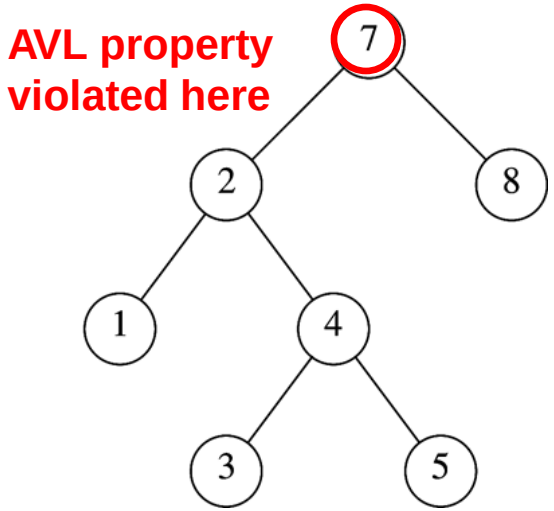
AVL TREE

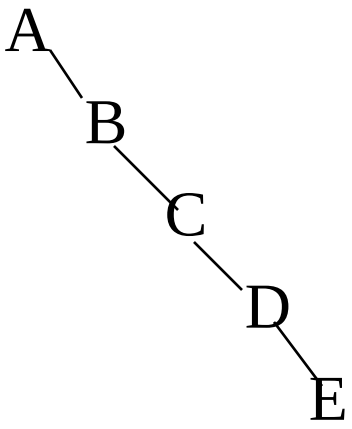
AVL Tree

- An AVL (Adelson-Velskii and Landis 1962) tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.

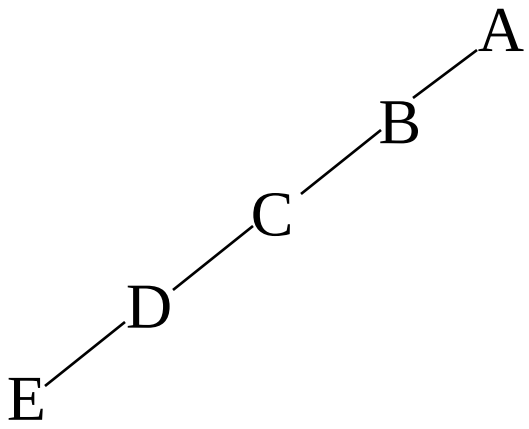




- The efficiency of many important operations on trees is related to the height of the tree –for example searching, insertion and deletion in a BST are all O(height). In general, the relation between the height of the tree and the number of nodes of the tree is O ($\log_2 n$) except in the case of right skewed or left skewed BST in which height is O(n). The right skewed or left skewed BST is one in which the elements in the tree are either on the left or right side of the root node.



Right-skewed

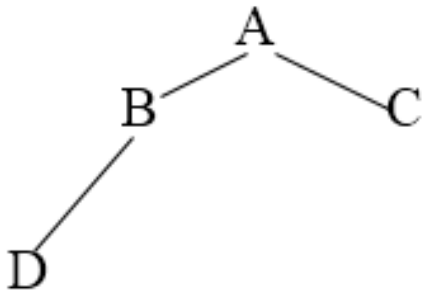


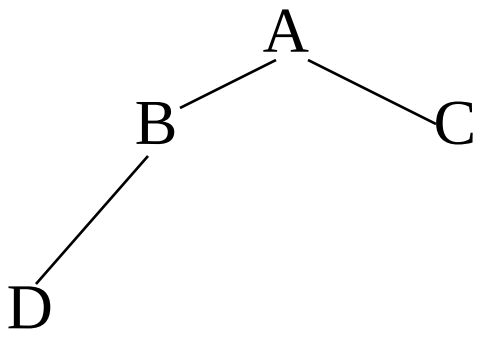
Left-skewed

- For efficiency sake, we would like to guarantee that h remains $O(\log_2 n)$. One way to do this is to force our trees to be height-balanced.

- Method to check whether a tree is height balanced or not is as follows:
  - Start at the leaves and work towards the root of the tree.
  - Check the height of the subtrees(left and right) of the node.
  - A tree is said to be height balanced if the difference of heights of its left and right subtrees of each node is equal to 0, 1 or -1

- Example:
  - Check whether the shown tree is balanced or not





Sol: Starting from the leaf nodes D and C, the height of left and right subtrees of C and D are each 0. Thus their difference is also 0

- Check the height of subtrees of B

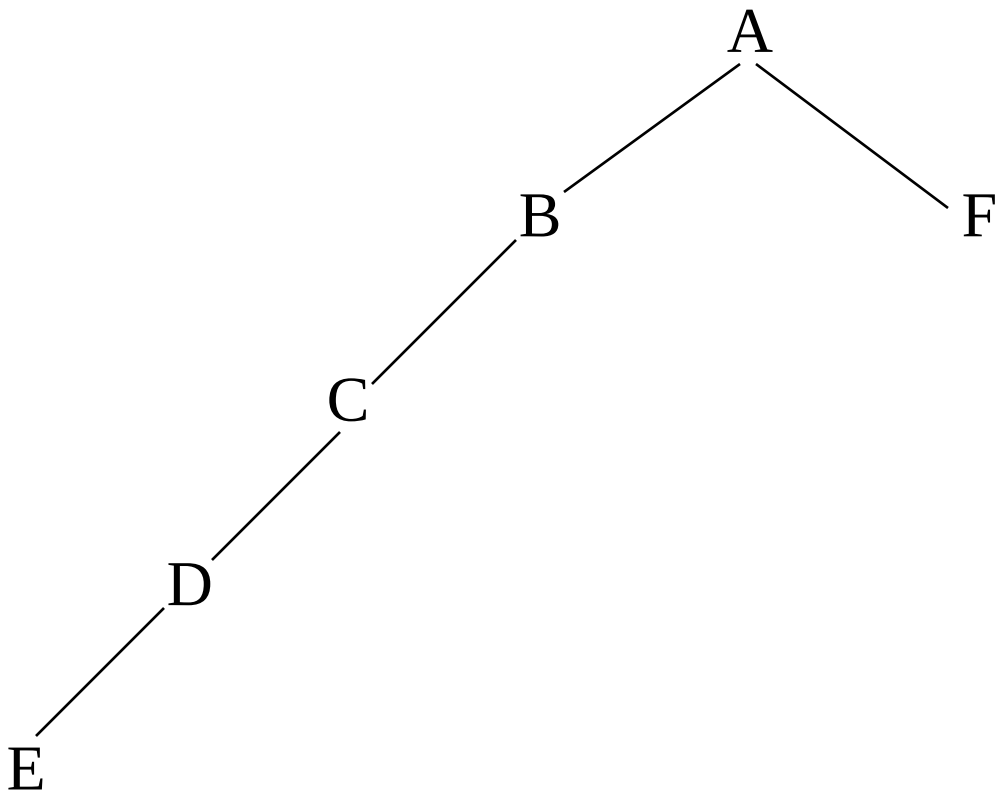
Height of left subtree of B is 1 and height of right subtree of B is 0. Thus the difference of two is 1 Thus B is not perfectly balanced but the tree is still considered to be height balanced.

- Check the height of subtrees of A

Height of left subtree of A is 2 while the height of its right subtree is 1. The difference of two heights still lies within 1.

- Thus for all nodes the tree is a balanced binary tree.

- Check whether the shown tree is balanced or not



Ans) No as node B is not balanced as difference of heights of left and right subtrees is 3-0 i.e more than 1.

Height-balanced Binary tree (AVL Tree)

The disadvantage of a skewed binary search tree is that the worst case time complexity of a search is O(n). In order to overcome this disadvantage, it is necessray to maintain the binary search tree to be of balanced height. Two Russian mathematicians , G.M. Adel and E.M. Landis gave a technique to balance the height of a binary tree and the resulting tree is called AVL tree.

Definition: An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given T^L and T^R to be the left and right subtrees of T and $h(T^L)$ and $h(T^R)$ be the heights of subtrees T^L and T^R respectively, **T^L and T^R are AVL trees and $|h(T^L)-h(T^R)| \leq 1$.**

$|h(T^L)-h(T^R)|$ is also called the balance factor (BF) and for an AVL tree the balance factor of a node can be either -1, 0 or 1

An AVL search tree is a binary search tree which is an AVL tree.

- A node in a binary tree that does not contain the BF of 0, 1 or -1, it is said to be unbalanced. If one inserts a new node into a balanced binary tree at the leaf, then the possible changes in the status of the node are as follows:

- The node was either left or right heavy and has now become balanced. A node is said to be left heavy if number of nodes in its left subtree are one more than the number of nodes in its right subtree.. In other words, the difference in heights is 1. Similar is the case with right heavy node where number of nodes in right subtree are one more than the number of nodes in left subtree

- The node was balanced and has now become left or right heavy

- The node was heavy and the new node has been inserted in the heavy subtree, thus creating an unbalanced subtree. Such a node is called a **critical node.**

Rotations- Inserting an element in an AVL search tree in its first phase is similar to that of the one used in a binary search tree. However, if after insertion of the element, the balance factor of any node in a binary search tree is affected so as to render the binary search tree unbalanced, we resort to techniques called Rotations to restore the balance of the search tree.

