ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT IV : TREES

PART - 2

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

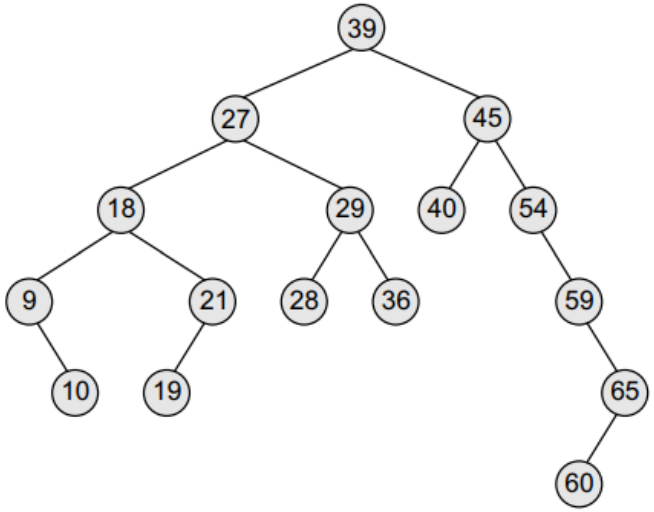
Contents

Trees:

- Trees: Basic Terminology in Trees,
- Binary Trees-Properties,
- Representation of Binary Trees using Arrays and Linked lists.
- Binary Search Trees- Basic Concepts,
- BST Operations: Insertion, Deletion, Tree Traversals
- Applications-Expression Trees,
- Heap Sort,
- Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations.

Binary Search Trees

- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value

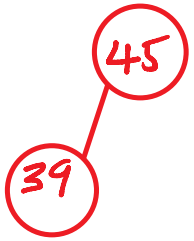


Creating a binary search tree

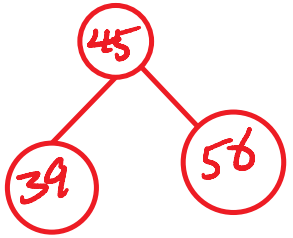
- Consider the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

step1:

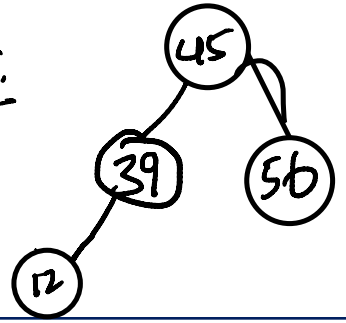
step2:



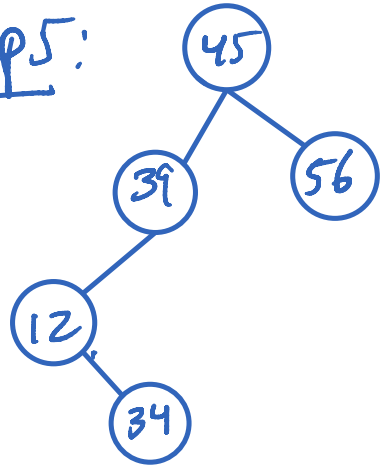
step3:



step4:

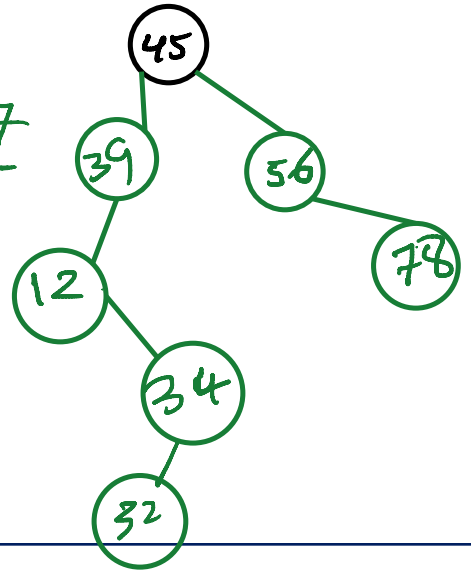


step5:

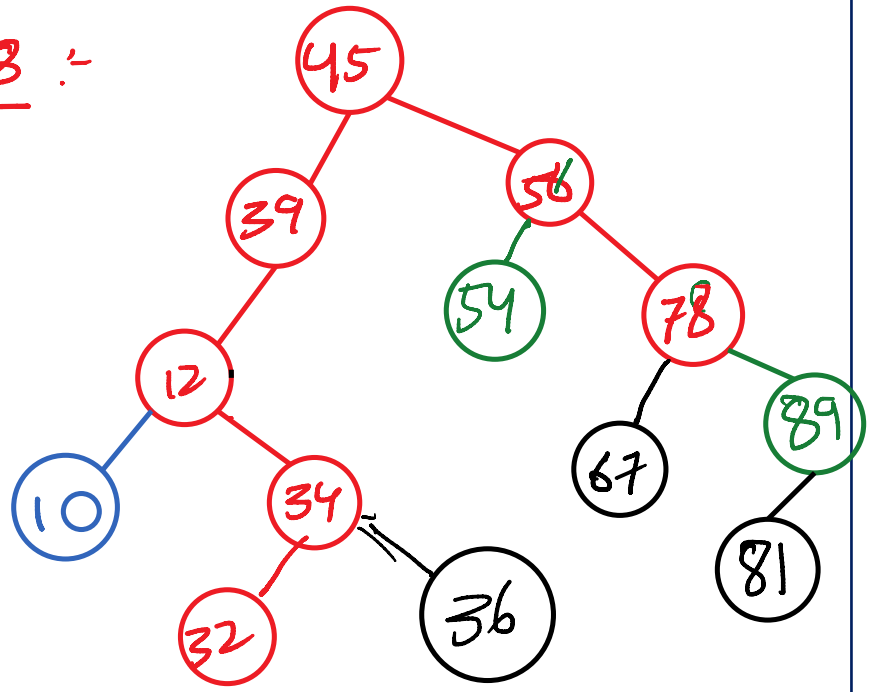


step6:

step7

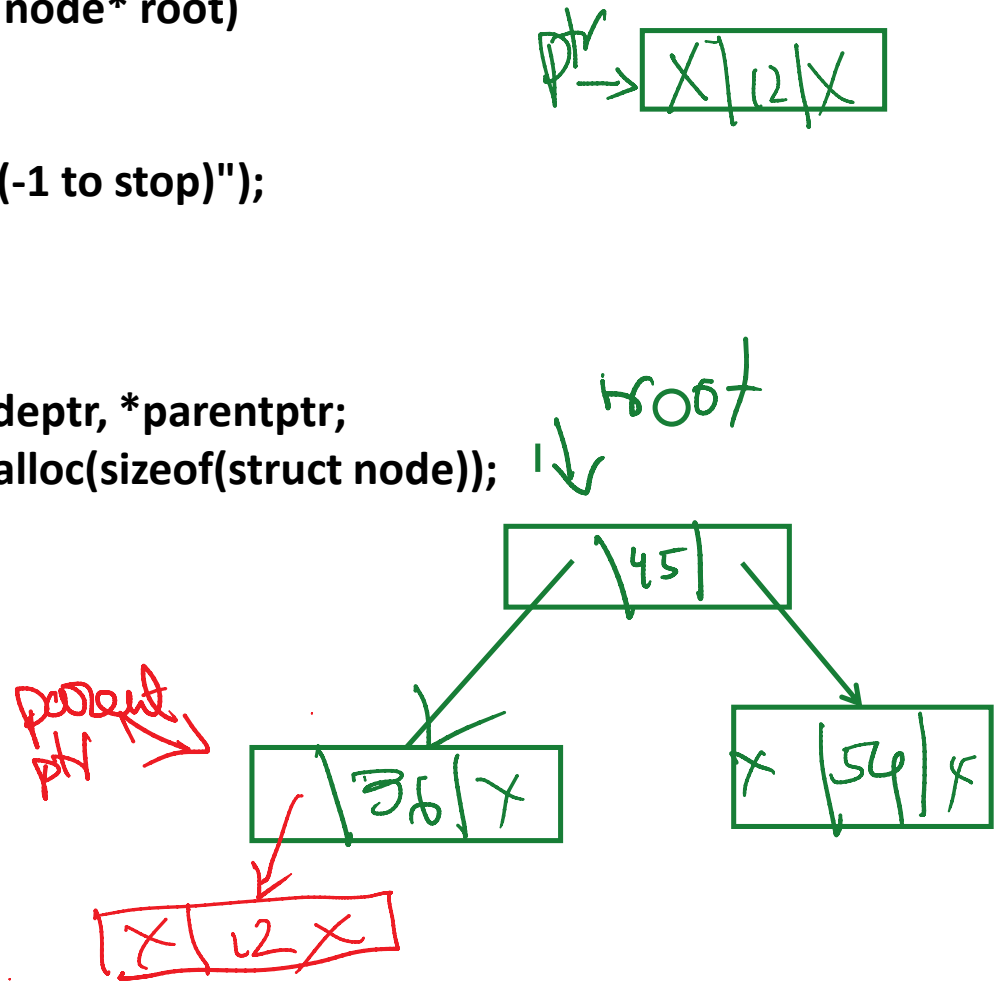


step8 :-



C program to Create a BST

```
struct node* create(struct node* root)
{
        int val;
        printf("\n enter data (-1 to stop)");
        scanf("%d",&val);
        while(val != -1)
        {
        struct node *ptr, *nodeptr, *parentptr;
       ptr = (struct node*)malloc(sizeof(struct node));
        ptr->data = val;
        ptr->left = NULL;
        ptr->right = NULL;
        if(root==NULL)
        {
                root=ptr;
        }
```




```
else        {
        parentptr=NULL;
        nodeptr=root;
        while(nodeptr!=NULL)
        {
                parentptr=nodeptr;
                if(val<nodeptr->data)
                        nodeptr=nodeptr->left;
                else
                        nodeptr = nodeptr->right;
        }
        if(val<parentptr->data)
                parentptr->left = ptr;
        else
                parentptr->right = ptr;
}
printf("\n enter data (-1 to stop)");
scanf("%d", &val);
}       return root;    }
```

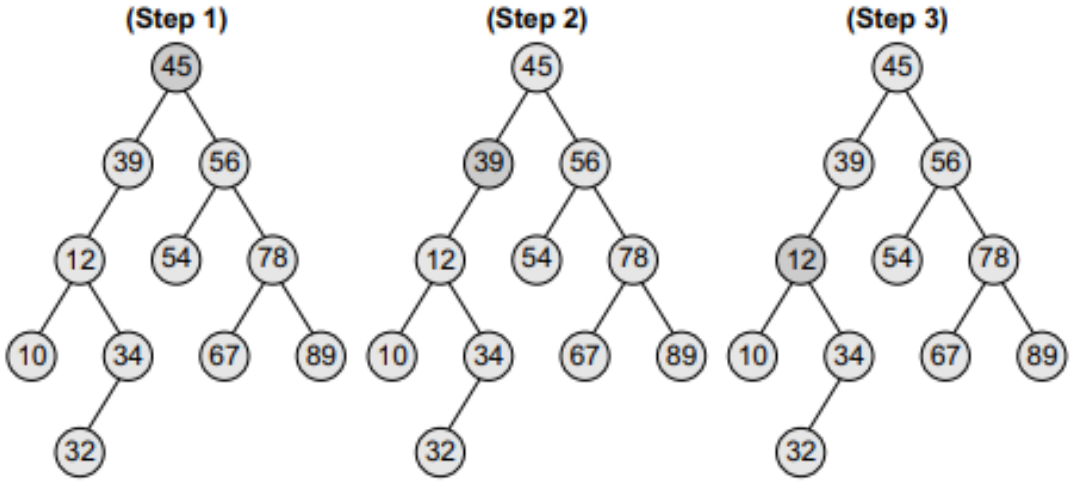
OPERATIONS ON BINARY SEARCH TREES

- Searching for a Node in a Binary Search Tree
- Inserting a New Node in a Binary Search Tree
- Deleting a Node from a Binary Search Tree
  1. Deleting a Node that has No Children
  2. Deleting a Node with One Child
  3. Deleting a Node with Two Children
- Determining the Height of a Binary Search Tree
- Determining the Number of Nodes
  1. Determining the Number of Internal Nodes
  2. Determining the Number of External Nodes
- Finding the Mirror Image of a Binary Search Tree
- Deleting a Binary Search Tree
- Finding the Smallest Node in a Binary Search Tree
- Finding the Largest Node in a Binary Search Tree

Searching for a Value in a BST

- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the node, it should be recursively called on the right child node.

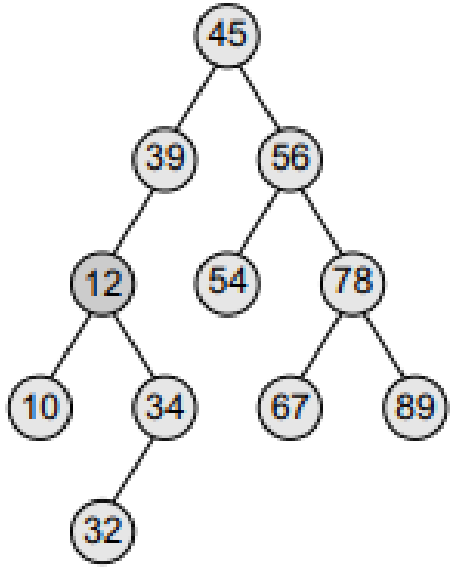
val = 12




```
if (root == NULL)
if (val == root->data)
else if (val < root->data)
else if (val > root->data)
```

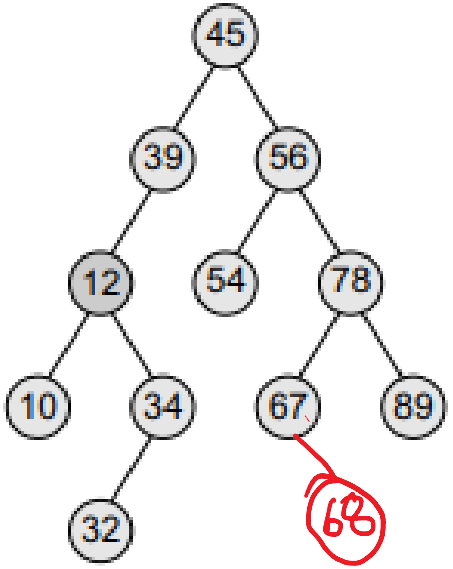
Algorithm to search for a given value in a binary search tree:

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```



Inserting a New Node in a Binary Search Tree

- The insert function is used to add a new node with a given value at the correct position in the binary search tree.
- Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.



val = 68

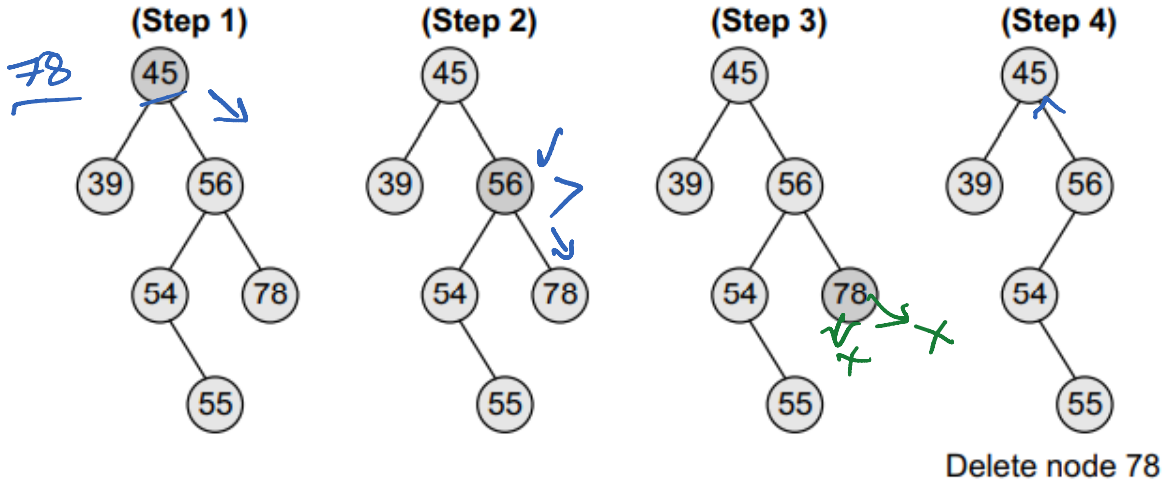
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Deleting a Value from a BST

- The delete function deletes a node from the binary search tree.
- However, care should be taken that the properties of the BSTs do not get violated and nodes are not lost in the process.
- The deletion of a node involves any of the three cases.

Case 1: Deleting a node that has no children.

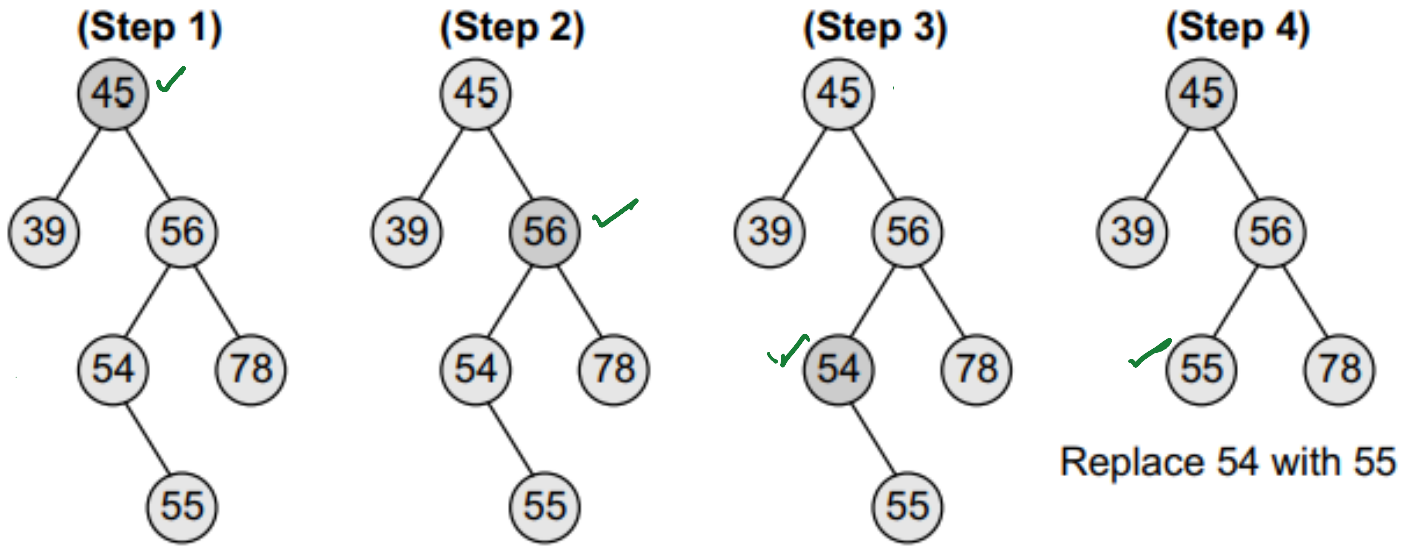
For example, deleting node 78 in the tree below. we can simply remove this node without any issue. This is the simplest case of deletion.



