**Binary Search Tree**

**Balanced Tree**

**AVL**

**I. Tree**

\* A tree <T> (Tree) is :

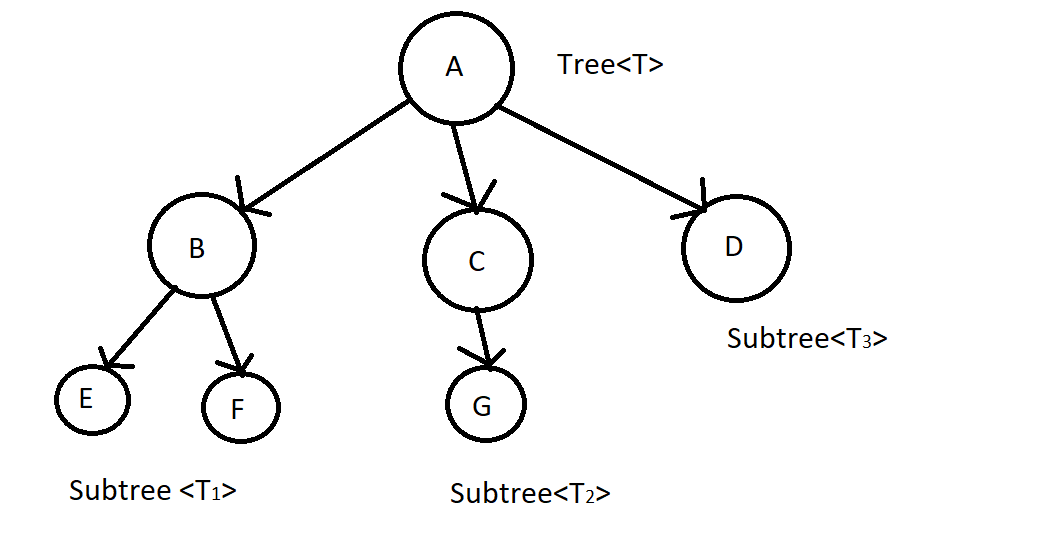
+A set of values ​​of the same type, elements of the same value are stored by a node, called node p1,p2,...

+ If a tree has no elements, it is called an empty tree.

+If the tree is not empty, there are some caveats:

\* Each tree has only 1 root node. The root node is the starting node of the tree.

\* The remaining nodes are divided into non-intersection sets (Ti ,Ti+1,…). Each <Ti> is called a subtree of <T> tree .Example:

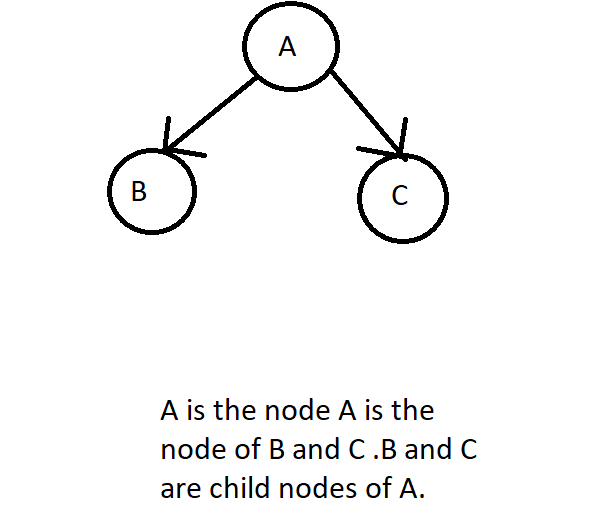


\*Properties:

+The root node has no parent node.

+Leaf node (external node):is a node that does not have any child nodes.

+ Parent node: is the node that we will be processed to do something before reaching another node that it can go to. Child node: It is the node that when the parent node has been processed, it comes to it.Example:



+Each node has only one node parent.

+Each node can have multiple children.

+Trees do not count cycles . That is, when a node 1 node goes to a child node, the child node cannot have any path in the tree to go against the node above. If possible it is called a graph.

+Node: is an element in tree. Node can contain any data. Each node can contain many types of data. For example, 1 node can contain 1 data as name, another data as age,... And importantly, all nodes must be the same in terms of data but may differ in the values ​​in the data that the node holds. That's why we say a tree is a data structure that collects elements of the same type.

+Branch: link 2 nodes together.

+ Sibling nodes: is a node with the same parent node.

+Degree of node: number of children of a node.

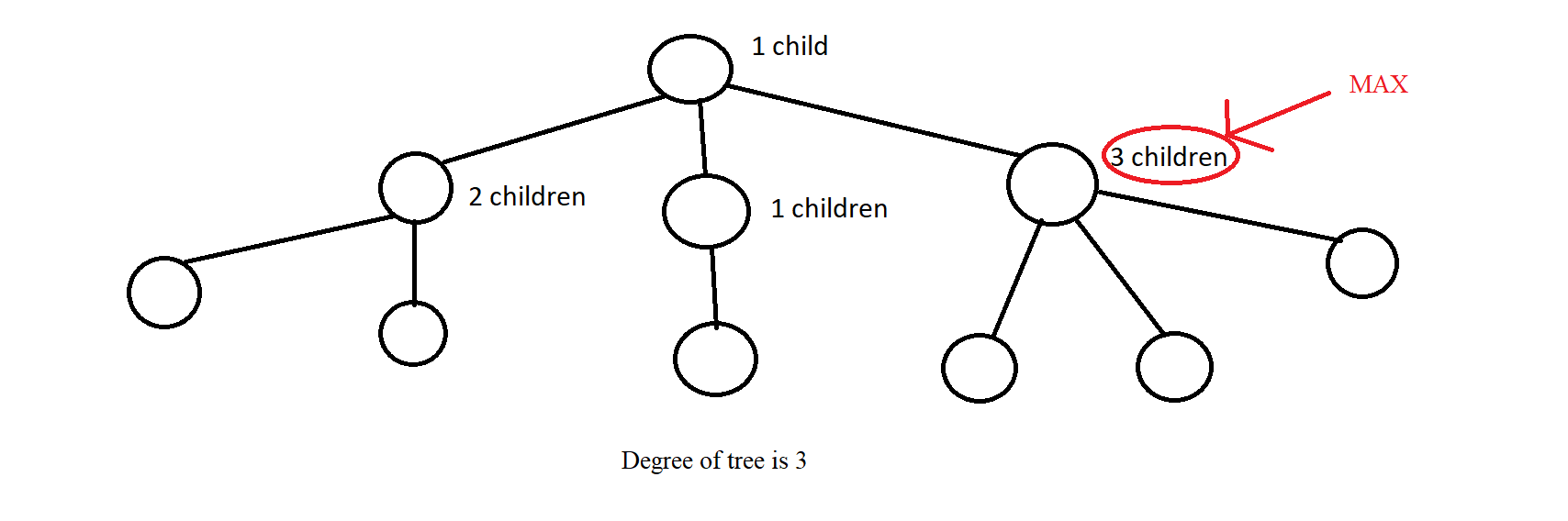
+Internal node: is a node that has both a parent node and a child node.

+Subtree: is a subtree of the original tree. Also starts with a node.

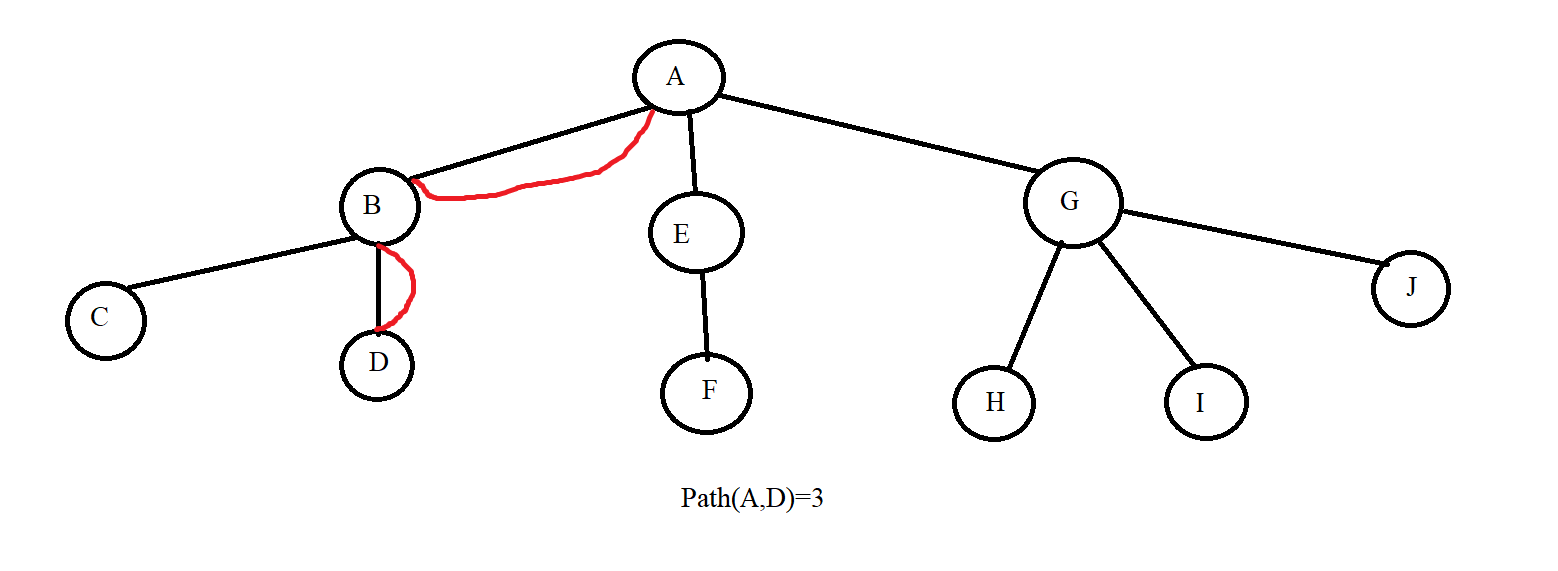
+Degree of tree: is the maximum value of the degree of node in the tree. When we interpret it with a formula, we can write:

Degree (<T>) = max {degree (pi) / pi ∈ <T>}

Example:



+Path between node pi to node pj: is a series of nodes from pi to pj and on those nodes there must be branches to connect 2 nodes together.Example:

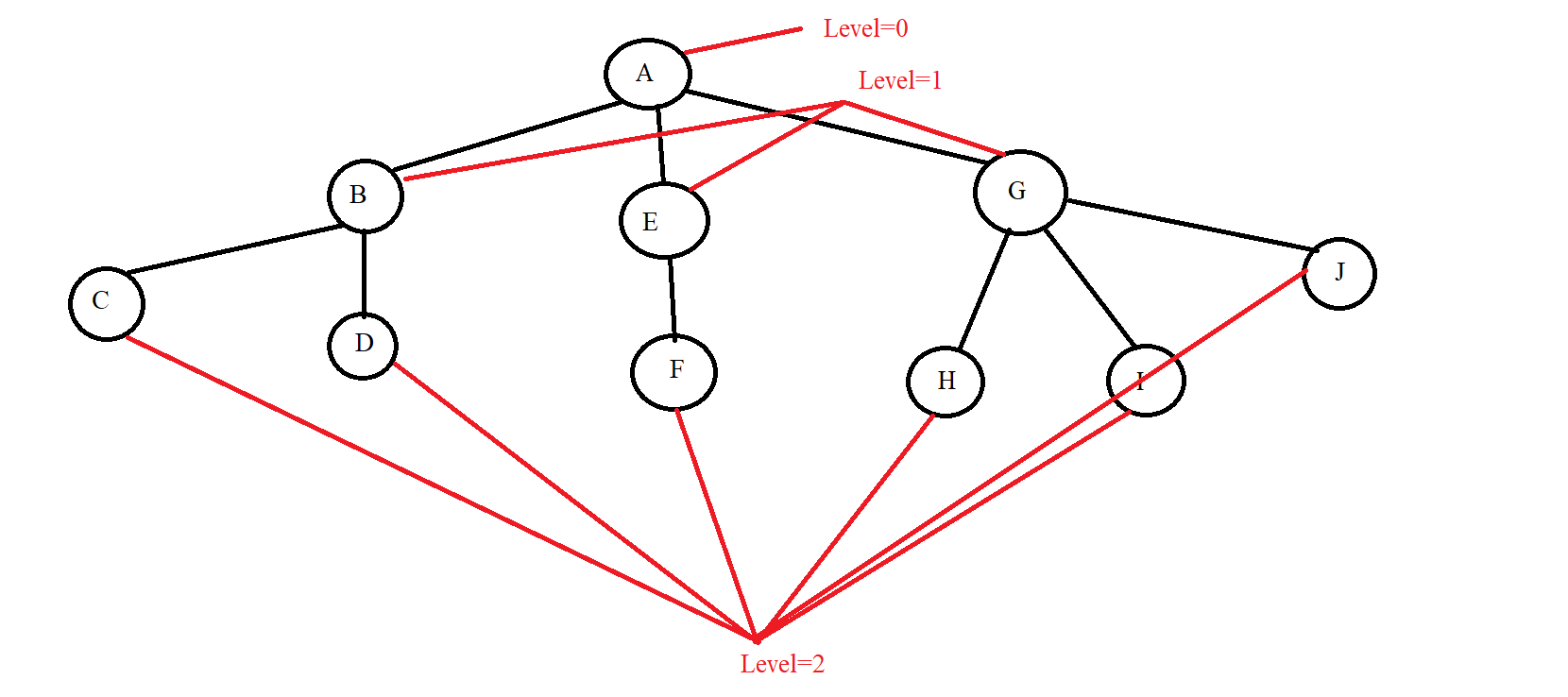


+Level: Just like how to calculate the floor of a whole tower, we apply it to trees as well. Where the ground floor is the root with level =0 and the upper floors are considered the bottom nodes of a node, the level is calculated as the parent node's level plus 1. From this we can deduce the formula:

Level(p) = 0 if p = root

Level(p) = 1 + level(parent(p)) if p! = Root

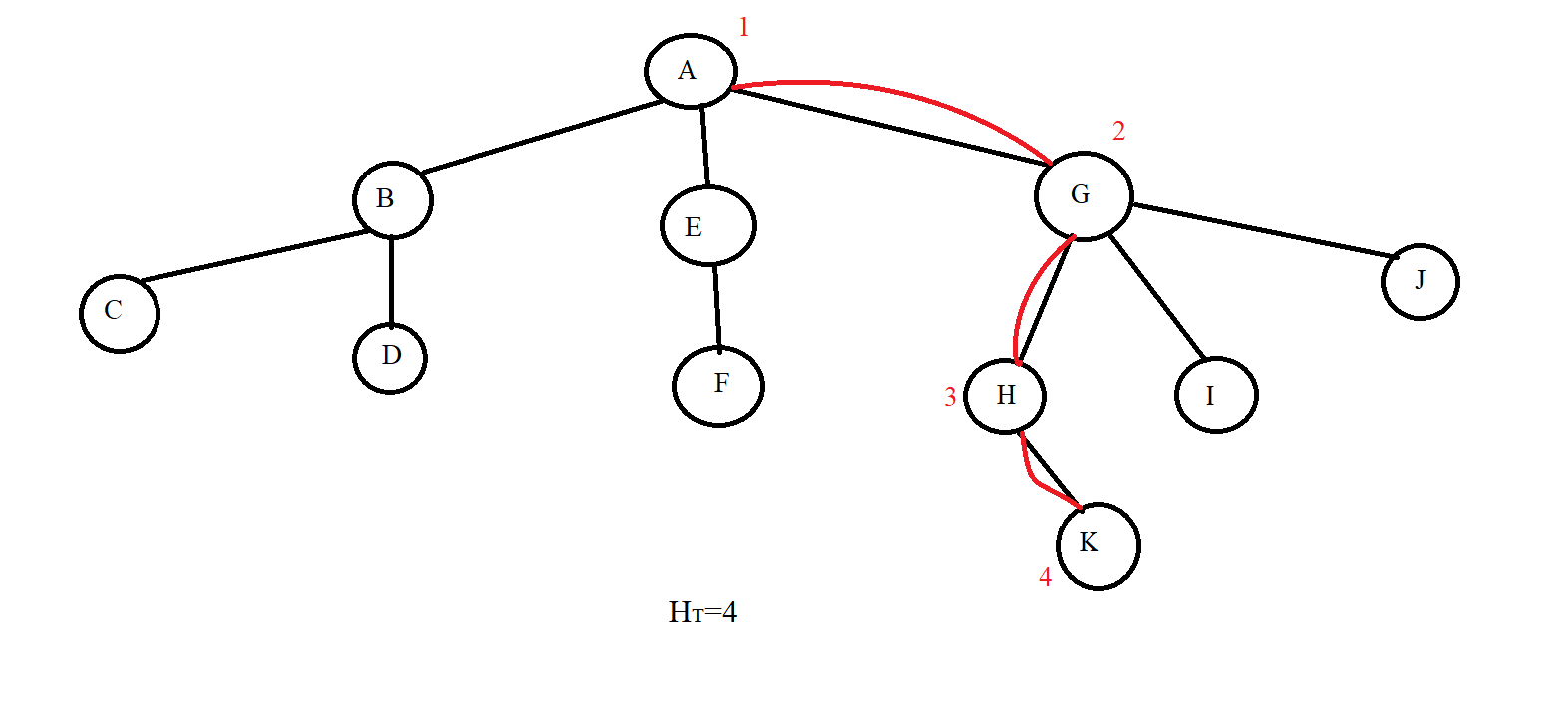
Example:



+Height of tree (hT): Maximum value of path from root to leaf node. We have the mathematical formula:

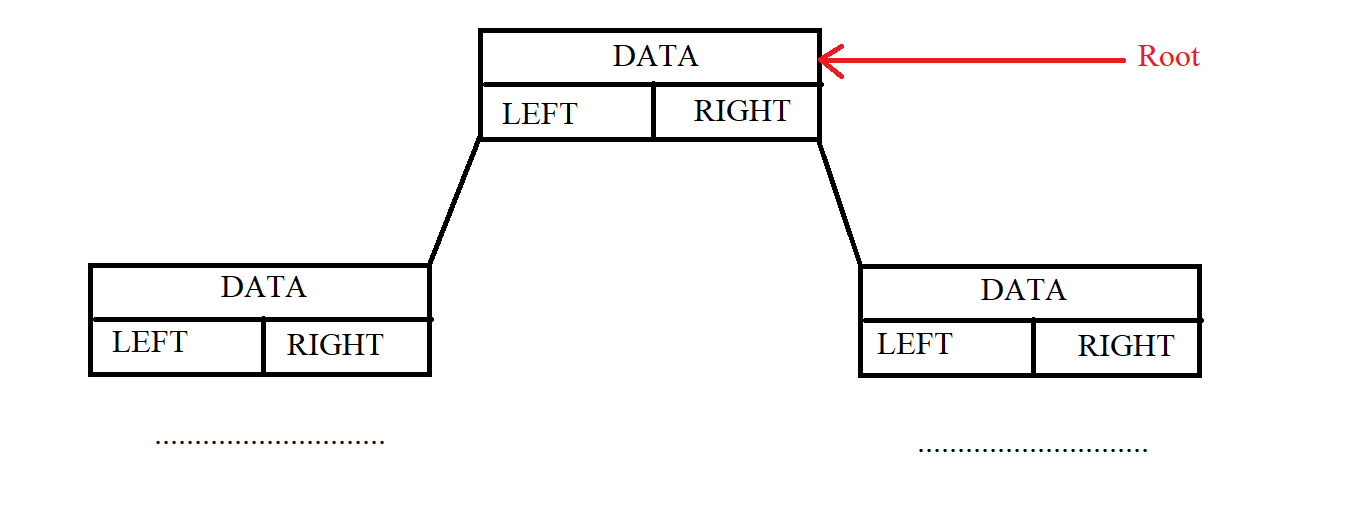
hT = max {Path(root, pi) | pi is the leaf node ∈ <T>}

Example:



**II. Binary tree**

\*A tree <T> (Tree) is a binary tree when the degree of the tree is 2.

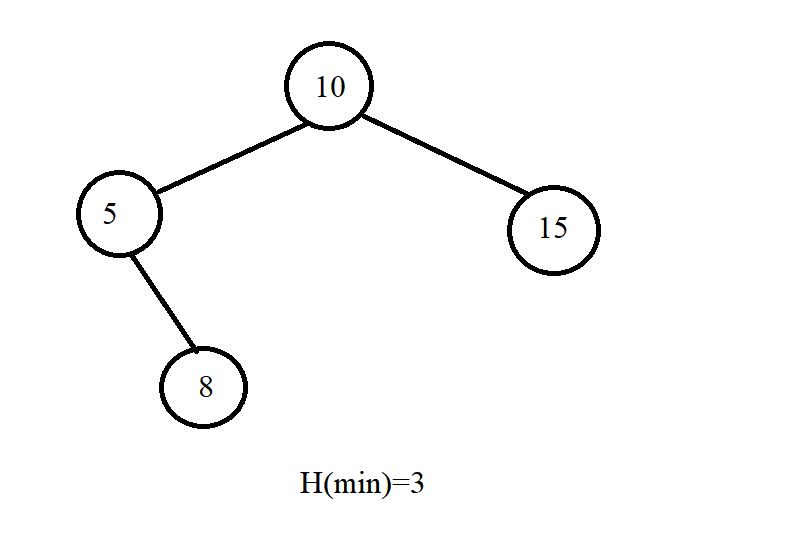


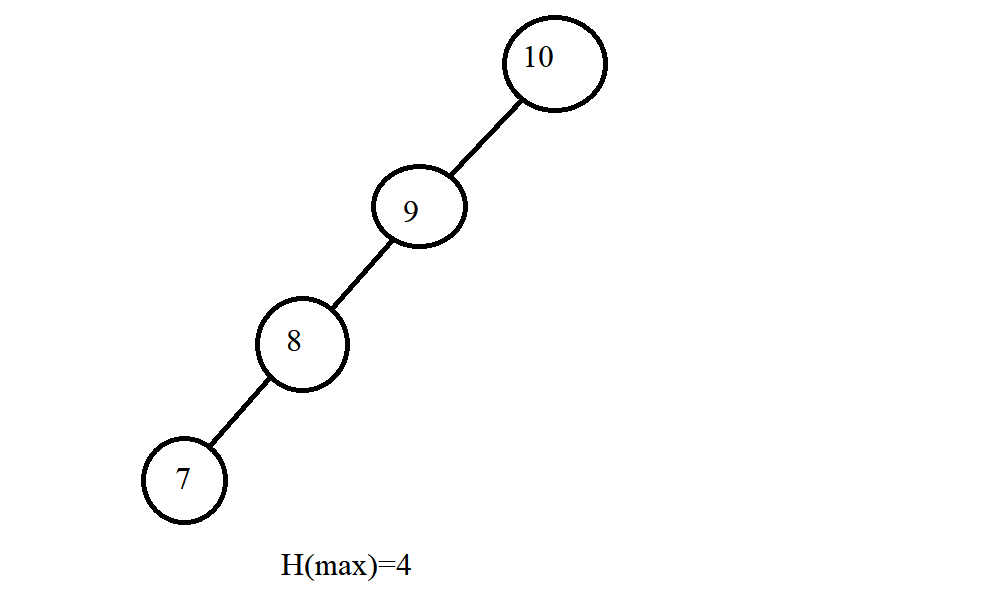
\*The height of a binary tree has N nodes:

hT(max) = N

hT(min) = + 1

Example: with N=4





\*There are 2 ways to organize a binary tree:

+Array.