- To perform rotations, it is necessary to identify the specific node A whose BF (balance factor) is neither 0,1 or -1 and which is nearest ancestor to the inserted node on the path from inserted node to the root.
- The rebalancing rotations are classified as LL, LR, RR and RL based on the position of the inserted node with reference to A

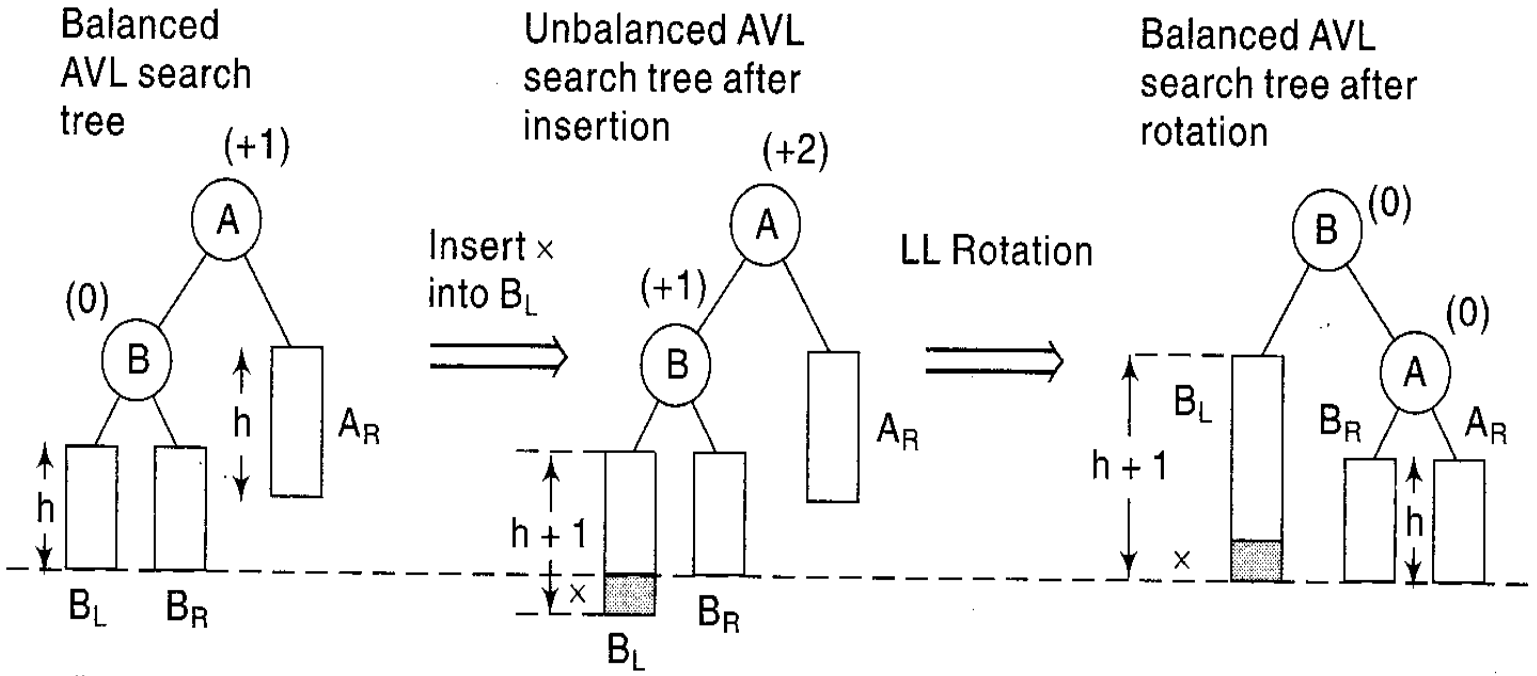
LL rotation: Inserted node in the left subtree of the left subtree of A

RR rotation: Inserted node in the right subtree of the right subtree of A

LR rotation: Inserted node in the right subtree of the left subtree of A

RL rotation: Inserted node in the left subtree of the right subtree of A

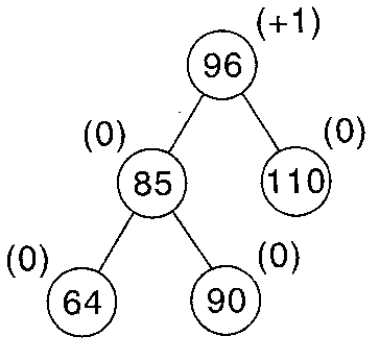
• **LL Rotation-** This rotation is done when the element is inserted in the left subtree of the left subtree of A. To rebalance the tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right child and B_R and A_R to be the left and right subtrees of A. The rotation results in a balanced tree.



B_L : Left subtree of B
B_R : Right subtree of B
A_R : Right subtree of A
h : Height

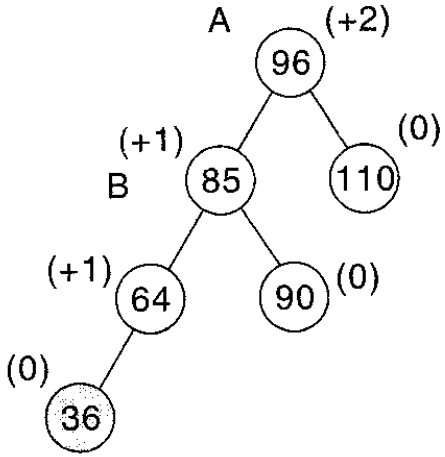
Give label "A" to violating node. Give label "B" to that child node of "A" where the insertion is done.
Then label the edges by verifying:

1) Insertion done on which sub-tree of "B" ?
2) "B" is lying on which side of "A" ?



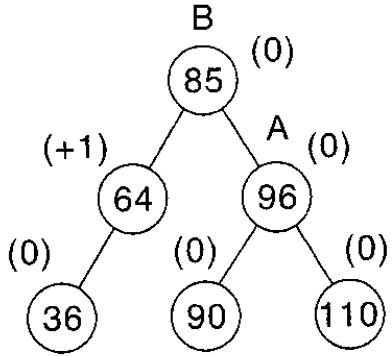
Initial AVL search tree
(a)

Insert 36



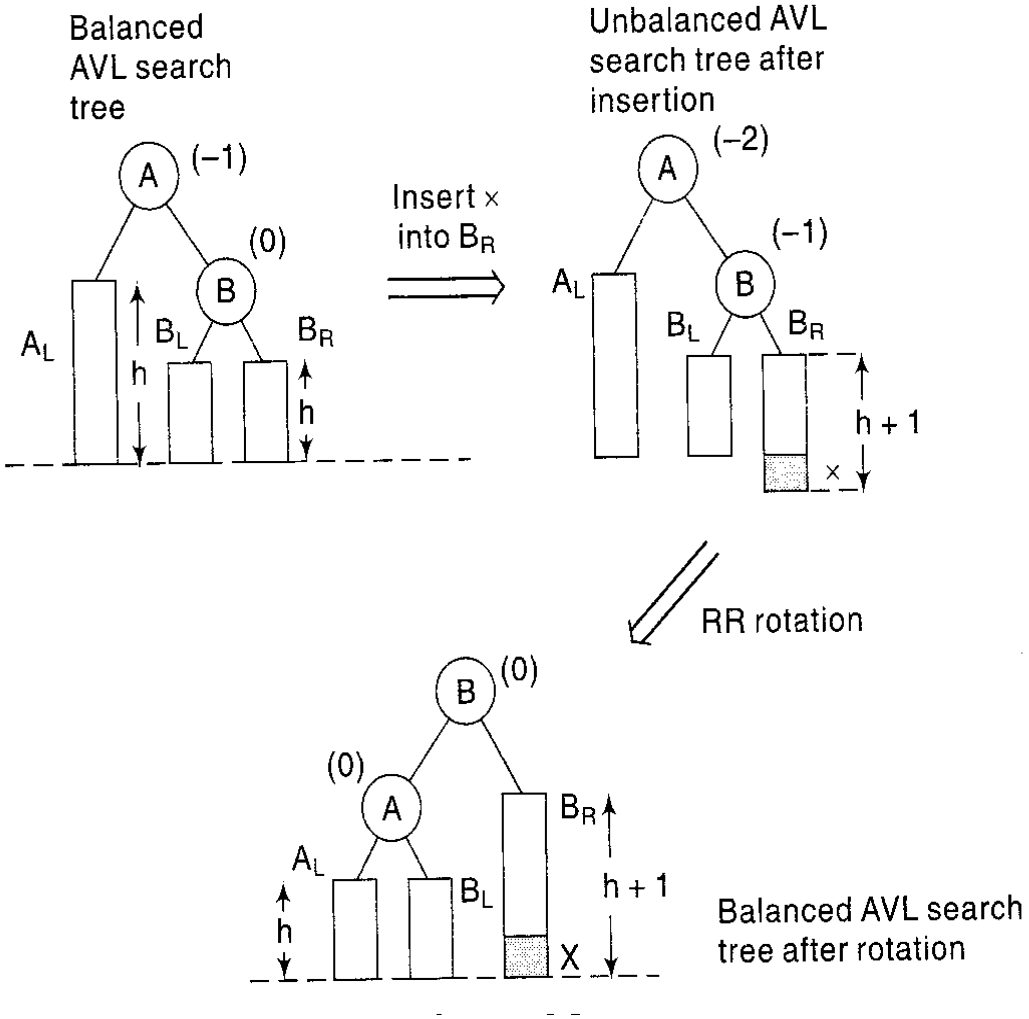
Unbalanced
AVL search tree
(b)