Deleting a Value from a BST

Case 2: Deleting a node with one child (either left or right).

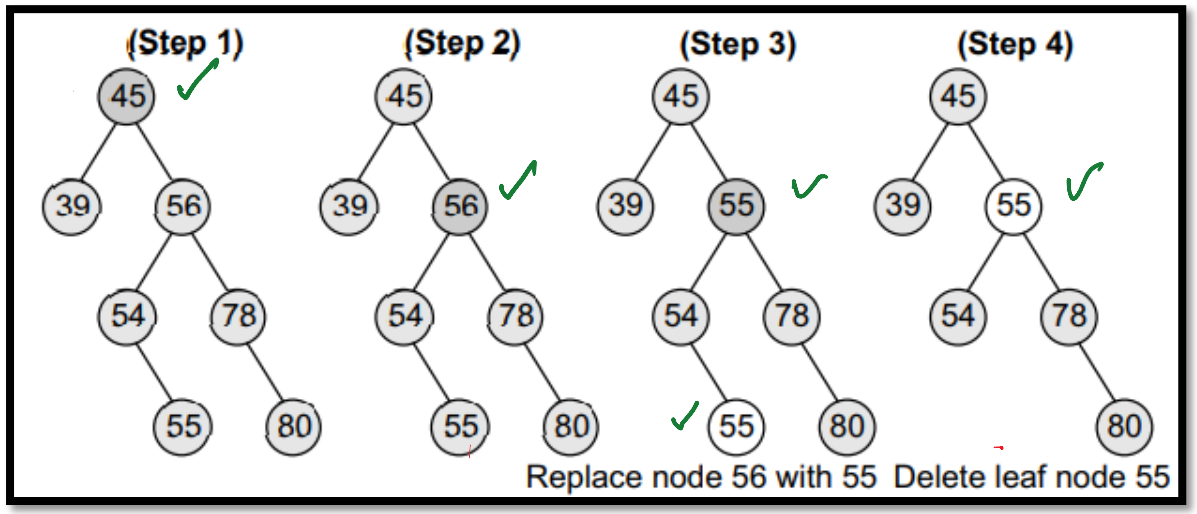
- To handle the deletion, the node's child is set to be the child of the node's parent.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.
- For example, deleting node 54 in the tree below.

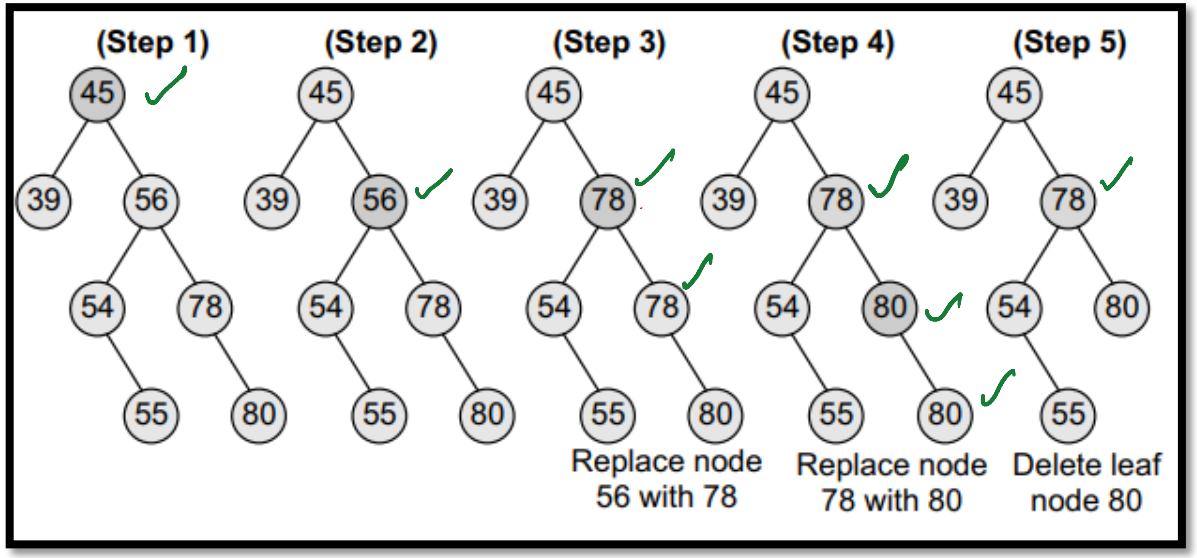


Deleting a Value from a BST

Case 3: Deleting a node with two children.

- To handle this case of deletion, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
- The in-order predecessor or the successor can then be deleted using any of the above cases.
- Deleting node 54 from the given binary search tree



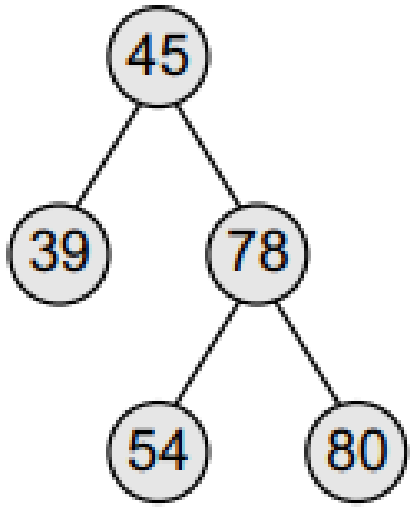


Algorithm to delete a given value from a binary search tree:

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
         IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Determining the Height of a Binary Search Tree

- In order to determine the height of a binary search tree, we calculate the height of the left sub-tree and the right sub-tree. Whichever height is greater, 1 is added to it.
- For example, if the height of the left sub-tree is greater than that of the right sub-tree, then 1 is added to the left sub-tree, else 1 is added to the right sub-tree.
- In the below Fig., the height of the right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 2 + 1 = 3.

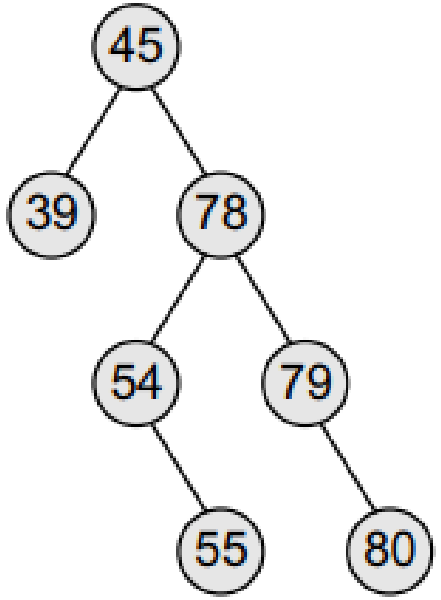


- **Algorithm**

```
Height (TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
         SET LeftHeight = Height(TREE -> LEFT)
         SET RightHeight = Height(TREE -> RIGHT)
          IF LeftHeight > RightHeight
               Return LeftHeight + 1
          ELSE
               Return RightHeight + 1
          [END OF IF]
        [END OF IF]
Step 2: END
```

Determining the Number of Nodes

- To calculate the total number of elements/nodes in a BST, we will count the number of nodes in the left sub-tree and the right sub-tree.
- Number of nodes = total Nodes(left sub-tree) + total Nodes(right sub-tree) + 1



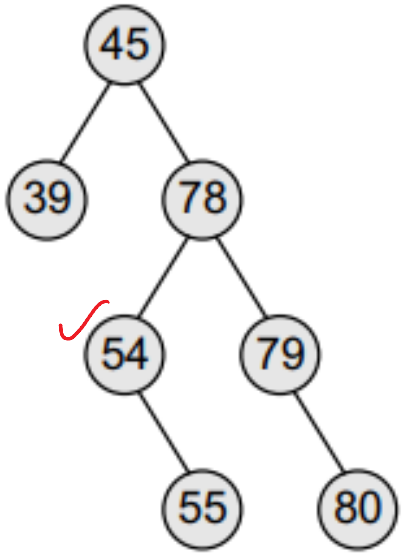
- **Algorithm**

```
totalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
            Return totalNodes(TREE -> LEFT)
                + totalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```

- Total nodes of left sub–tree = 1
- Total nodes of right sub–tree = 5
- Total nodes of tree = 1+5+1 = 7

Determining the Number of Internal Nodes

- To calculate the total number of internal nodes or non-leaf nodes, we count the number of internal nodes in the left sub-tree and the right sub-tree and add 1 to it (1 is added for the root node).



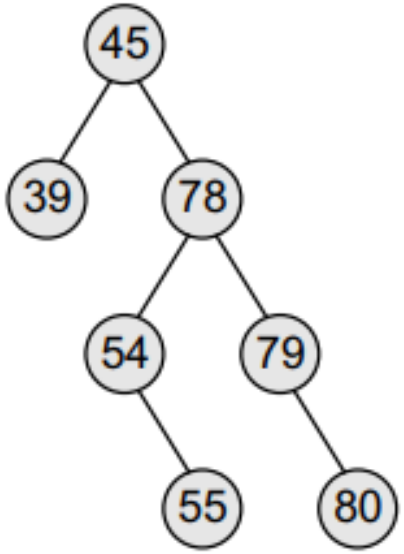
- **Algorithm**

```
totalInternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        [END OF IF]
        IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 0
        ELSE
            Return totalInternalNodes(TREE -> LEFT) +
            totalInternalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```

Total internal in left subtree = 0
Total internal in right subtree = 3
Total internal = 0+3+1

Determining the Number of External Nodes

- To calculate the total number of external nodes or leaf nodes, we add the number of external nodes in the left sub-tree and the right sub-tree.
- However if the tree is empty, that is TREE = NULL, then the number of external nodes will be zero.
- But if there is only one node in the tree, then the number of external nodes will be one.



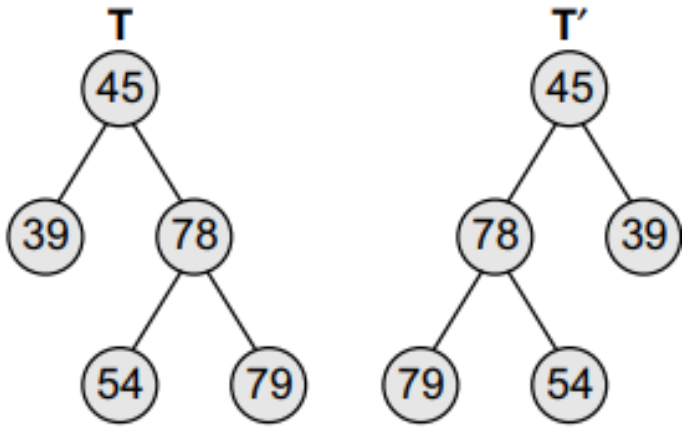
External of left subtree = 1
External of right subtree = 2
total External node = 1+2

- **Algorithm**

```
totalExternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 1
        ELSE
            Return totalExternalNodes(TREE -> LEFT) +
            totalExternalNodes(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

Finding the Mirror Image of a Binary Search Tree

- Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.
- For example, given a tree T, the mirror image of T can be obtained as T'. Consider the tree T given in Fig.

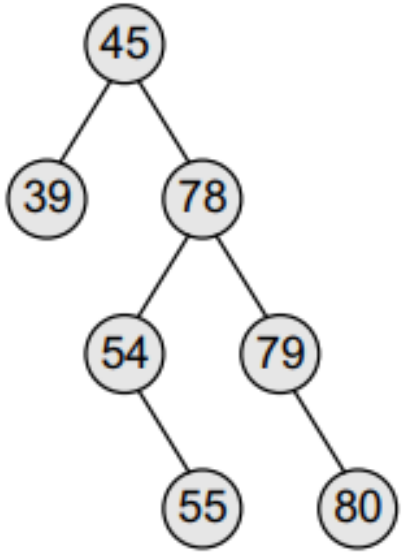


- **Algorithm**

```
MirrorImage(TREE)

Step 1: IF TREE != NULL
            MirrorImage(TREE -> LEFT)
            MirrorImage(TREE -> RIGHT)
            SET TEMP = TREE -> LEFT
            SET TREE -> LEFT = TREE -> RIGHT
            SET TREE -> RIGHT = TEMP
        [END OF IF]
Step 2: END
```

Finding the Smallest Node in a Binary Search Tree

- The very basic property of the binary search tree states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of the root node will be smallest as compared to the nodes in the right sub-tree.
- So, to find the node with the smallest value, we find the value of the leftmost node of the left sub-tree.



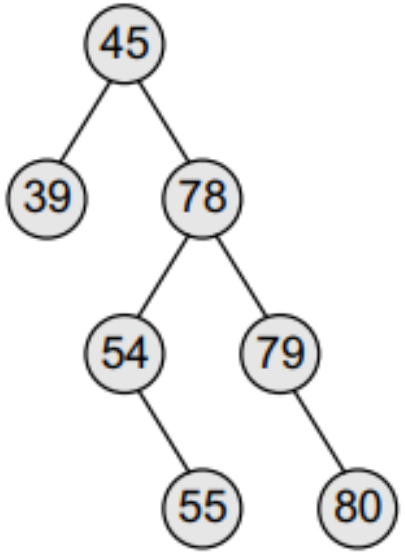
- **Algorithm**

```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
            Returen TREE
        ELSE
            Return findSmallestElement(TREE -> LEFT)
        [END OF IF]
Step 2: END
```

Finding the Largest Node in a Binary Search Tree

- To find the node with the largest value, we find the value of the rightmost node of the right sub-tree.
- However, if the right sub-tree is empty, then the root node will be the largest value in the tree.

- **Algorithm**




```
findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
           Return TREE
        ELSE
           Return findLargestElement(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

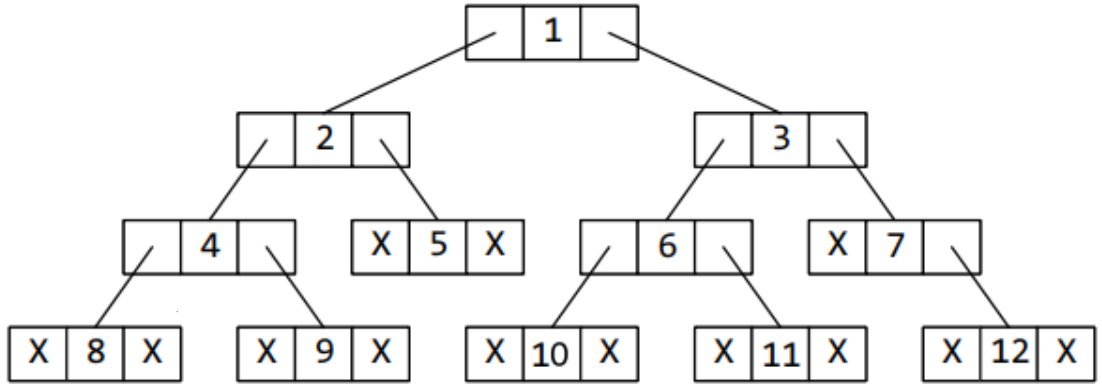
Deleting a Binary Search Tree

- To delete/remove an entire binary search tree from the memory, we first delete the elements/nodes in the left sub-tree and then delete the nodes in the right sub-tree.
- **Algorithm**

```
deleteTree(TREE)

Step 1: IF TREE != NULL
            deleteTree (TREE -> LEFT)
            deleteTree (TREE -> RIGHT)
            Free (TREE)
        [END OF IF]
Step 2: END
```

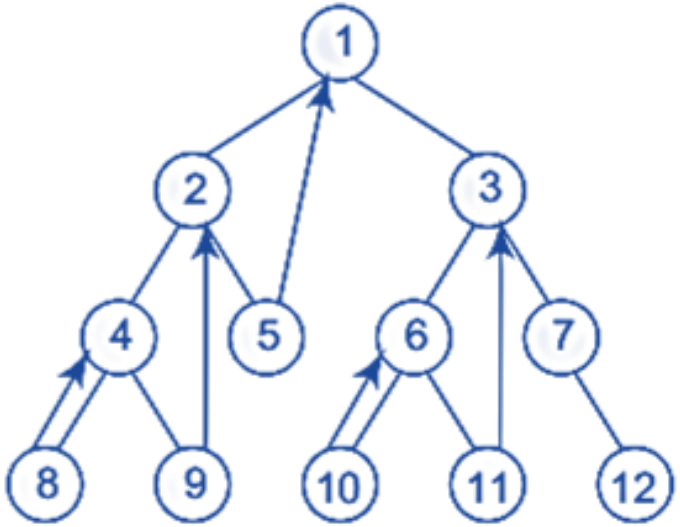
THREADED BINARY TREES

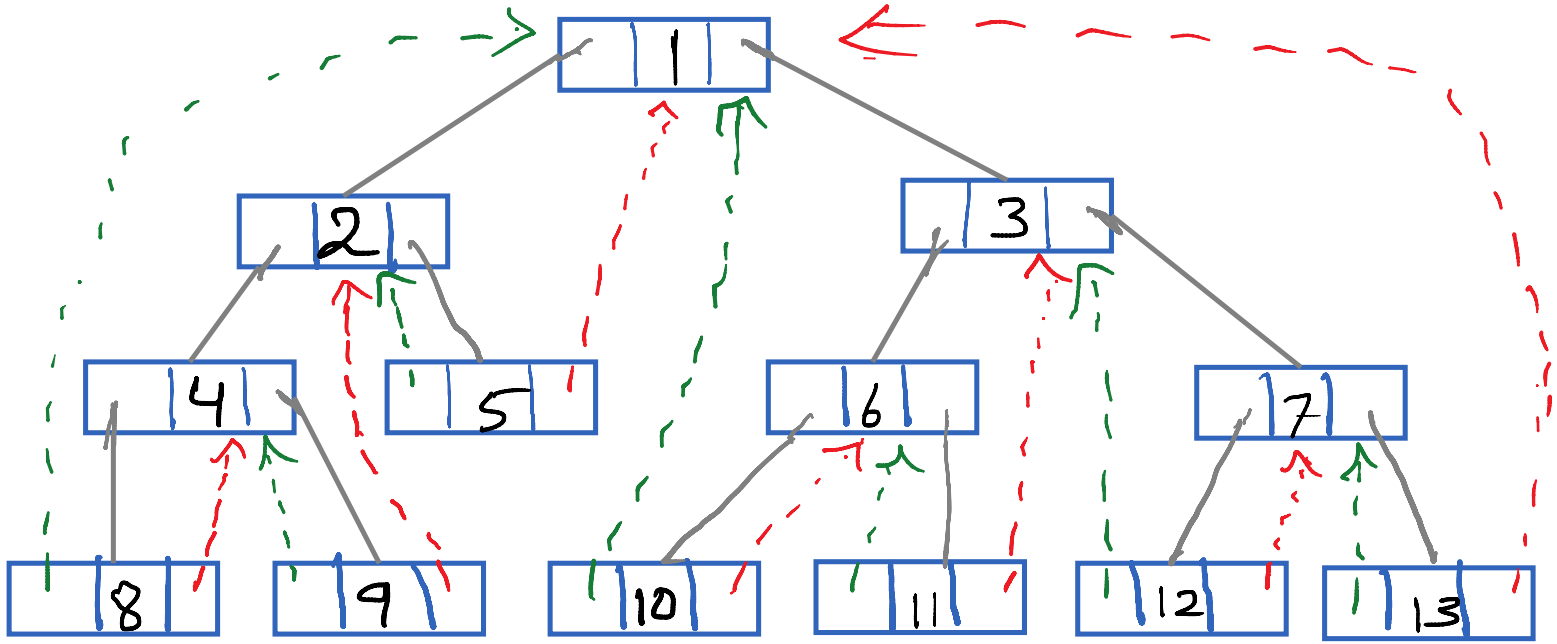
- A threaded binary tree is same as that of a binary tree but with a difference in storing NULL pointers.
- In the linked representation of a BST, a number of nodes contain a NULL pointer either in their left or right fields or in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information.
- For example, the NULL entries can be replaced to store a pointer to the in-order predecessor, or the in-order successor of the node. These special pointers are called **threads** and binary trees containing threads are called **threaded trees**.
- In the linked representation of a threaded binary tree, threads will be denoted using dotted lines.