+ Structure pointers.

\* Here it is best to use pointers because it is easier to use pointers to delete, insert and search. Pointers make it easier to know the child node of a node.

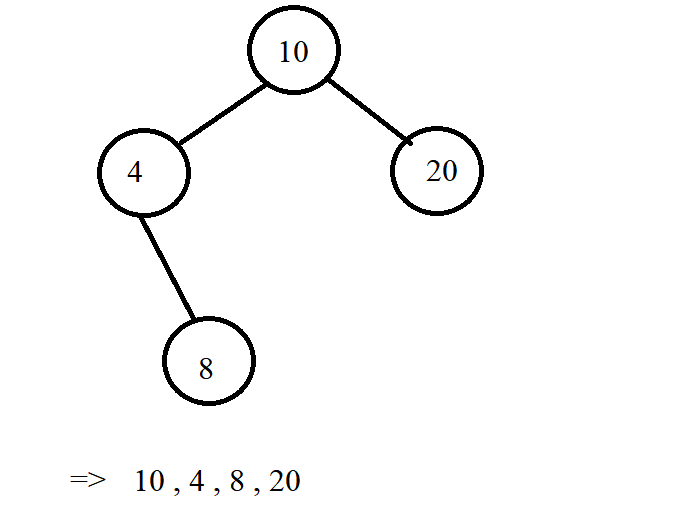


\*Traverse in Tree: We have many different ways of traversing such as NLR, LRN, RNL, ... but we only consider 3 basic traversals: NLR, LRN and LRN.

+ Pre-Order (NLR): It will process the current node and then recursively call the left node and finally the right node. Put the case we want in the buttons in we have the following examples

|  |
| --- |
|  |

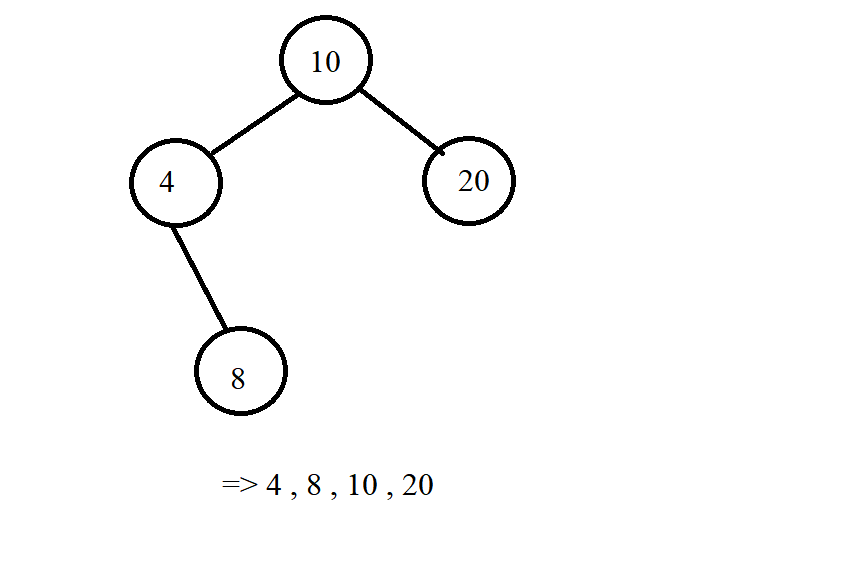
Example:



+ In-Order (LNR) : It will recursively call the left node. Then it will process the current node and finally recursively call the right node.

|  |
| --- |
|  |

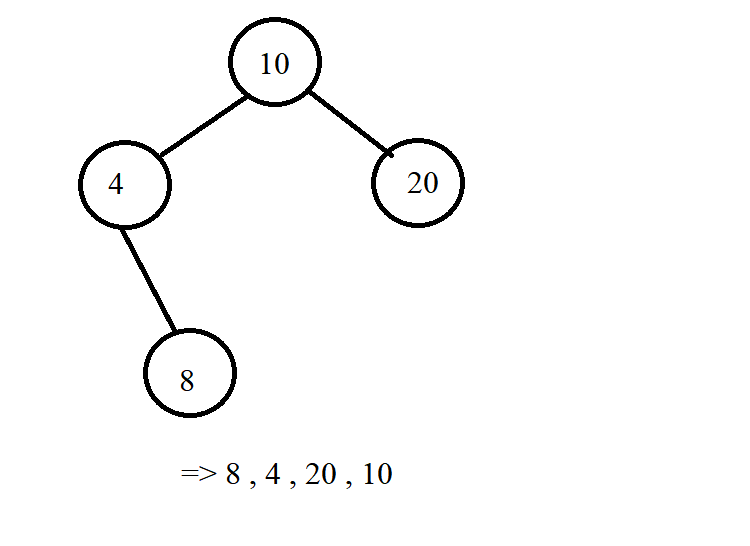
Example:



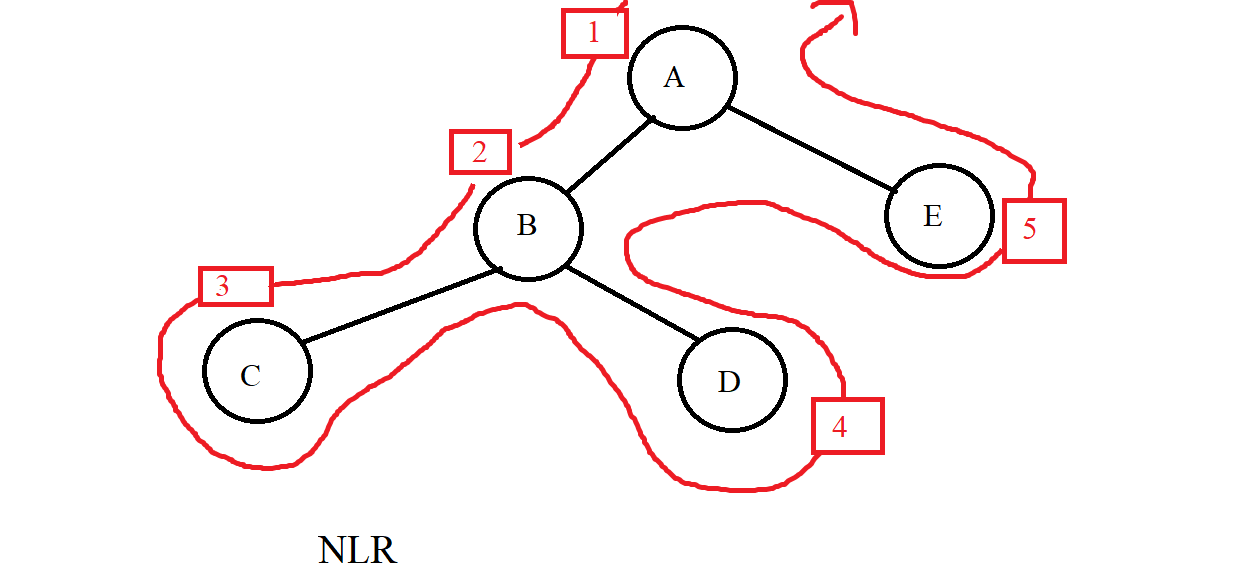
+Post-Order (LRN): We will recursively call the left node, go to the right node and finally process the current node.

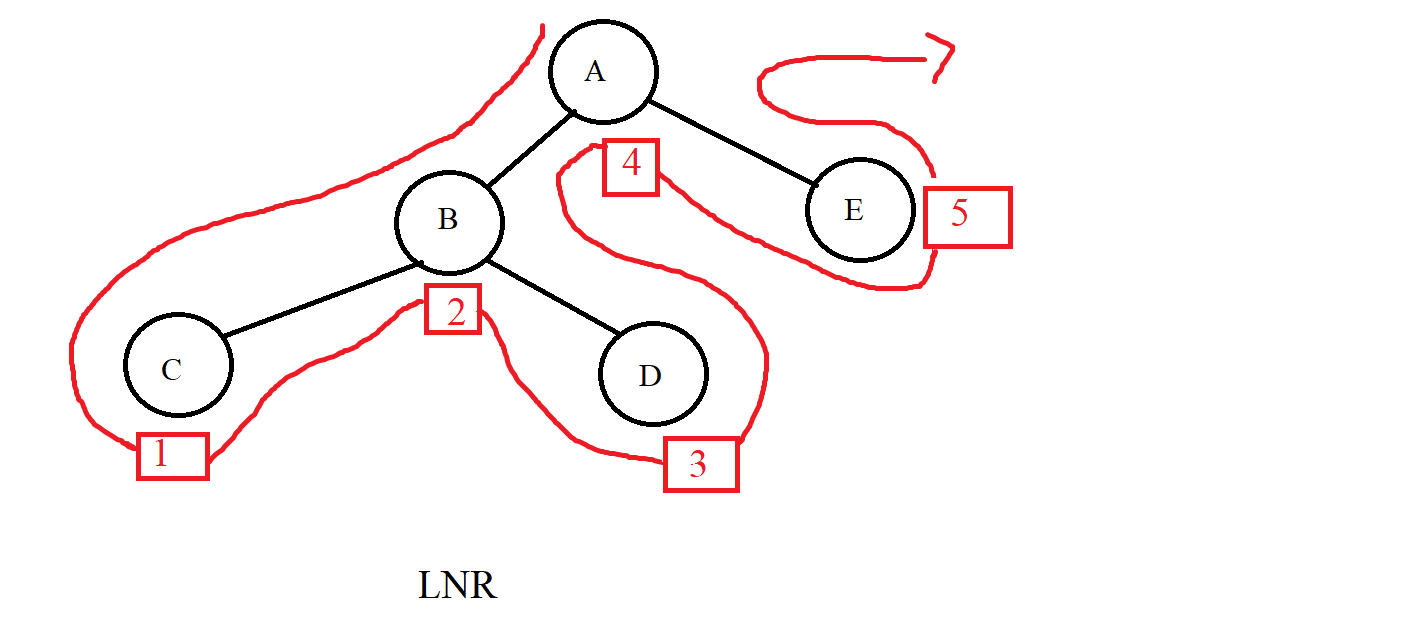
|  |
| --- |
|  |

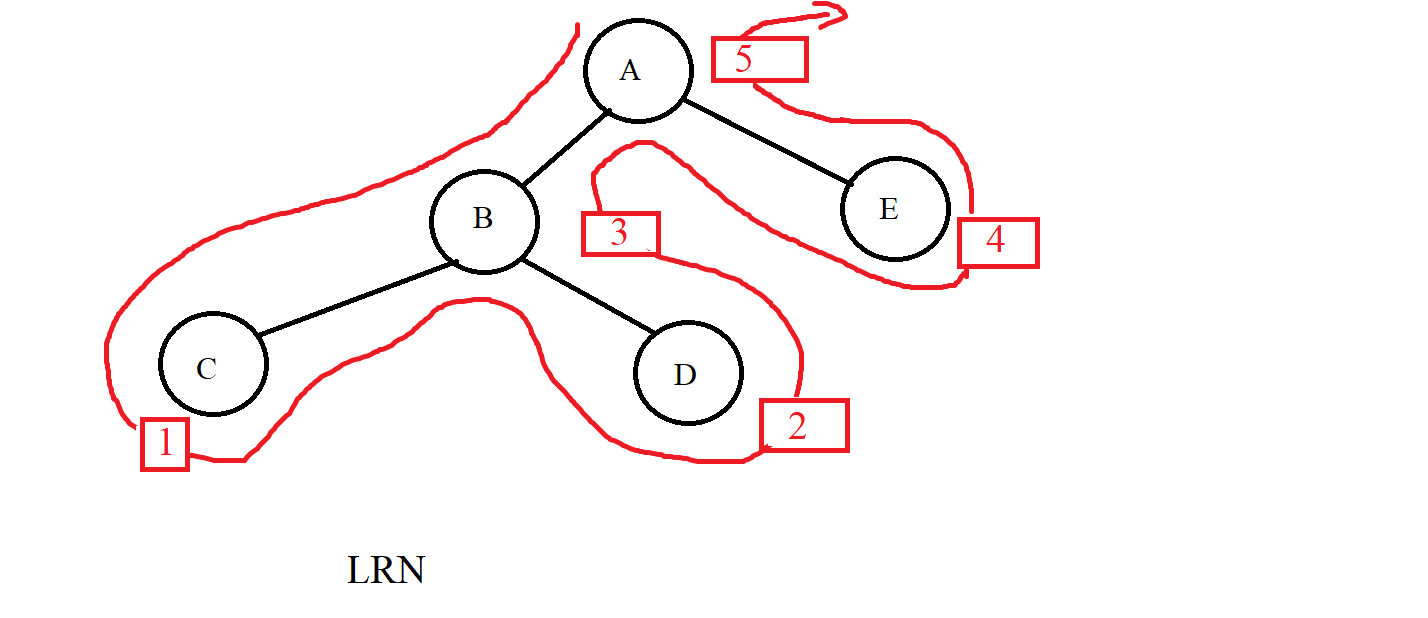
Example:



*\*Note*: In addition to printing nodes like the examples above, we can change the place (cout<< pCurr->Data << " ") to whatever command we want to handle the query , the subject we want. Below is an illustration of the traversal of the three methods above.







**III.Binary search tree (BST)**

**-** The binary search tree is:

+A binary tree : That is, each node has at most 2 child nodes.

+Each value node exists only once in the tree.

+ Each node must satisfy the condition:

. All nodes of the left subtree must be less than the current node.

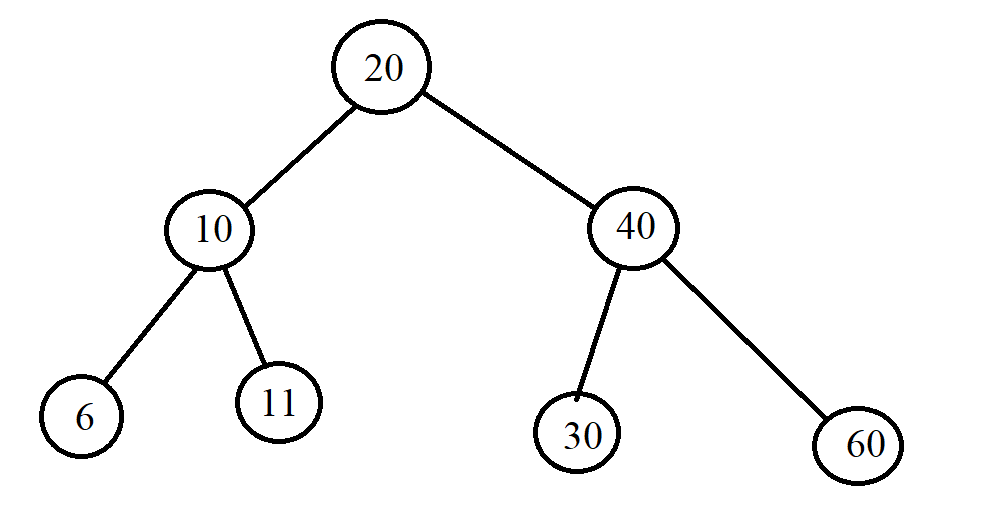
. All nodes of the right subtree must be less than the current node.

. Assuming the current node is node (p), then we have :

∀L ∈ p-> pLeft: L-> Data <p-> Data

∀R ∈ p-> pRight: R-> Data> p-> Data

Example:



- Operations in BST:

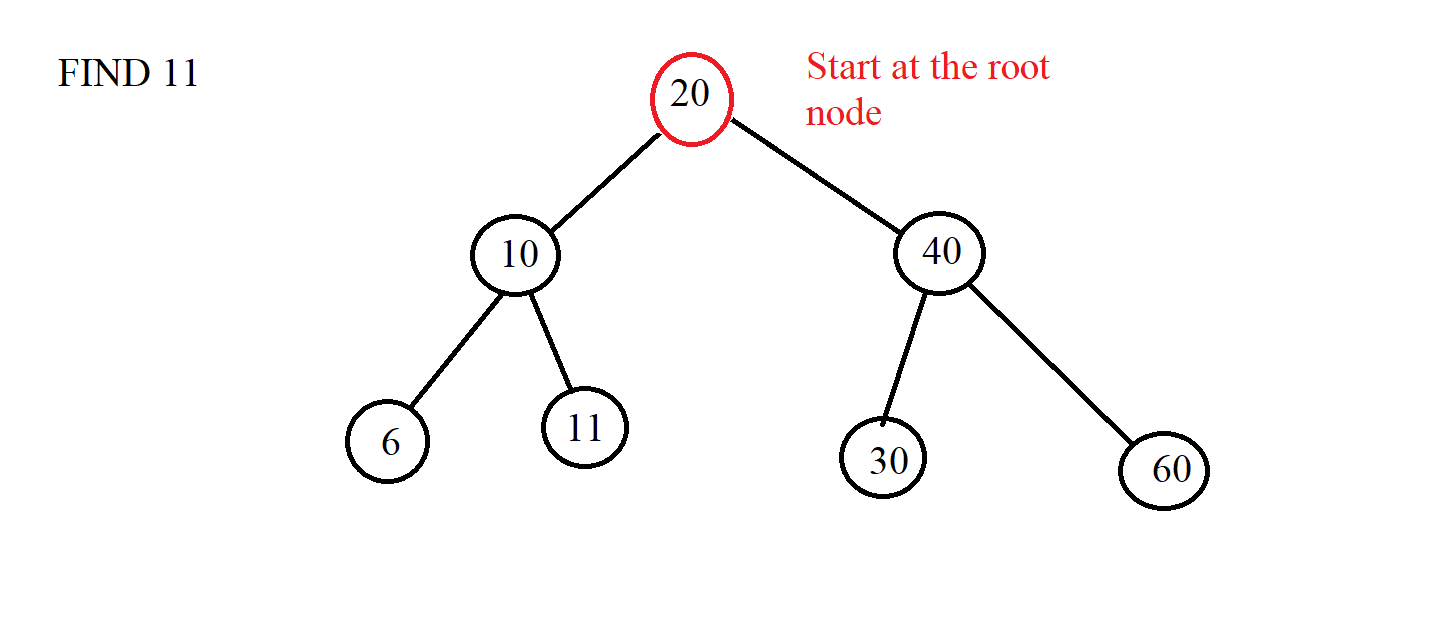
+ Create a empty tree.

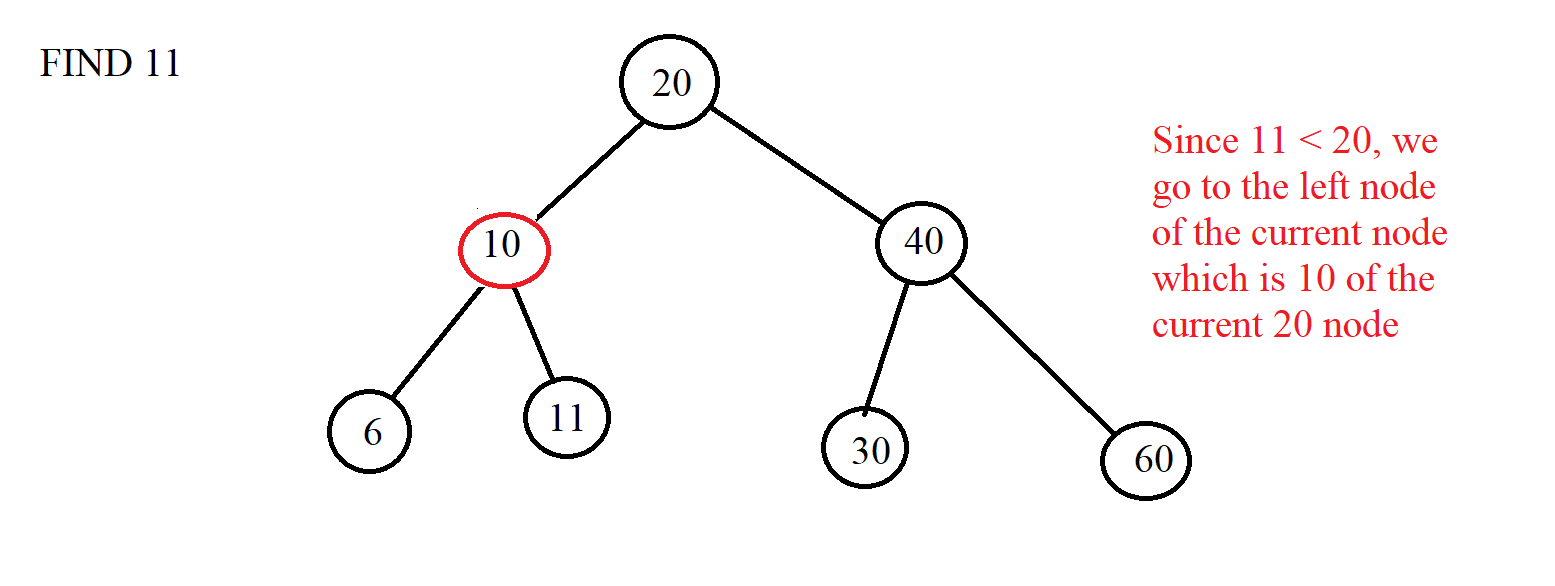


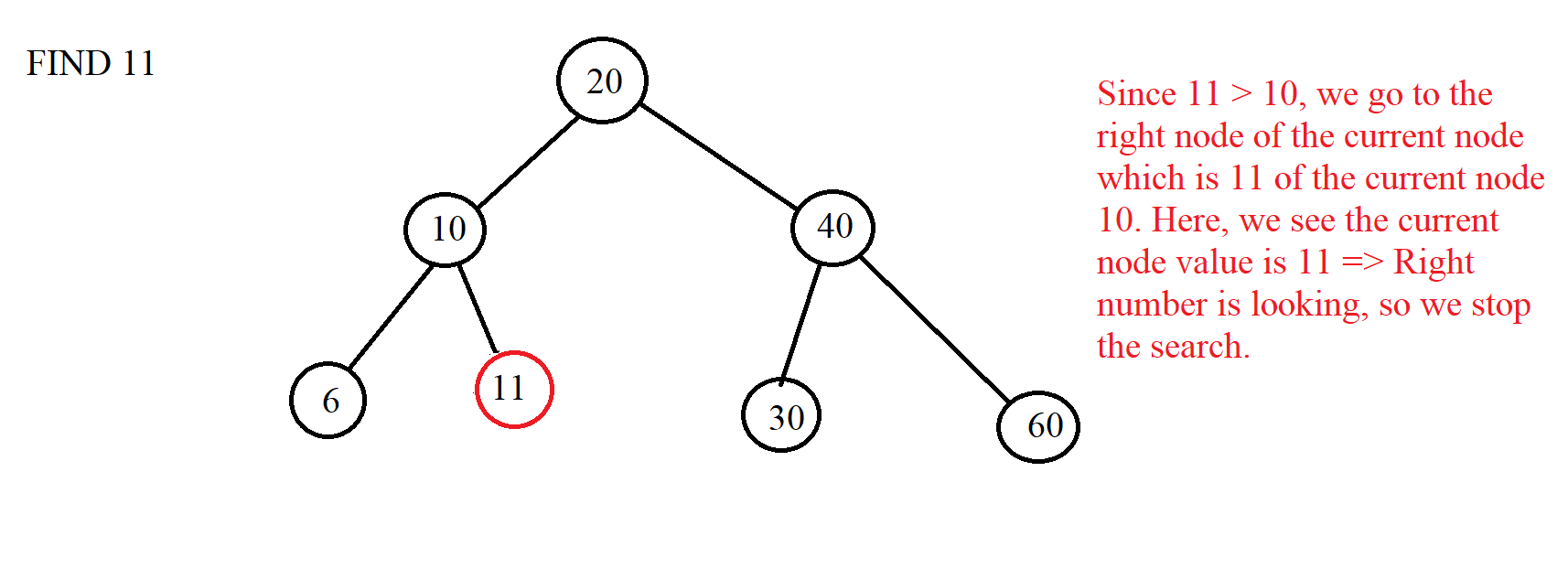
+ Check the empty tree.