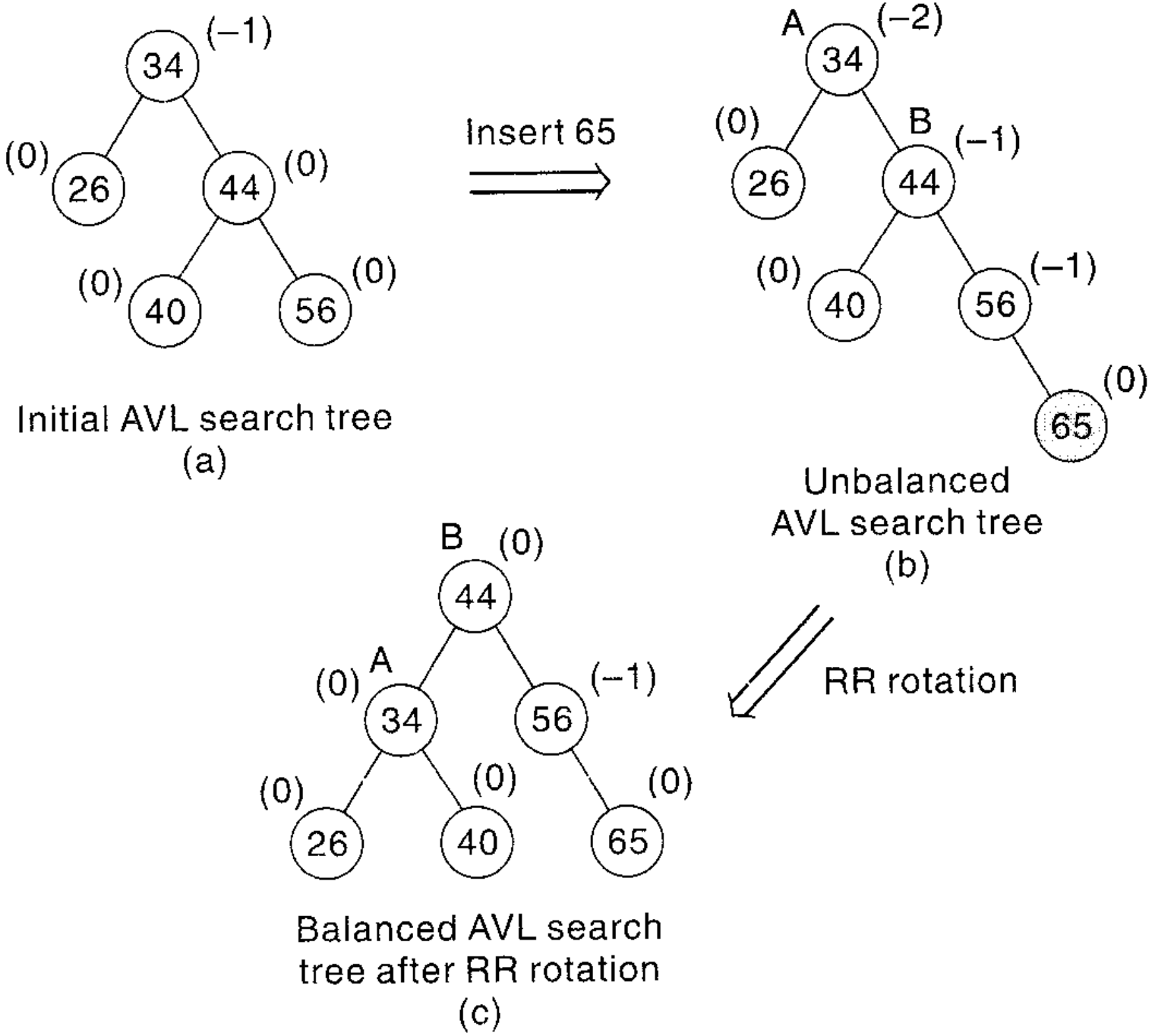
LL rotation



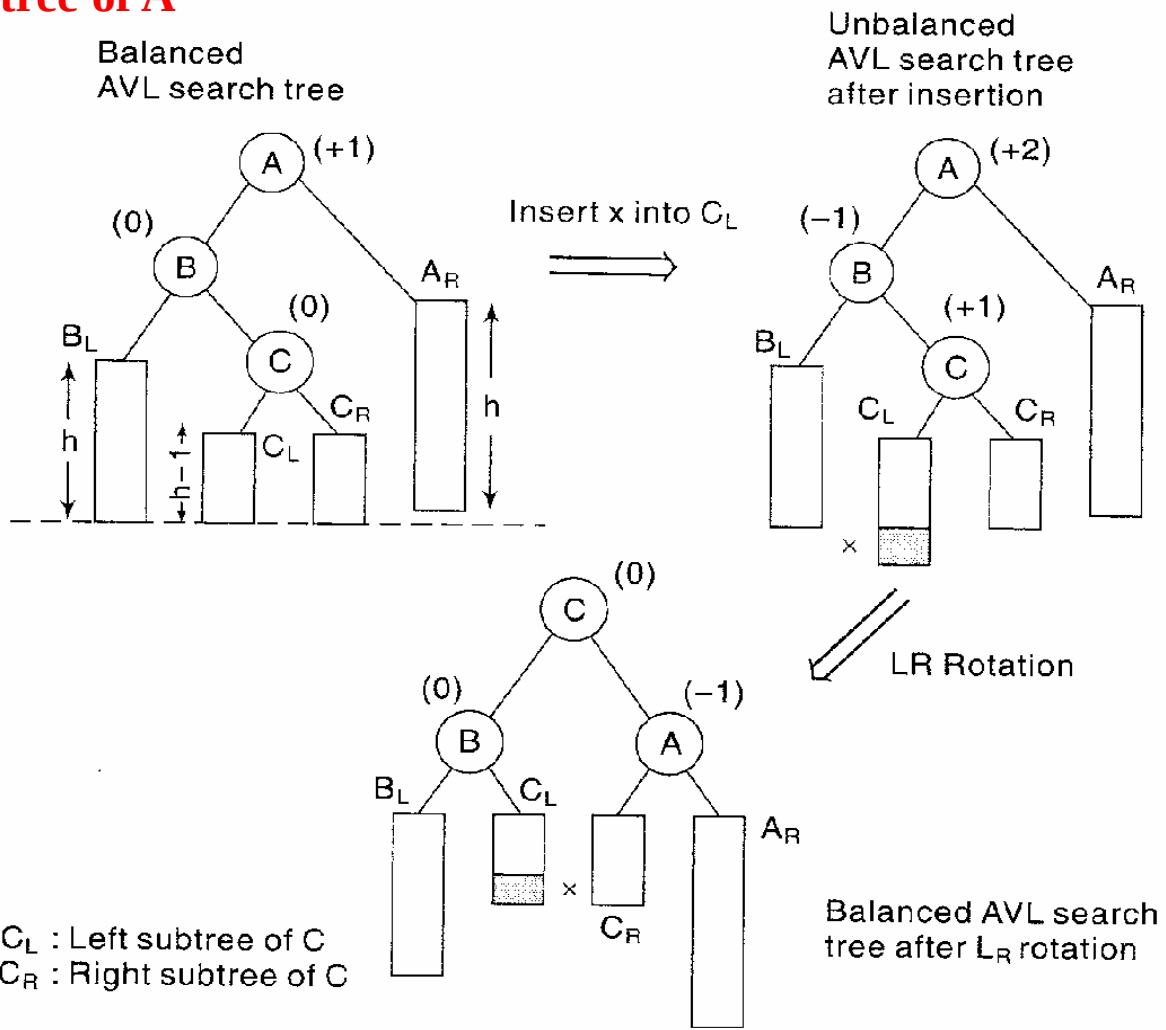
Balanced AVL search
tree after LL rotation
(c)

- **RR Rotation**-This rotation is applied if the new element is inserted right subtree of right subtree of A. The rebalancing rotation pushes B upto the root with A as its left child and B_R as its right subtree and A_L and B_L as the left and right subtrees of A