THREADED BINARY TREES (One-way Threading)

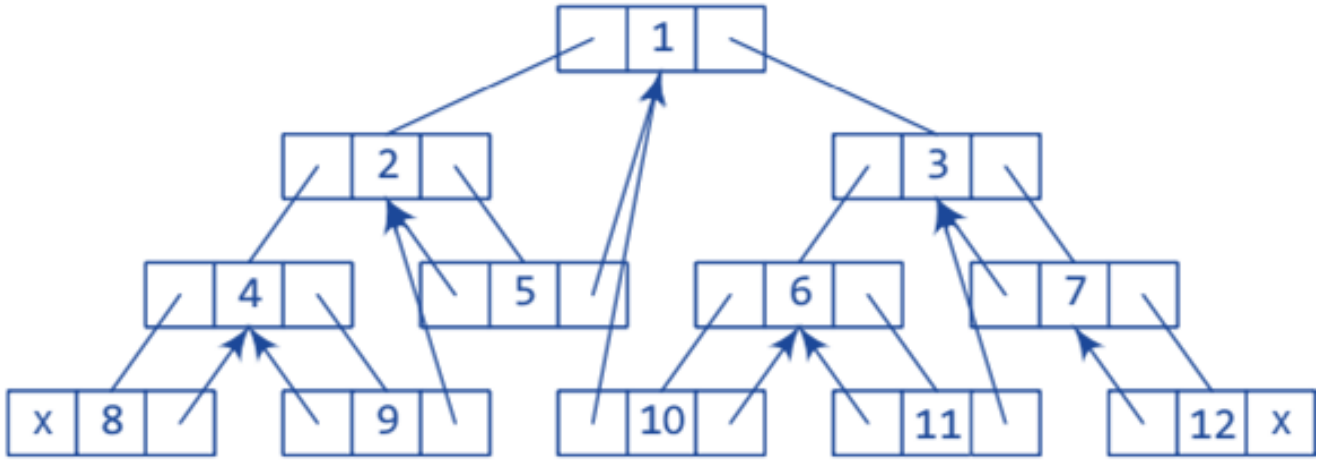
- In one way threading, a thread will appear either in the **right field** or the **left field** of the node.
- If the thread appears in the **left field**, then it points to the **in-order predecessor** of the node. Such a one way threaded tree is called a **left threaded binary tree**.
- If the thread appears in the **right field**, then it will point to the **in-order successor** of the node. Such a one way threaded tree is called a **right threaded binary tree**.





THREADED BINARY TREES (Two-way Threading)

- A two way threaded tree, also called a **doubled threaded tree**, threads will appear in both the left and right fields of the node.
- While the **left field** will point to the **in-order predecessor** of the node, the **right field** will point to its **successor**.
- A two way threaded binary tree is also called a **fully threaded binary tree**.



- It enables **linear traversal** of elements in the tree.
- Linear traversal **eliminates the use of stacks** which in turn consume a lot of memory space and computer time.

Balanced Binary Trees- AVL Trees

- AVL tree is a self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one. Because of this property, AVL tree is also known as a **height-balanced tree**.
- AVL Trees are named after their inventors, Adelson-Velsky and Landis.
- The key advantage of using an AVL tree is that it takes O(logn) time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to O(logn)).
- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the **BalanceFactor.**

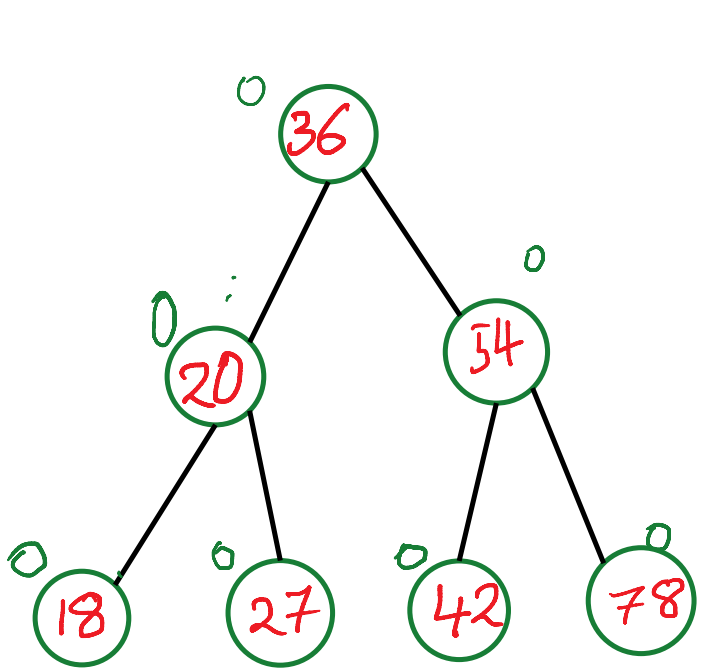
AVL Trees

- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

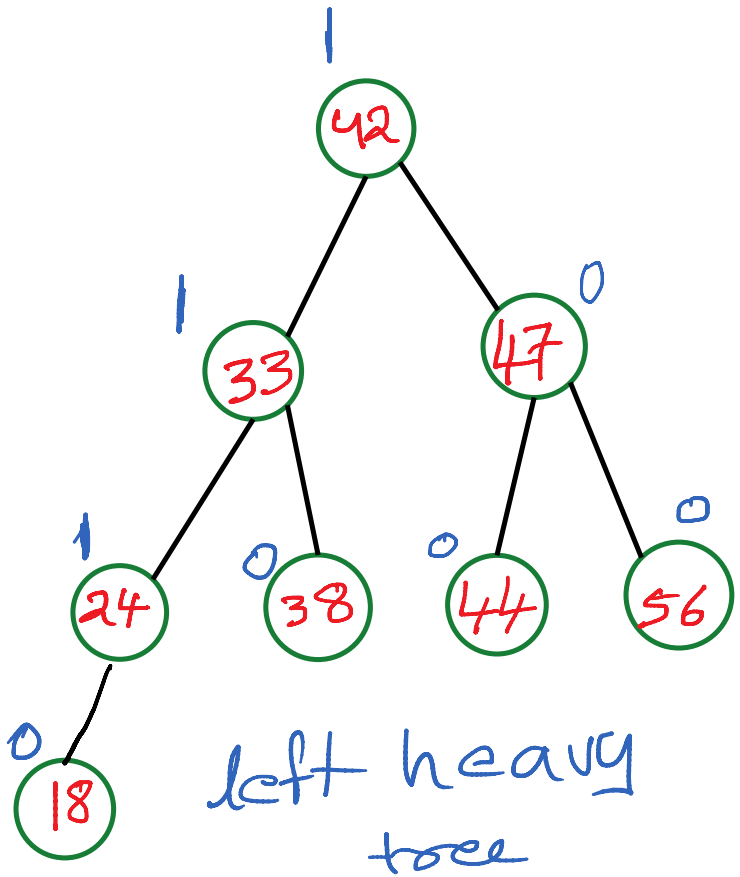
Balance factor = Height (left sub-tree) – Height (right sub-tree)

- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be height balanced.
- A node with any other balance factor is considered to be unbalanced and requires rebalancing.
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called **Left-heavy tree**.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is **equal** to the height of its right sub-tree. Such a tree is called **Balanced tree**
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called **Right-heavy tree**.

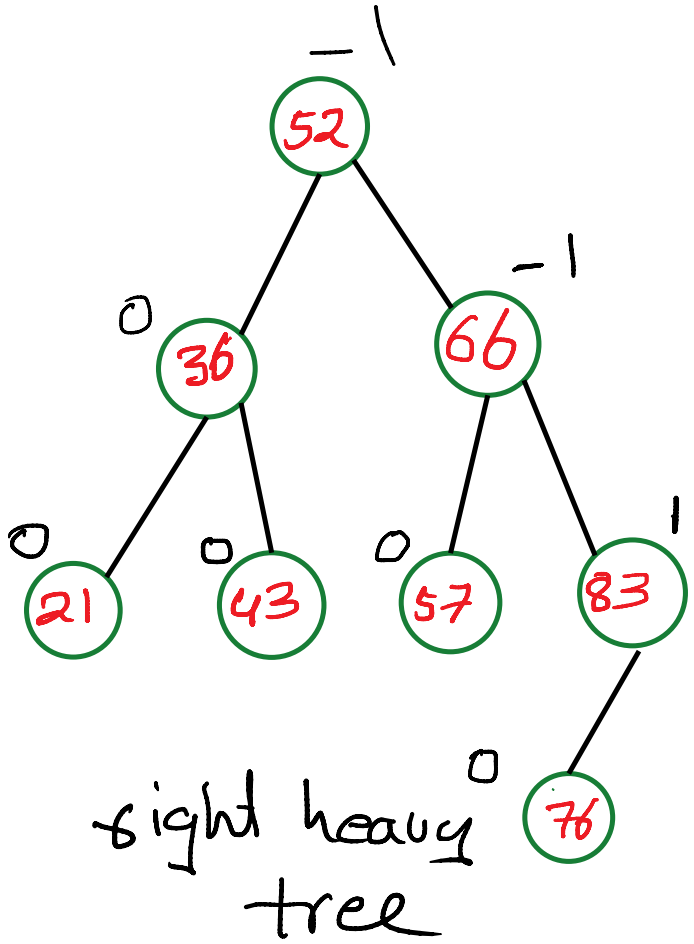
Examples of AVL trees:



Balanced tree



left heavy tree



right heavy tree

Operations on an AVL Tree

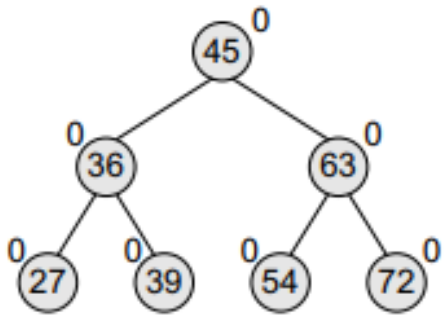
- The following operations are performed on AVL tree
  - Search
  - Insertion
  - Deletion

Searching for a Node in an AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes O(log n) time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken

Search Operation in AVL Tree

- In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree.
- We use the following steps to search an element in AVL tree.



Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then continue the search process in right subtree.

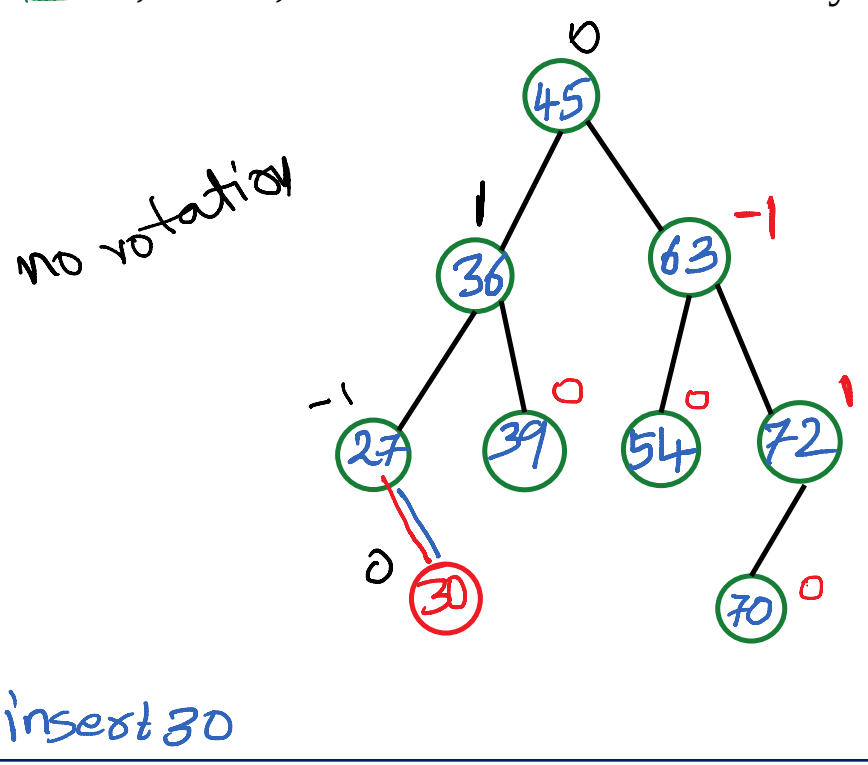
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

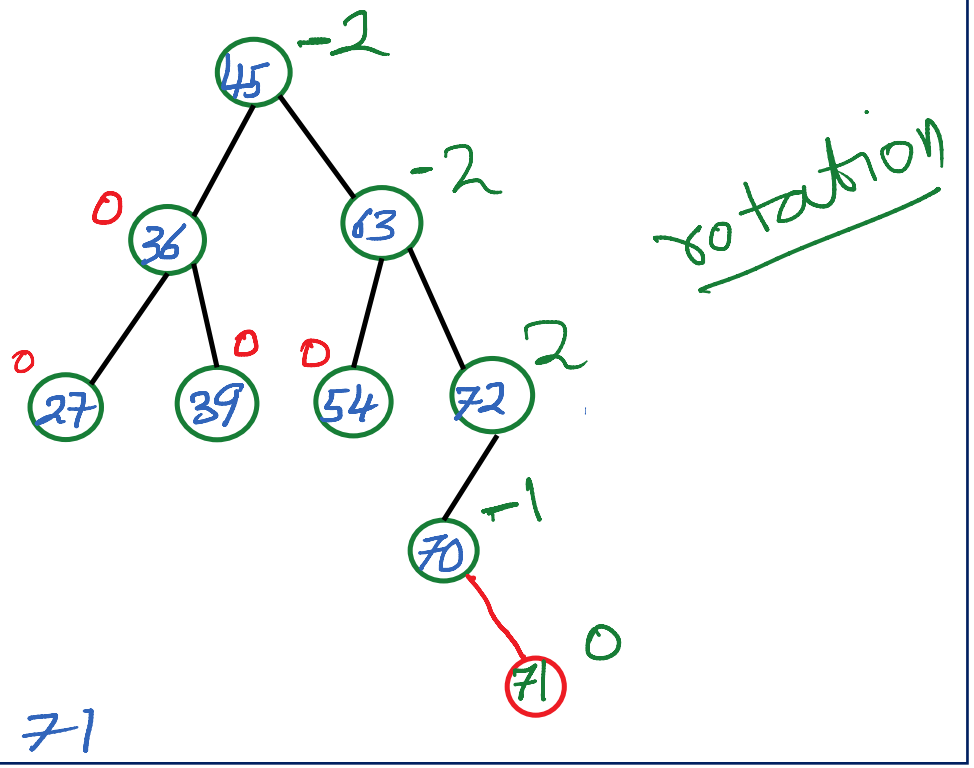
Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Inserting a Node in an AVL Tree

- Since an AVL tree is also a variant of binary search tree, insertion is also done in the same way as it is done in case of a binary search tree.
- Like in binary search tree, the new node is always inserted as the **leaf node**. But the step of insertion is usually followed by an additional step of **rotation**.
- Rotation is done to restore the **balance** of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.



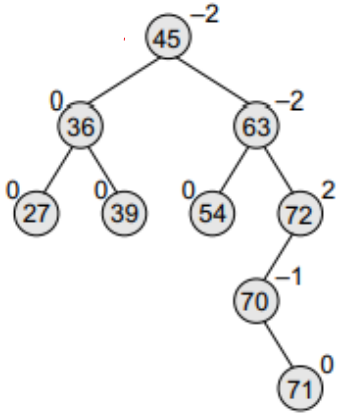


Inserting a Node in an AVL Tree

- During insertion, the new node is inserted as the leaf node, so it will always have balance factor equal to zero.
- The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.
- The possible changes which may take place in any node on the path are as follows:

1. Initially the node was either left or right heavy and after insertion has become balanced.
2. Initially the node was balanced and after insertion has become either left or right heavy.
3. Initially the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree thereby creating an unbalanced sub-tree. Such a node is said to be a **critical node**.

- **Critical node** is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1.
- Balance a sub-tree by rotating from a Critical node.

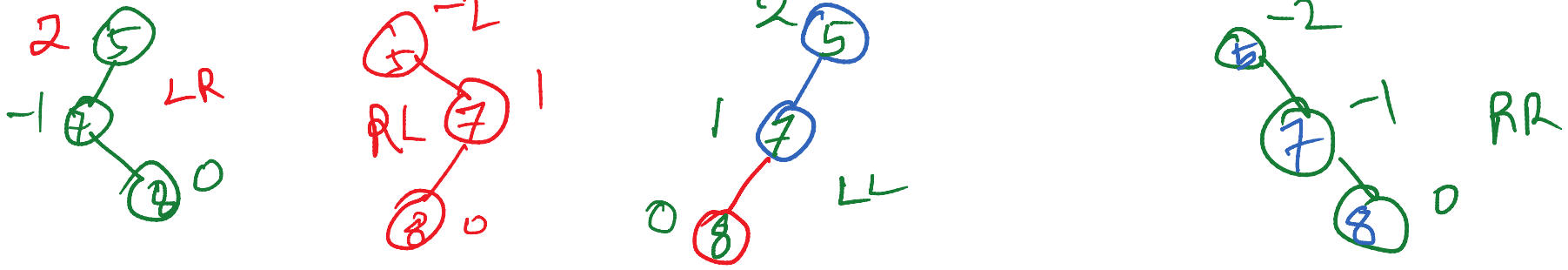


critical node
72

Rotations to Balance AVL Trees

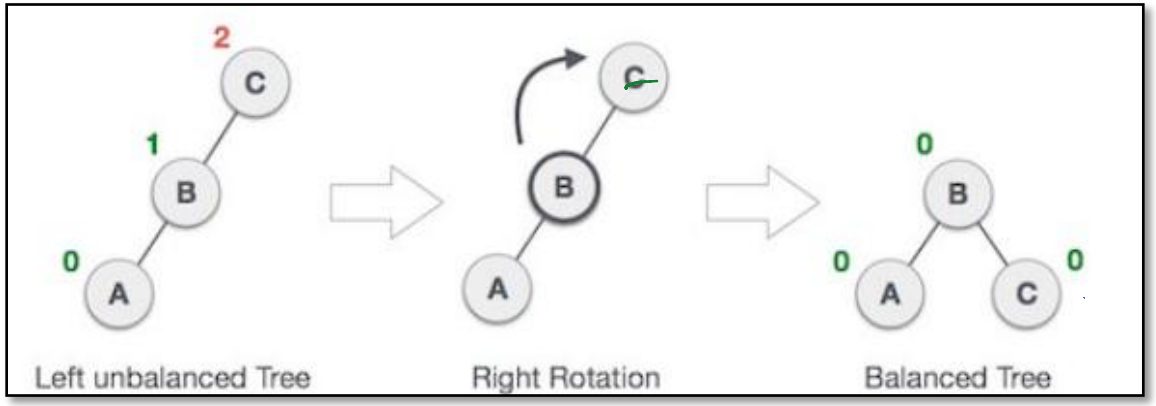
- To perform rotation, our first work is to find the **critical node**.
- **Critical node** is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- The second task is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and their application depends on the position of the inserted node with reference to the critical node.