+Find an element : Using the same condition for each node in the BST that the left child is less than the current node and the right node is greater than the current node, we will apply it to finding the element in the BST tree. Accordingly, if the current node that the tree is traversing has data larger than the element we are looking for, we will recursively go to the left node and vice versa, we will recursively go to the right. If we find the value we want to find, we stop and if we can't find it, the node is Null. Example:



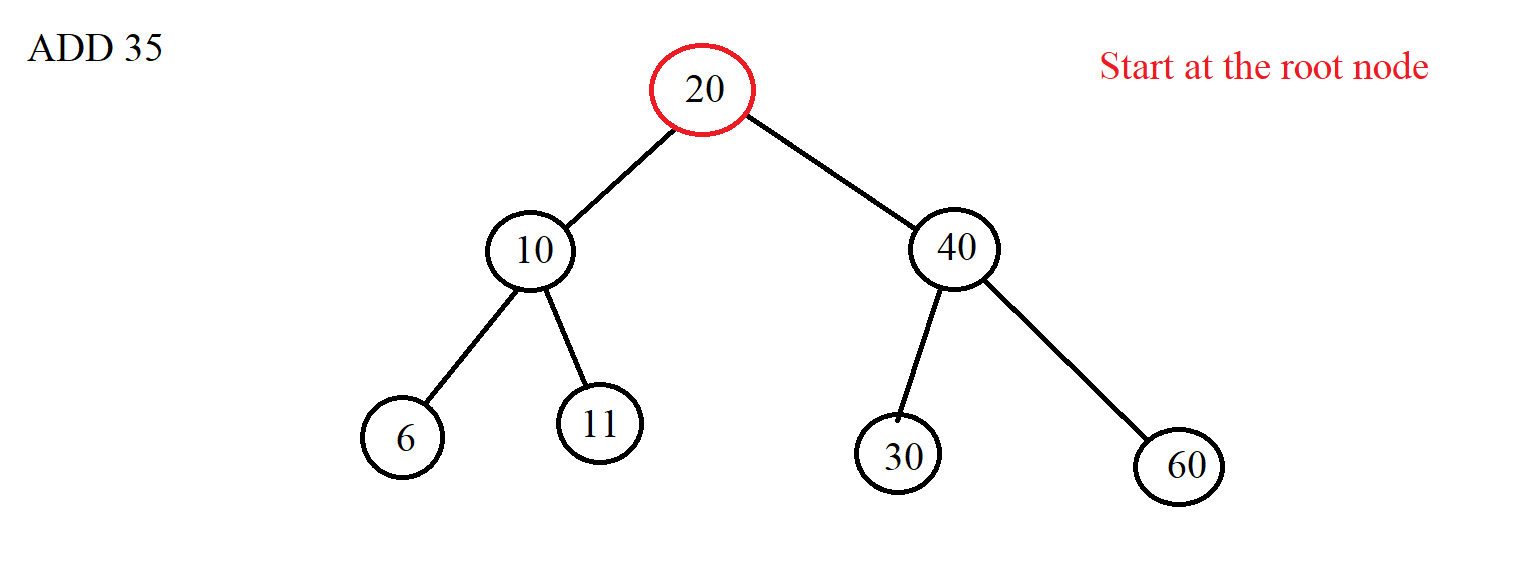


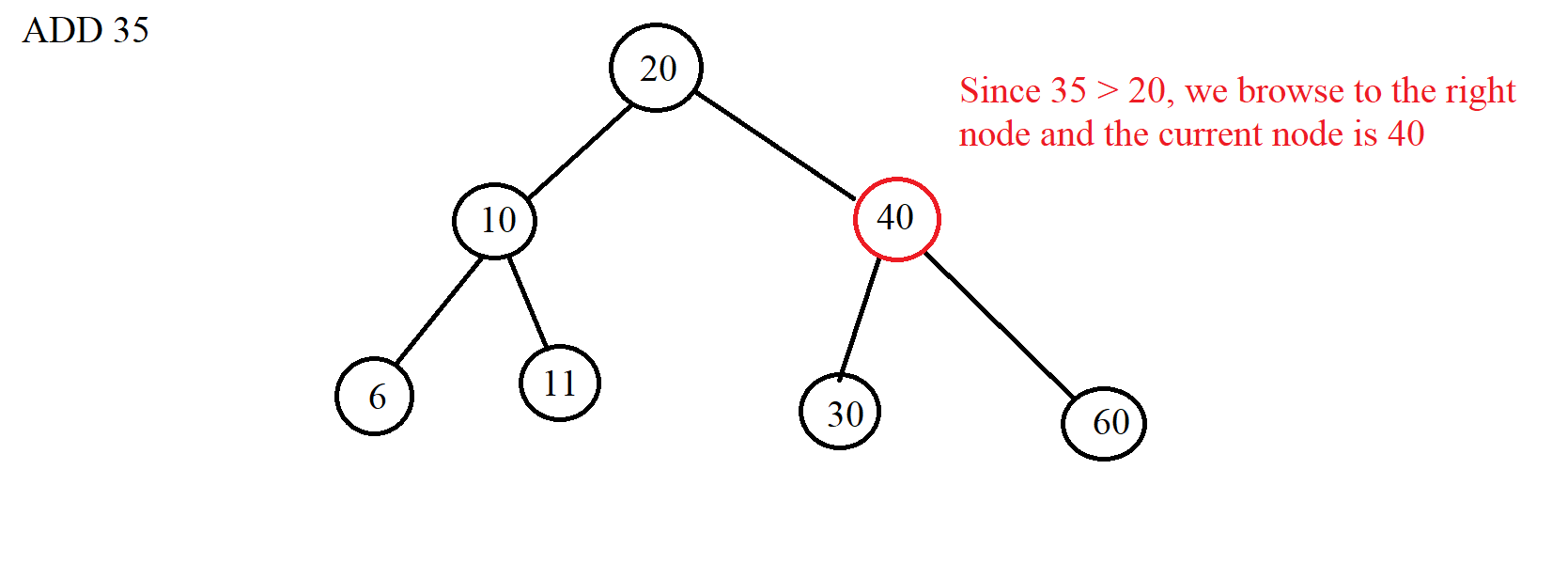


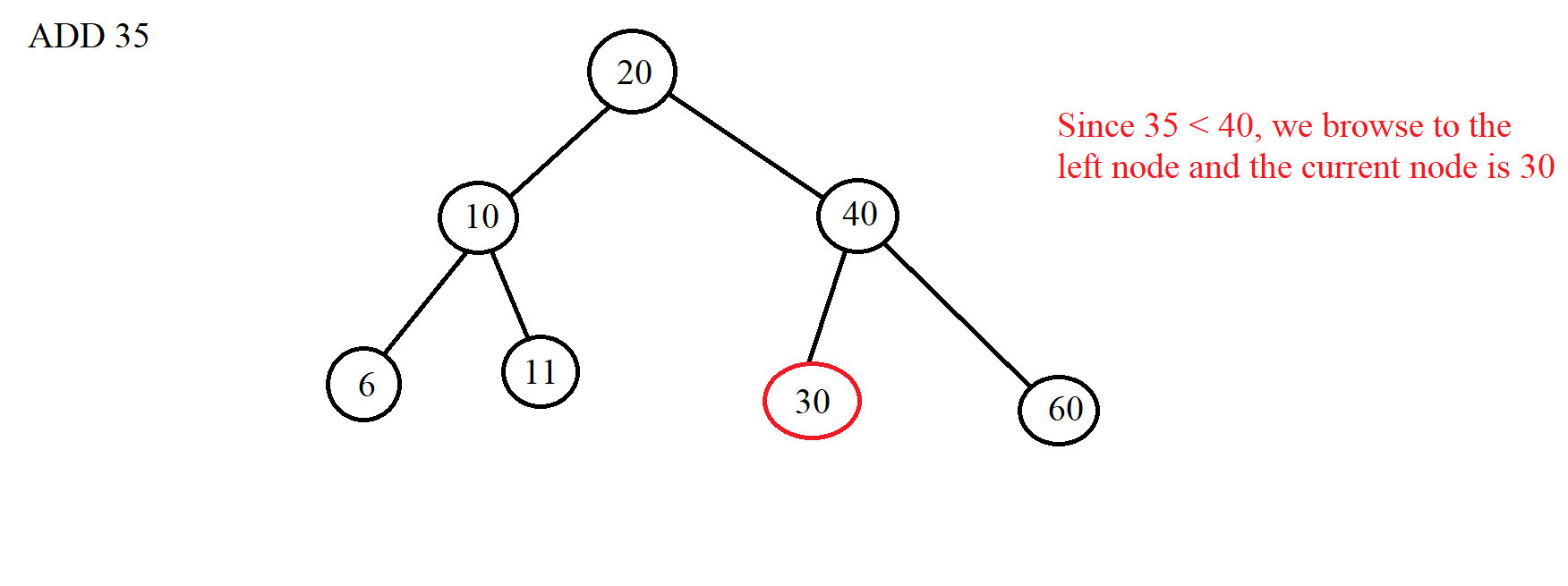


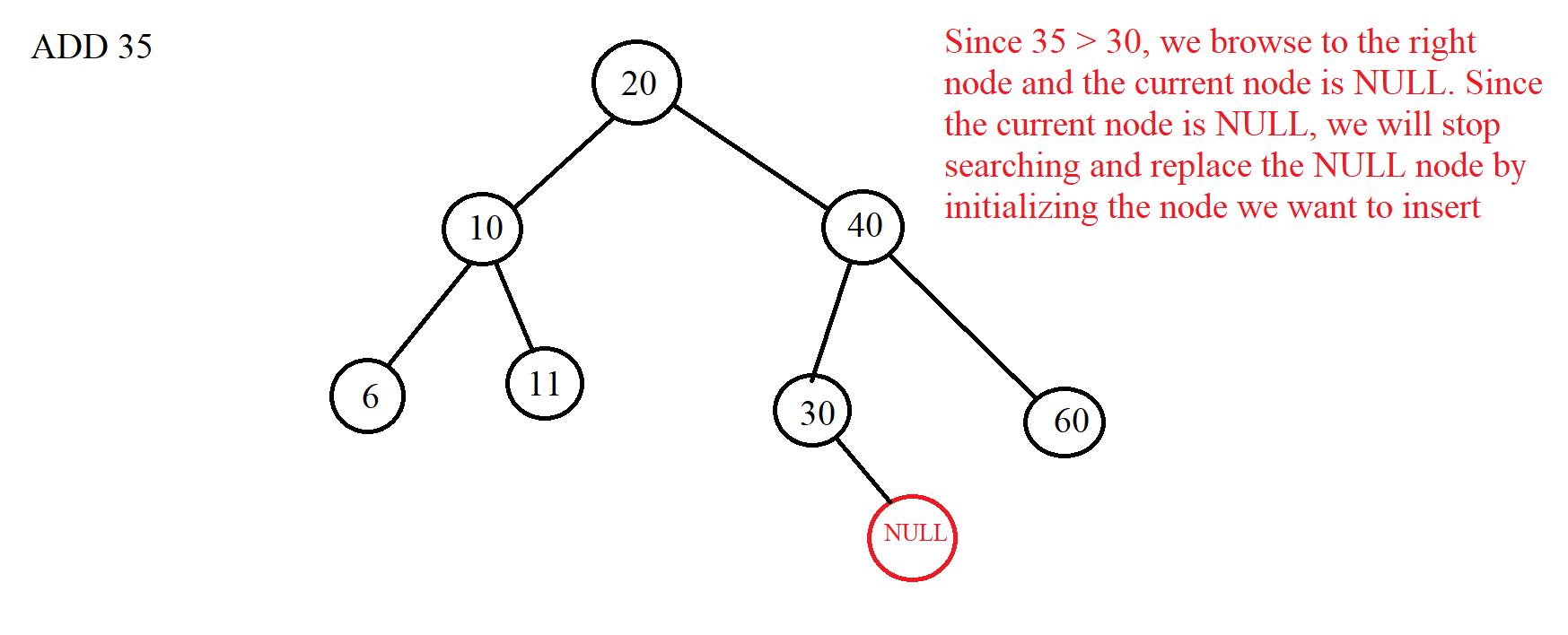
+ Add 1 element: Just like the search is based on conditions, we will start at the root node and will traverse left if the inserted element is greater than the current node and vice versa. We will do this until the current node is a NULL node and finally, we replace the NULL node with the node that we initialize with the value we want to insert.

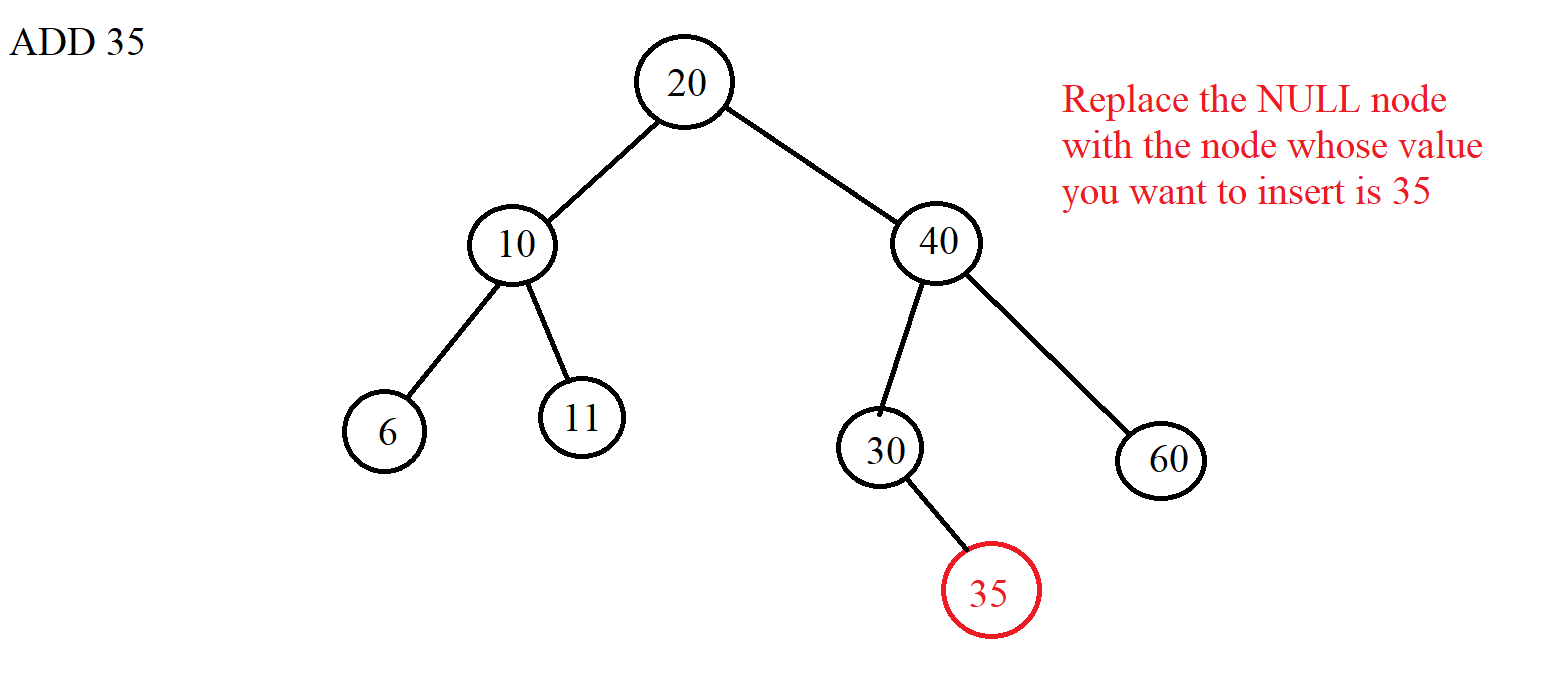
Example:







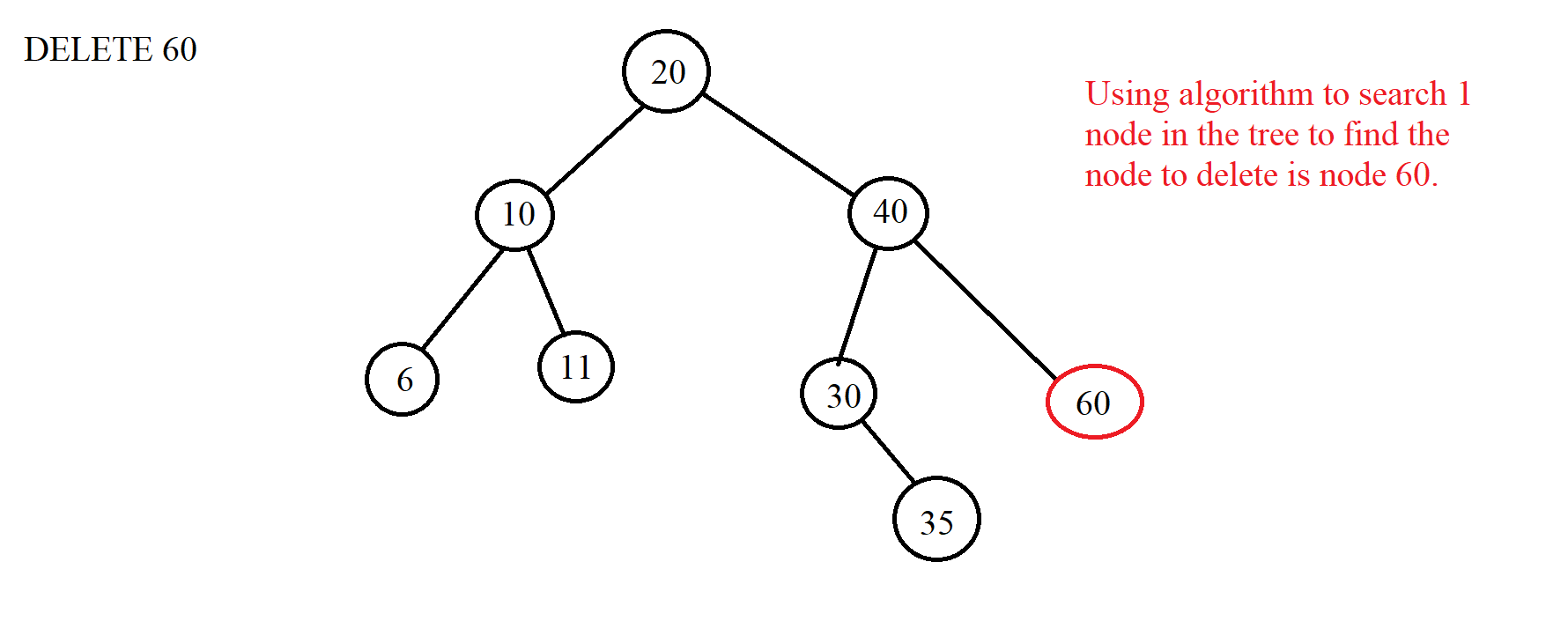


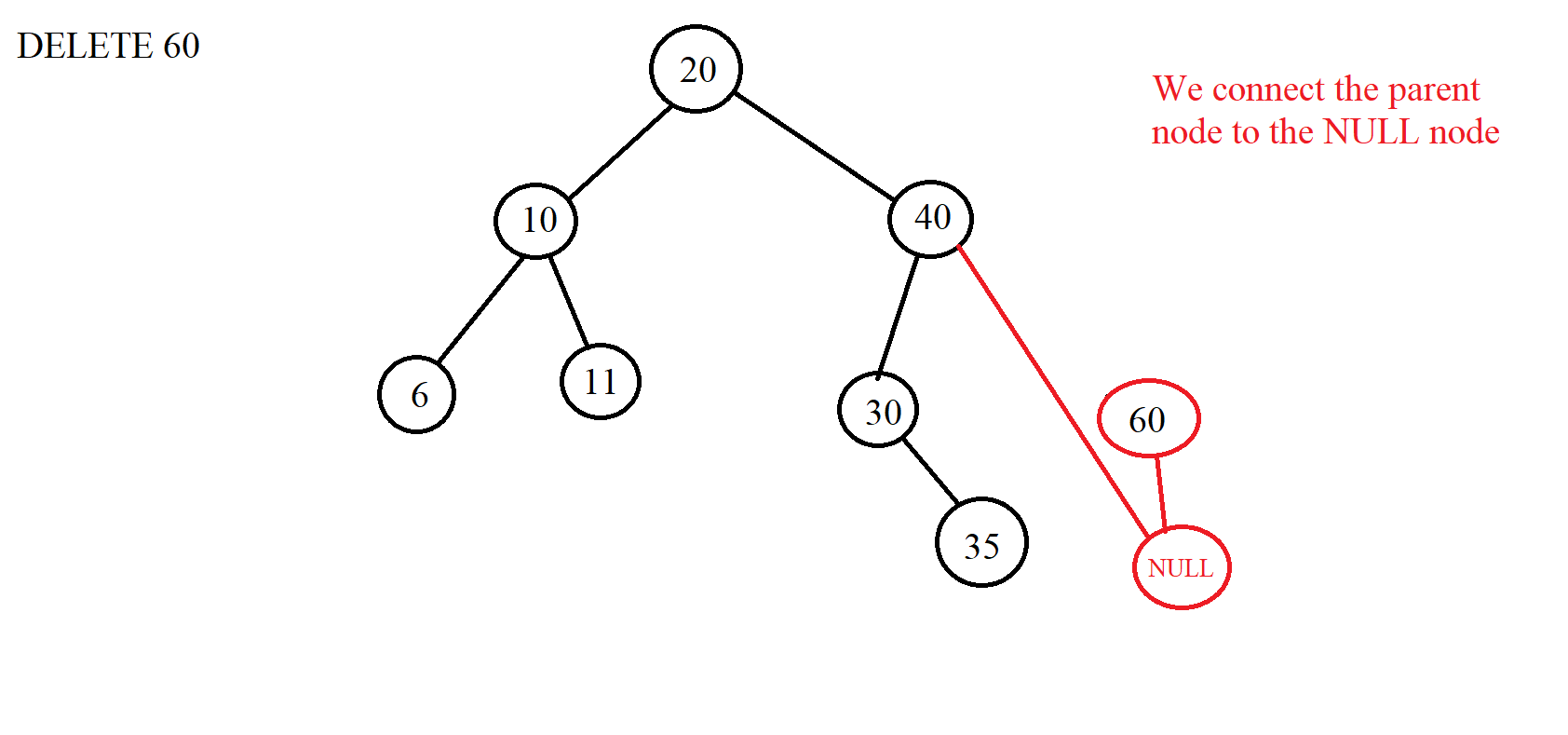


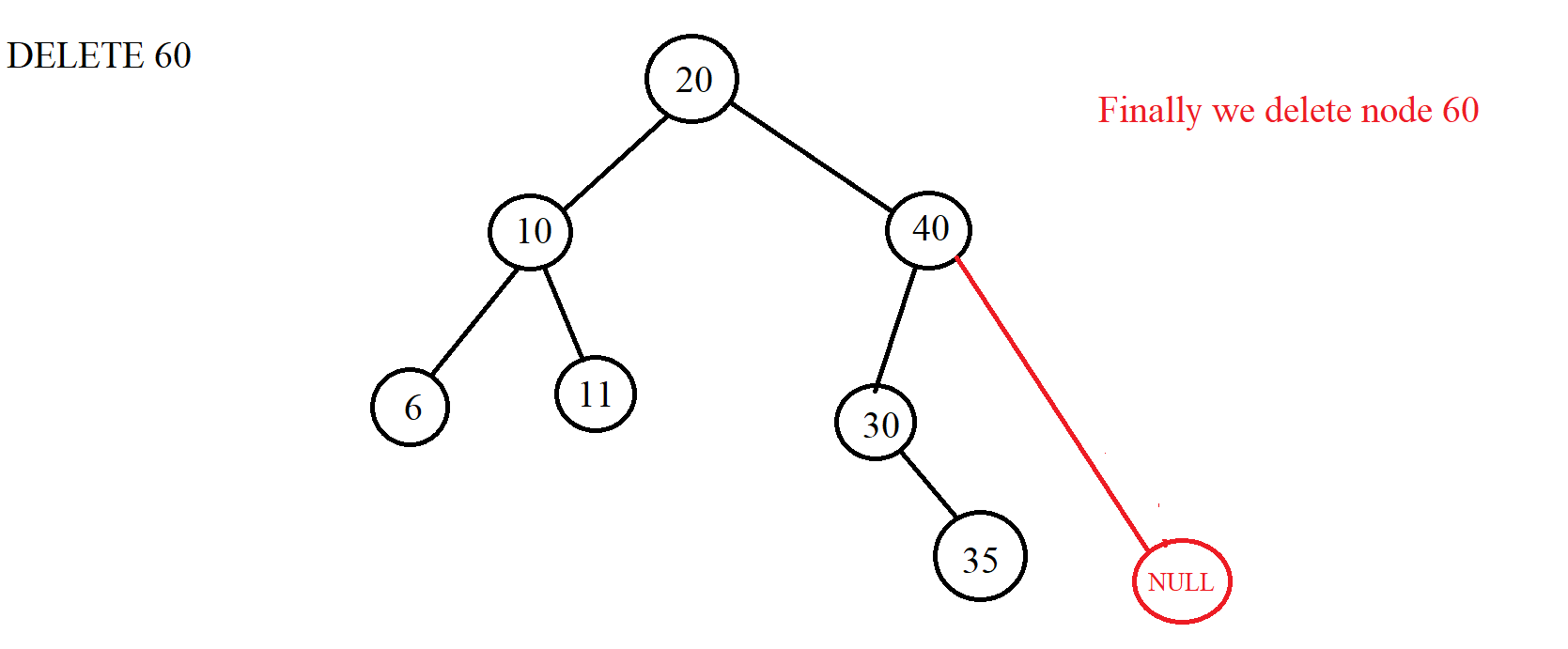


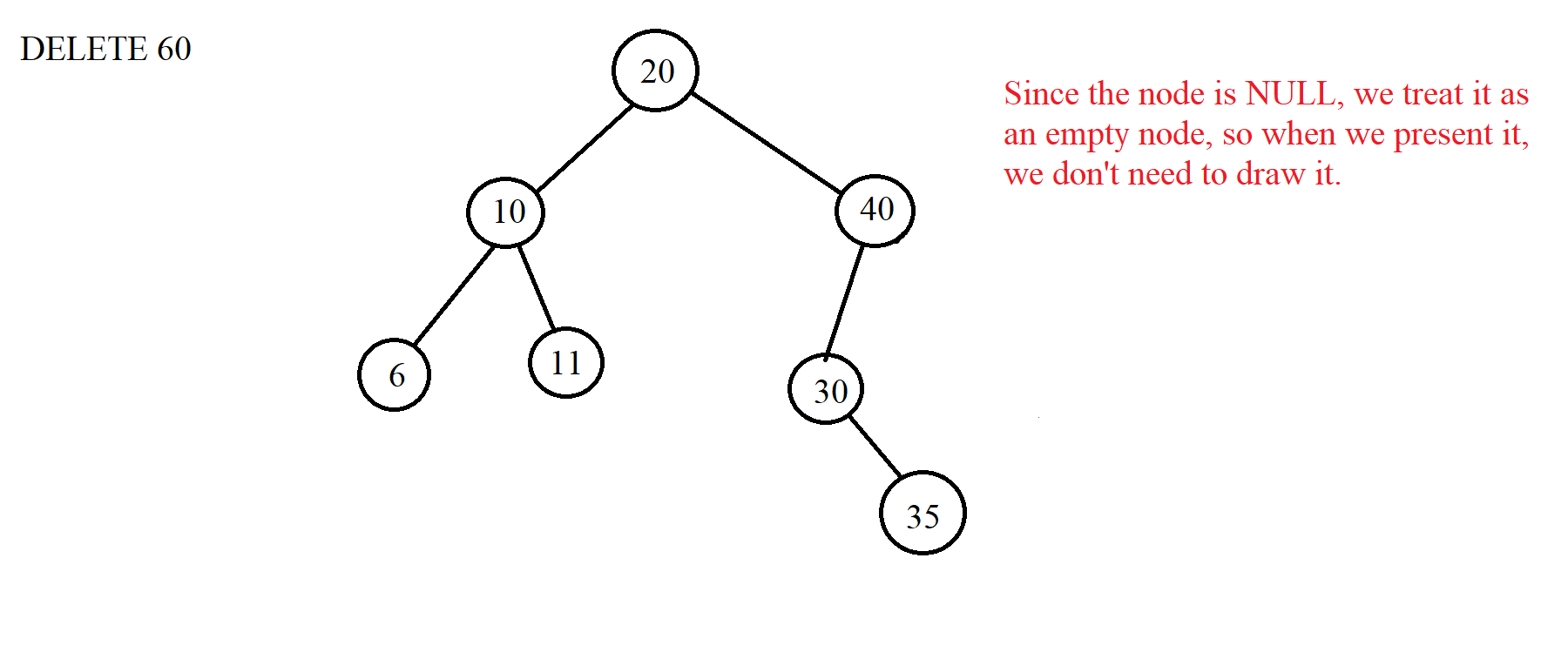
+ Delete 1 element: We use the above tree search algorithm to find the element we want to delete. If you can't find it, it means there is no element in the tree. And if there is, then we divide into 3 cases to delete.

***Case 1*** : Delete node without any child node => We just need to delete that node and let the parent node of the node to be deleted connect to the NULL node. Example:

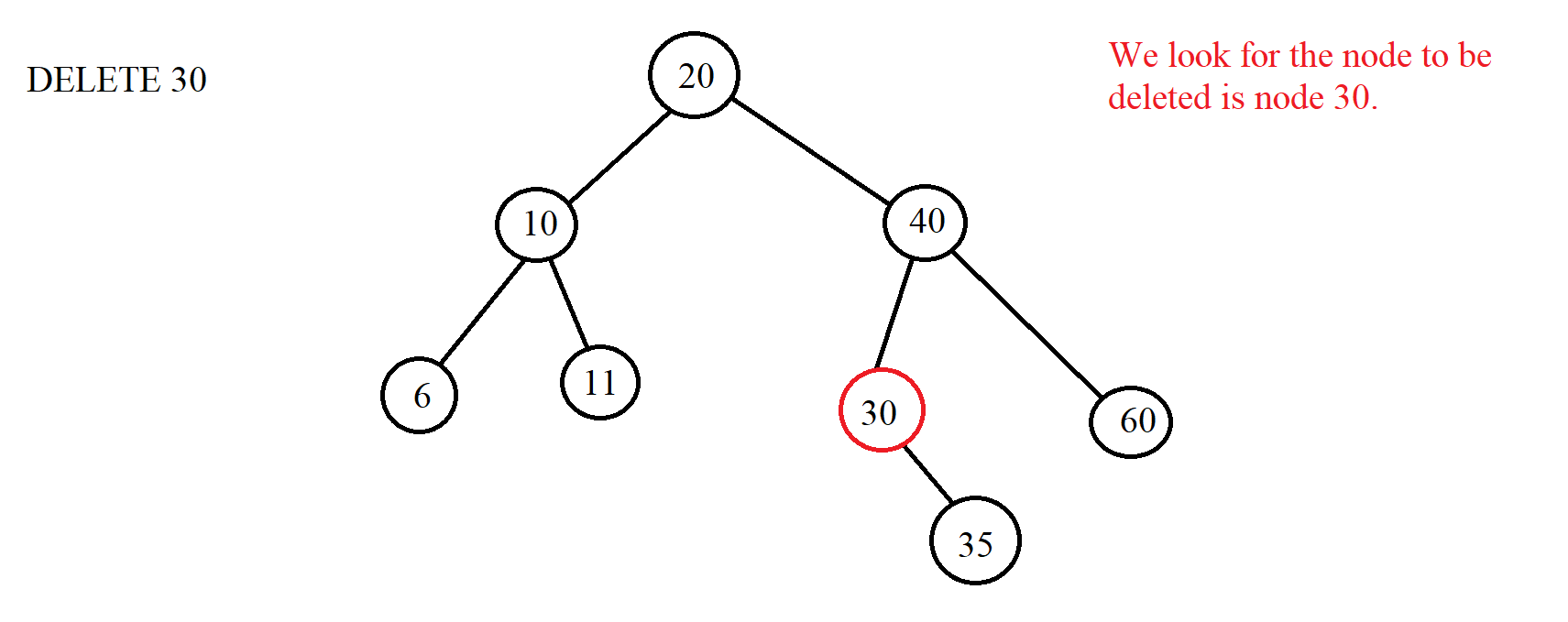


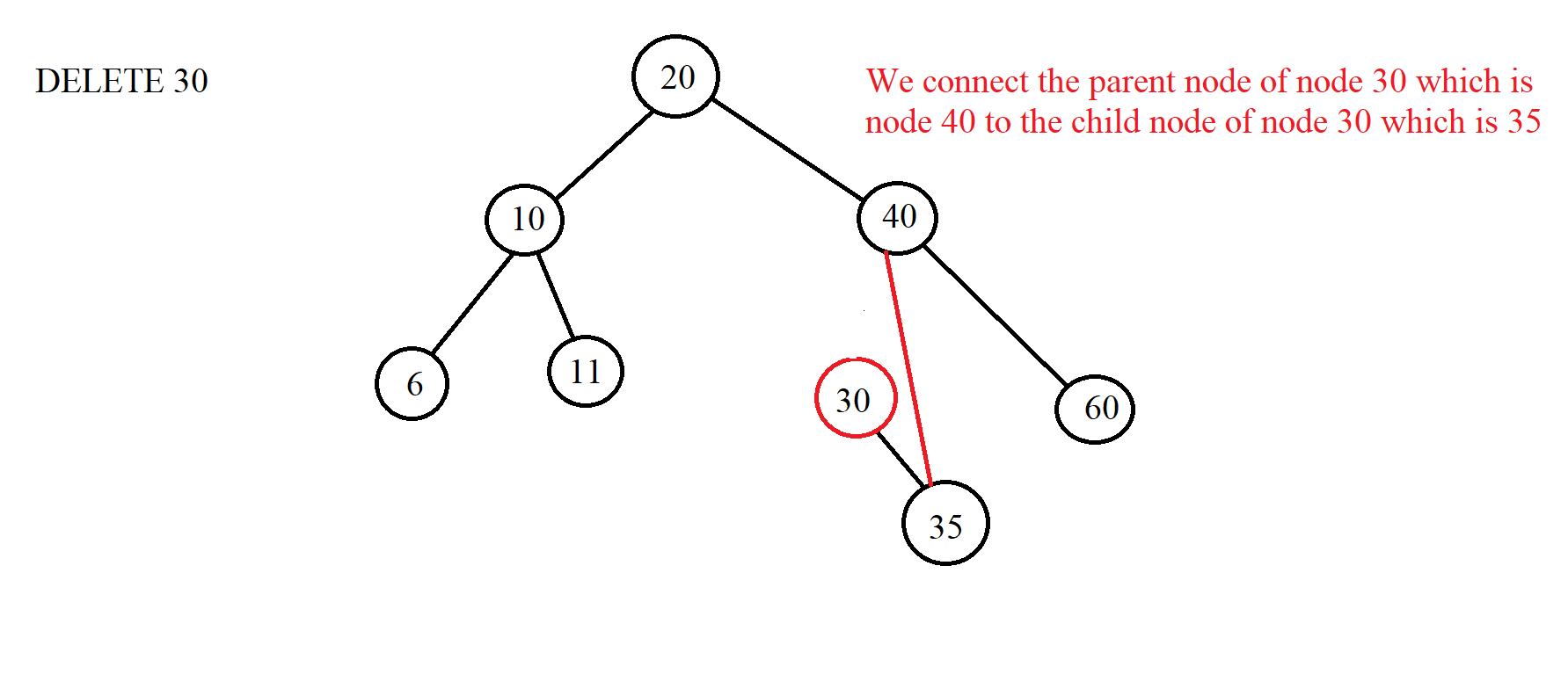


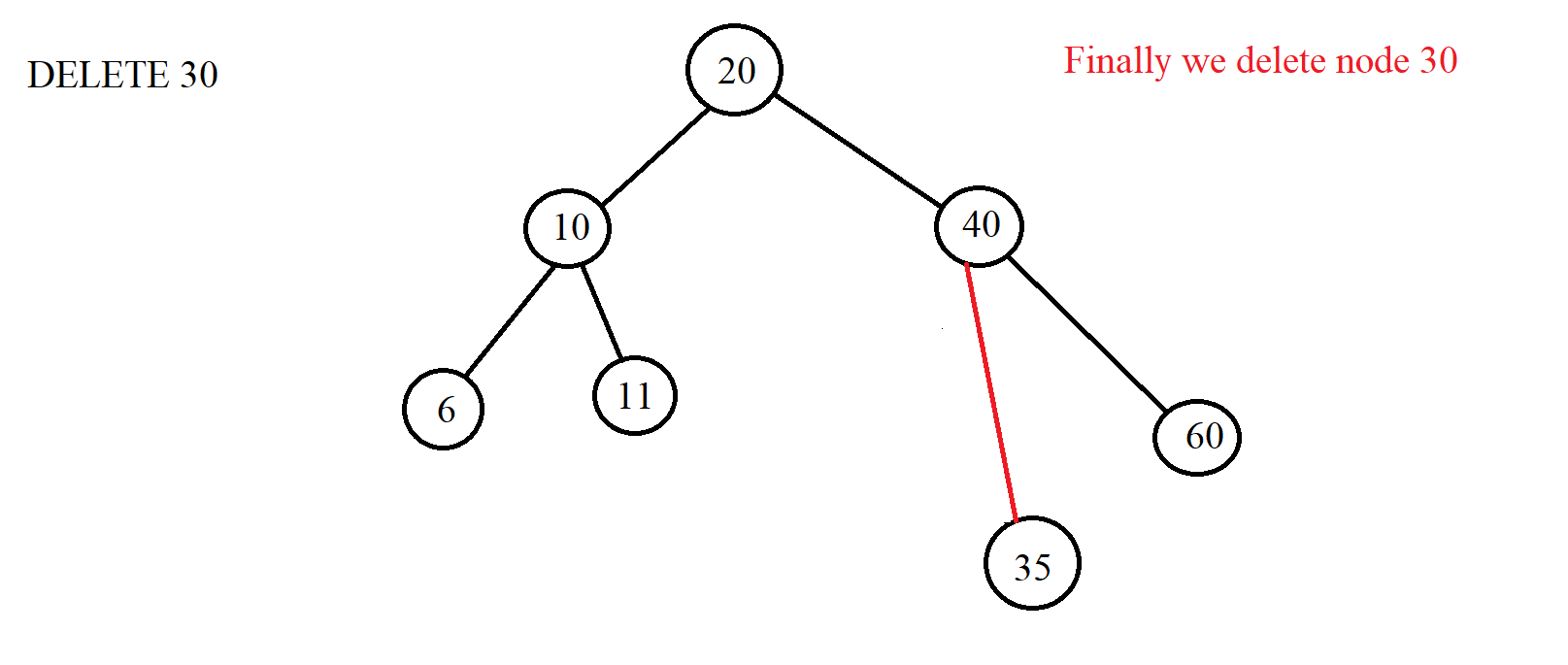




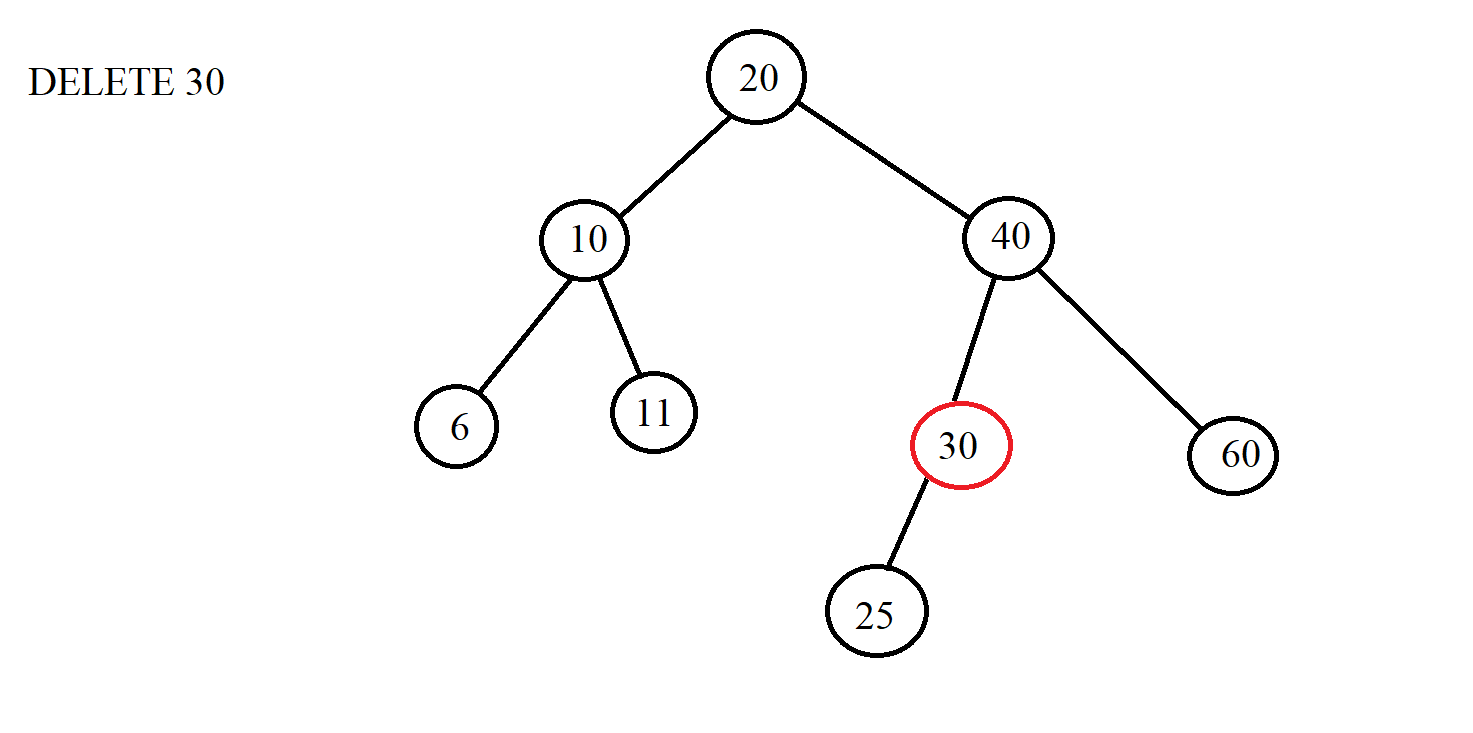
***Case 2***: Delete node with 1 child node => When we want to delete the element to be deleted in this case, we must connect the parent node of the node to be deleted with the child node of that node because that way the new tree will not be broken. When the connection is complete, we just need to delete it. When connecting the parent node of the node to be deleted with its child, we need to pay attention that if the left child node of that node is a NULL node, we will connect to the right child node and vice versa. In fact, this case is similar to case 1, but only the difference is that a child node is NULL and here is a valid node. Example we delete node 30 with 35 as the right child:

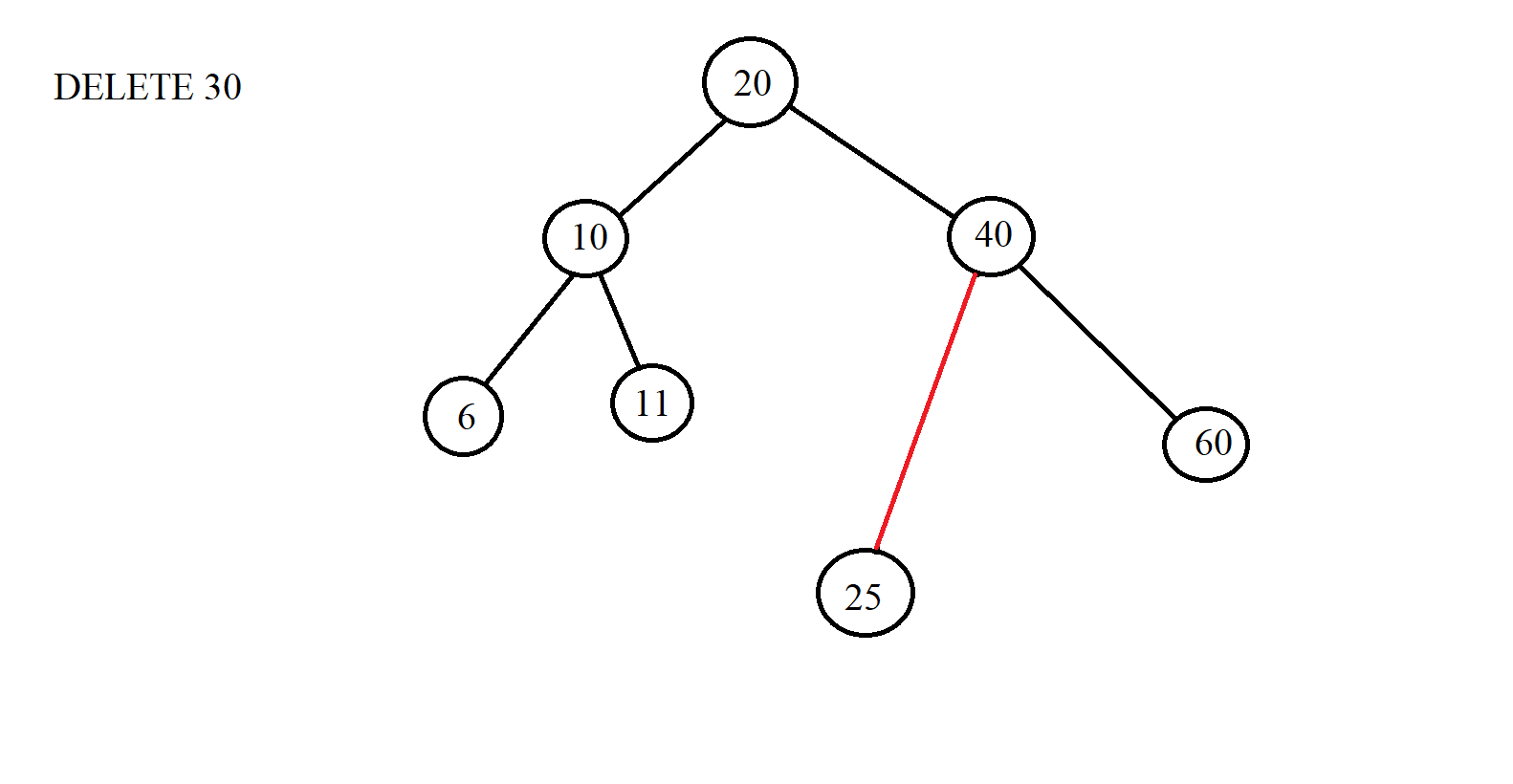






Similarly when we delete 30 and 25 are left child nodes:

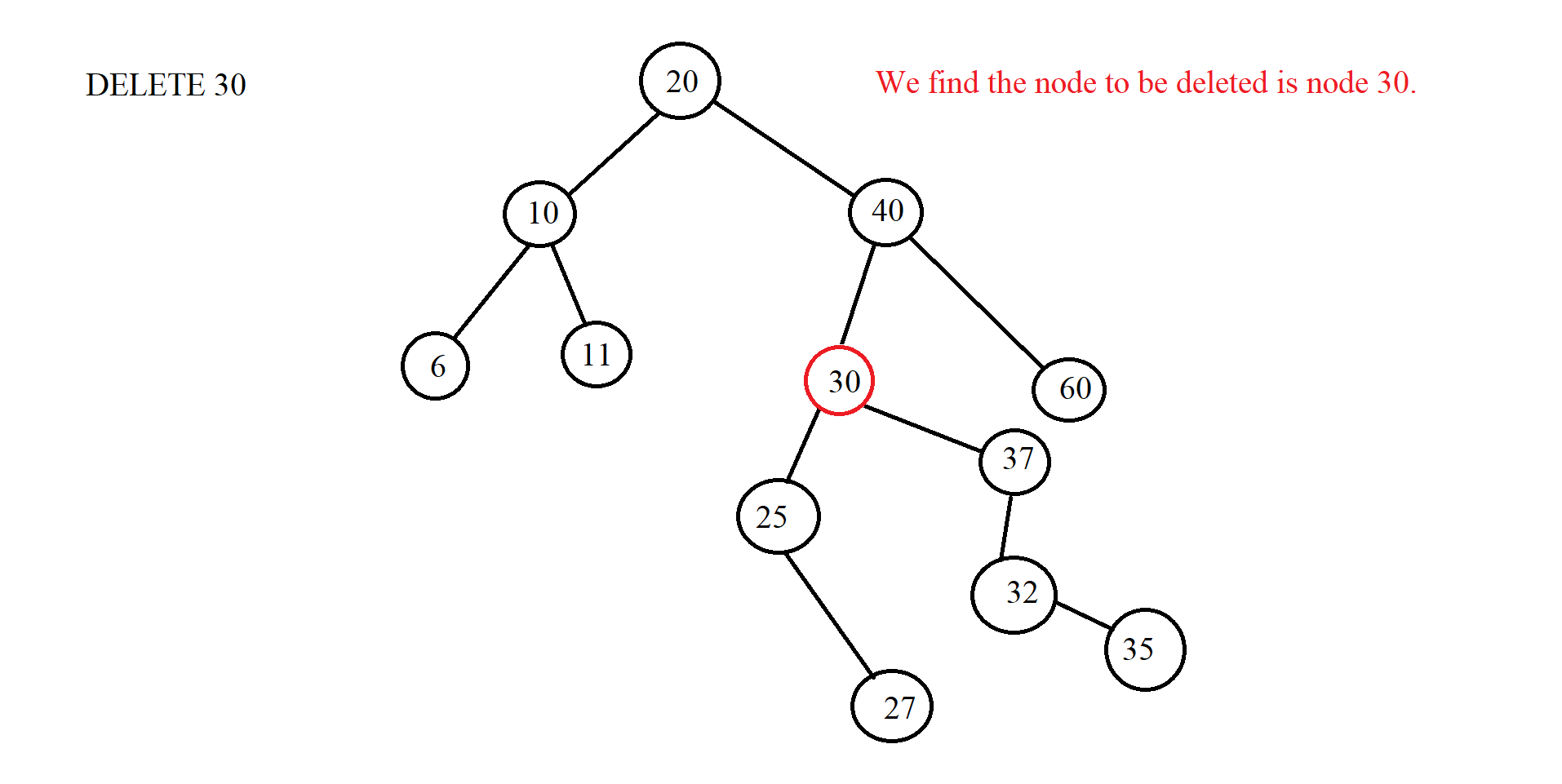


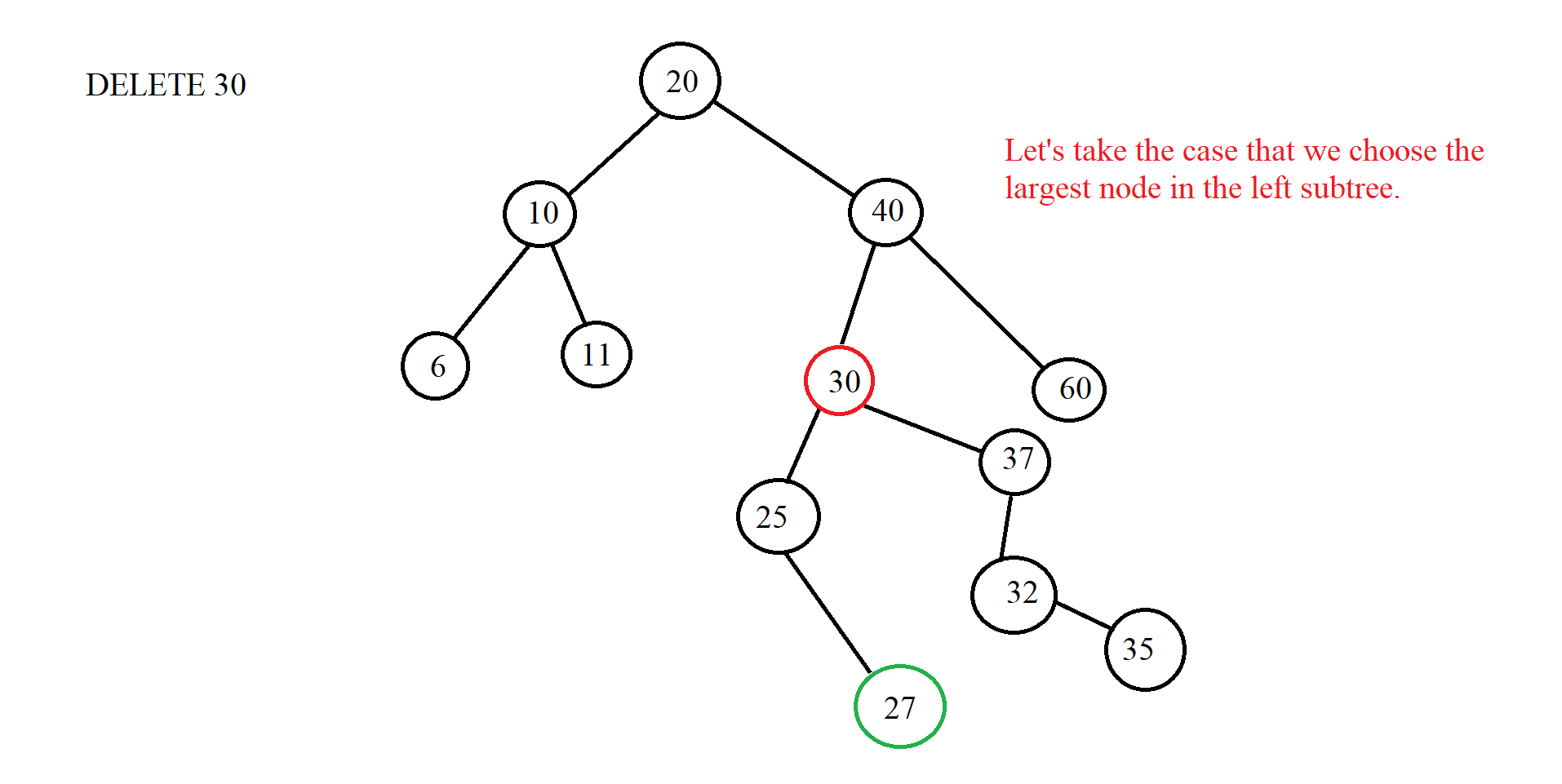
***Case 3***: Delete node with 2 children => Instead of deleting that node, we delete the node for which we get the value to replace the node that needs to be deleted. Then the node that we want to delete will be replaced with another value. Therefore, the node so that we can substitute the value for the node we want to delete is the node instead of the node we want to delete must satisfy the condition of the BST tree. From there, we have 2 ways to do it:

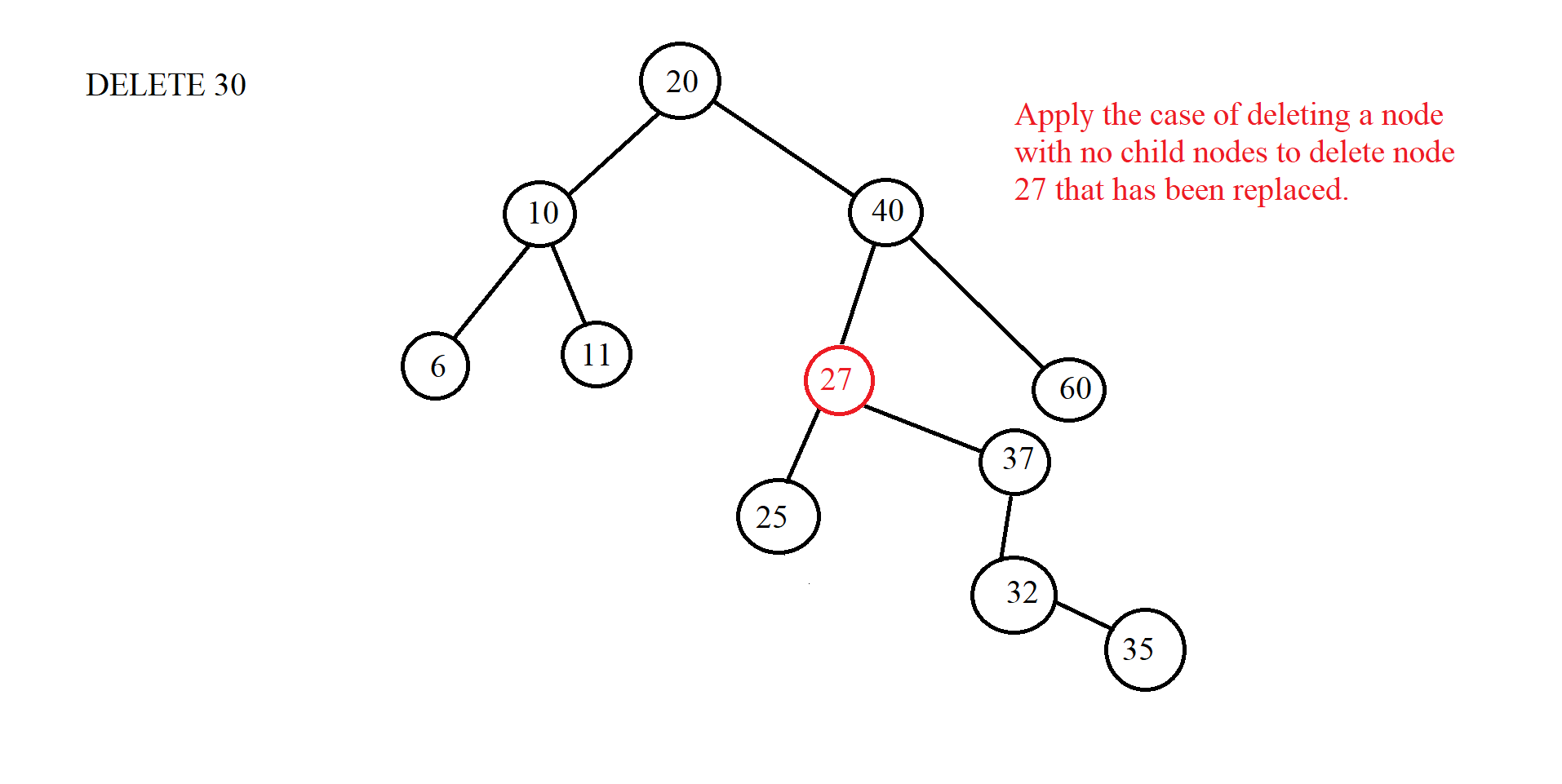
\*Node 1: We will choose the node with the largest value of the left subtree of the replaced node.

\*Node 2: We choose the smallest node of the right subtree.

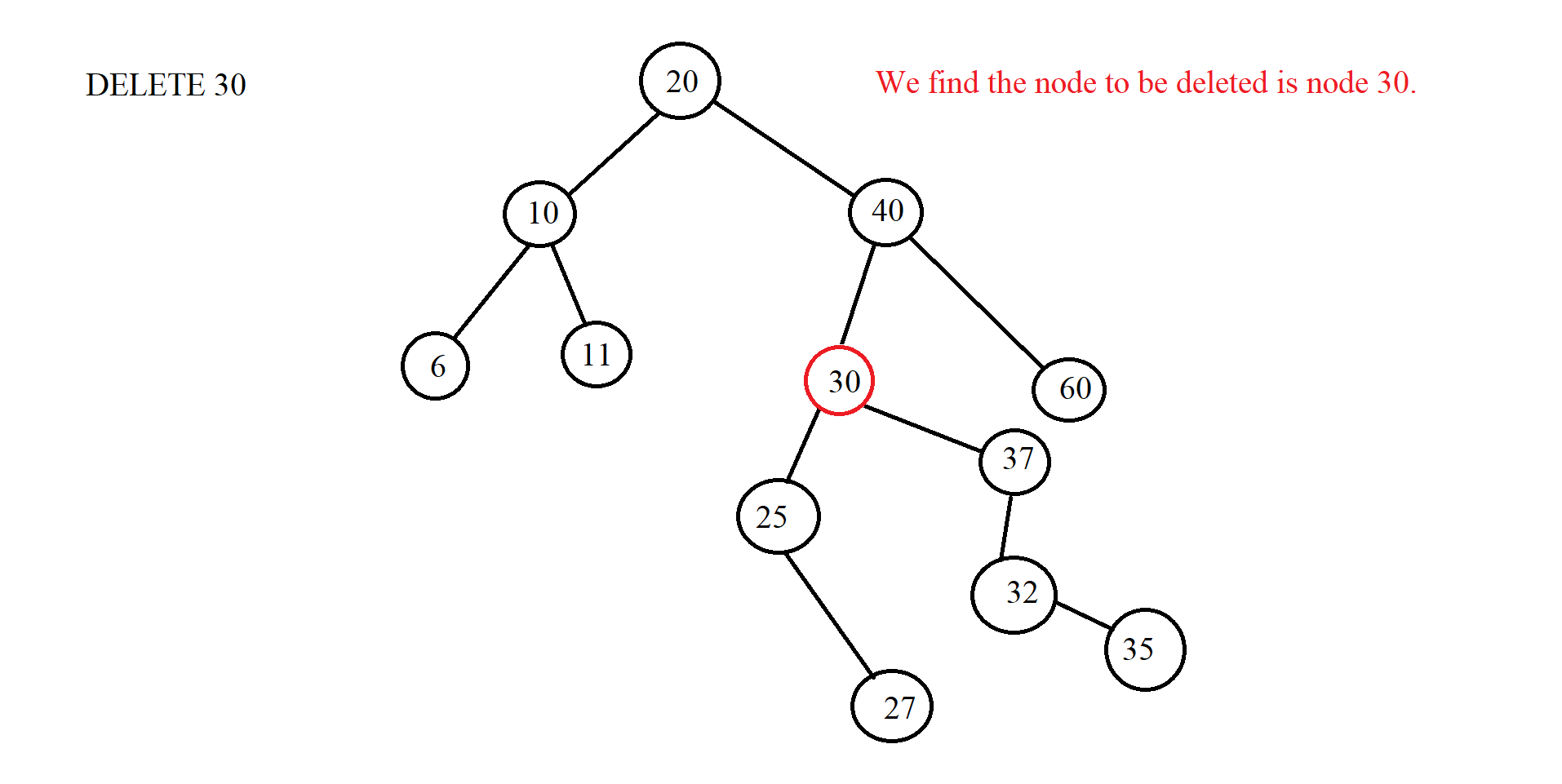
When the replacement is complete, we delete the node that has been taken to replace the node we want to delete. Here is the largest node of the left subtree or the smallest node of the right subtree. If the replacement node is deleted, it will fall into one of the two cases 1 or 2 above. Example:

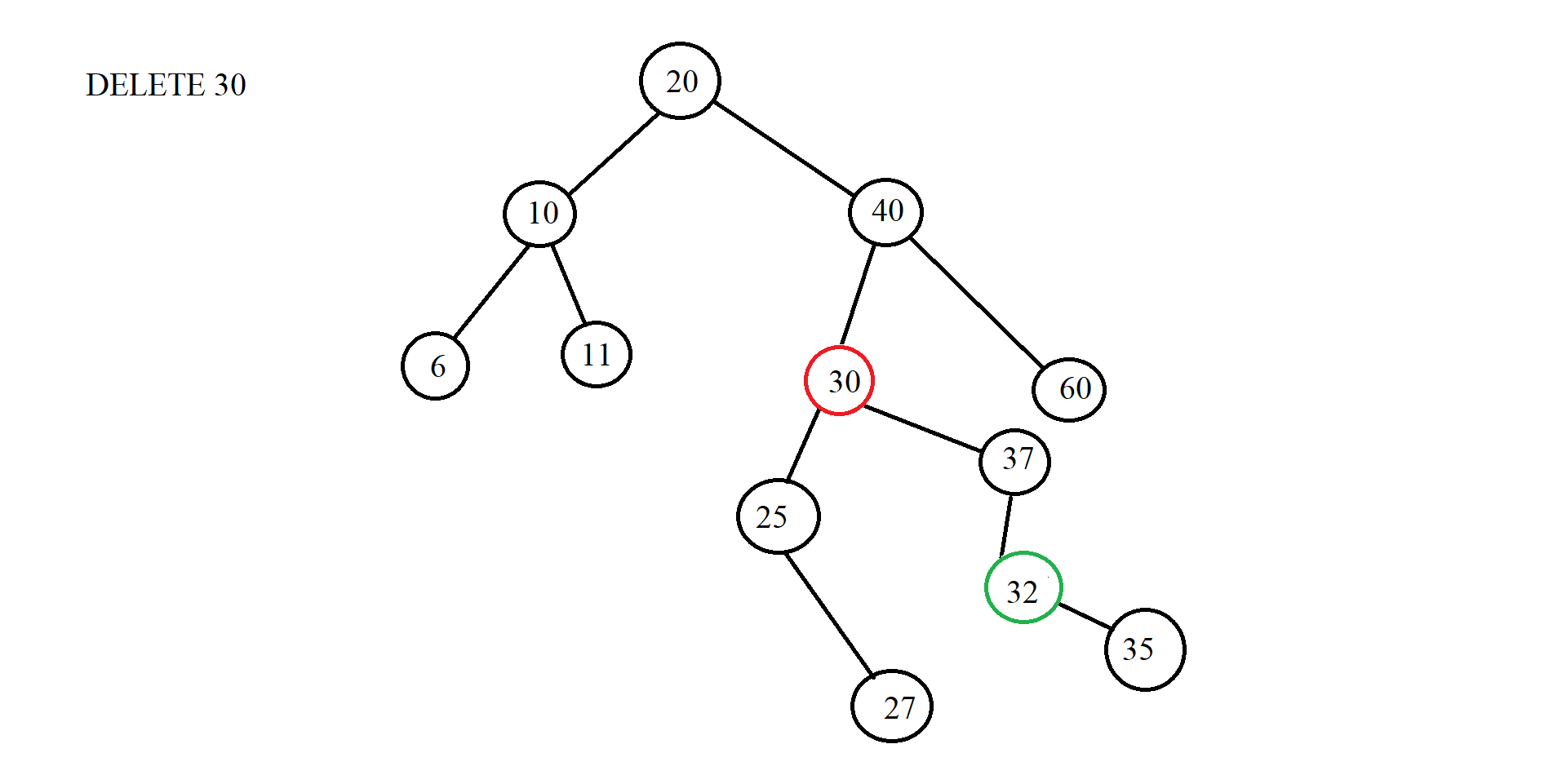


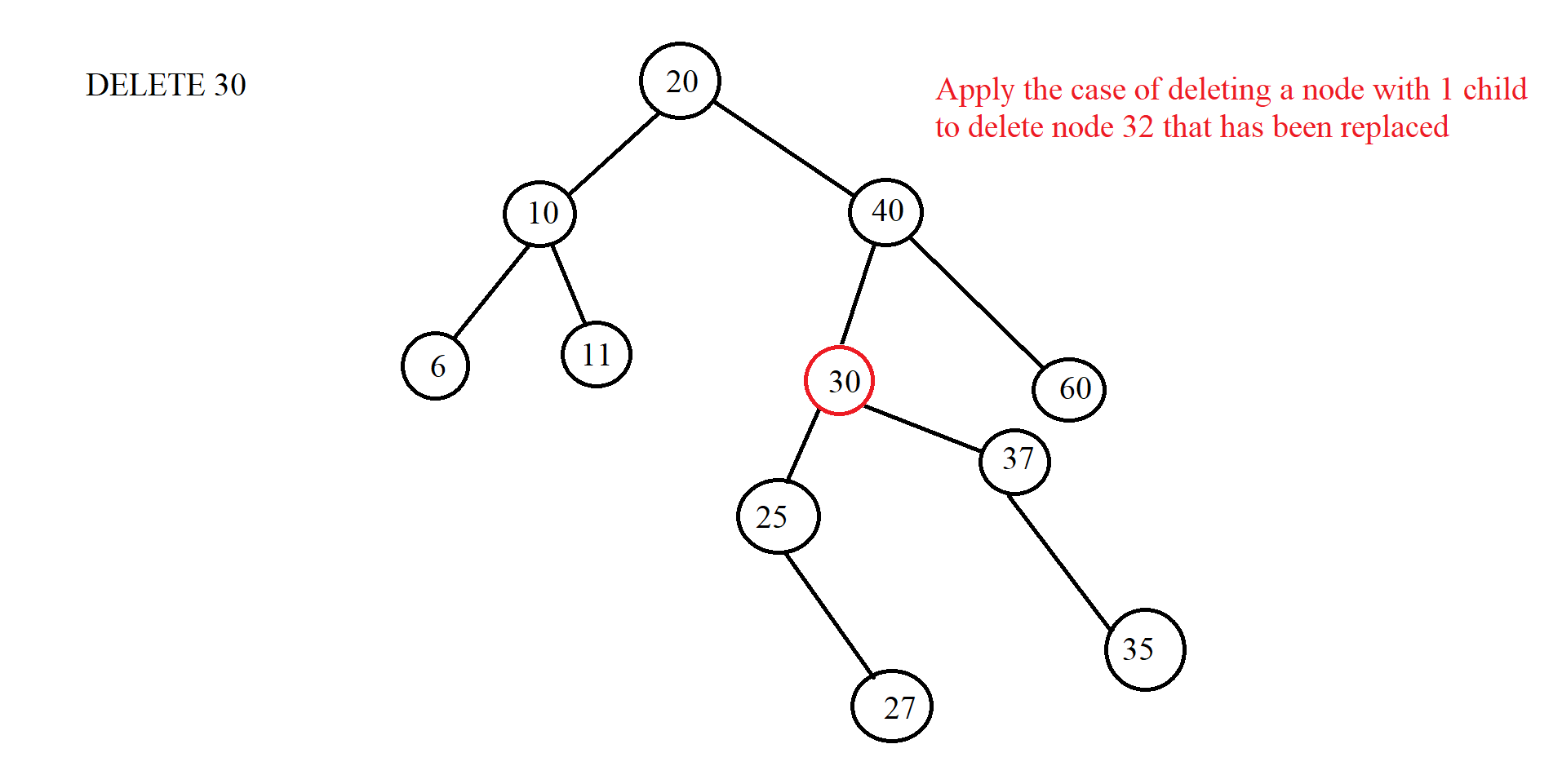




(Let's say we want to get the smallest node of the right subtree instead)









|  |
| --- |
|  |

***\*Illustration:***

*Example 1:*

* **INPUT:**

-Insert:32, 20, 10, 15, 40, 25, 27, 26, 5, 13, 14.