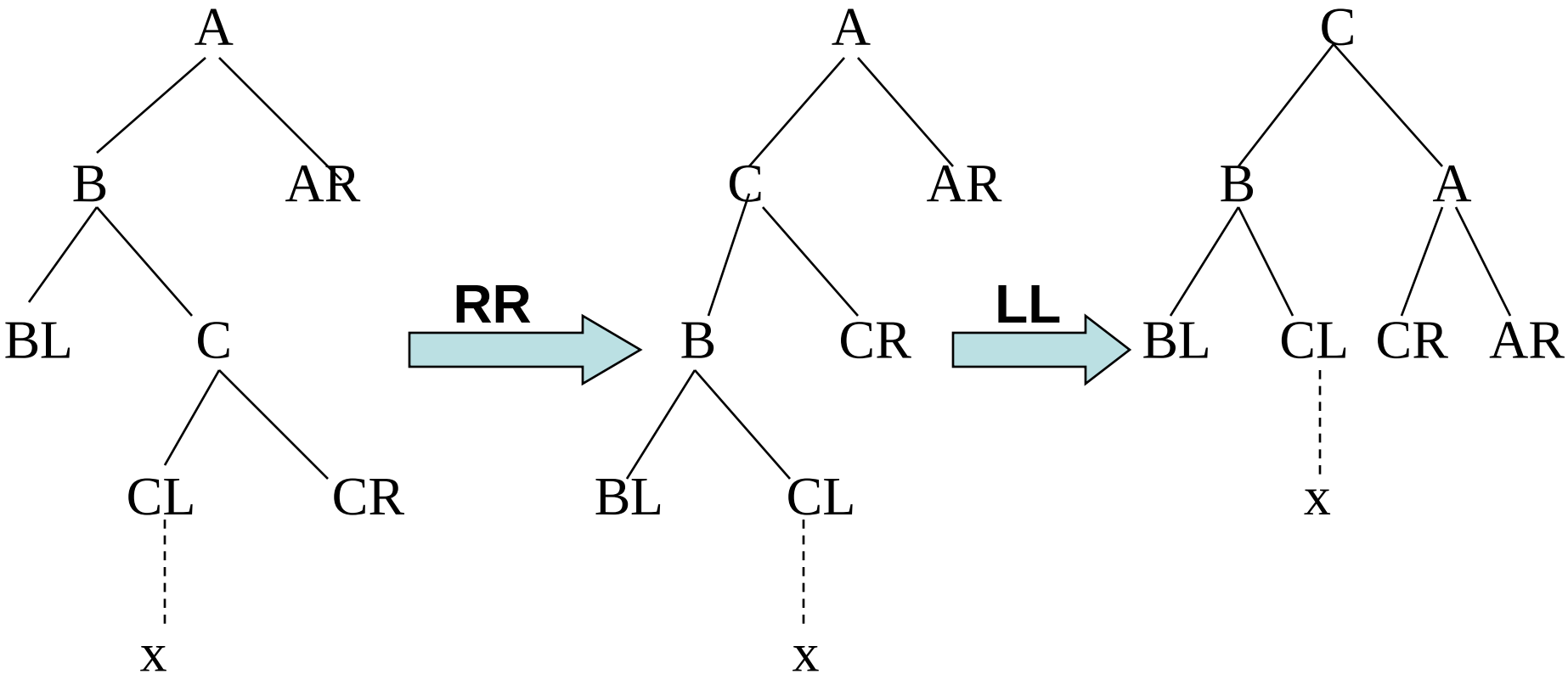



(−1)
34
(0)
26
44
(0)
(0)
40
56
(0)
Initial AVL search tree
(a)
Insert 65
A
(−2)
34
(0)
B
(−1)
26
44
(0)
40
56
(−1)
(0)
65
Unbalanced
AVL search tree
(b)
RR rotation
B
(0)
44
(0)
A
34
(−1)
56
(0)
26
(0)
40
(0)
65
Balanced AVL search
tree after RR rotation
(c)

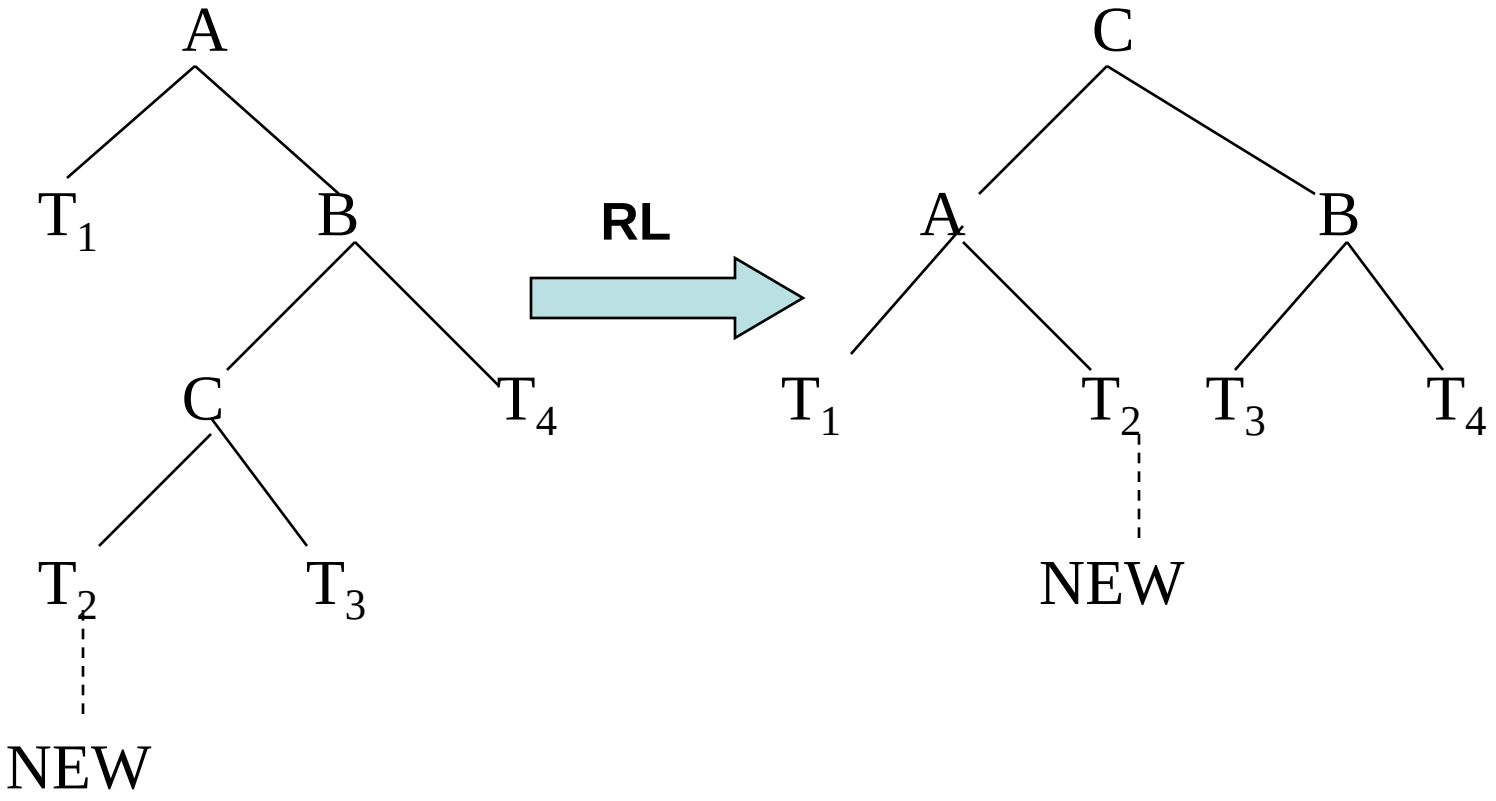
LR and RL rotations- The balancing methodology of LR and RL rotations are similar in nature but are mirror images of one another. Amongst the rotations, **LL and RR rotations** are called as **single rotations** and **LR and RL** are known as **double rotations** since LR is accomplished by RR followed by LL rotation and RL can be accomplished by LL followed by RR rotation. **LR rotation is applied when the new element is inserted in right subtree of the left subtree of A. RL rotation is applied when the new element is inserted in the left subtree of right subtree of A**



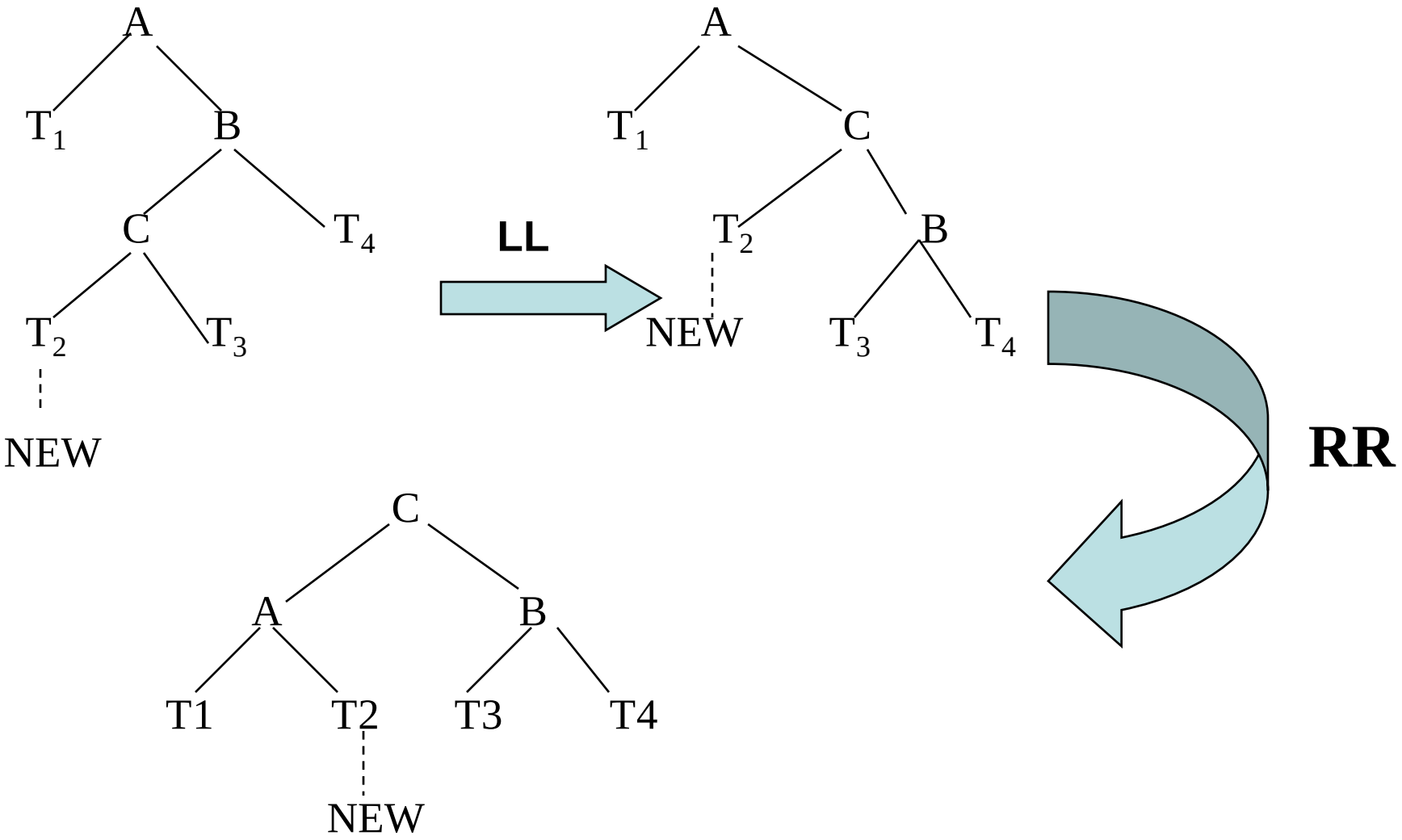
LR Rotation- this rotation is a combination of RR rotation followed by LL rotation.