1. **LL rotation**: The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
2. **RR rotation**: the new node is inserted in the right sub-tree of the right sub-tree of the critical node
3. **LR rotation**: the new node is inserted in the right sub-tree of the left sub-tree of the critical node
4. **RL rotation**: the new node is inserted in the left sub-tree of the right sub-tree of the critical node



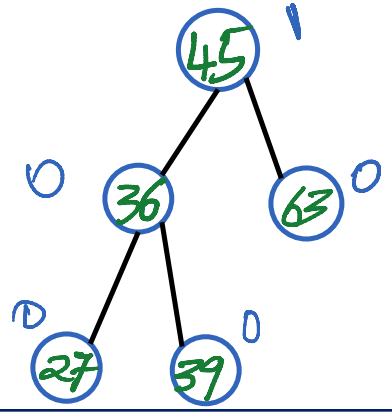
LL Rotation

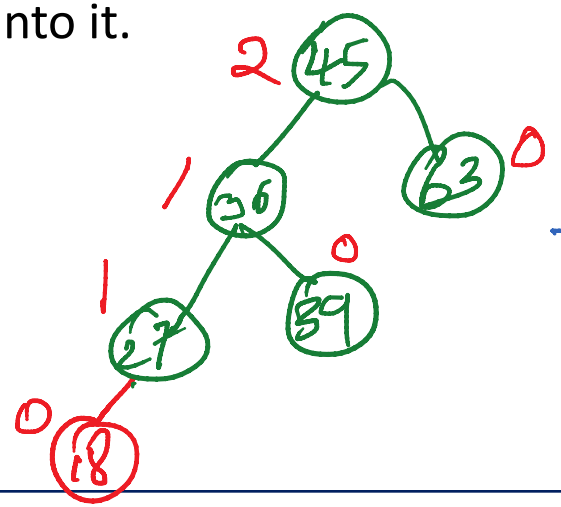
- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of critical node, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

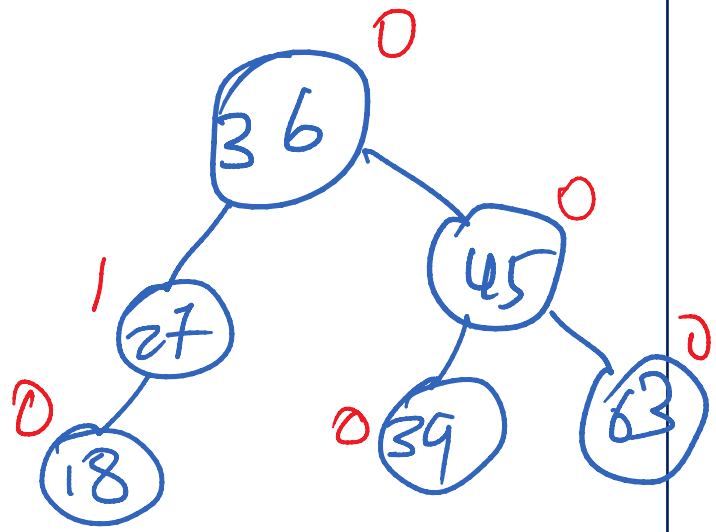


critical node = 45
LL rotation

- Consider the AVL tree given in Fig. and insert 18 into it.

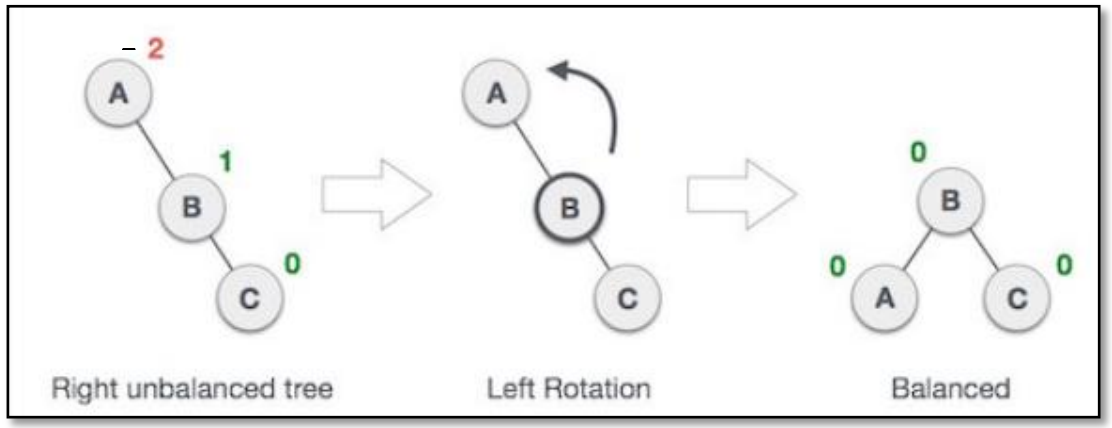






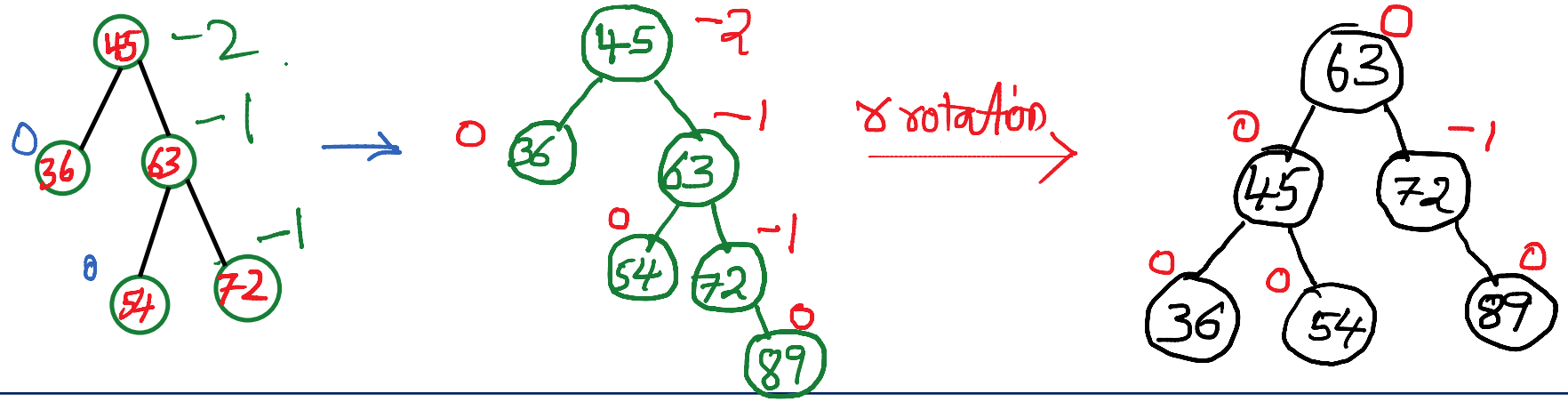
RR Rotation

- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of critical node, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



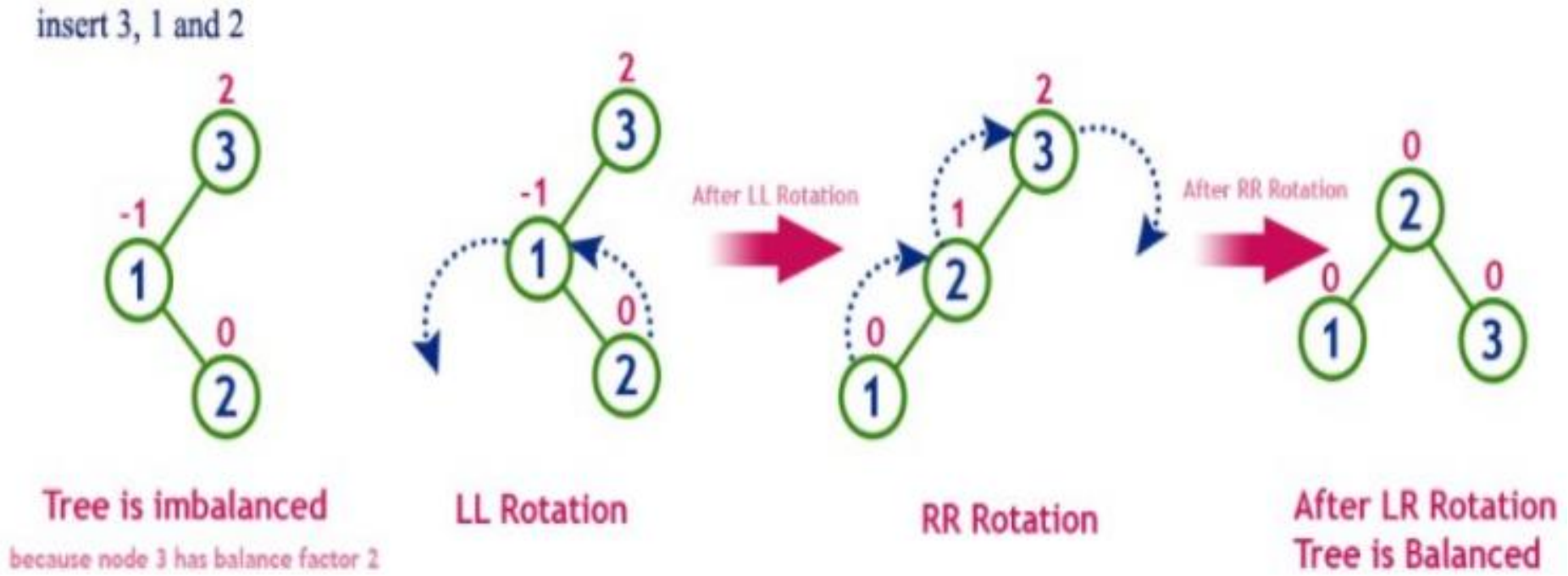
Critical node = 45
RR rotation

- Consider the AVL tree given in Fig. and insert 89 into it.

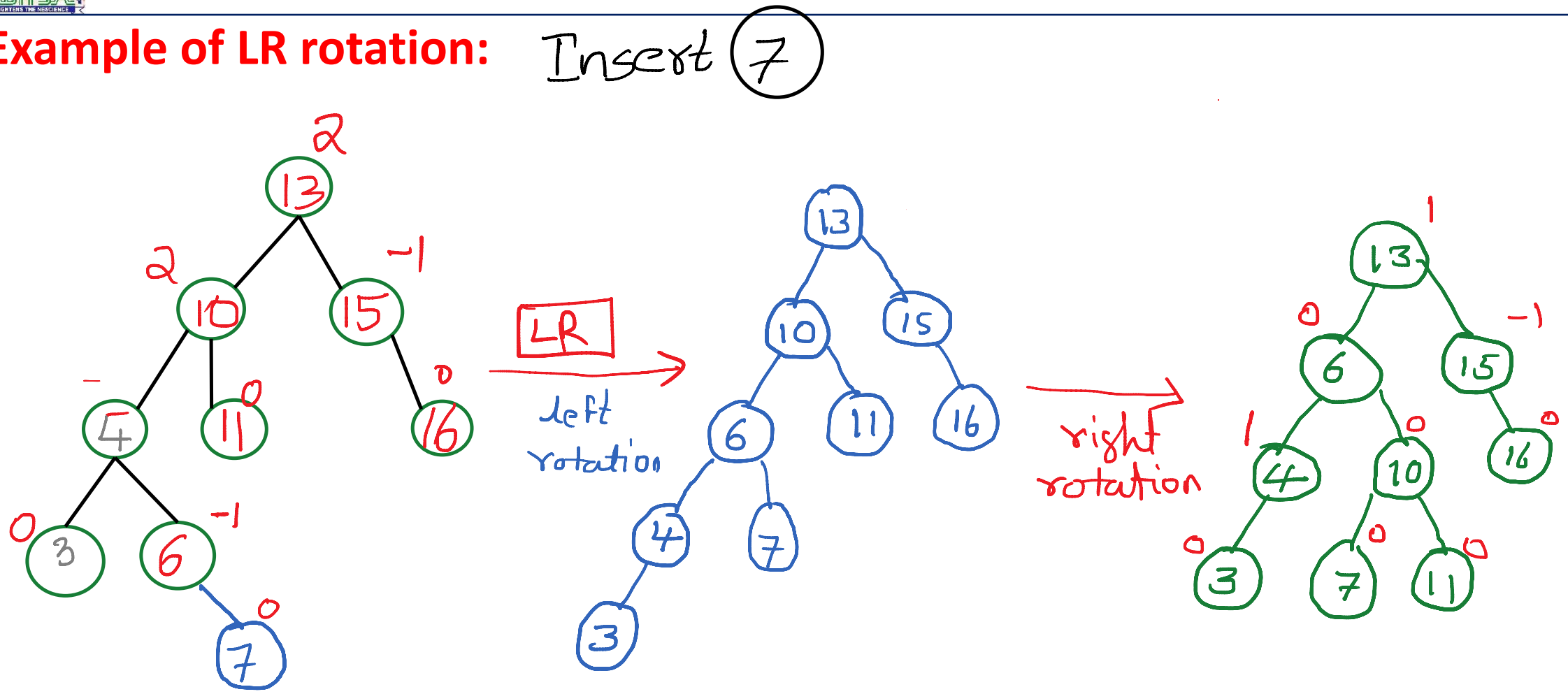


Left Right Rotation (LR Rotation)

- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AVL Tree.

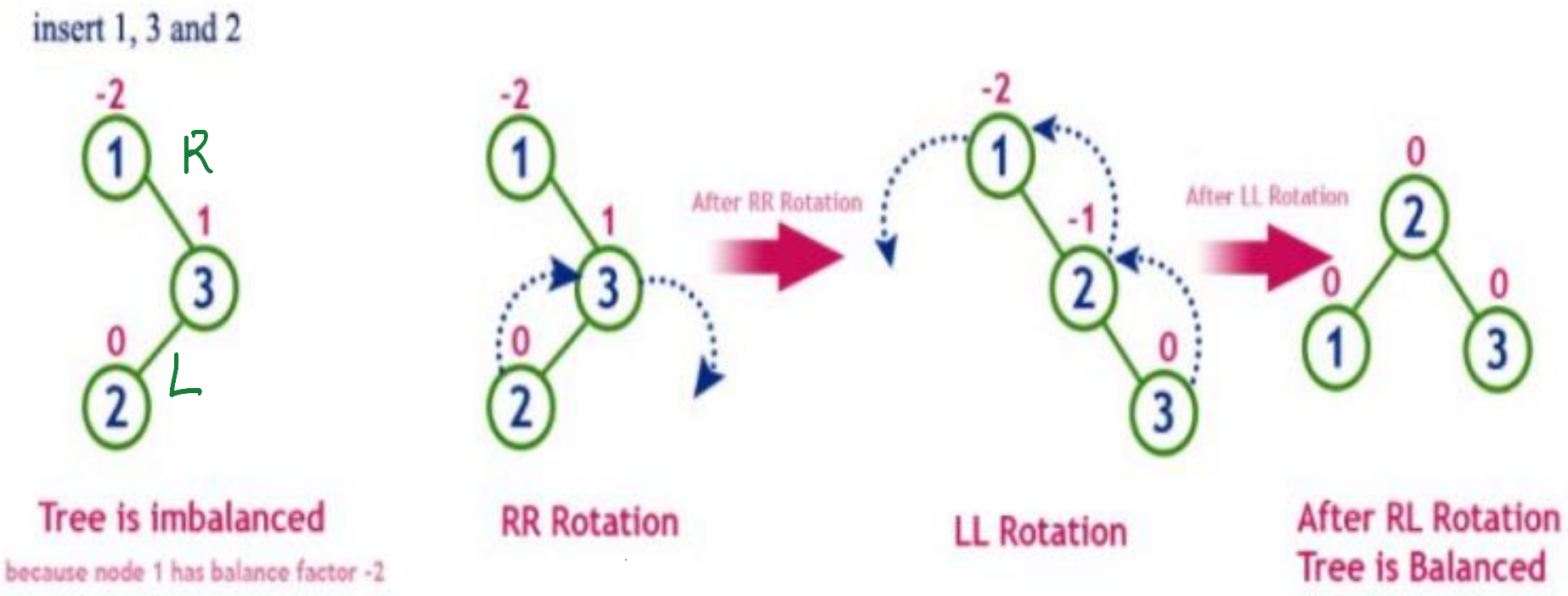


Example of LR rotation:

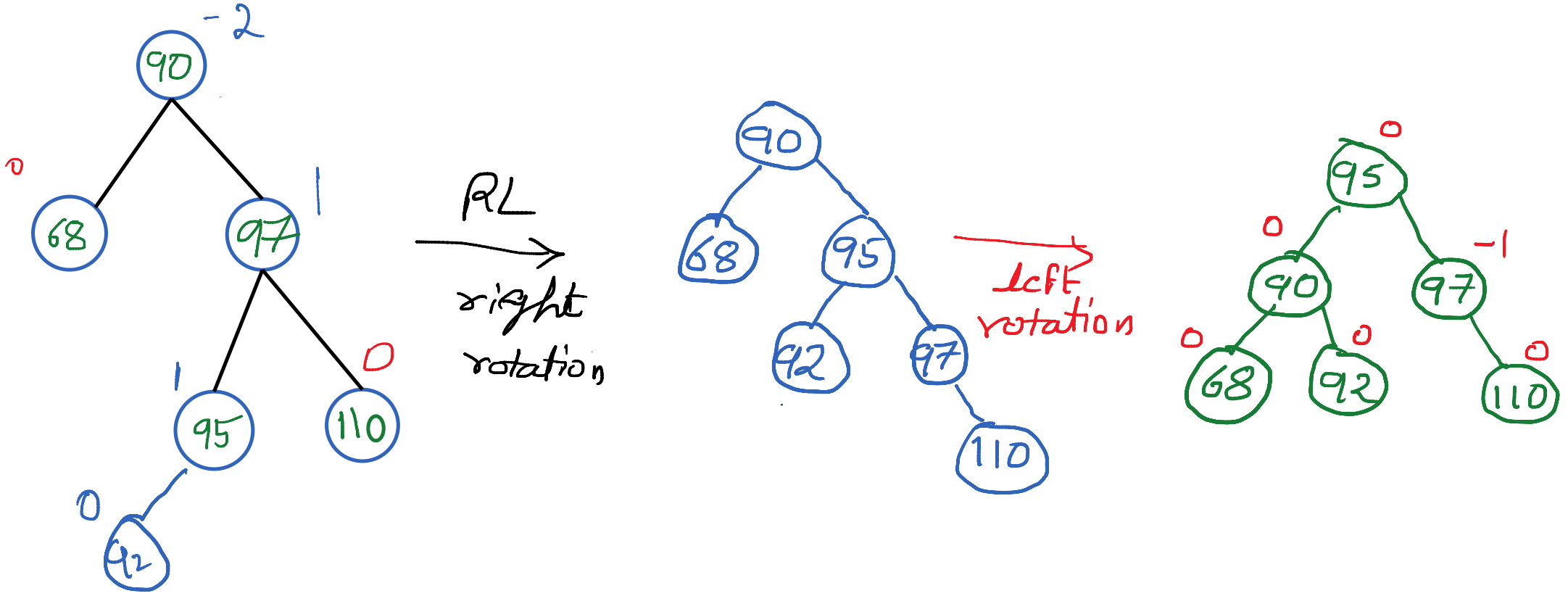


Right Left Rotation (RL Rotation)

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- To understand RL Rotation, let us consider the following insertion operation in AVL Tree



Example of RL rotation: insert 92



Insertion Operation in AVL Tree

- In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node.
- The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the Balance Factor of every node.