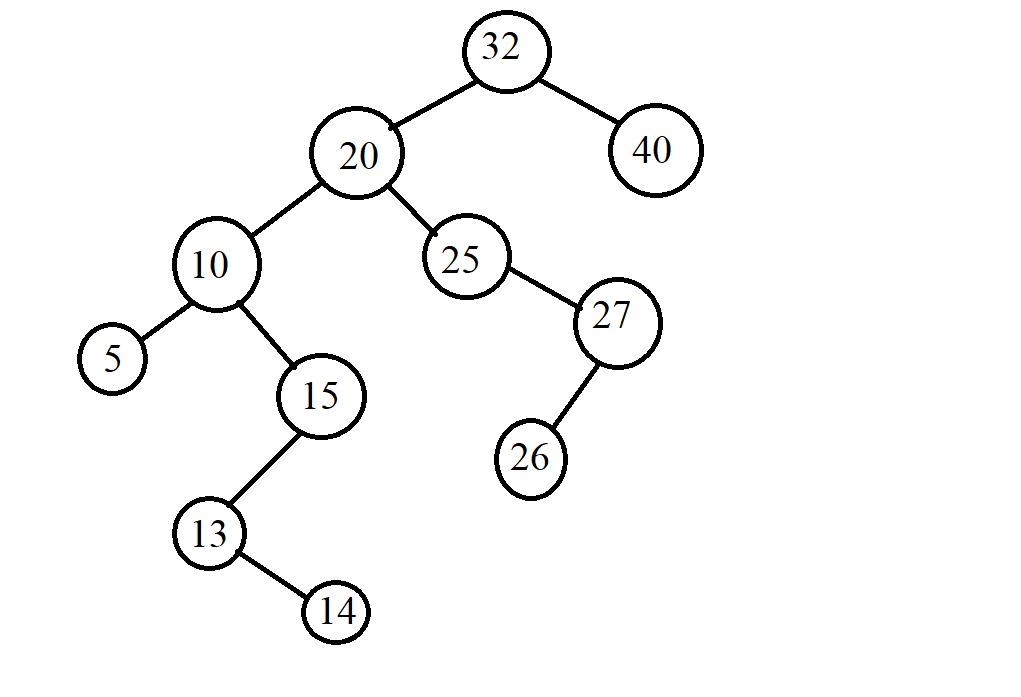
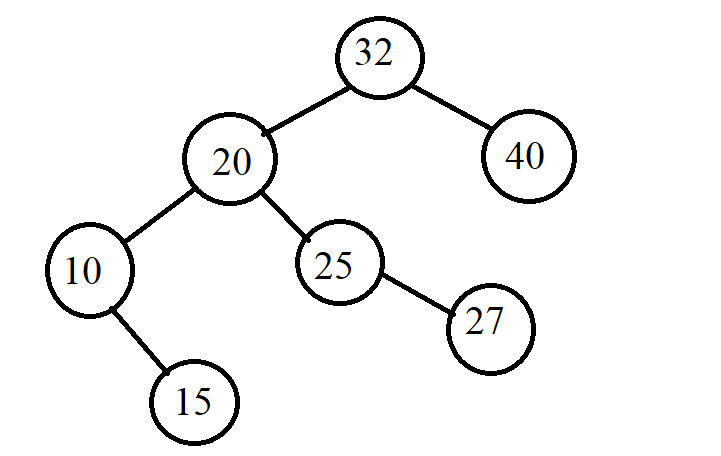
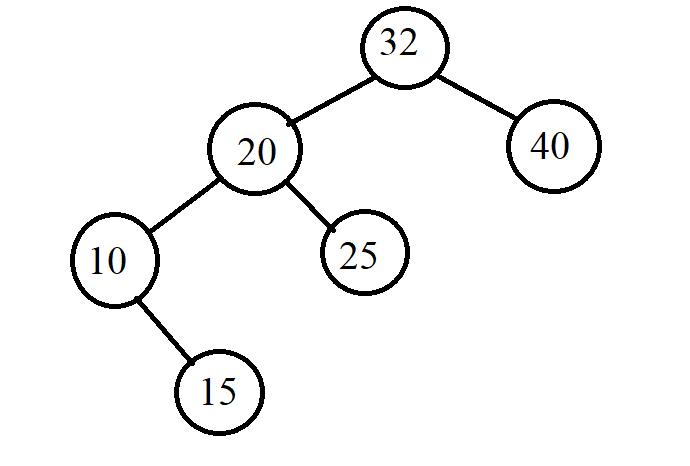
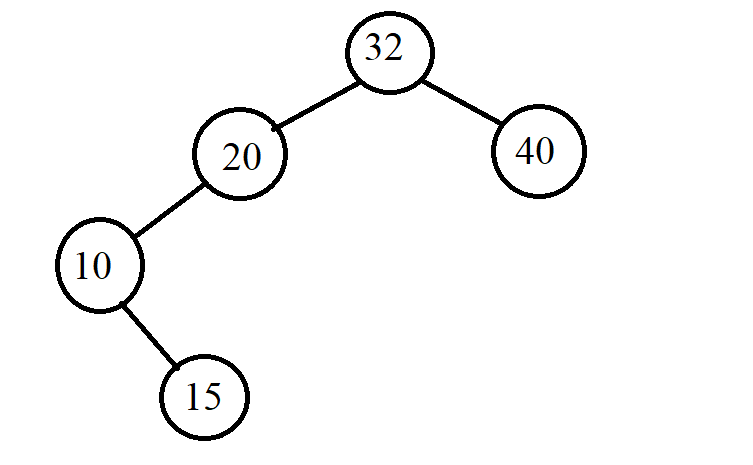
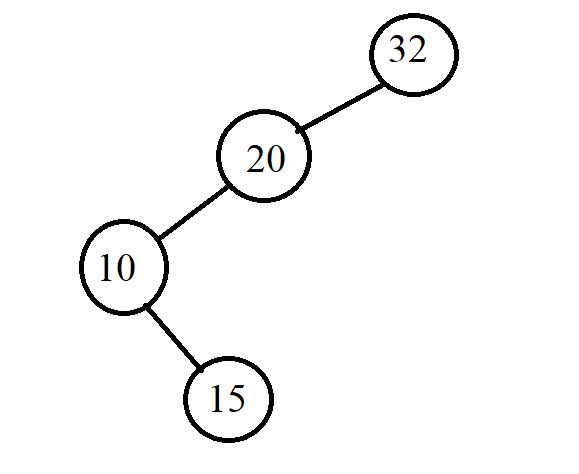
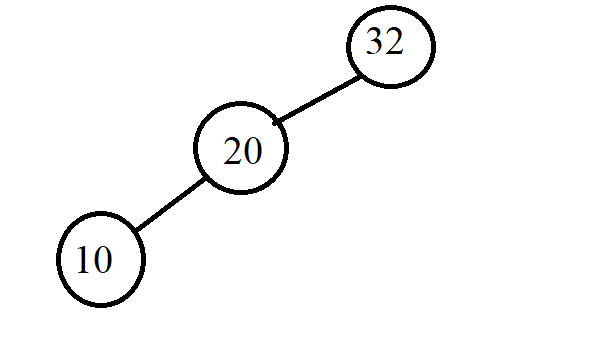
-Find: 15, 29

-Delete: 25, 40 , 20

* **OUTPUT:**

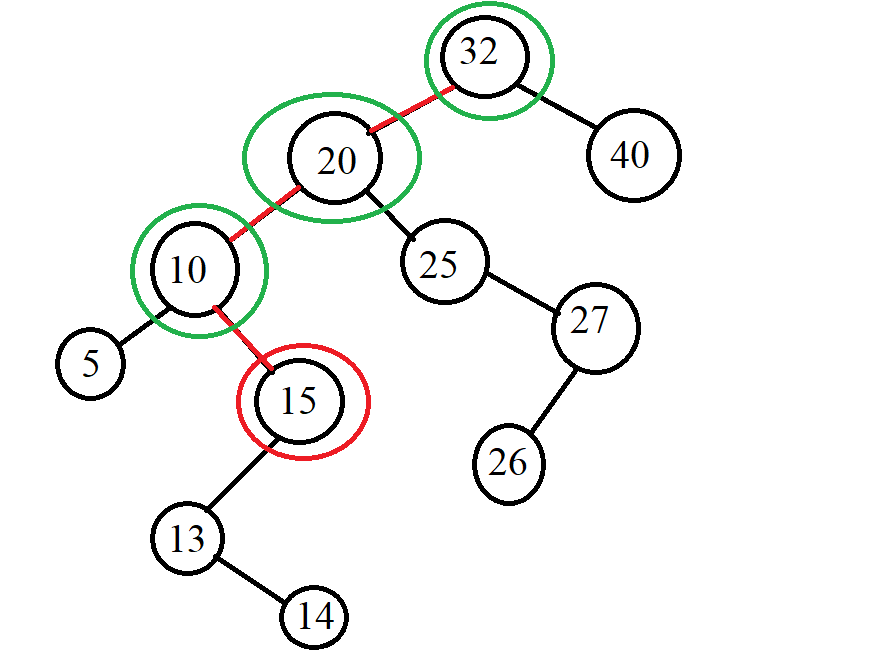
-Insert:



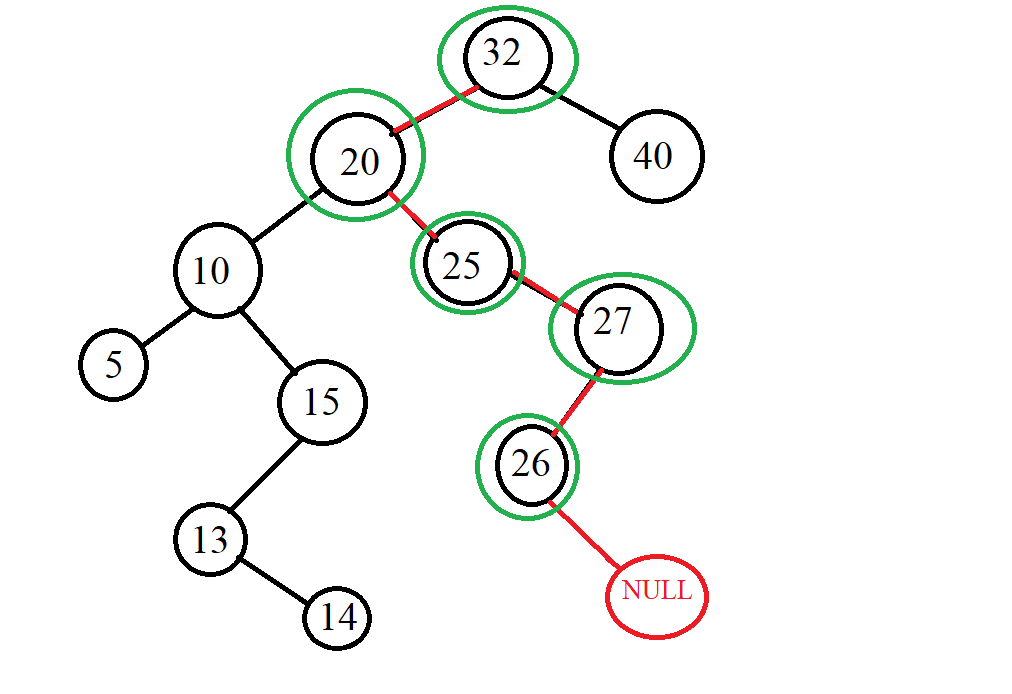


-Find:

Find 15:

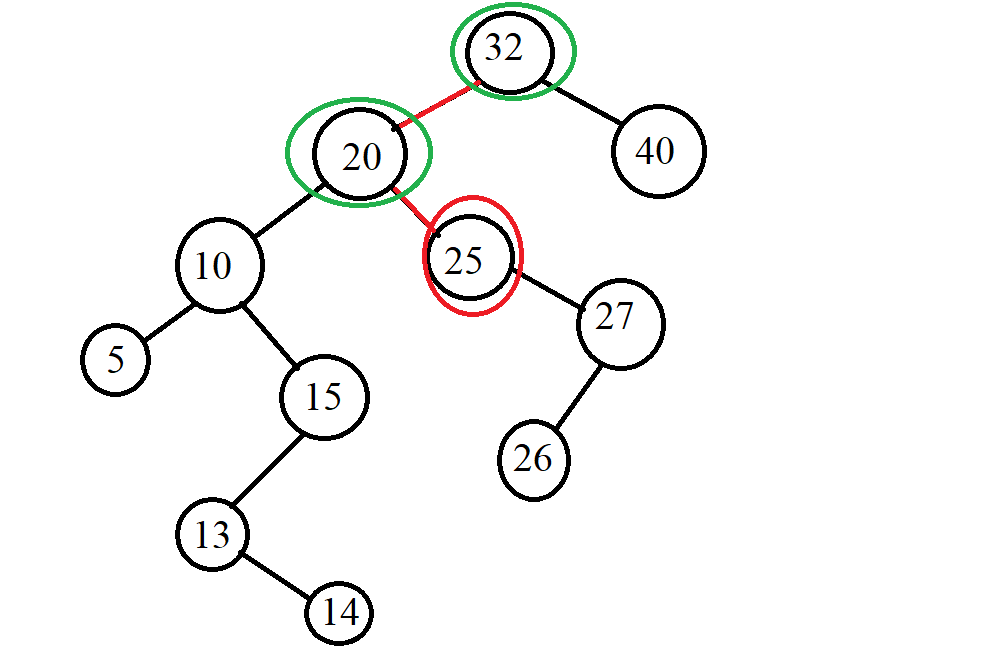
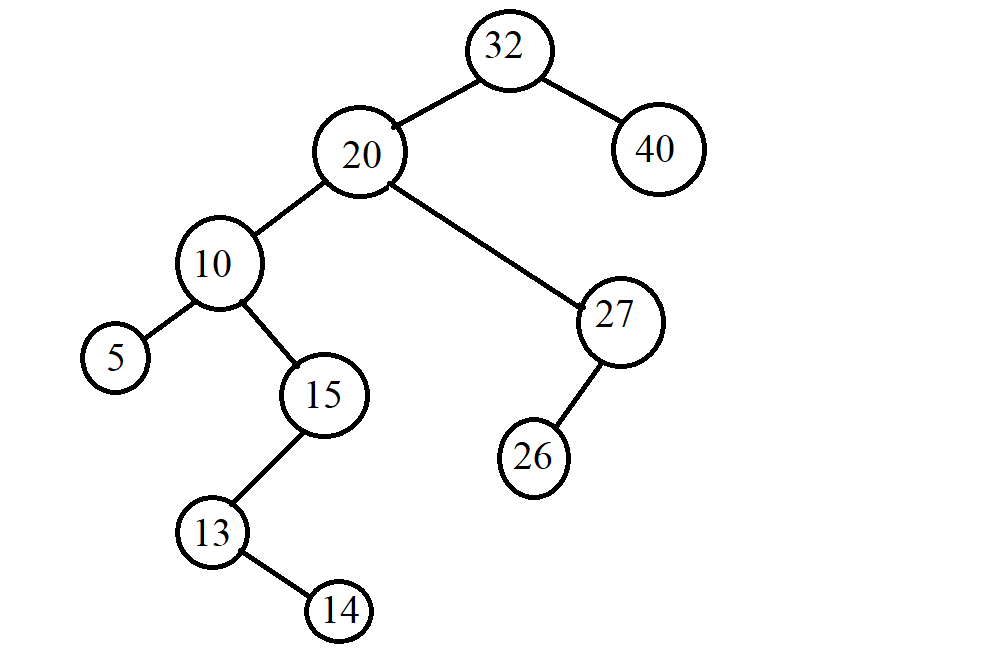


Find 29:

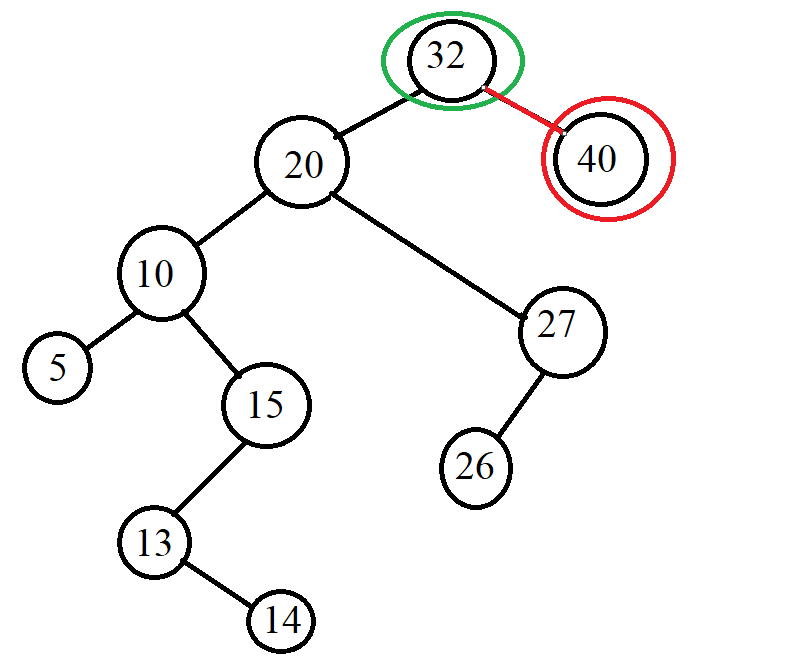


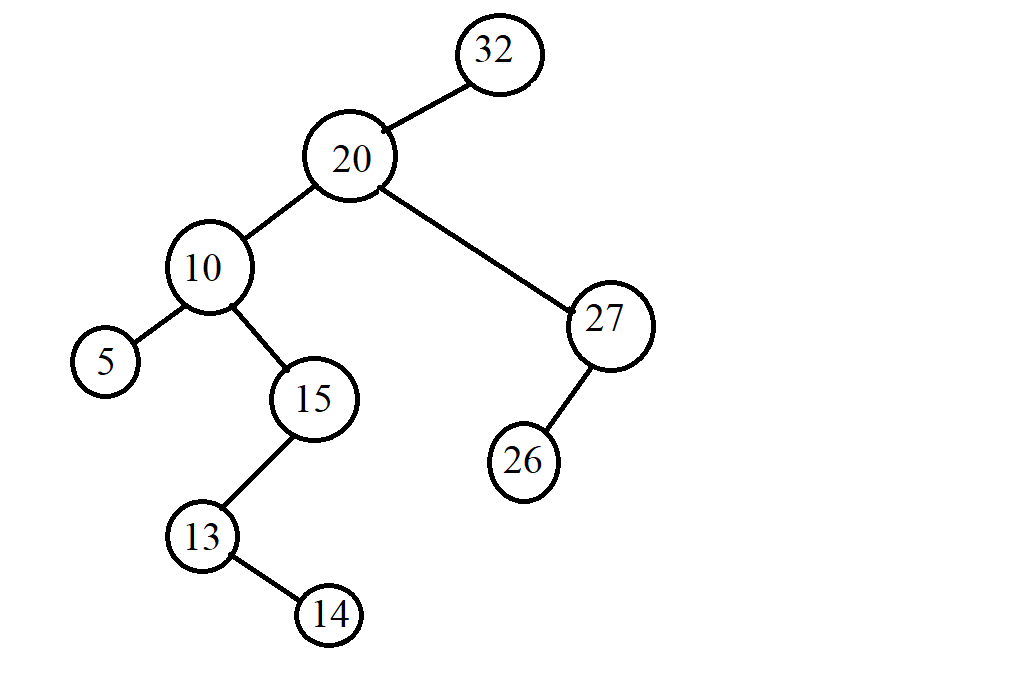
-Delete:

Delete 25

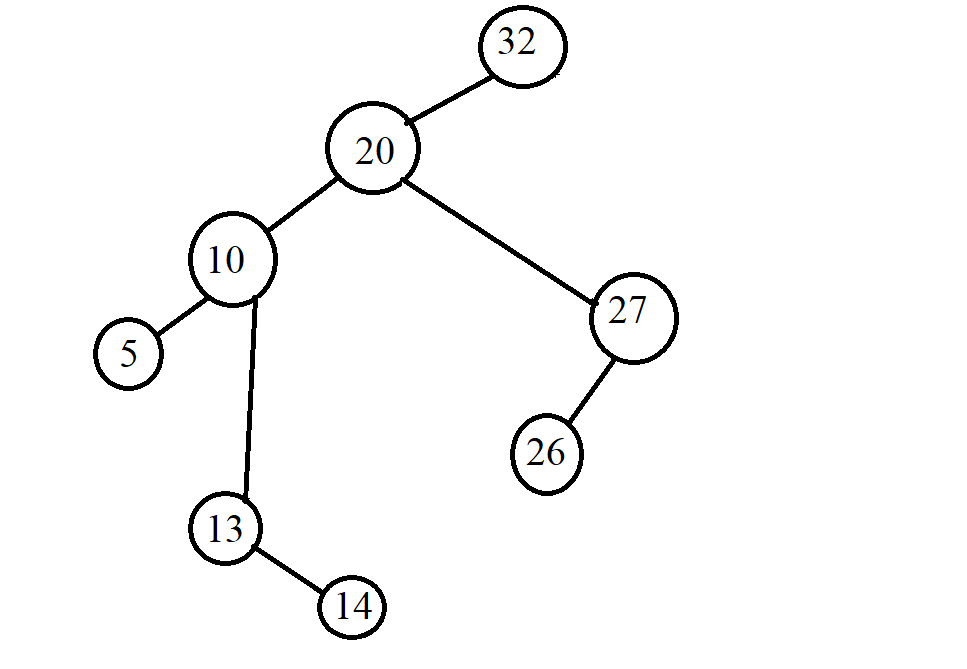
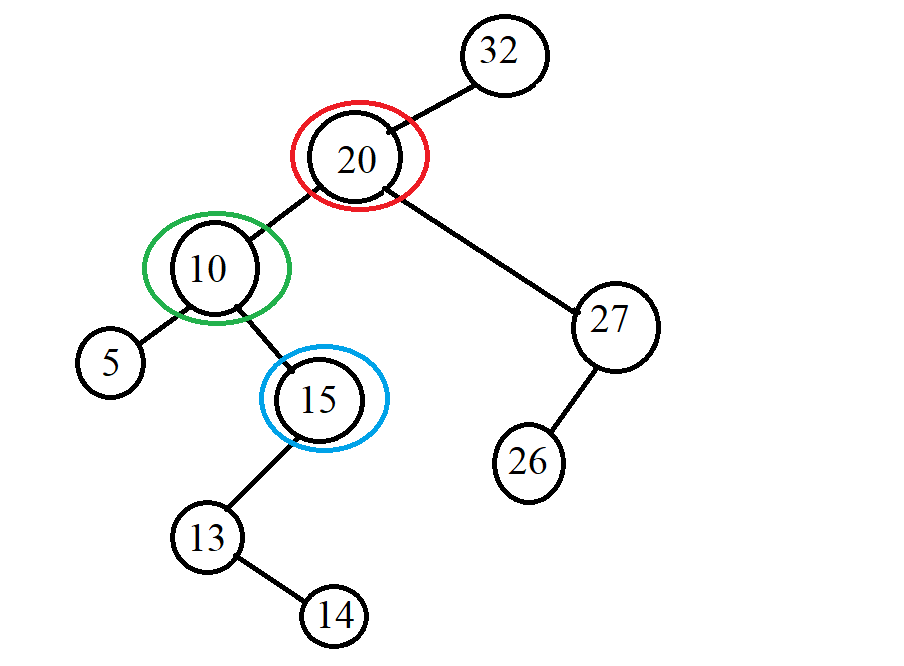
 

Delete 40:





Delete 20:



*Example 2:*

* **INPUT:**

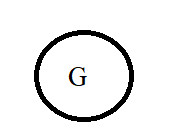
-Insert: G, J, A, T, B, W, C, K.