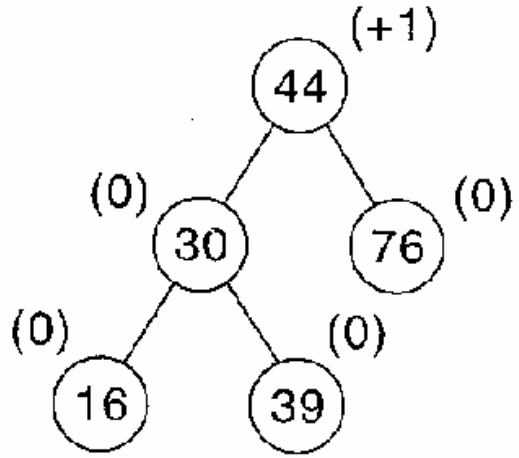


RL Rotation-This rotation occurs when the new node is inserted in left subtree of right subtree of A. It's a combination of LL followed by RR



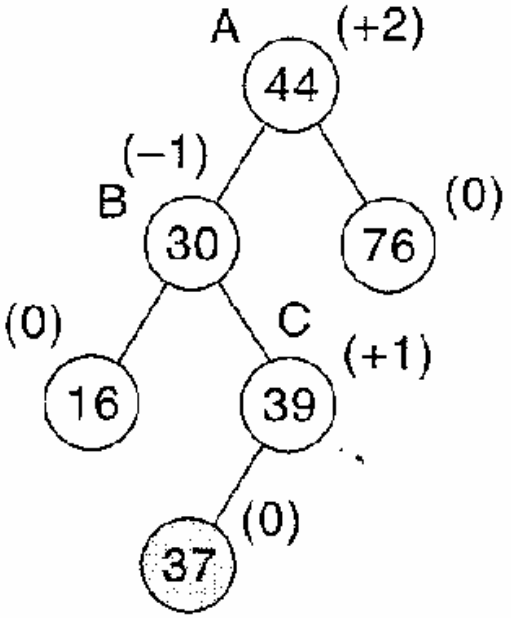
- RL Rotation- This rotation occurs when the new node is inserted in right subtree of left subtree of A.





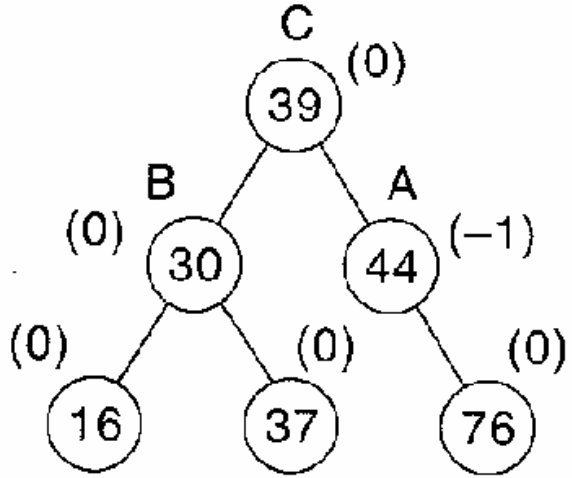
Initial AVL search tree
(a)

Insert 37



Unbalanced
AVL search tree
(b)

LR rotation



Balanced AVL search
tree after rotation
(c)

Problem: Construct an AVL search tree by inserting the following elements in the order of their occurrence

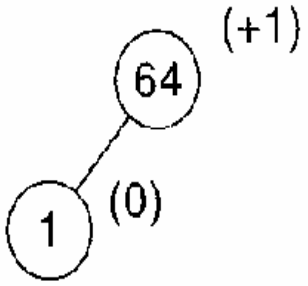
64, 1, 14, 26, 13, 110, 98, 85

Problem: Construct an AVL search tree by inserting the following elements in the order of their occurrence

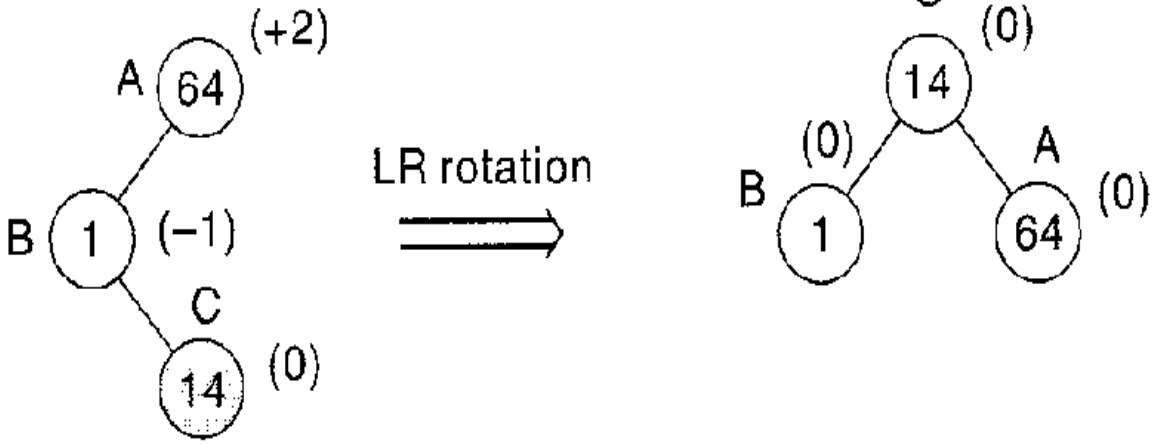
64, 1, 14, 26, 13, 110, 98, 85

Sol:

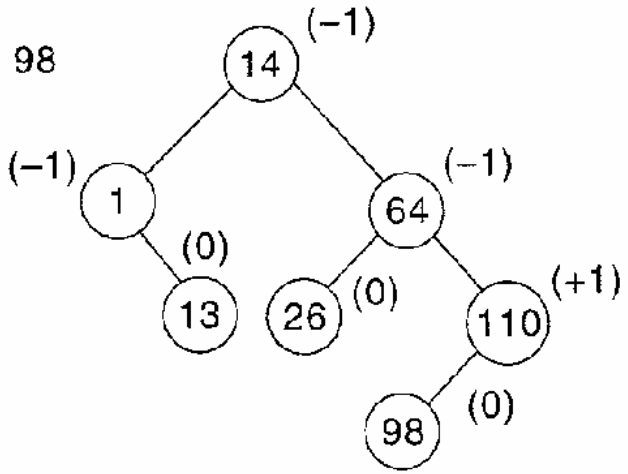
Insert 64, 1



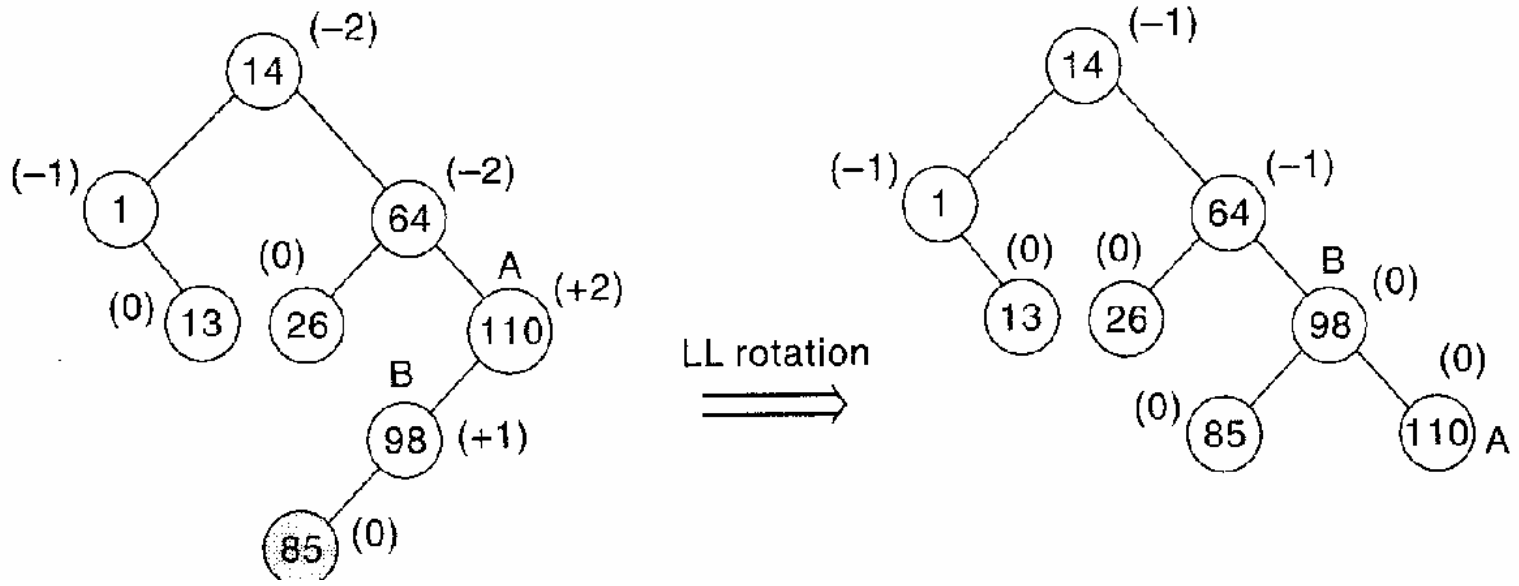
Insert 14



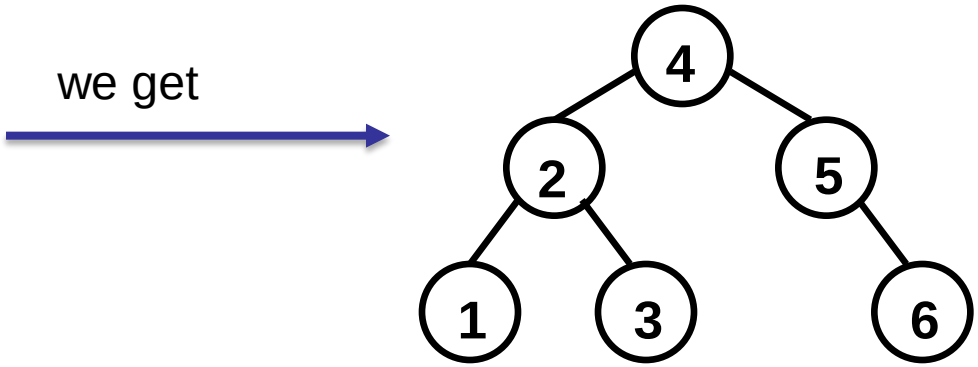
Insert 26, 13, 110, 98



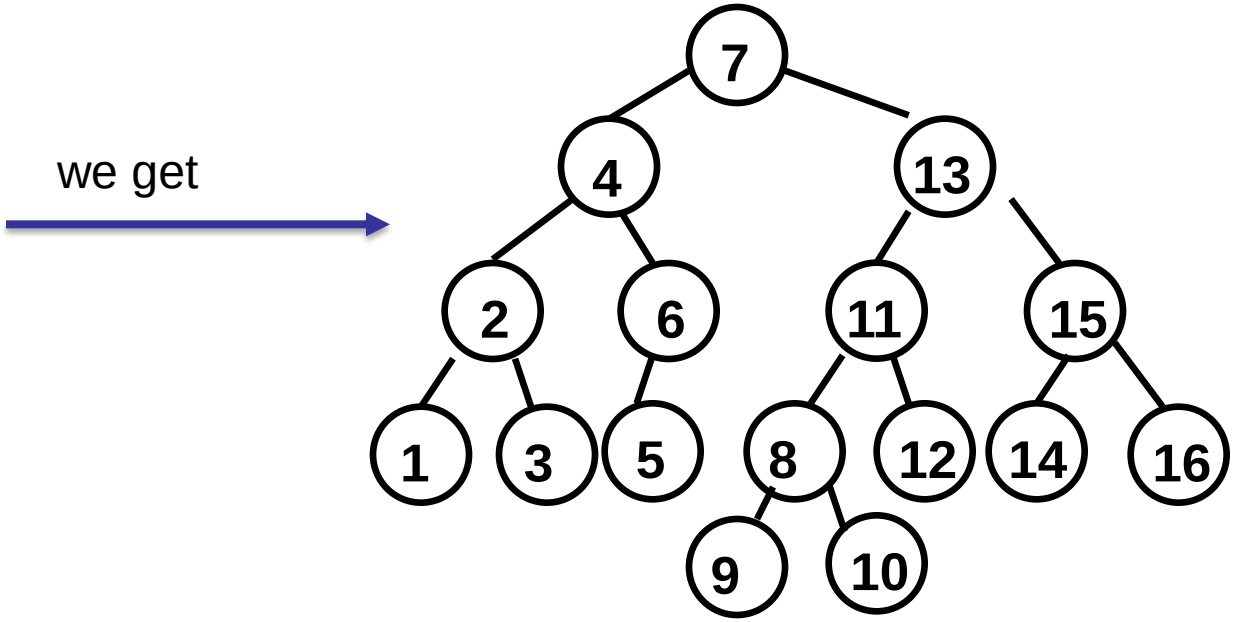
Insert 85



- Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL Tree



- If we continue to insert 7, 16, 15, 14, 13, 12, 11, 10, 8, 9



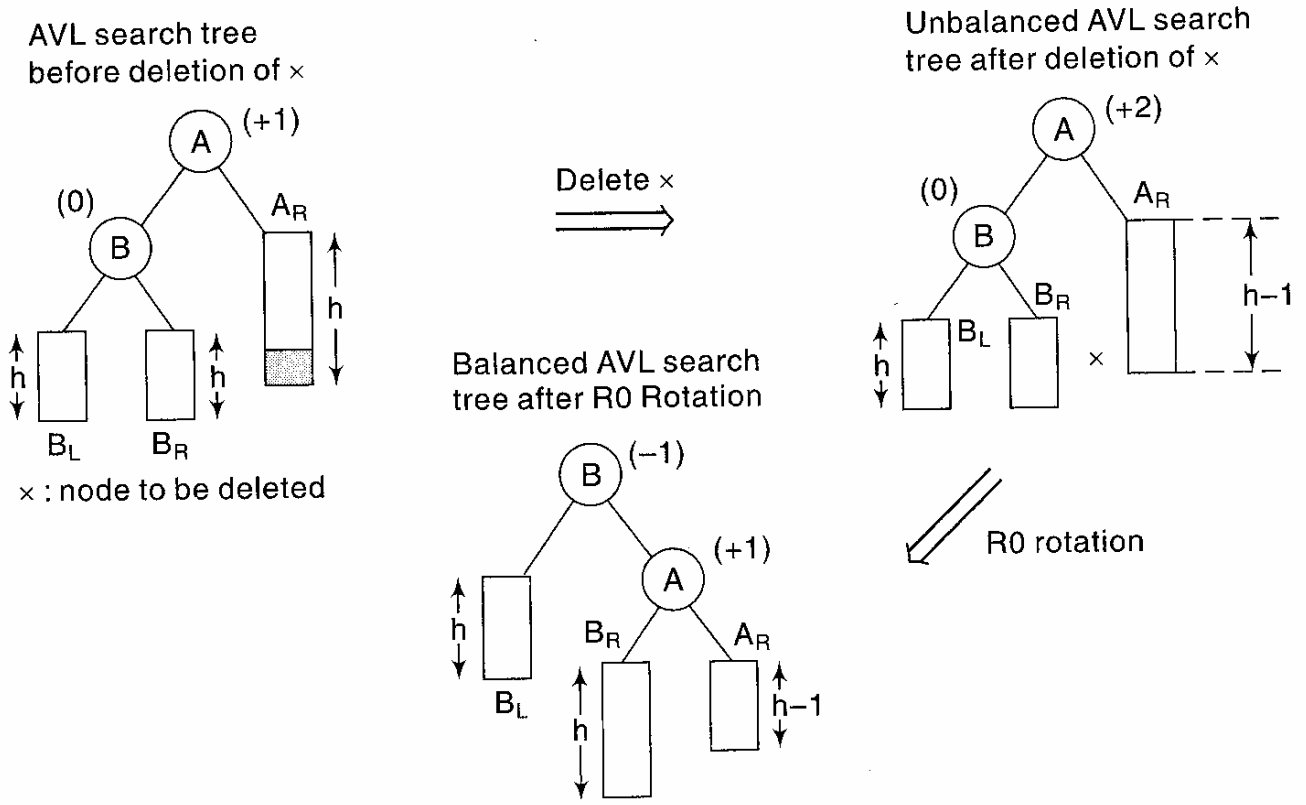
Deletion in an AVL search Tree

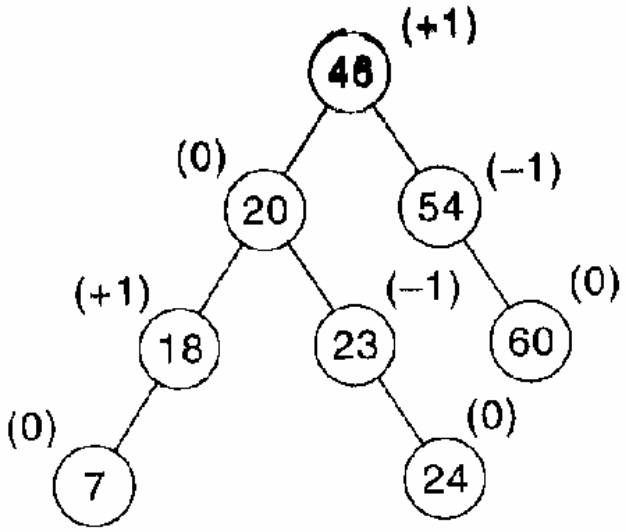
- The deletion of element in AVL search tree leads to imbalance in the tree which is corrected using different rotations. The rotations are classified according to the place of the deleted node in the tree.