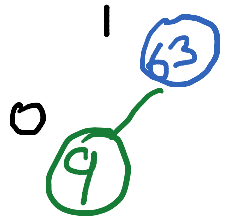
Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation

. **Step 4** - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation

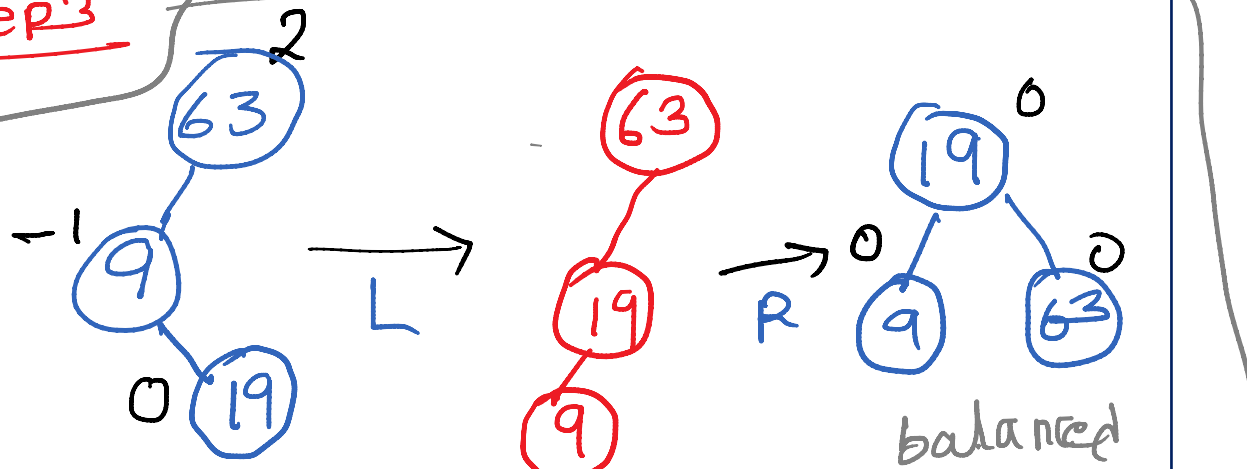
- Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

step 1

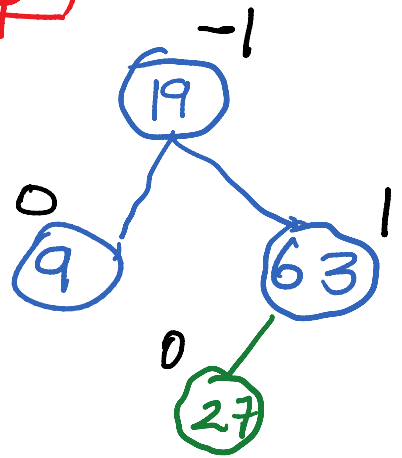
step 2



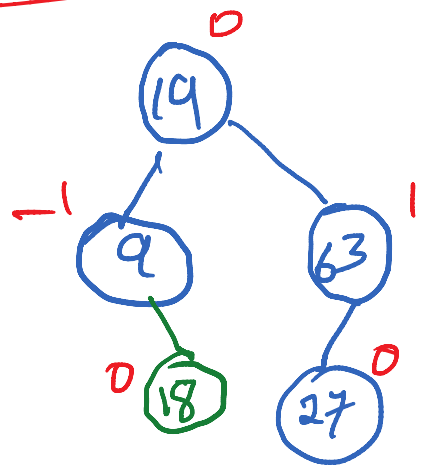
step 3



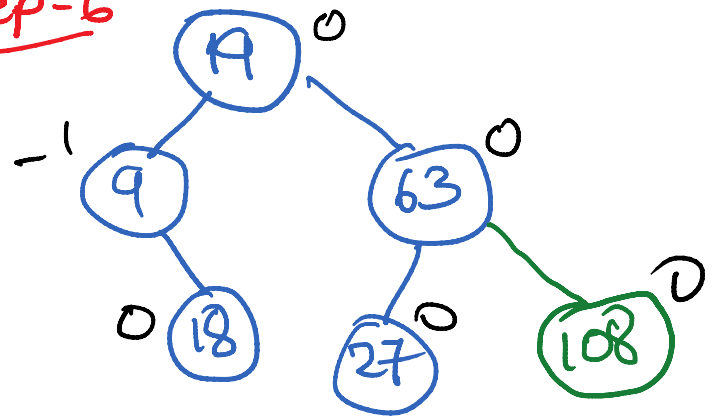
step 4



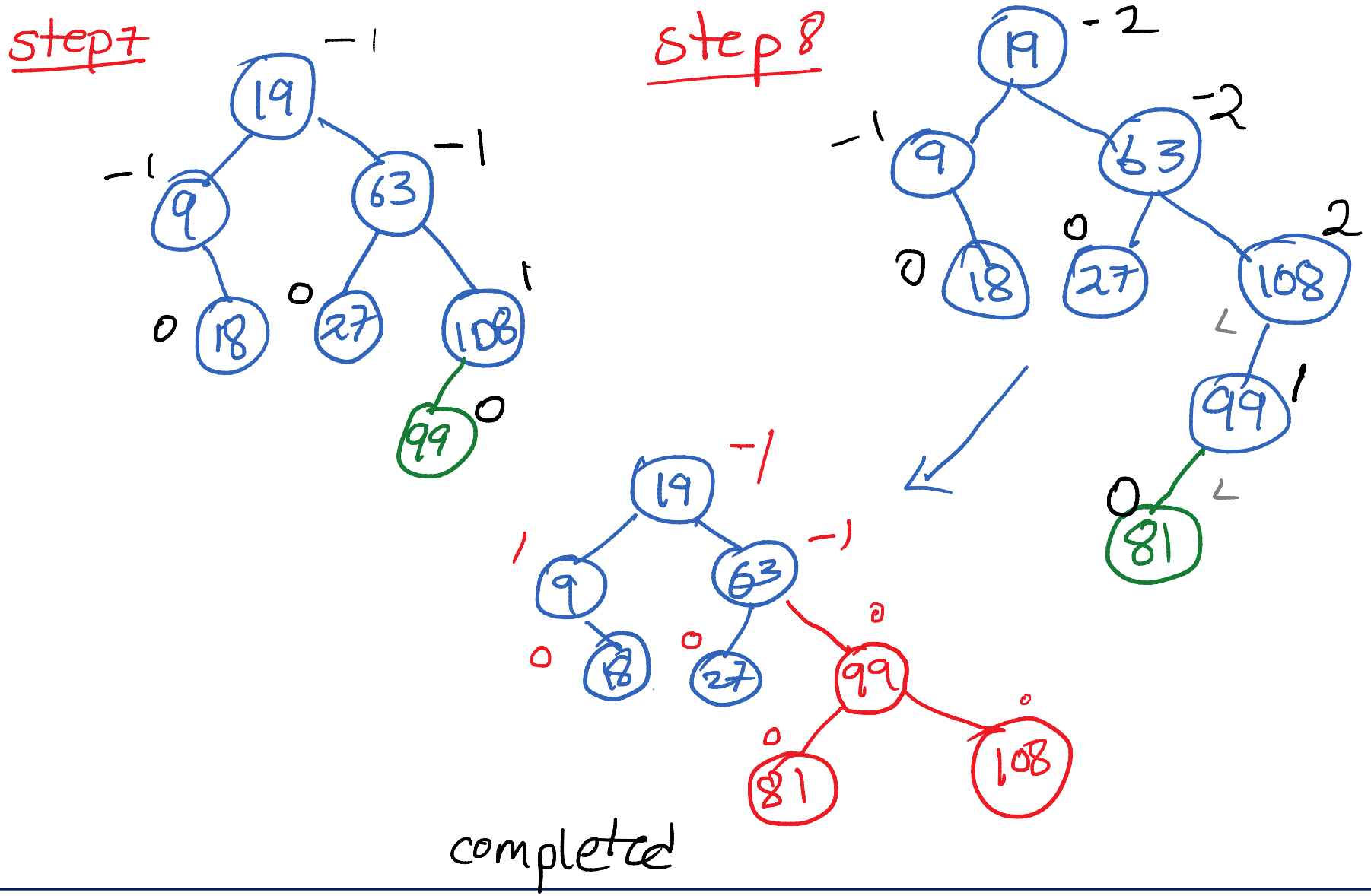
step-5



step-6



➢ Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.



C program to insert a node into AVL tree

```
struct node * insert(struct node *root,int x)
{
    if(root==NULL)
    {
        root=(struct node*)malloc(sizeof(struct node));
        root->data=x;
        root->left=NULL;
        root->right=NULL;
    }
    else
    if(x > root->data) // insert in right subtree
    {
        root->right=insert(root->right,x);
        if(Balancefactor(root)==-2)
        if(x>root->right->data)
            root=RR(root);
        else
            root=RL(root);
    }
```

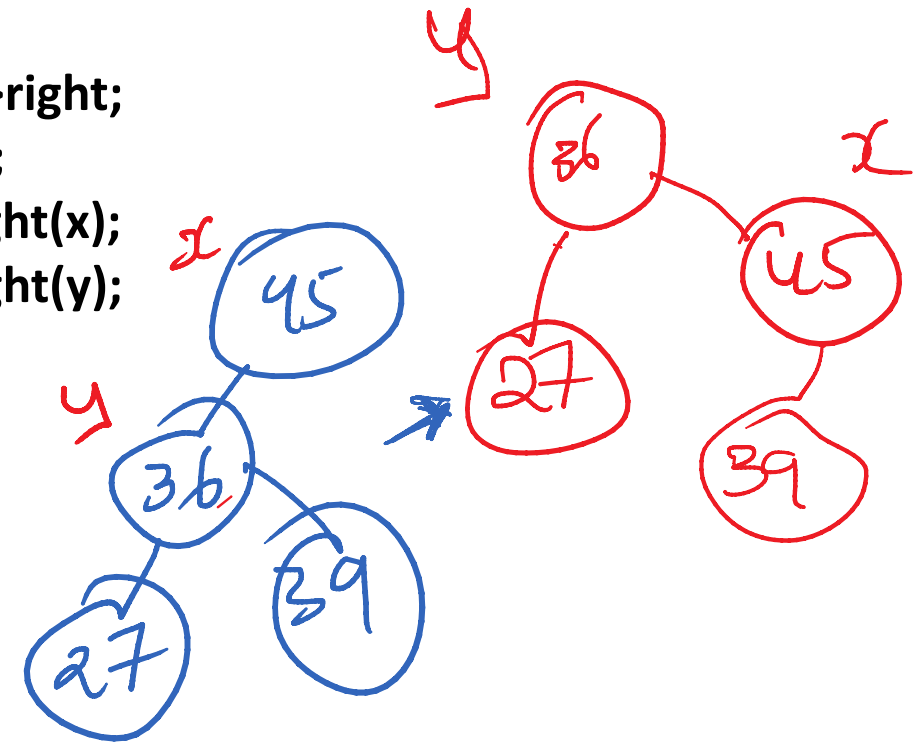
```
    else if(x<root->data)
    {
        root->left=insert(root->left,x);
        if(Balancefactor(root)==2)
        if(x < root->left->data)
            root=LL(root);
        else
            root=LR(root);
    }
    root->ht=height(root);
    return(root);
}
```

C program to insert a node into AVL tree

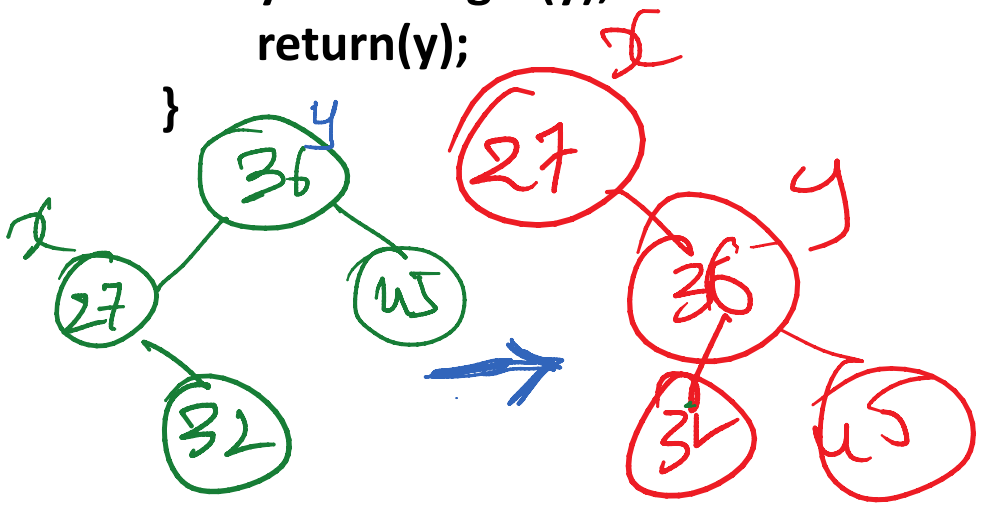
```
int height(struct node *root)
{
        int lh,rh;
        if(root==NULL)
                return(0);
        if(root->left==NULL)
                lh=0;
        else
                lh=1+root->left->ht;
        if(root->right==NULL)
                rh=0;
        else
                rh=1+root->right->ht;
        if(lh>rh)
                return(lh);
        return(rh);
}
```

C program to insert a node into AVL tree

```
struct node * rotateright(struct node *x)
{
        struct node *y;
        y=x->left;
        x->left=y->right;
        y->right=x;
        x->ht=height(x);
        y->ht=height(y);
        return(y);
}
```




```
struct node * rotateleft(struct node *x)
{
        struct node *y;
        y=x->right;
        x->right=y->left;
        y->left=x;
        x->ht=height(x);
        y->ht=height(y);
        return(y);
}
```



C program to insert a node into AVL tree

```
struct node * RR(struct node *root)
{
        root=rotateleft(root);
        return(root);
}

struct node * LL(struct node *root)
{
        root=rotateright(root);
        return(root);
}
```

```
struct node * LR(struct node *root)
{
        root->left=rotateleft(root->left);
        root=rotateright(root);
        return(root);
}

struct node * RL(struct node *root)
{
        root->right=rotateright(root->right);
        root=rotateleft(root);
        return(root);
}
```

C program to insert a node into AVL tree

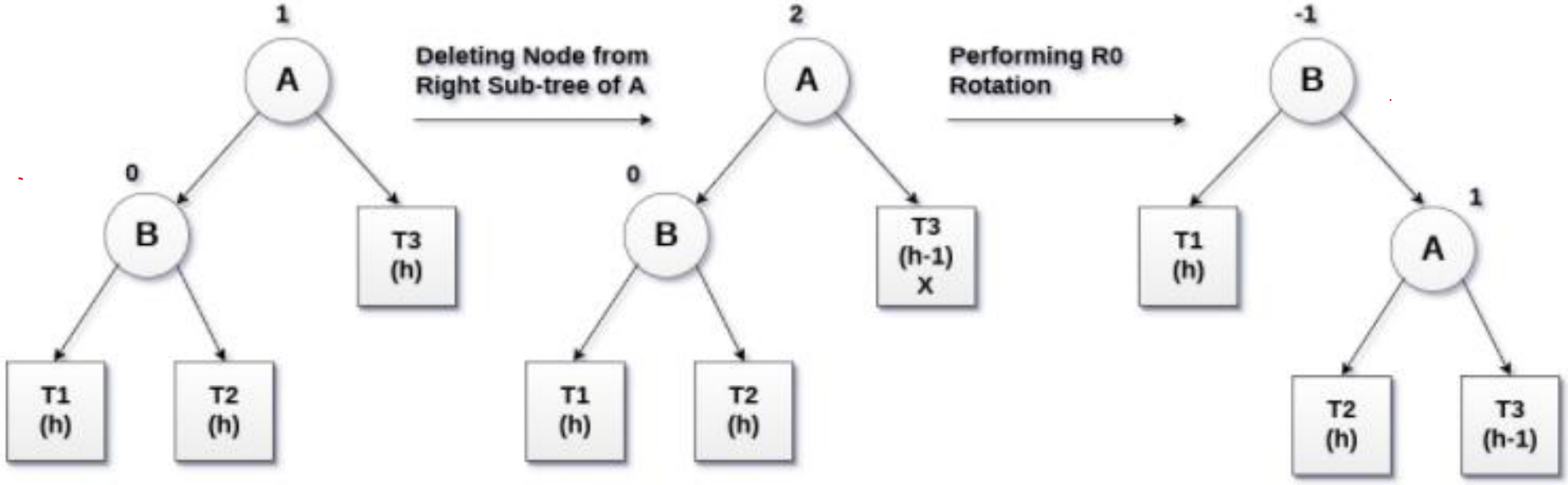
```
int Balancefactor(struct node *root)
{
        int lh,rh;
        if(root==NULL)
                return(0);
        if(root->left==NULL)
                lh=0;
        else
                lh=1+root->left->ht;

        if(root->right==NULL)
                rh=0;
        else
                rh=1+root->right->ht;

        return(lh - rh);
}
```

Deleting a Node from an AVL Tree

- Deletion of a node in an AVL tree is similar to that of binary search trees.
- But deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- There are two classes of rotation that can be performed on an AVL tree after deleting a given node: R rotation and L rotation.
- If the node to be deleted is present in the left sub-tree of the critical node, then L rotation is applied else if node is in the right sub-tree, R rotation is performed.
- Further there are three categories of L and R rotations.
- The variations of L rotation are: L-1, L0 and L1 rotation.
- Correspondingly for R rotation, there are R0, R-1 and R1 rotations.
- L rotations are the mirror images of R rotations.