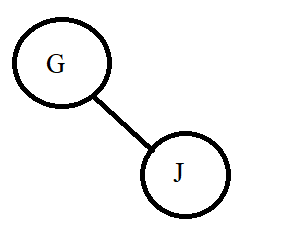
-Find: A, Z

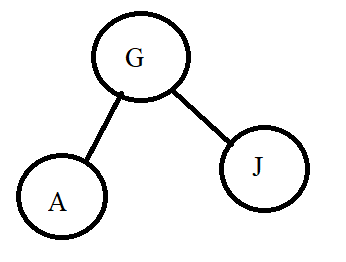
-Delete: J, A

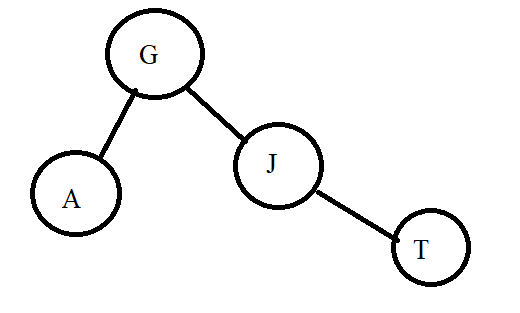
* **OUTPUT:**

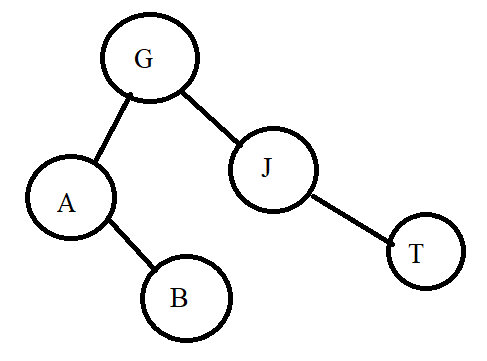
-Insert:

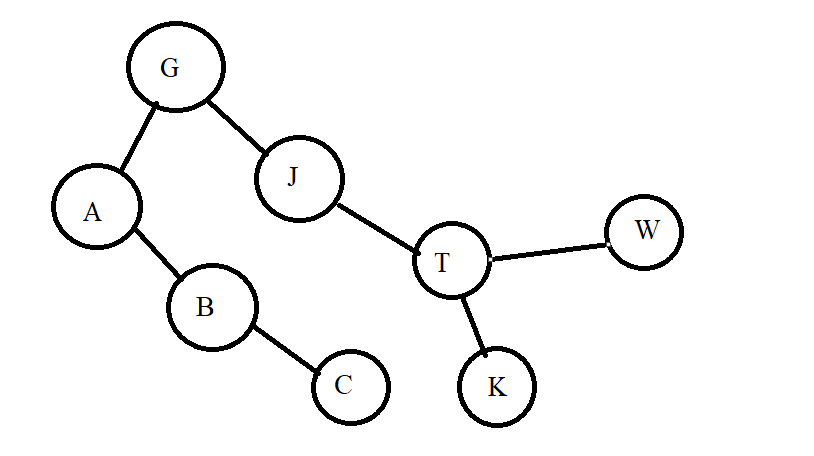






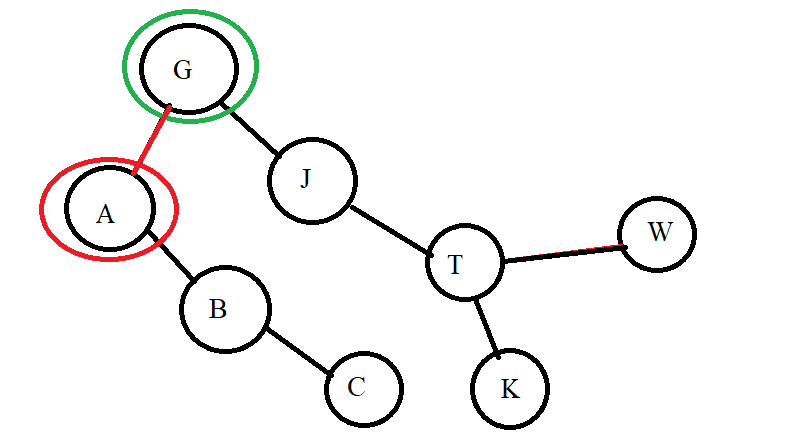




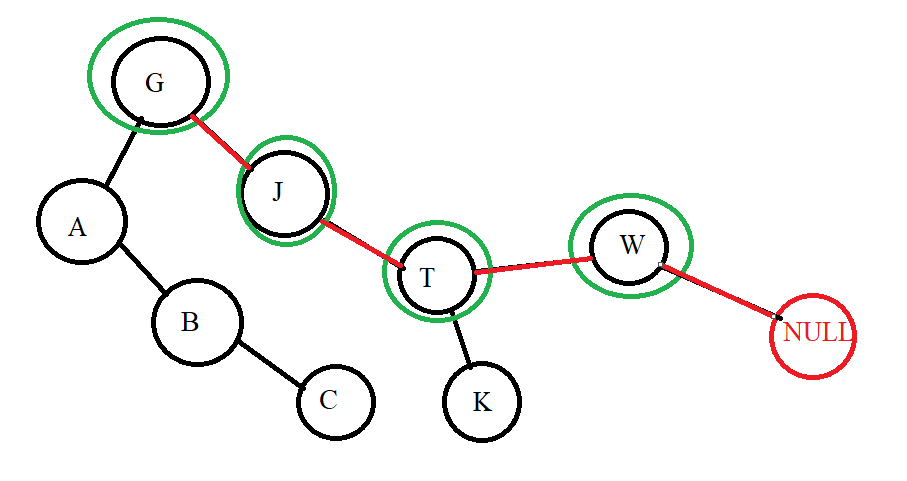


-Find:

Find A

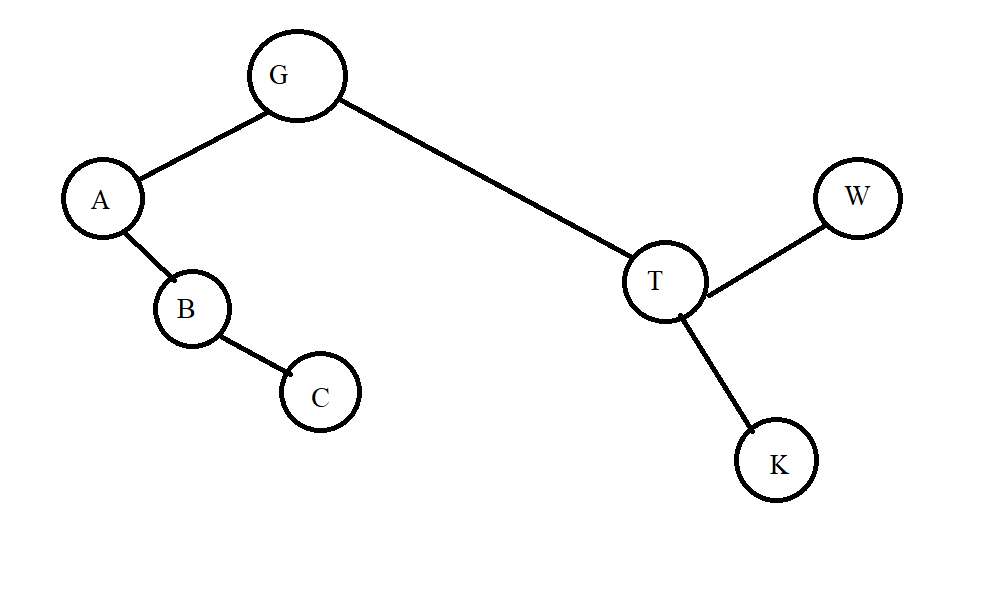
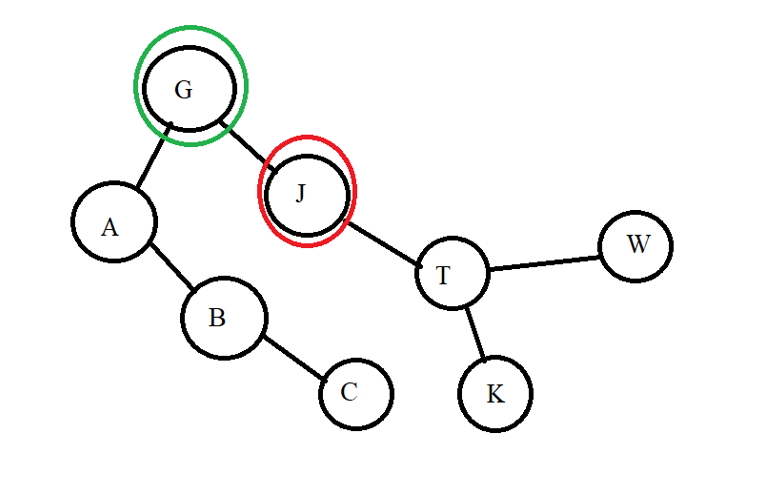


.Find Z

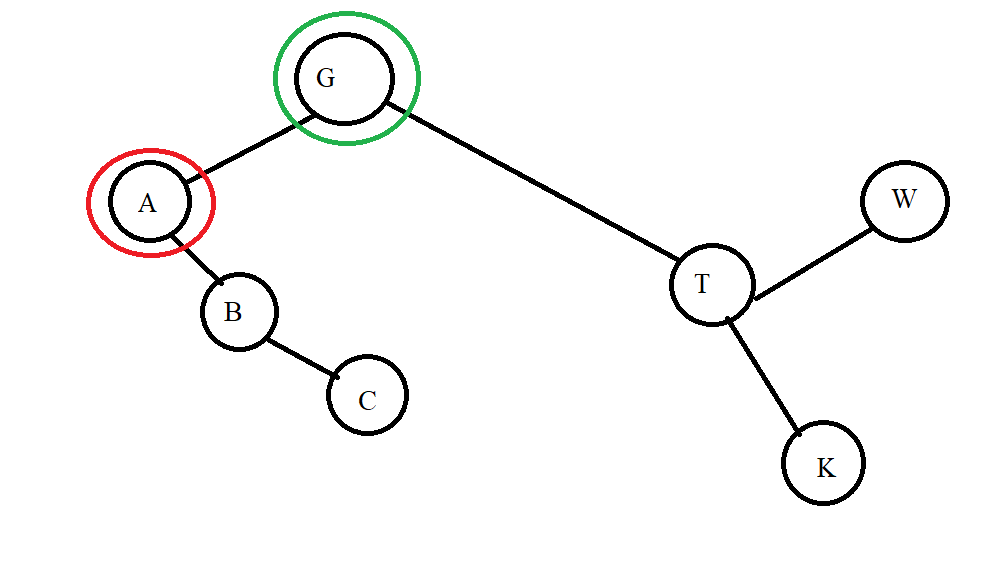


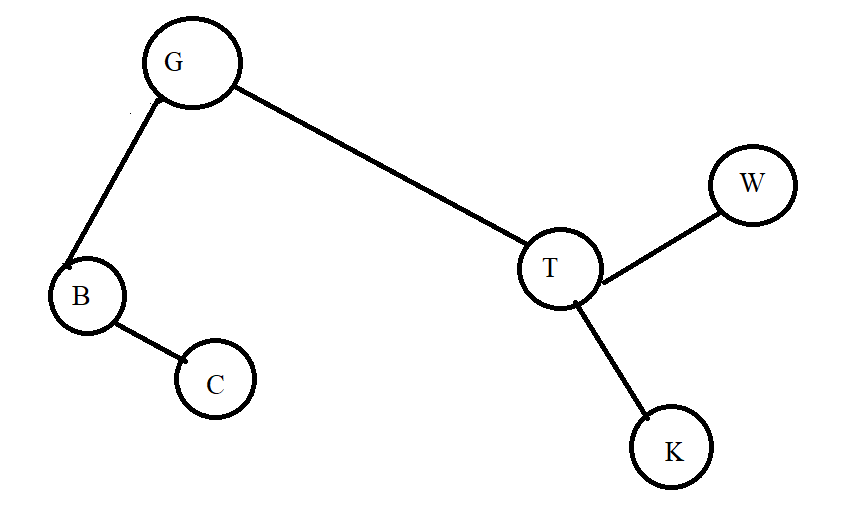
-Delete:

Delete J:



Delete A:





*Example 3:*

* **INPUT:** input is a data pair of the form (x,y) where x is an integer and y is a letter. We will compare the number first if the number is equal, we compare the letter.

**-**Insert: (5,g), (4,b), (5,a), (9,e), (2,p), (1,z)

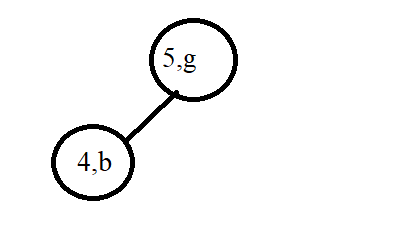
**-**Find: (9,e), (7,a)

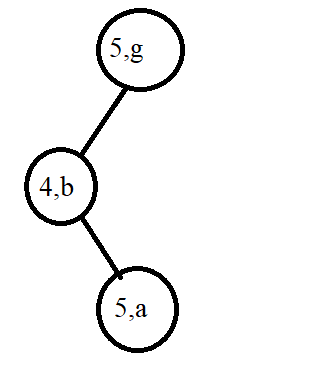
**-**Delete: (5,g), (2,p)

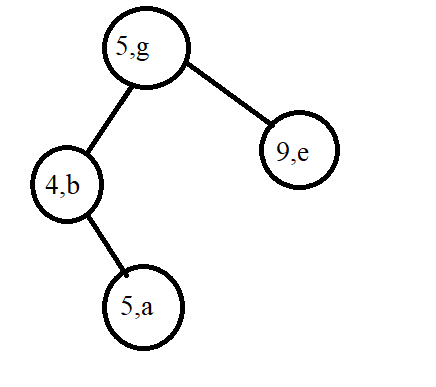
* **OUTPUT:**

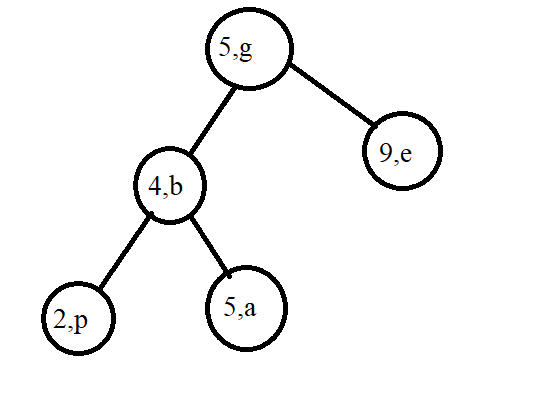
-Insert:

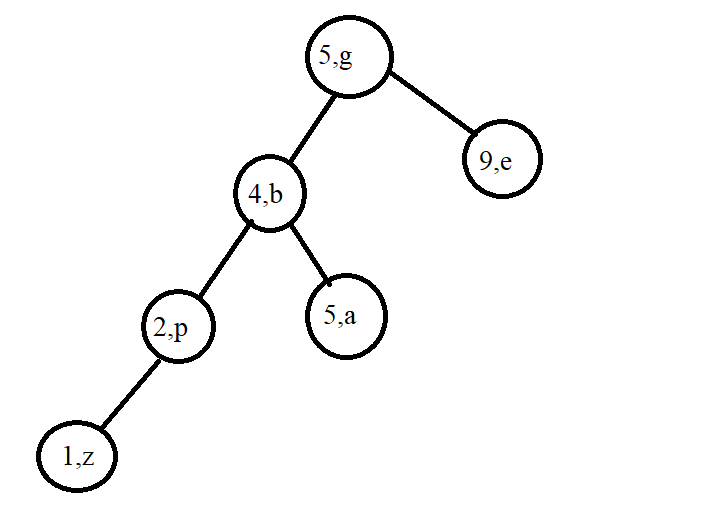






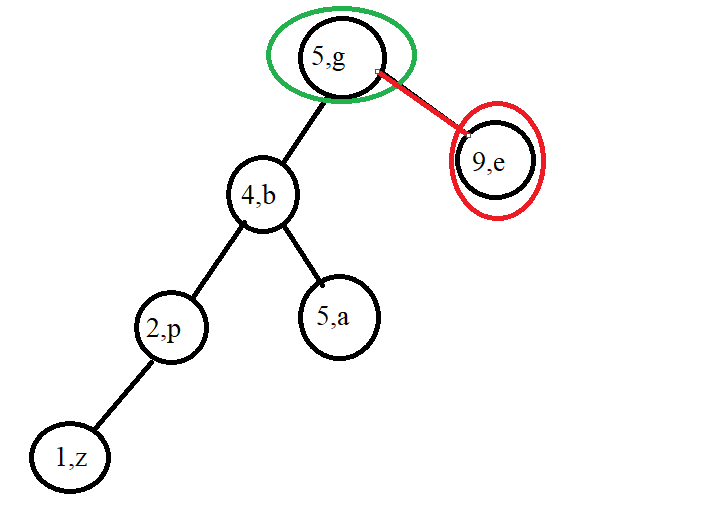




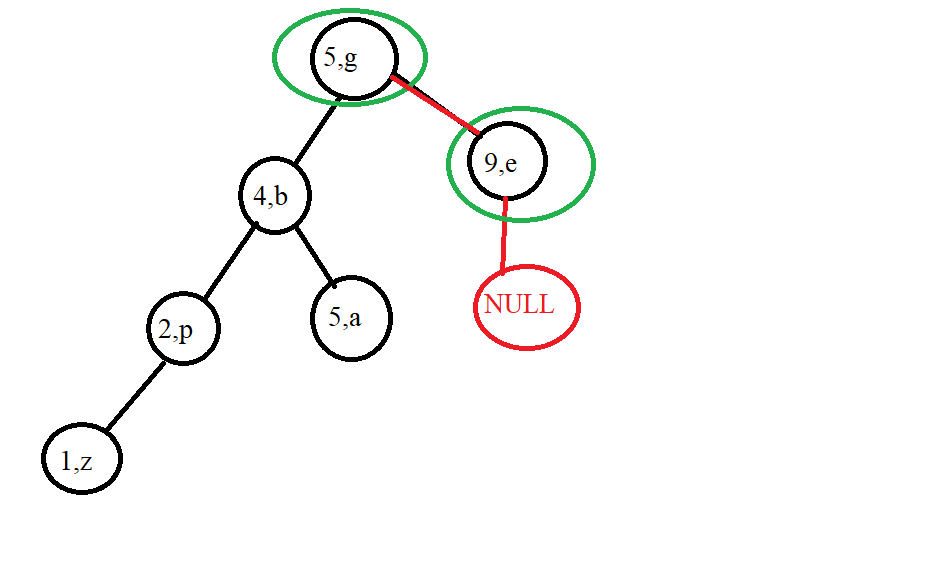


-Find:

.Find (9,e)

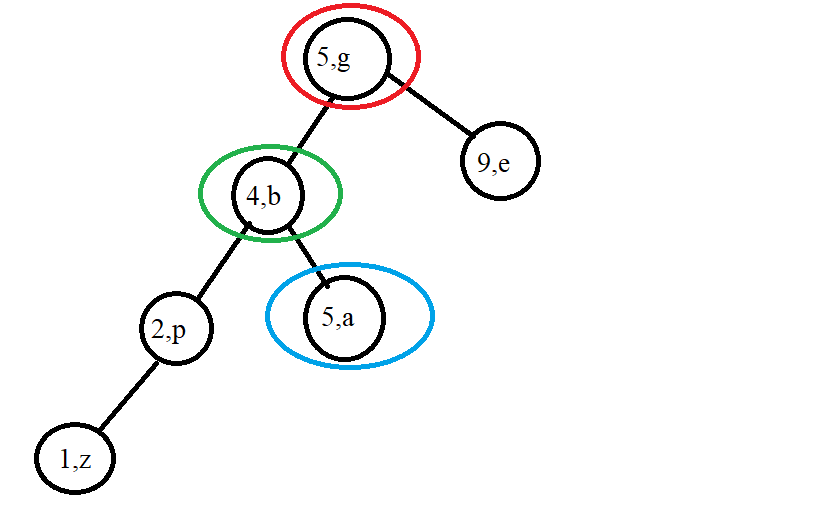


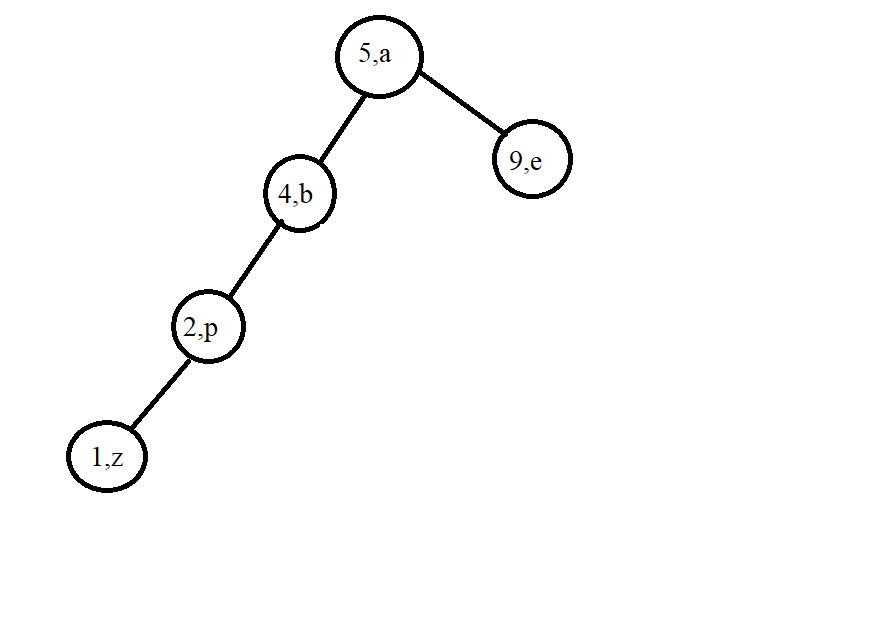
.Find (7,a)