- On deletion of a node X from AVL tree, let A be the closest ancestor node on the path from X to the root node with balance factor of +2 or -2 .To restore the balance, the deletion is classified as L or R depending on whether the deletion occurred on the left or right sub tree of A.

- Depending on value of BF(B) where B is the root of left or right sub tree of A, the R or L rotation is further classified as R0, R1 and R-1 or L0, L1 and L-1. The L rotations are the mirror images of their corresponding R rotations.

R0 Rotation- This rotation is applied when the BF of B is 0 after deletion of the node

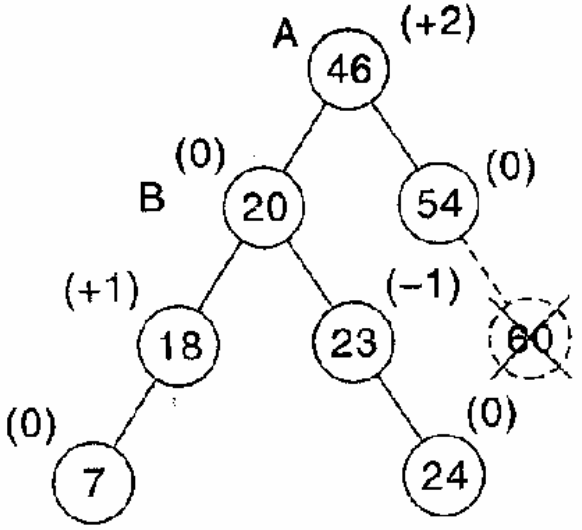




AVL search tree
before deletion

(a)

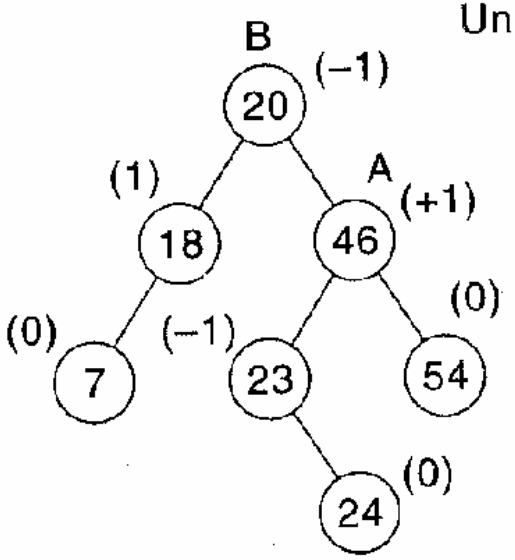
Delete 60



Unbalanced AVL search tree
after deletion of 60

(b)

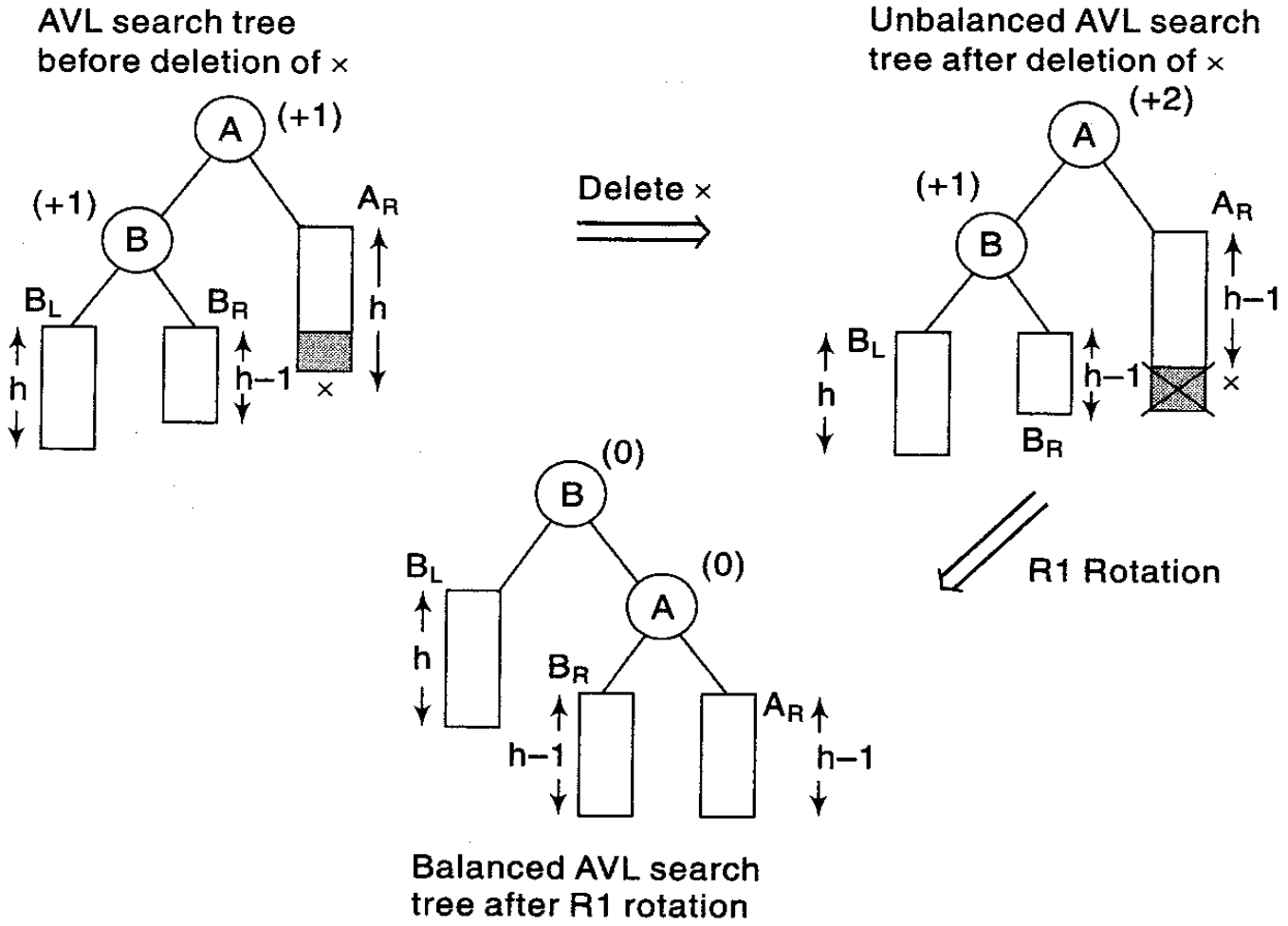
R0 rotation

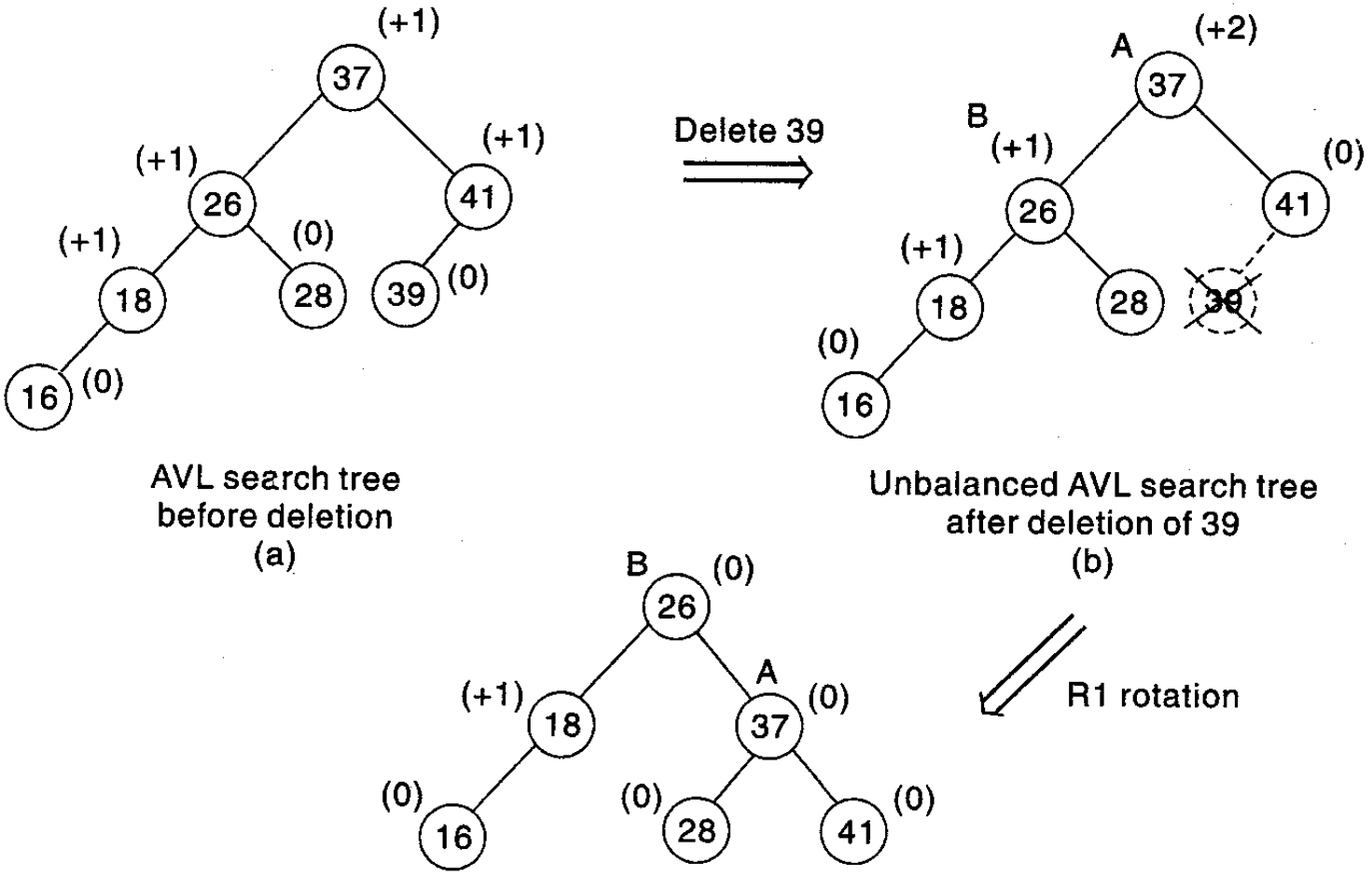


Balanced AVL search tree after R0 rotation

(c)