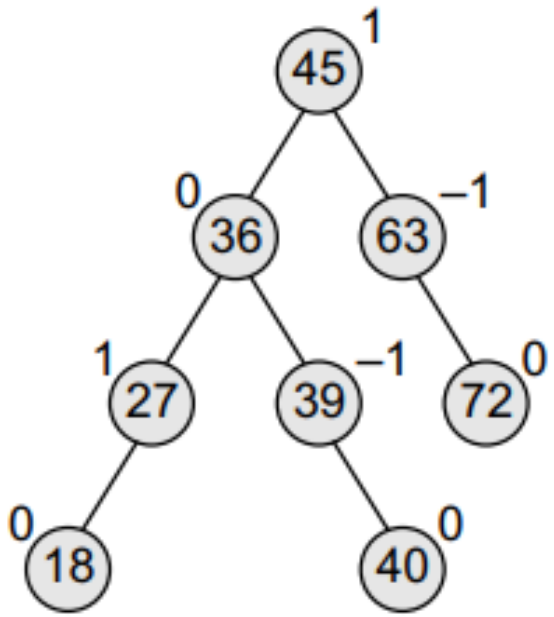
R0 rotation (Node B has balance factor 0)

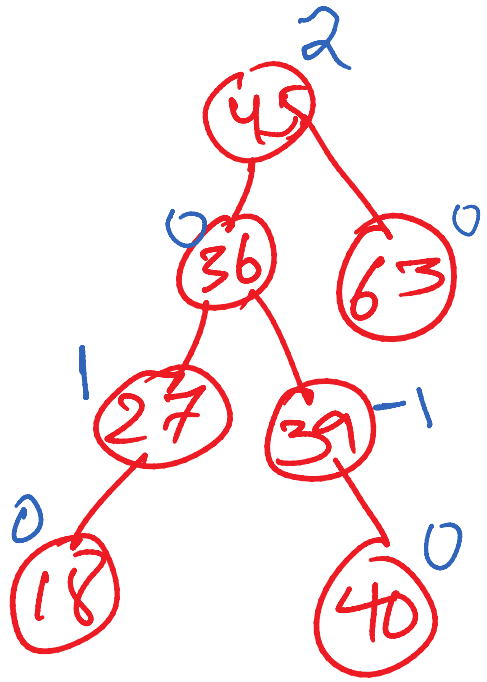
- Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations.
- If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.
- The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.



Example for R0 rotation:

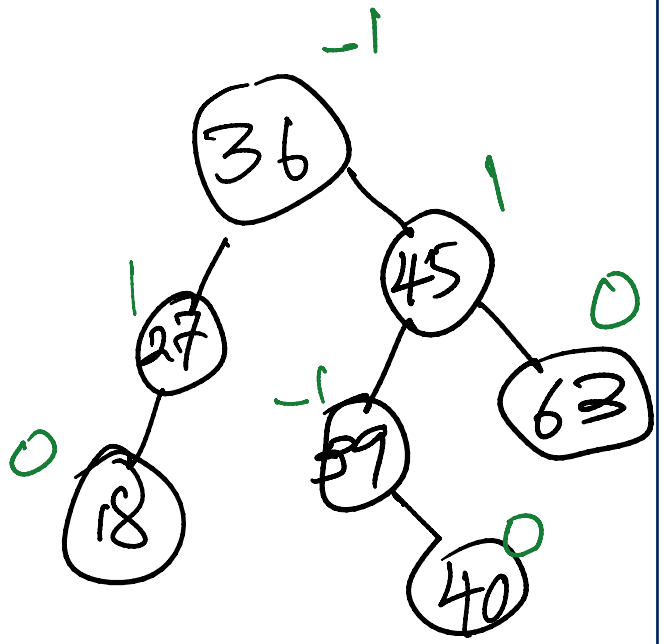
Delete the node 72 from the AVL tree shown in the following image.





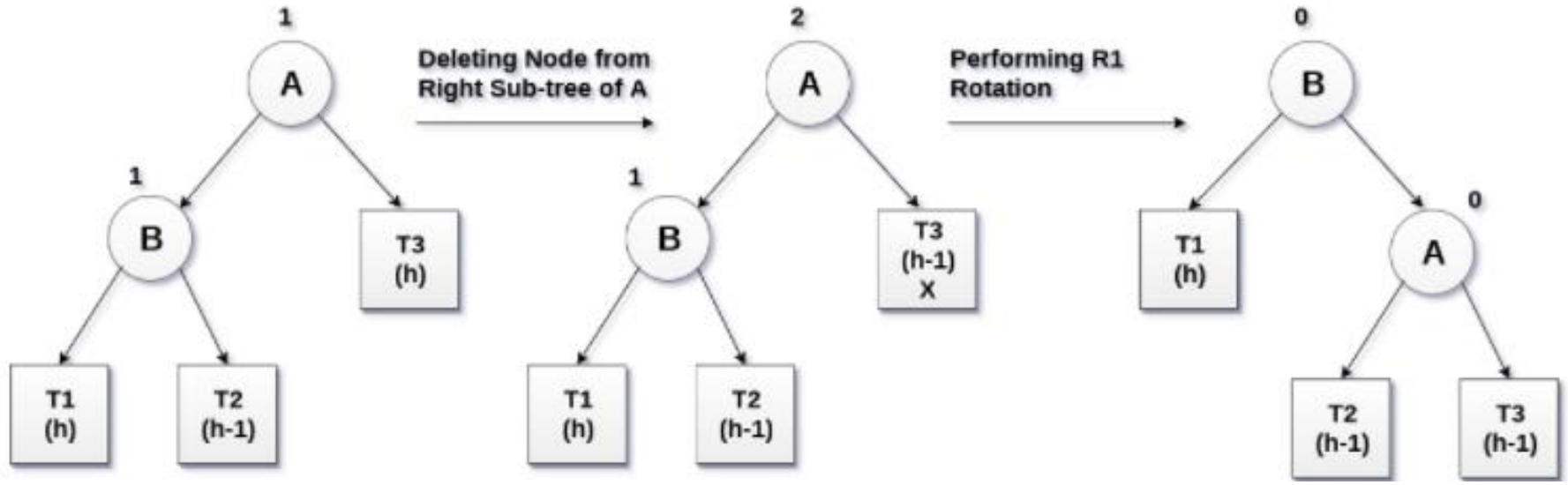
critical node = 45

R0 right rotation



R1 Rotation (Node B has balance factor 1)

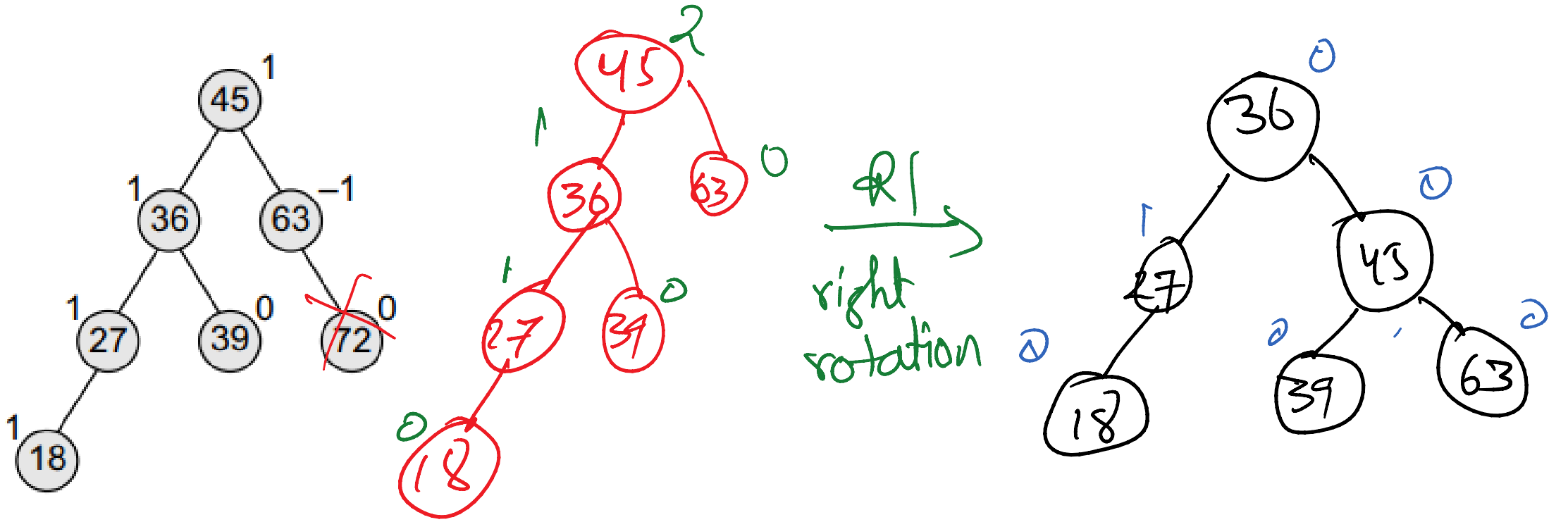
- R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.
- The process involved in R1 rotation is shown in the following image.



- Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors.

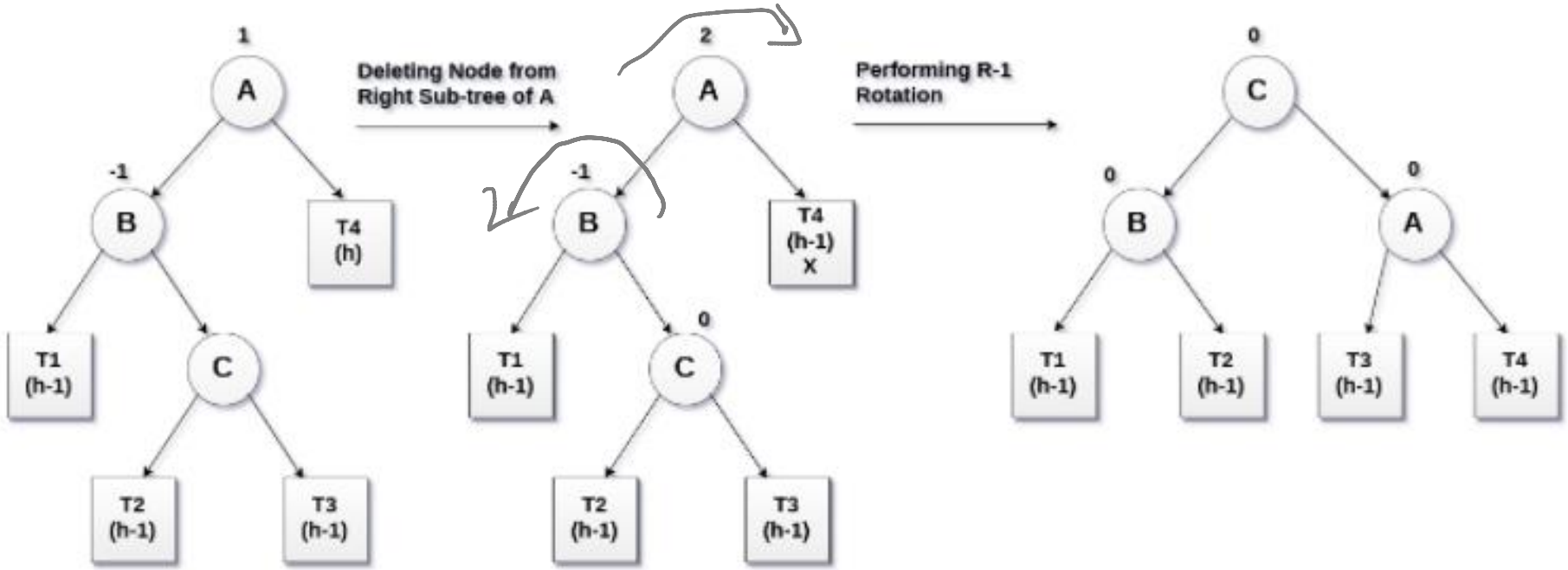
Example for R1 rotation:

Delete the node 72 from the AVL tree shown in the following image.



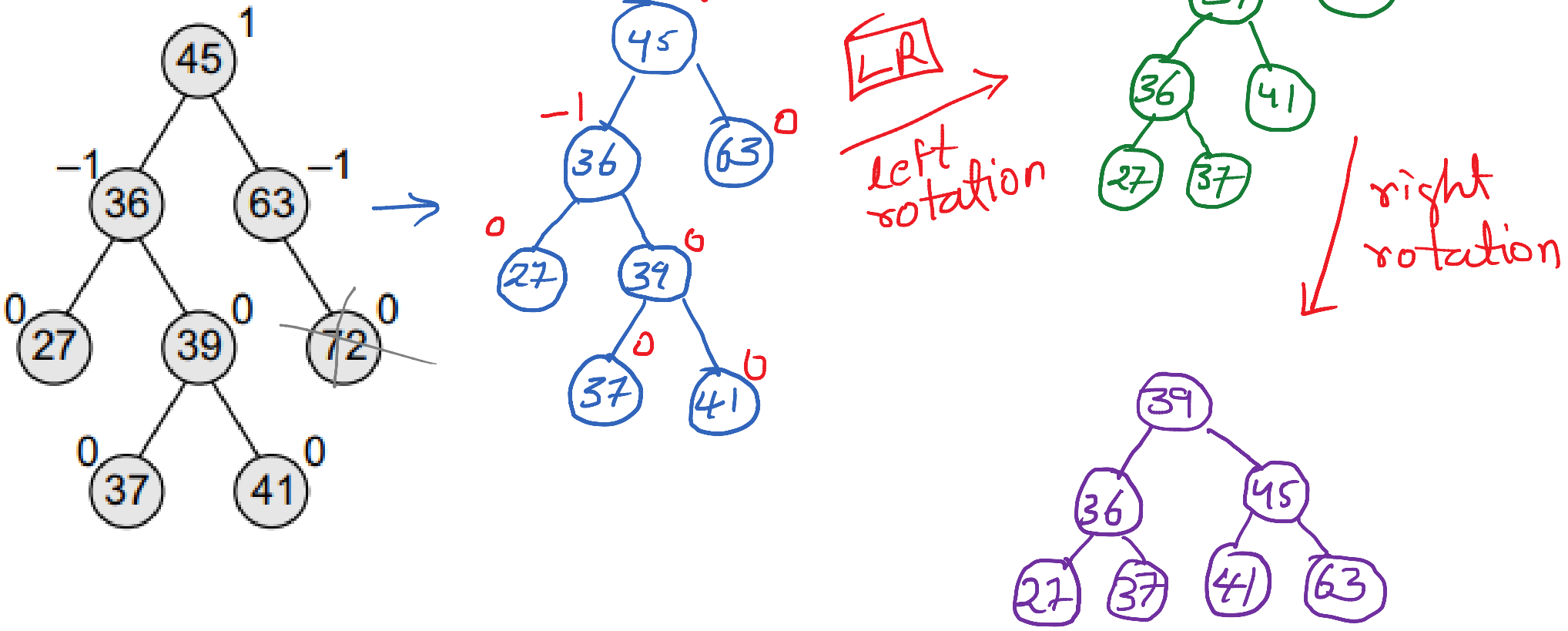
R-1 Rotation (Node B has balance factor -1)

- R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.
- The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.
- The process involved in R-1 rotation is shown in the following image.

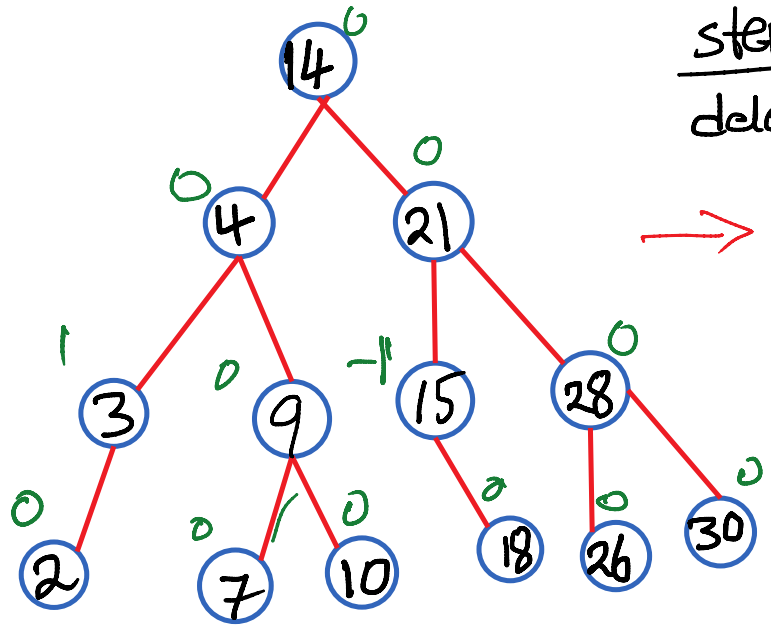


Example for R-1 rotation:

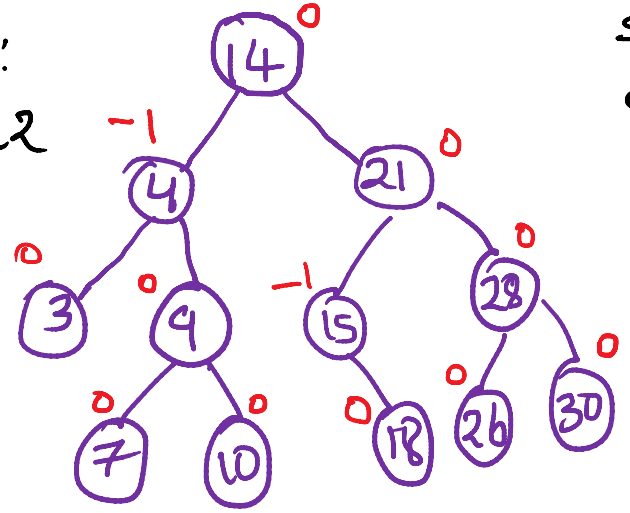
Delete the node 72 from the AVL tree shown in the following image.