R1 Rotation- This rotation is applied when the BF of B is 1



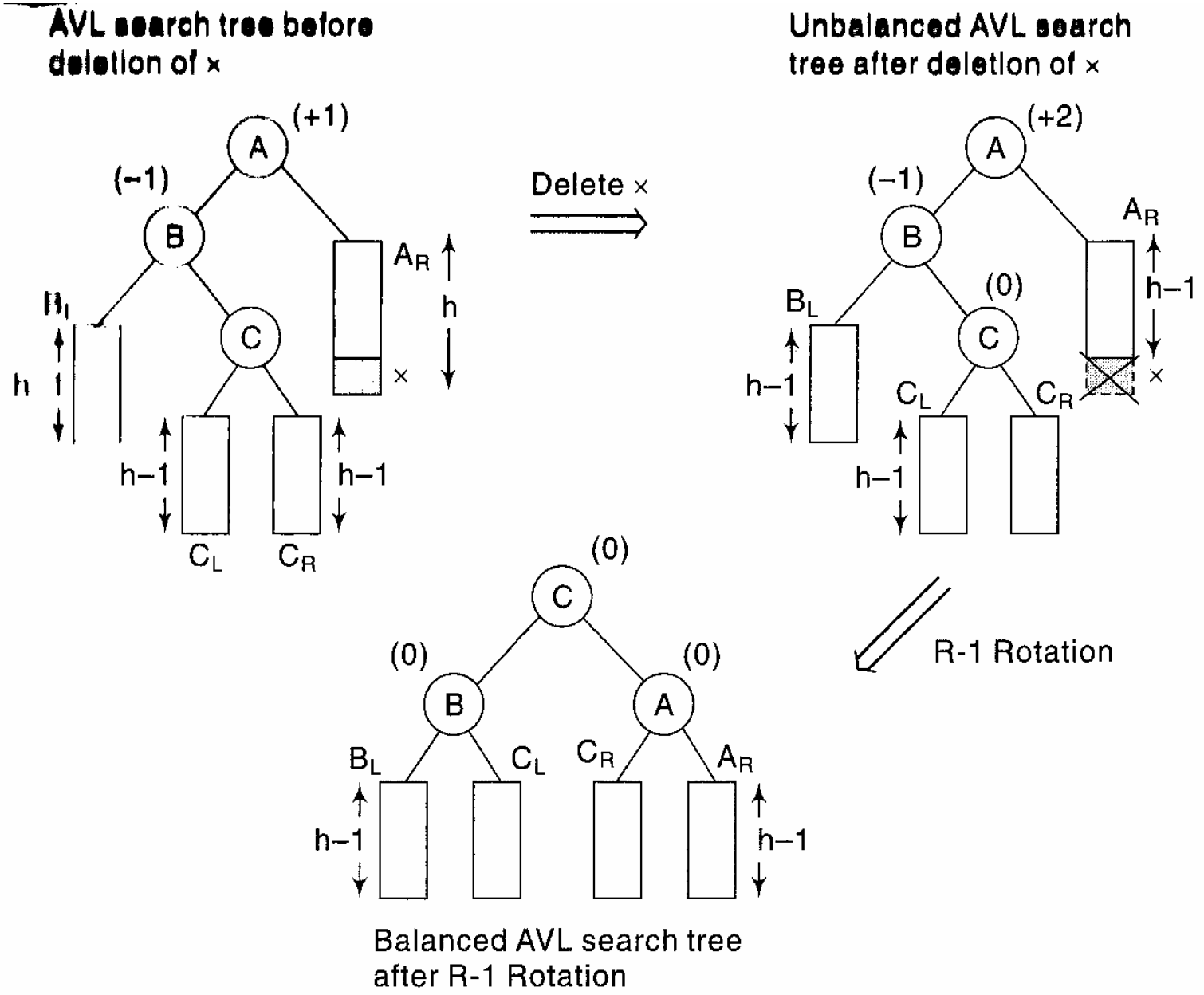


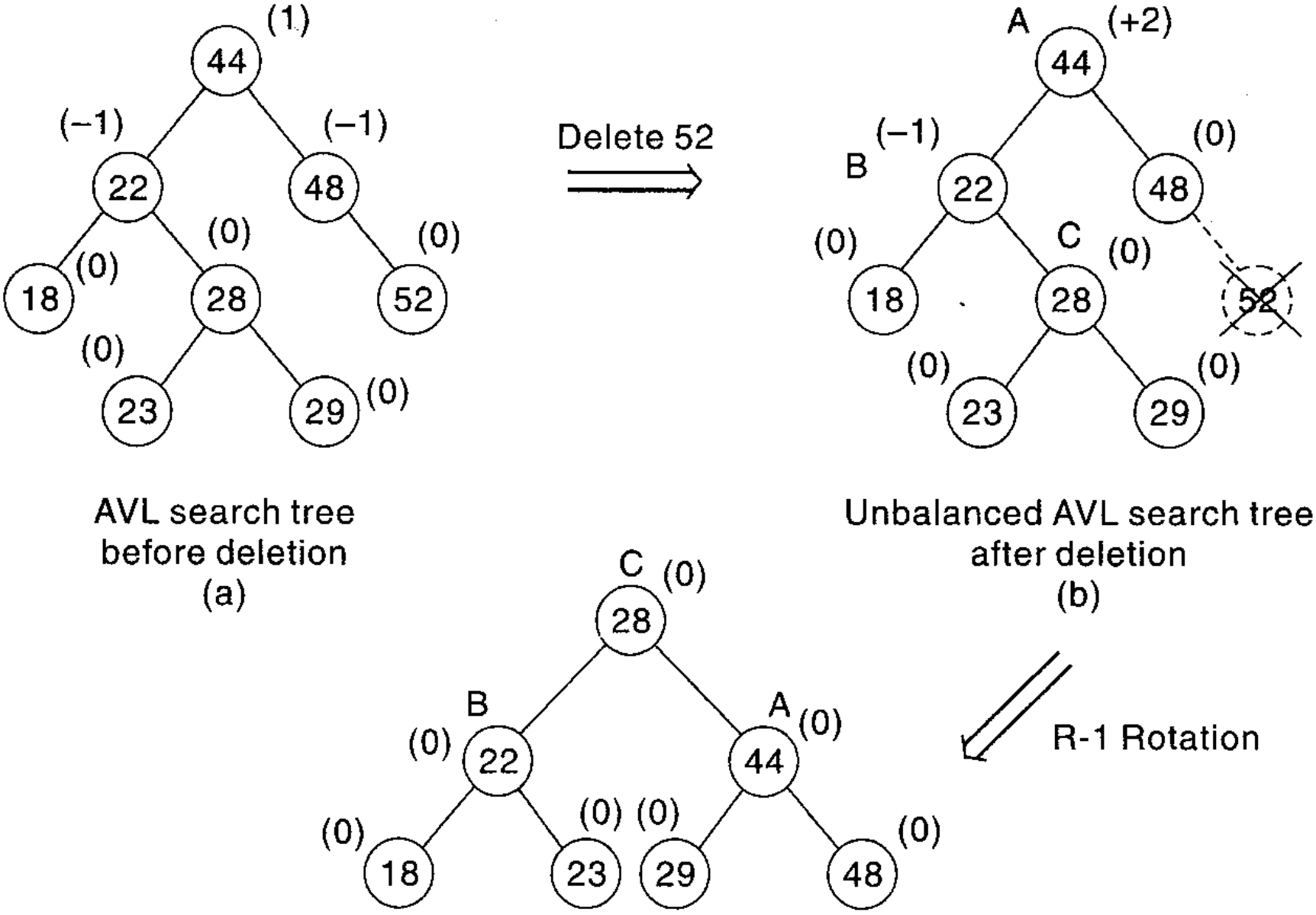
AVL search tree
before deletion
(a)

Unbalanced AVL search tree
after deletion of 39
(b)

Balanced AVL search tree after R1 rotation
(c)

R-1 Rotation- This rotation is applied when the BF of B is -1





AVL search tree
before deletion
(a)

Unbalanced AVL search tree
after deletion
(b)

Balanced AVL search tree after R-1 Rotation
(c)

- L rotations are the mirror images of R rotations. Thus L0 will be applied when the node is deleted from the left subtree of A and the BF of B in the right subtree is 0

- Similarly, L1and L-1 will be applied on deleting a node from left subtree of A and if the BF of root node of right subtree of A is either 1 or -1 respectively.