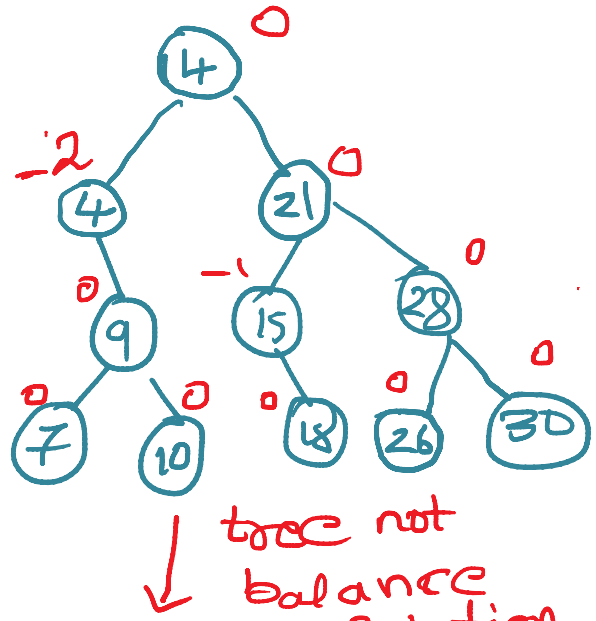
➢ Delete 2, 3, 10, 18, 4, 9, 14, 7, 15, from the following AVL tree.



step1: delete 2

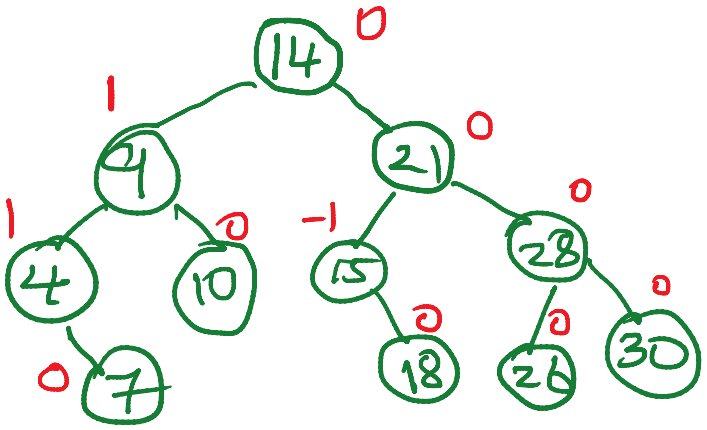


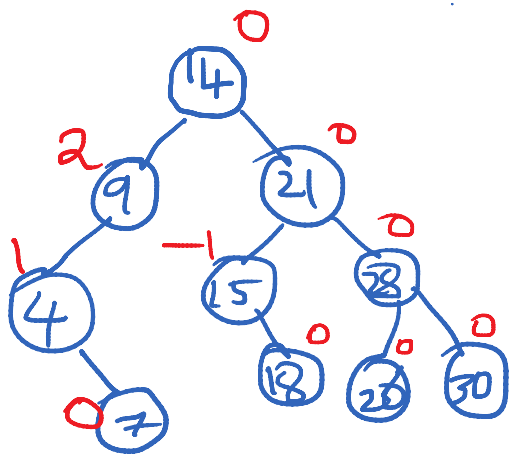
step2: delete 3



tree not balance LO Rotation

step3: delete 10

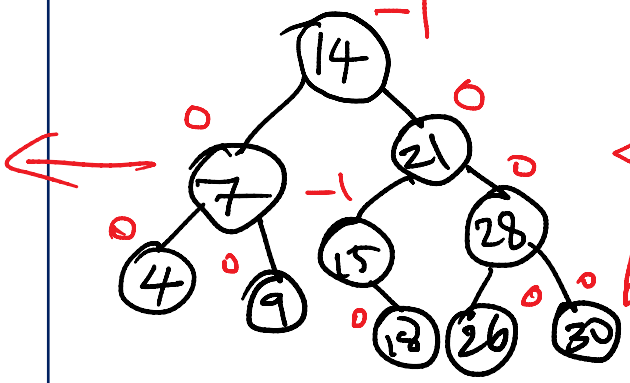




not balanced

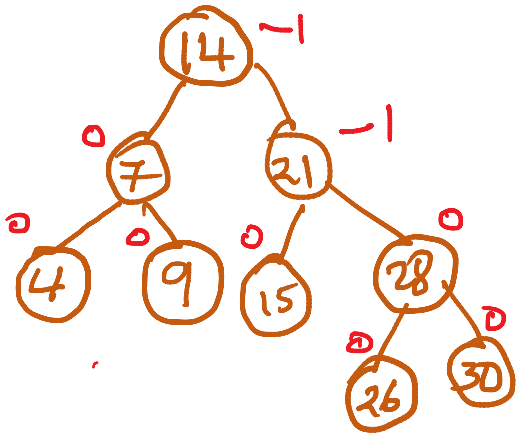
R-1

LR

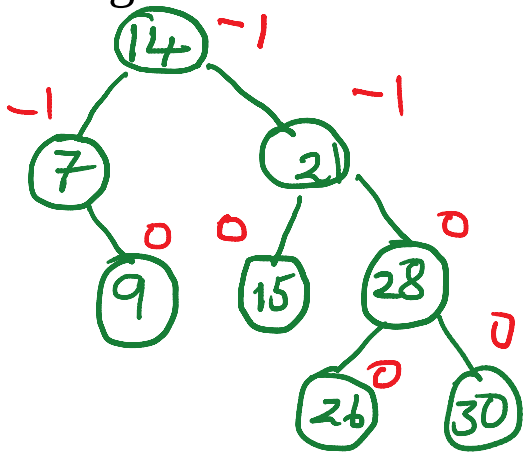


➤ Delete 2, 3, 10, 18, 4, 9, 14, 7, 15, from the following AVL tree.

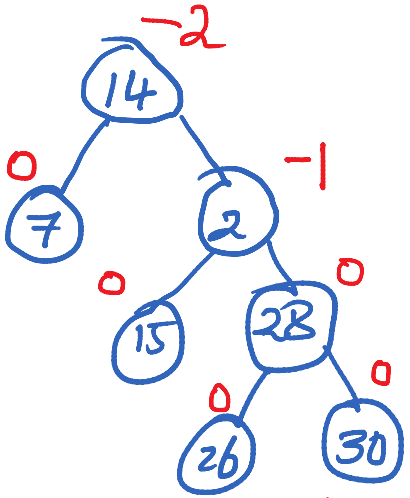
step4: delete 18



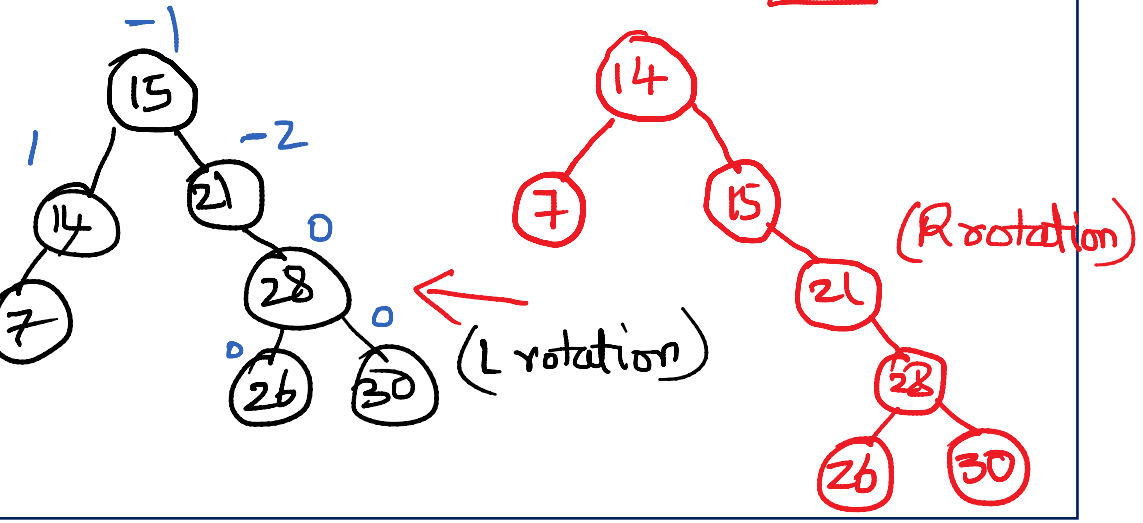
step5 : delete 4



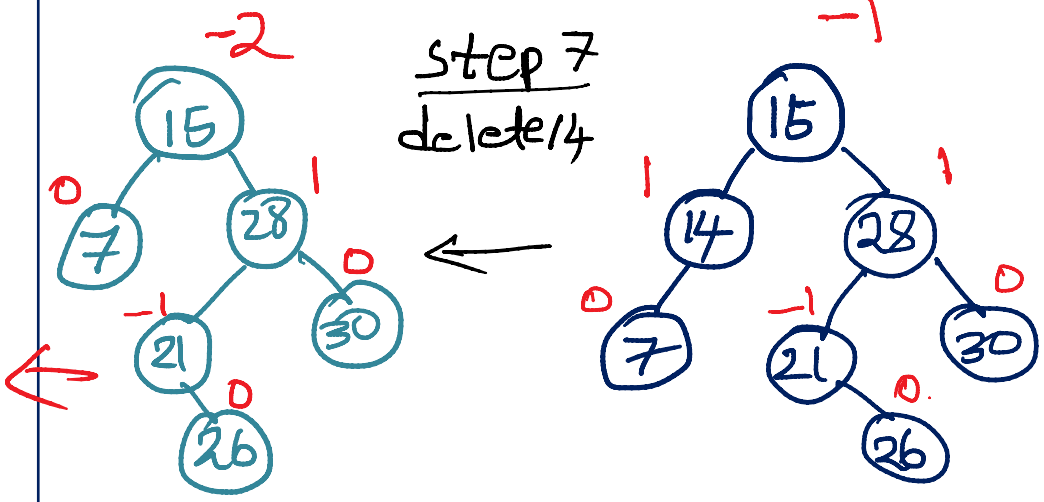
step6 delete 9

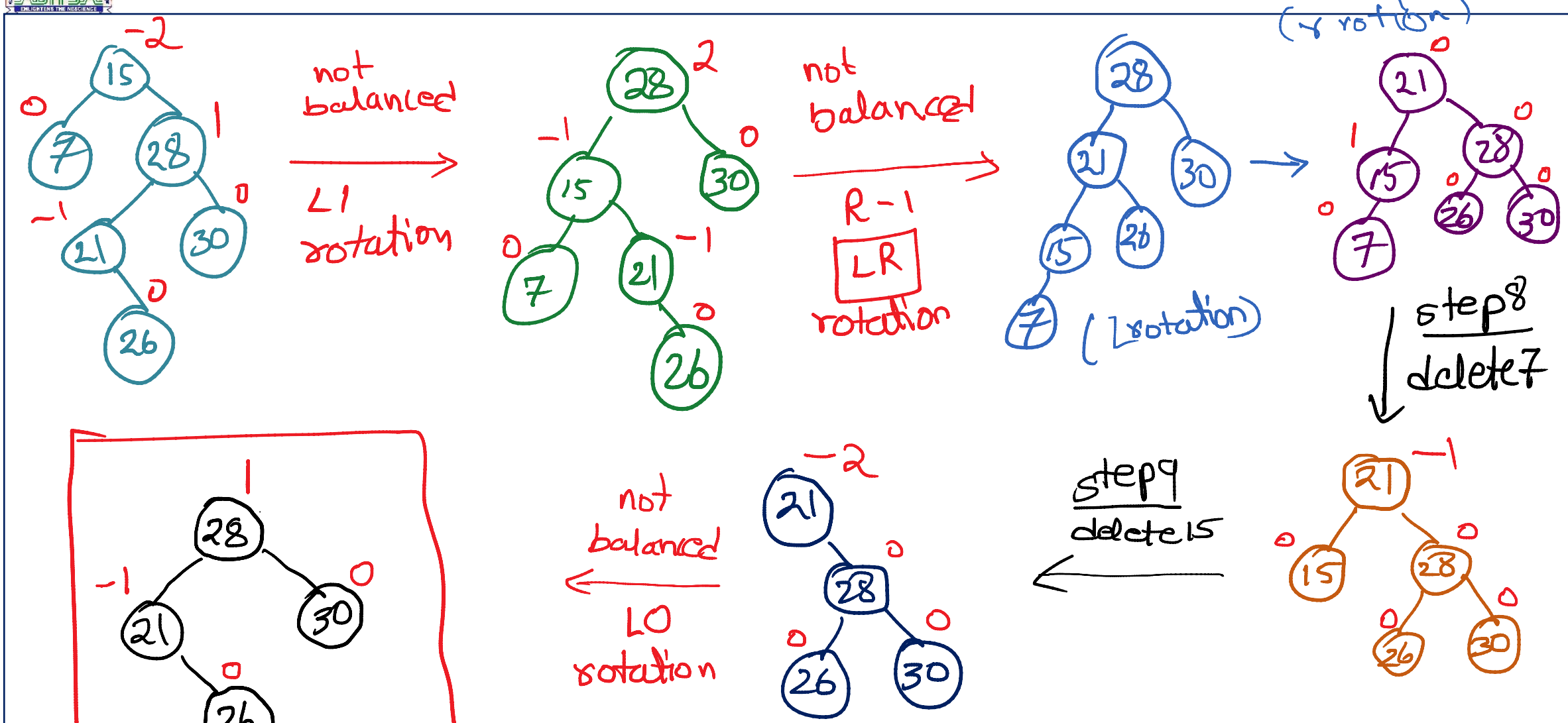


not balanced
L-1 rotation
RL



step 7 delete 14



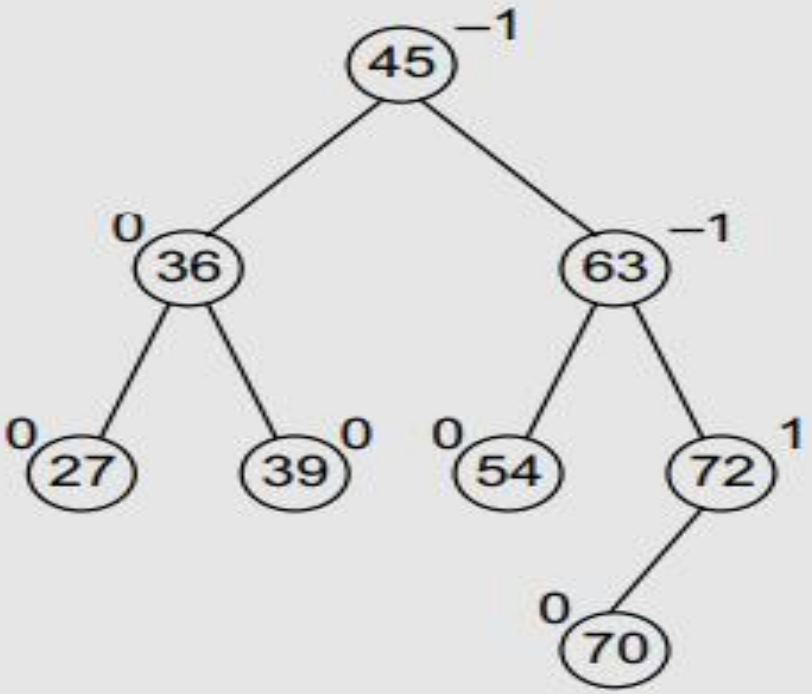

-2
15
0
7
28
1
-1
21
30
0
26
not balanced
L1 rotation
2
28
-1
15
30
0
7
21
-1
0
26
not balanced
R-1
LR
rotation
28
21
30
15
26
7
(L rotation)
(R rotation)
21
0
1
15
28
0
0
26
30
0
7
step8
delete7
21
-1
0
15
28
0
0
26
30
step9
delete15
not balanced
L0 rotation
-2
21
0
28
0
26
30
1
28
-1
21
30
0
26

C program to delete a node from a AVL tree

```
struct node * delete_node(struct node *root, int x)
{
        struct node *ptr;
        if(root==NULL)
                return NULL;
        else
                if(x > root->data) // node at right subtree
                {
                        root->right=delete_node(root->right,x);
                        if(Balancefactor(root)==2)
                                if(Balancefactor(root->left)>=0)
                                        root=LL(root);
                                else
                                        root=LR(root);
                }
else if(x<root->data)
        {
                root->left=delete_node(root->left,x);
                if(Balancefactor(root)==-2)
                        if(Balancefactor(root->right)<=0)
                                root=RR(root);
                        else
                                root=RL(root);
        }
```

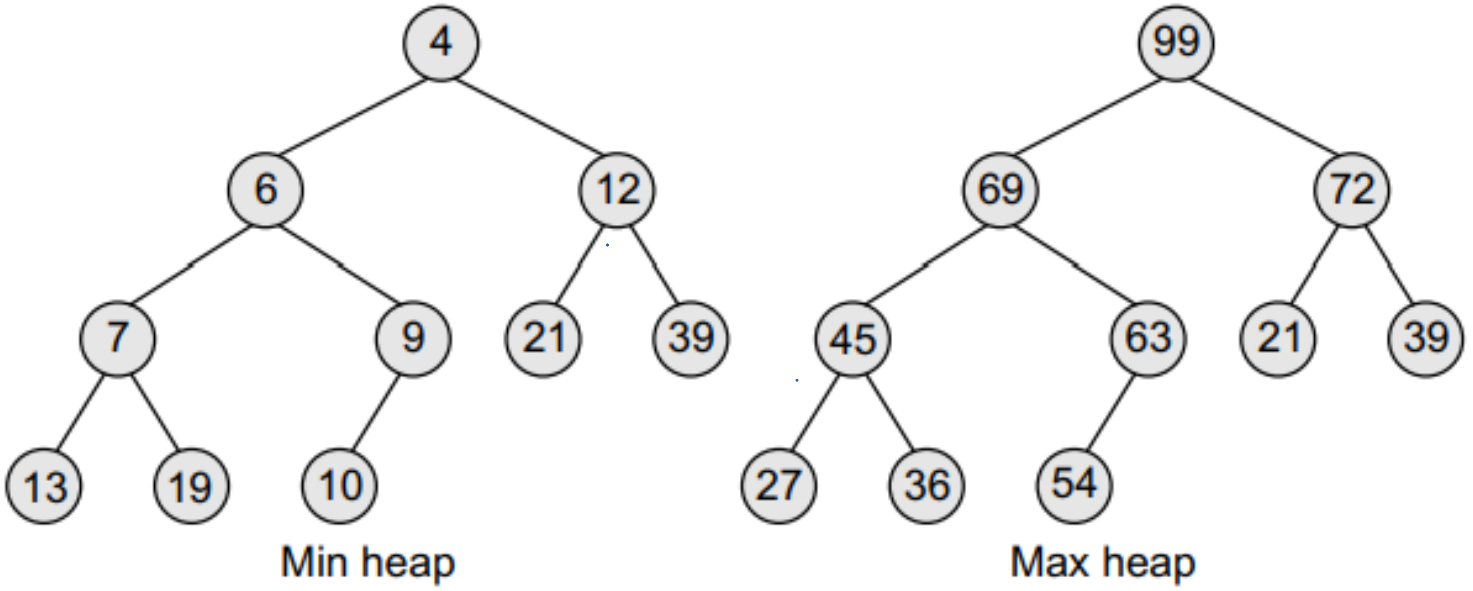
```
                //data to be deleted is found
                else    if(root->right!=NULL)
                { //delete its inorder succesor
                        ptr=root->right;
                        while(ptr->left!= NULL)
                                ptr=ptr->left;
                        root->data=ptr->data;
                        root->right=delete_node(root->right,ptr->data);
                        if(Balancefactor(root)==2)
                        if(Balancefactor(root->left)>=0)
                                root=LL(root);
                        else
                                root=LR(root);
                }
        else
                return(root->left);

root->ht=height(root);
return(root);
}
```

Consider the AVL tree given below and insert 18, 81, 29, 15, 19, 25, 26, and 1 in it.
Delete nodes 39, 63, 15, and 1 from the AVL tree formed after solving the above question.



BINARY HEAPS

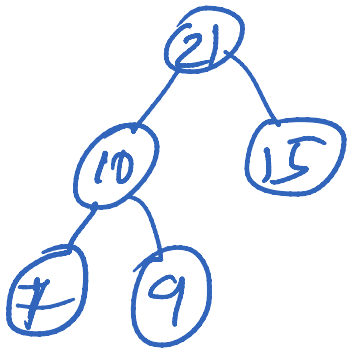
- A binary heap is a **complete binary tree** in which every node satisfies the heap property.
- elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a **max-heap**.
- elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a **min-heap**.

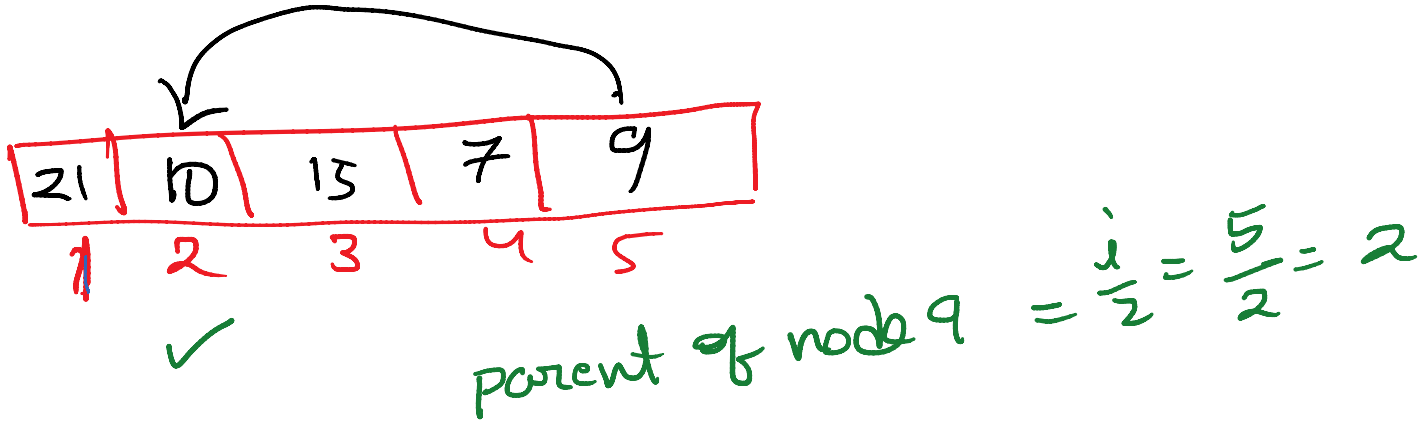


BINARY HEAPS

The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array.
  - if an element is at position i in the array,
  - Its left child is stored at position 2i
  - Its right child at position 2i+1.
  - Its parent stored at position i/2.
- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as log_2 n, where n is the number of elements.
- Heaps are a very popular data structure for implementing priority queues.





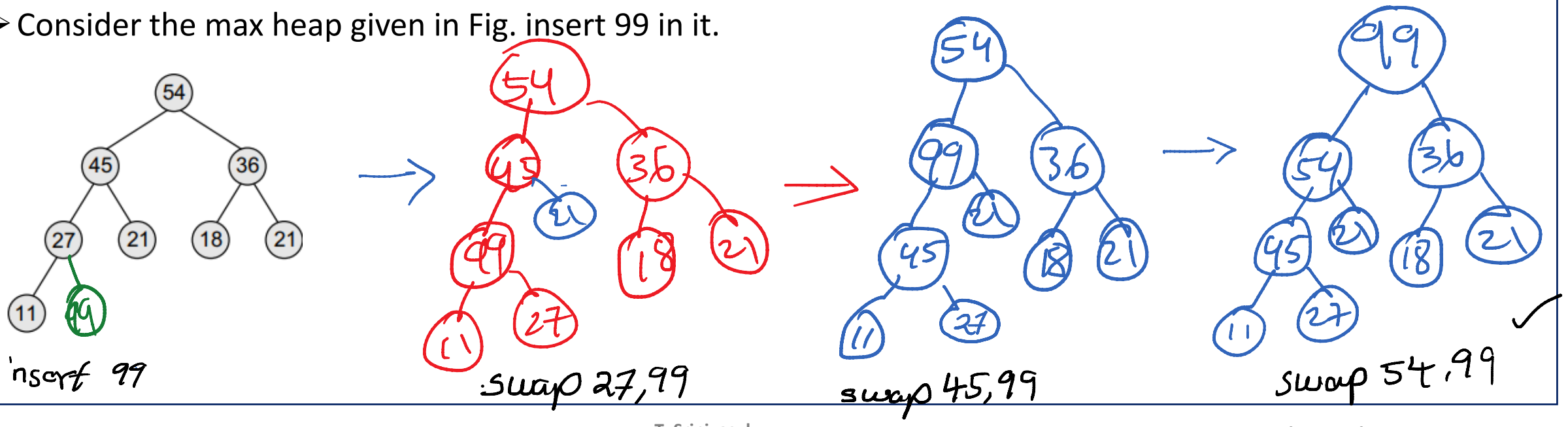
Inserting a New Element in a Binary Heap

- Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

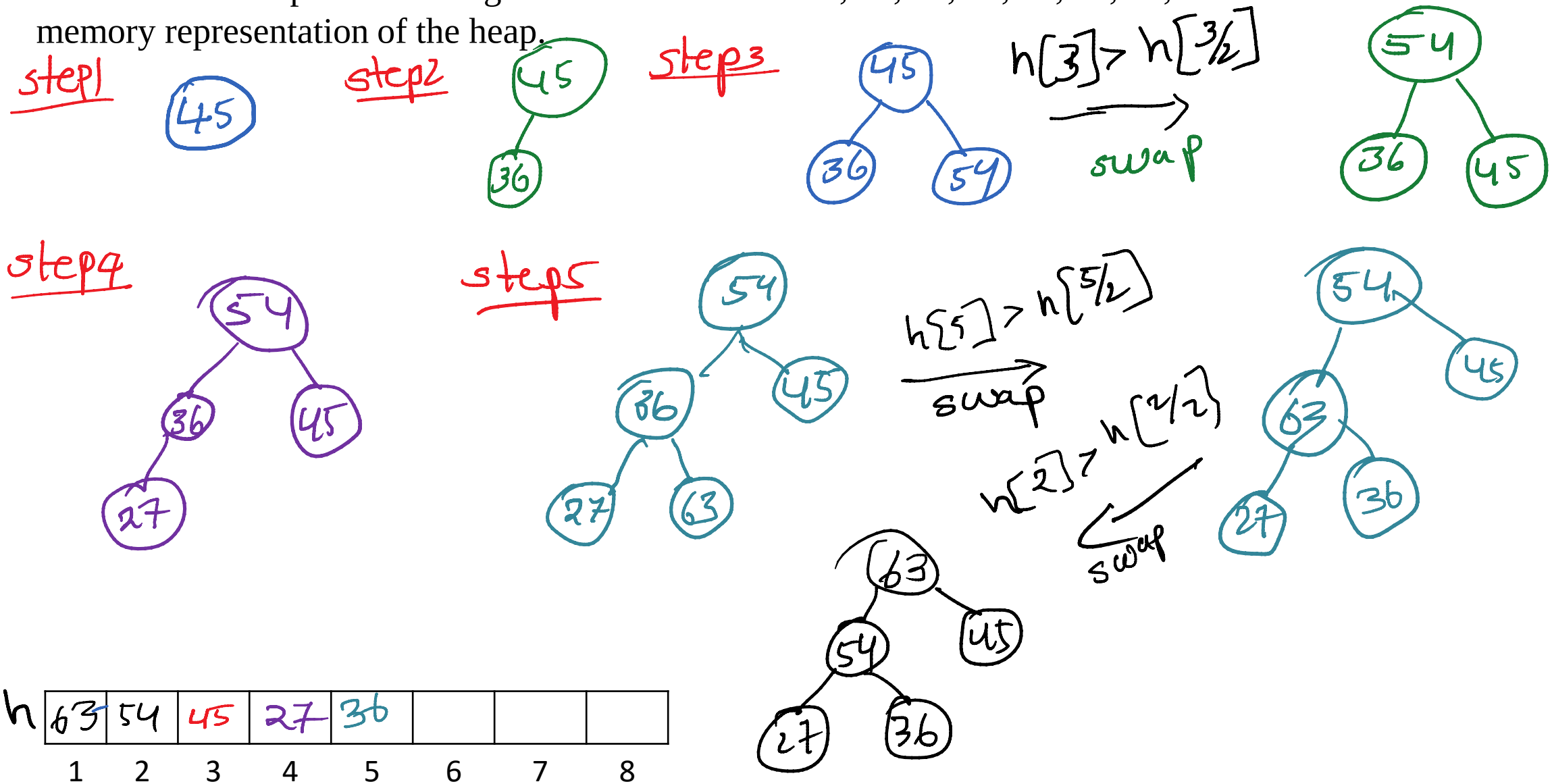
1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

   To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

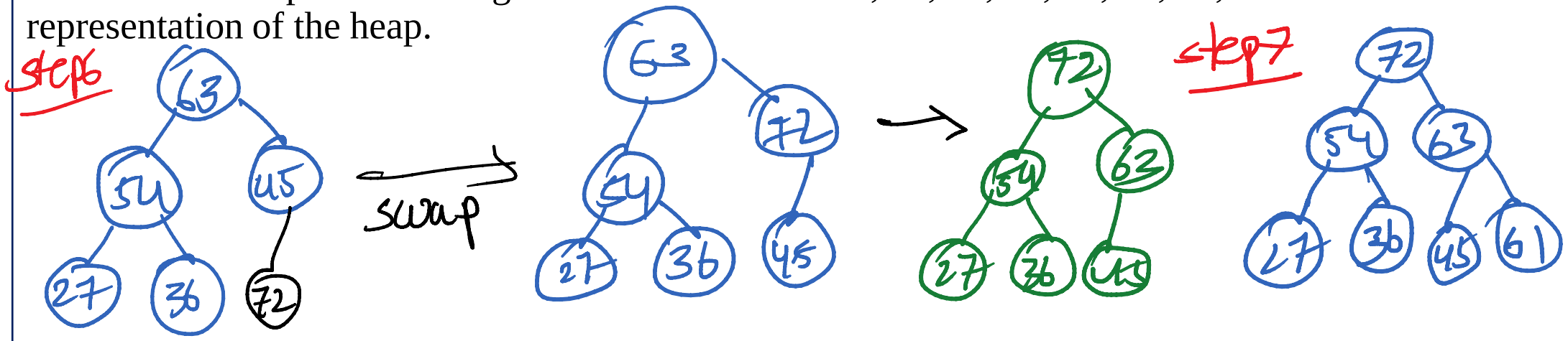
- Consider the max heap given in Fig. insert 99 in it.

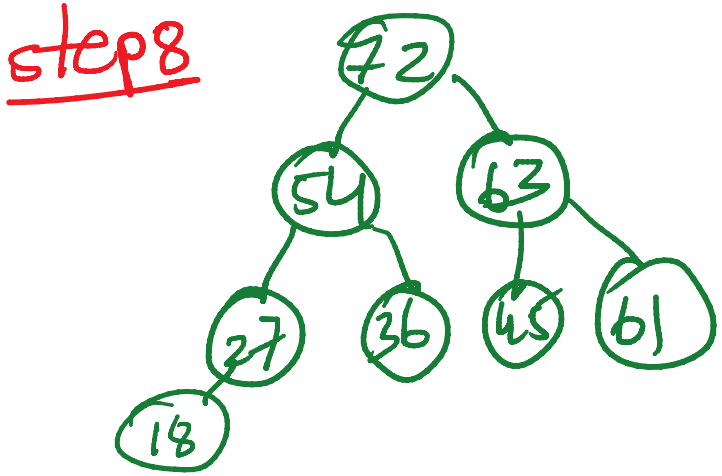


- Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.



Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.



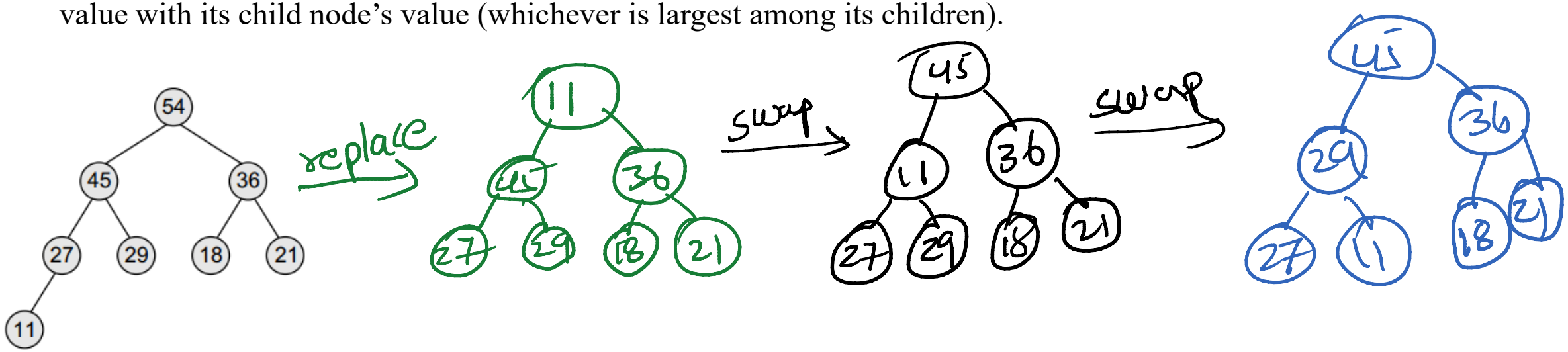


1	2	3	4	5	6	7	8
72	54	63	27	36	45	61	18

Deleting an element from a Binary Heap

- Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).



HEAP SORT

- we already know how to build a heap H from an array, how to insert a new element in an already existing heap, and how to delete an element from H.
- Now, using these basic concepts, we will discuss the application of heaps to write an efficient algorithm of heap sort (also known as tournament sort) that has a running time complexity of O(n log n).
- Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

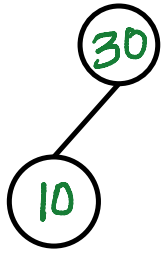
  In phase 1, build a heap H using the elements of ARR.

  In phase 2, repeatedly delete the root element of the heap formed in phase 1.
- In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order.

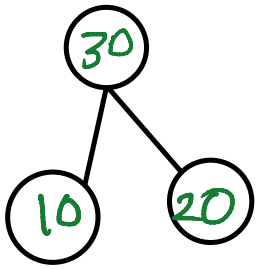
Example: By using heap sort, sort the following values in ascending order 30 10 20 15 40

Create heap:

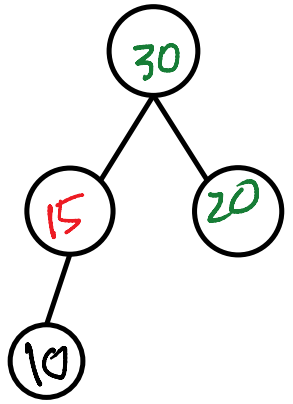
30				
1	2	3	4	5



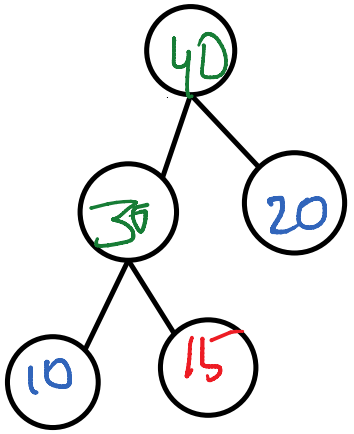
30	10			
1	2	3	4	5



30	10	20		
1	2	3	4	5

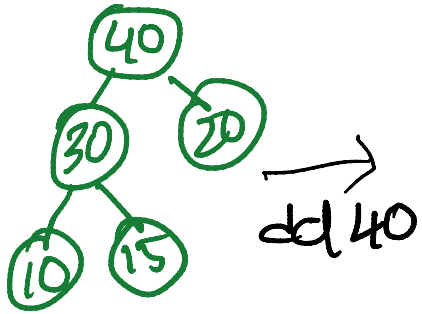


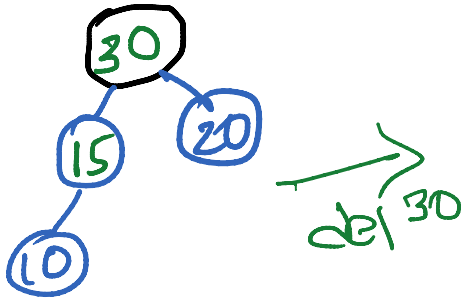
30	15	20	10	
1	2	3	4	5

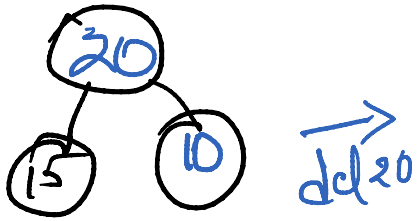


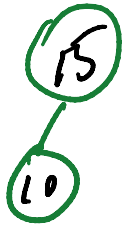
40	30	20	10	15
1	2	3	4	5

Delete a root node:









40	30	20	10	15
1	2	3	4	5

30	15	20	10	40
1	2	3	4	5

20	15	10	30	40
1	2	3	4	5

15	10	20	30	40
1	2	3	4	5

10	15	20	30	40
1	2	3	4	5

C program to implement Heap sort:

```
#include <stdio.h>
#define MAX 10
void HeapUp(int *,int);
void HeapDown(int*,int,int);

int main()
{
	int Heap[MAX],n,i,j;
	printf("\n Enter the number of elements : ");
	scanf("%d", &n);
	printf("\n Enter the elements : ");
	for(i=1;i<=n;i++)
	{
		scanf("%d", &Heap[i]);
		HeapUp(Heap, i);
	}
// Delete the root element and heapify the heap
	j=n;
	for(i=1;i<=j;i++)
	{
		int temp;
		temp=Heap[1];
		Heap[1]= Heap[n];
		Heap[n]=temp;
	// The element Heap[n] is supposed to be deleted
		n = n-1;
	HeapDown(Heap,1,n);
	}
	n=j;
	printf("\n The sorted elements are: ");
	for(i=1;i<=n;i++)
		printf("%4d",Heap[i]);
	return 0;
}
```

```
void HeapUp(int *Heap, int index)
{
	int val = Heap[index];
	while( (index>1) && (Heap[index/2] < val) )// Check parent's value
	{
		Heap[index]=Heap[index/2];
		index /= 2;
	}
	Heap[index]=val;
}
```

```
void HeapDown(int *Heap, int index, int n)
{
	int val = Heap[index];
	int j=index*2;
	while(j<=n)
	{
		if( (j<n) && (Heap[j] < Heap[j+1]))
			j++;
		if(Heap[j] < Heap[j/2]) //parent's value
			break;
		Heap[j/2]=Heap[j];
		j=j*2;
	}
	Heap[j/2]=val;
}
```