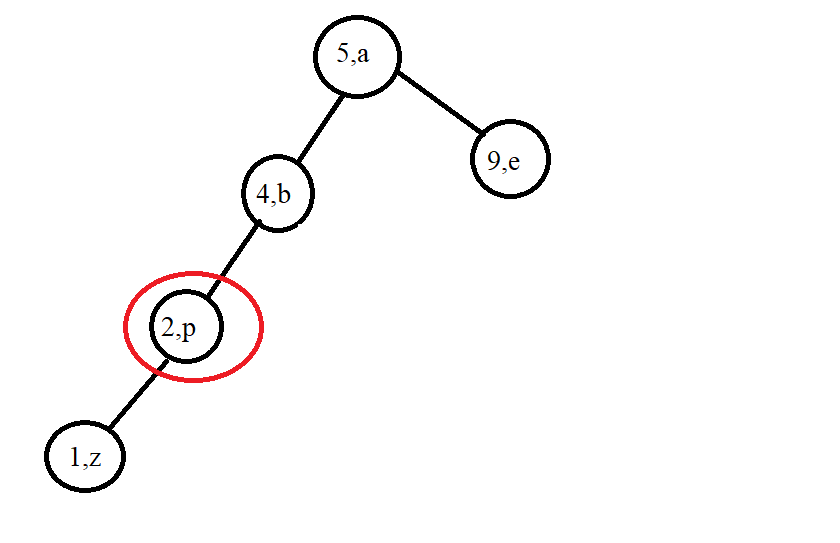
Delete:

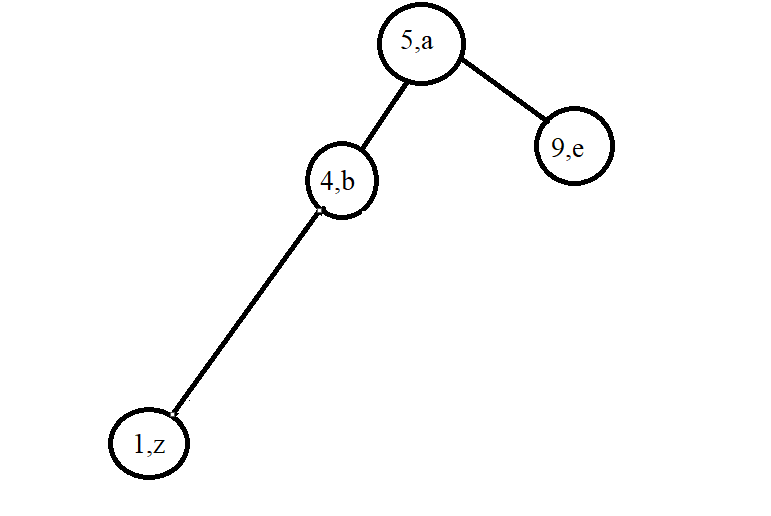
.Delete (5,g)





.Delete (2,p)

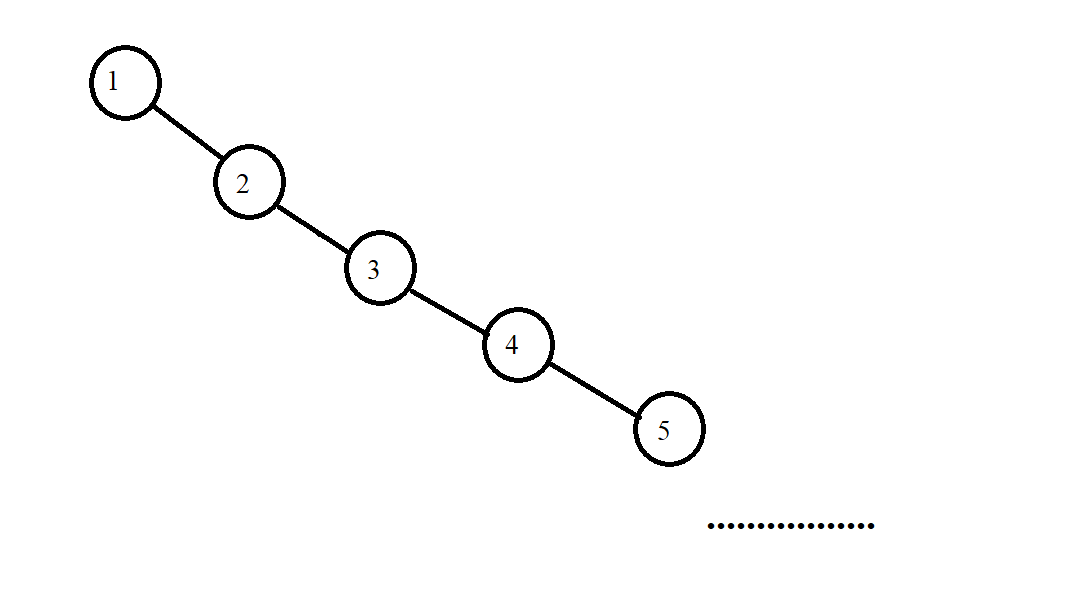




1. **Balance tree (AVL):**

**\***Question:

-If we keep inserting elements into the BST tree in ascending order of 1,2,3,4,.... or descending, for example, 1000,999,....then when we insert millions of numbers or the whole input ratio in that order, we will create a tree that leans left or right. Example:



\*The main content of the balanced tree:

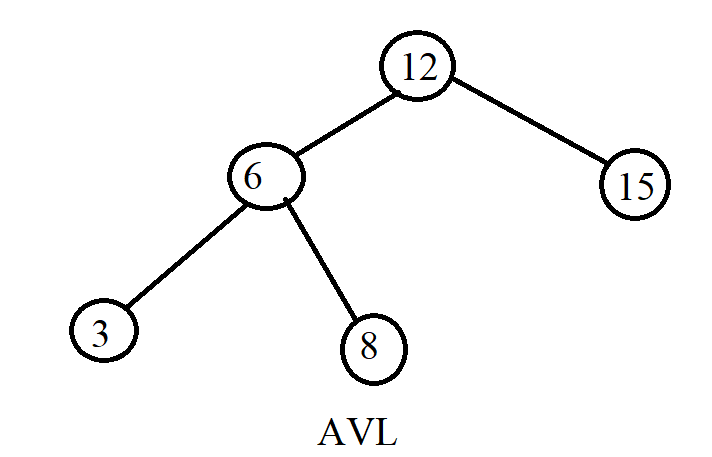
-The AVL is:

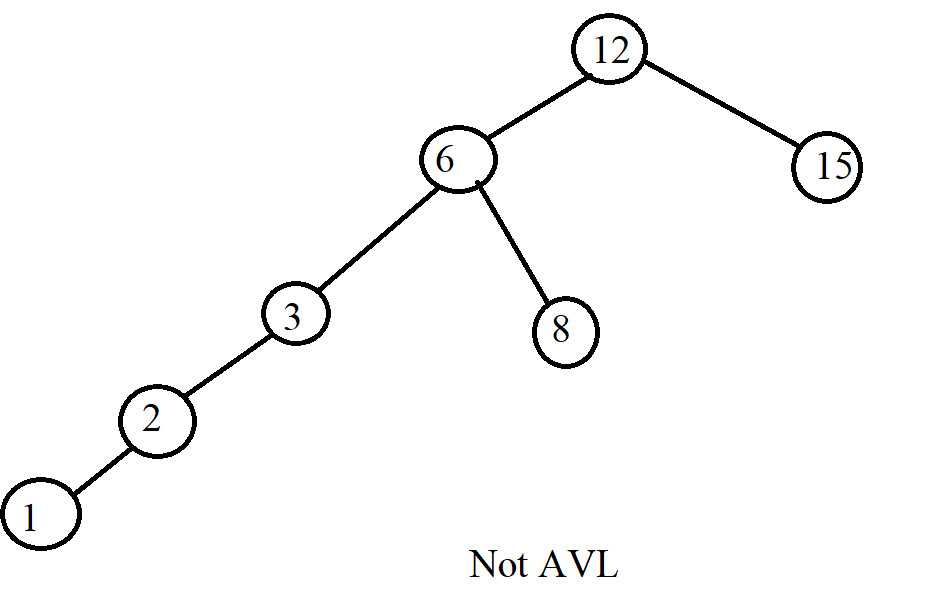
+A search binary tree: That is, it also has only 2 children in each node, where the left child node is smaller than the parent node and the right child node is larger than the parent node.

+The key to a balanced tree is: At each node, the difference between left and right subtrees is less than or equal to 1. Note here that the concept of equilibrium is not that the left and right subtrees are equal, but that their difference is less than or equal to 1 because in the case of two equal left and right subtrees, it is too good to achieve. I have the formula:

∀p ∈ TAVL :(hp −> Left − hp −> Right) ≤ 1

Example:





+ How to declare in a node:



-Balance: We call getBalance a function that tells us whether it is unbalanced or unbalanced. If it is out of balance, it will tell us which side it is unbalanced.

+If getBalance = 1: Right child node is more than left child node.

+If getBalance = -1: Left child node is more than right child node

+ The rest is in the case of a balanced tree.

// A utility function to get height

// of the tree

int height(Node\* N)

{

if (N == NULL)

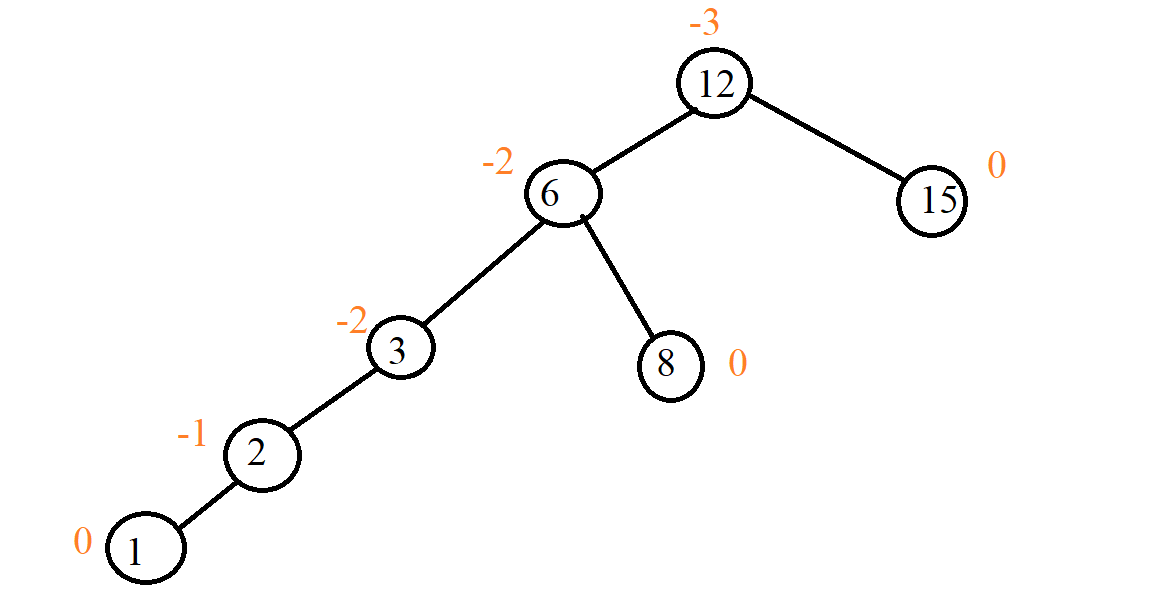
return 0;

return N->height;

}

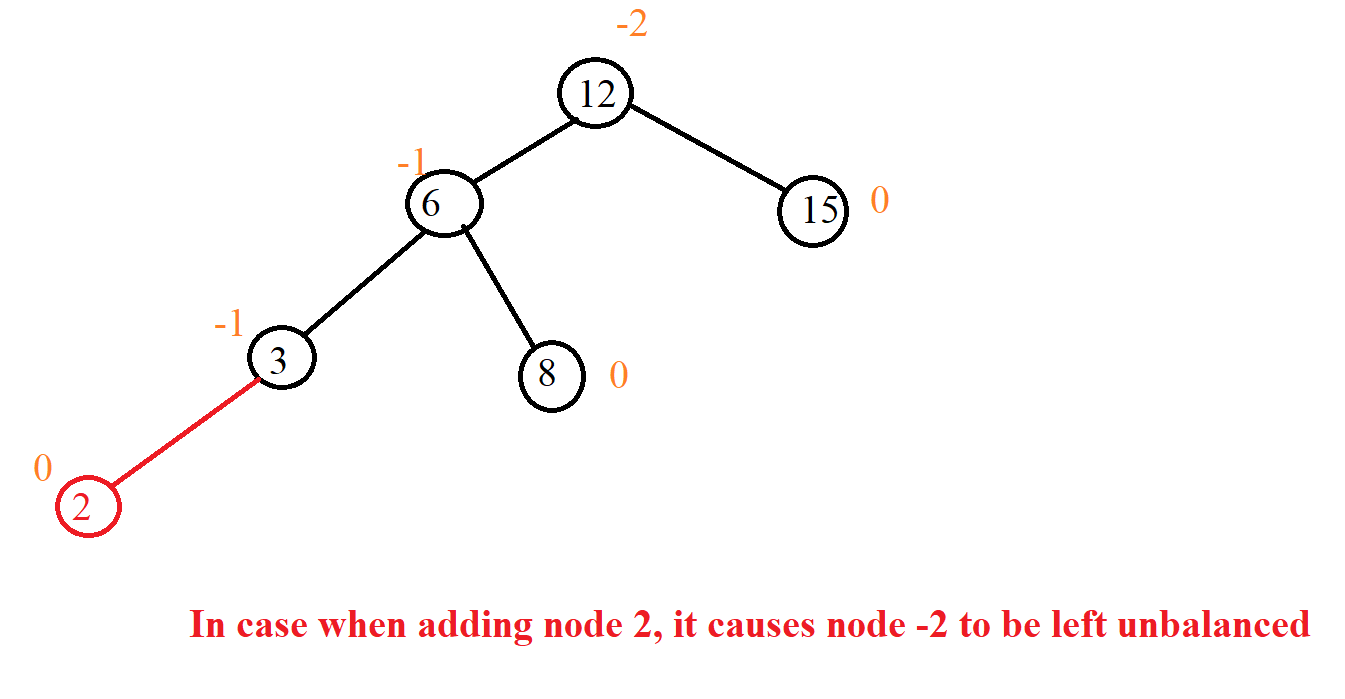


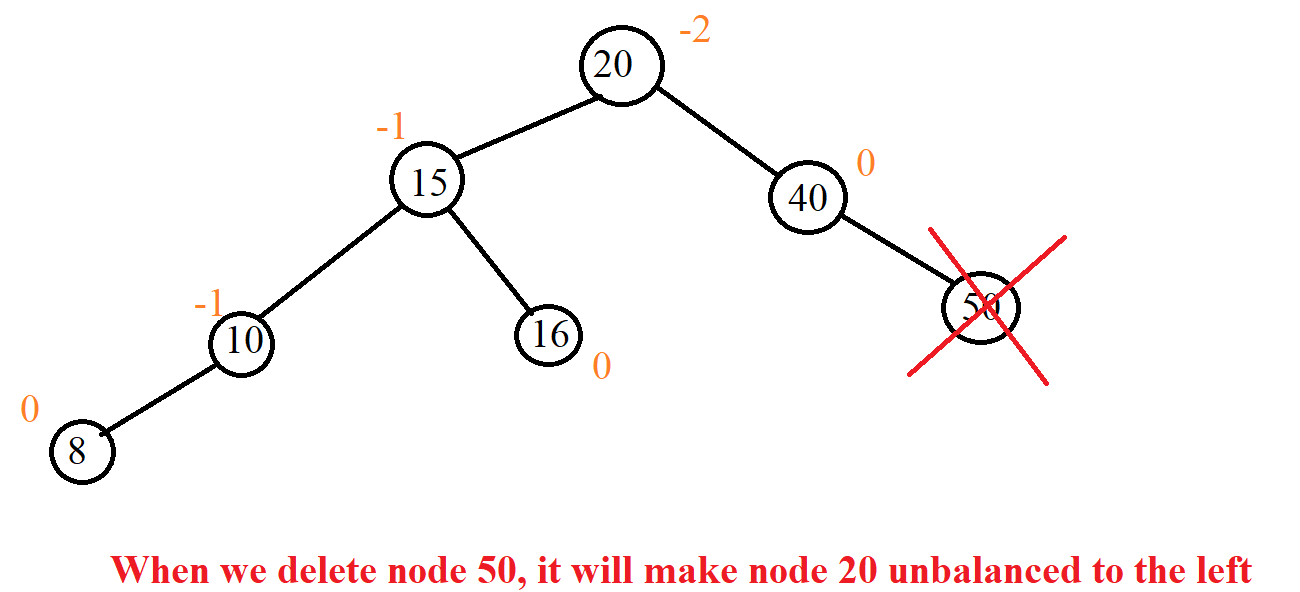
Example:



- Adjust the tree to balance: There are two operations on the tree that make the tree unbalanced: insertion and deletion because then we have changed the state of the tree. In the case of inserting an element into the tree, we have the ability to make the parent node of the node we just inserted unbalanced or the nodes on it also fall into the unbalanced case and in the case of deletion, it will too.

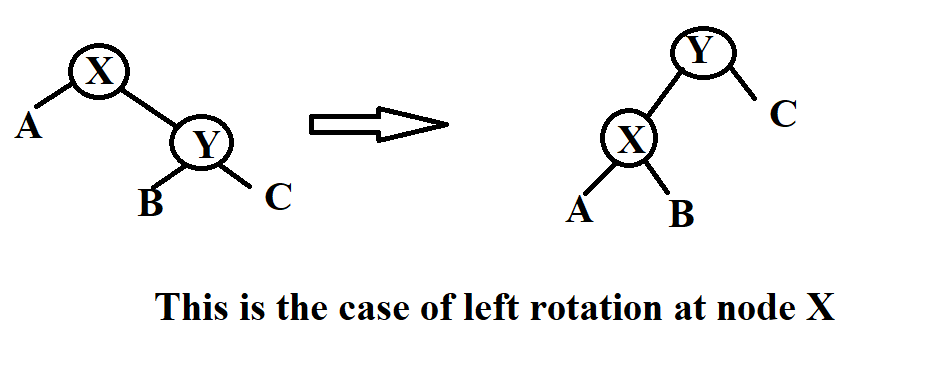
Example:





So the question is, what should we do when we fall into that situation? The answer is that we use tree correction.

- Before going into editing the tree, we have to see how to rotate 2 nodes because actually when editing the tree, we rotate the node for it to become an AVL tree.



Node\* leftRotate(Node\* x)

{ Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left),

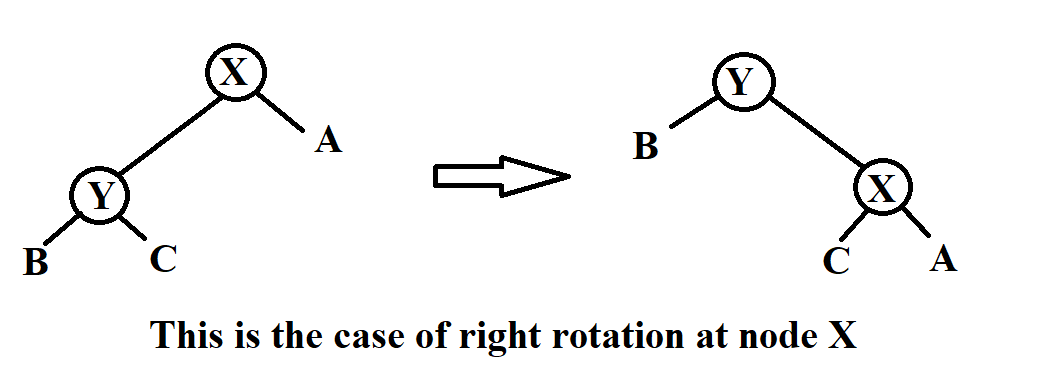
height(x->right)) + 1;

y->height = max(height(y->left),

height(y->right)) + 1;

// Return new root

return y; }



Node\* rightRotate(Node\* y)

{

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left),

height(y->right)) + 1;

x->height = max(height(x->left),

height(x->right)) + 1;

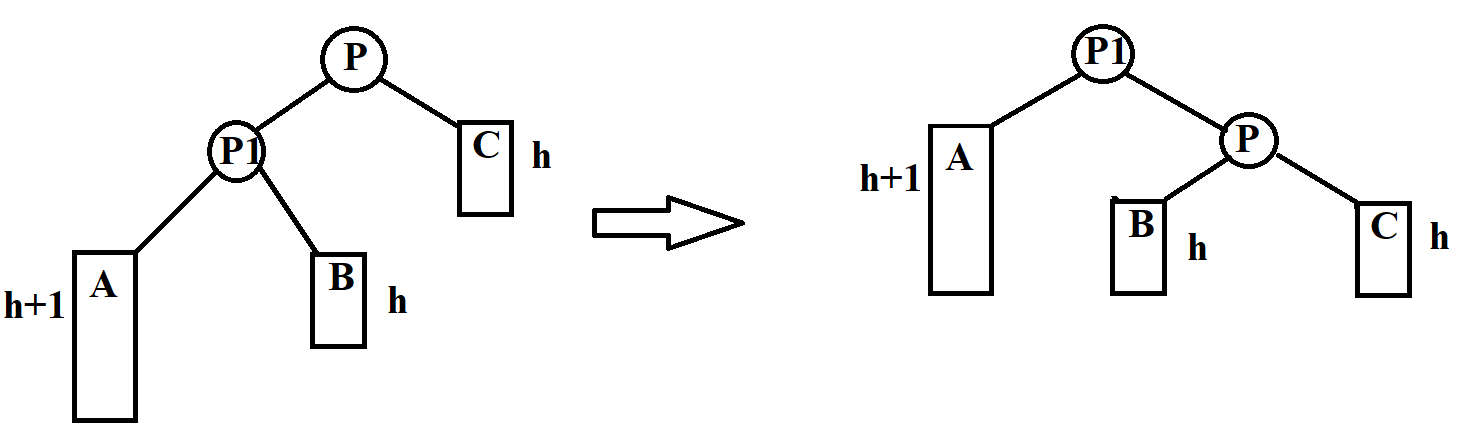
// Return new root

return x;

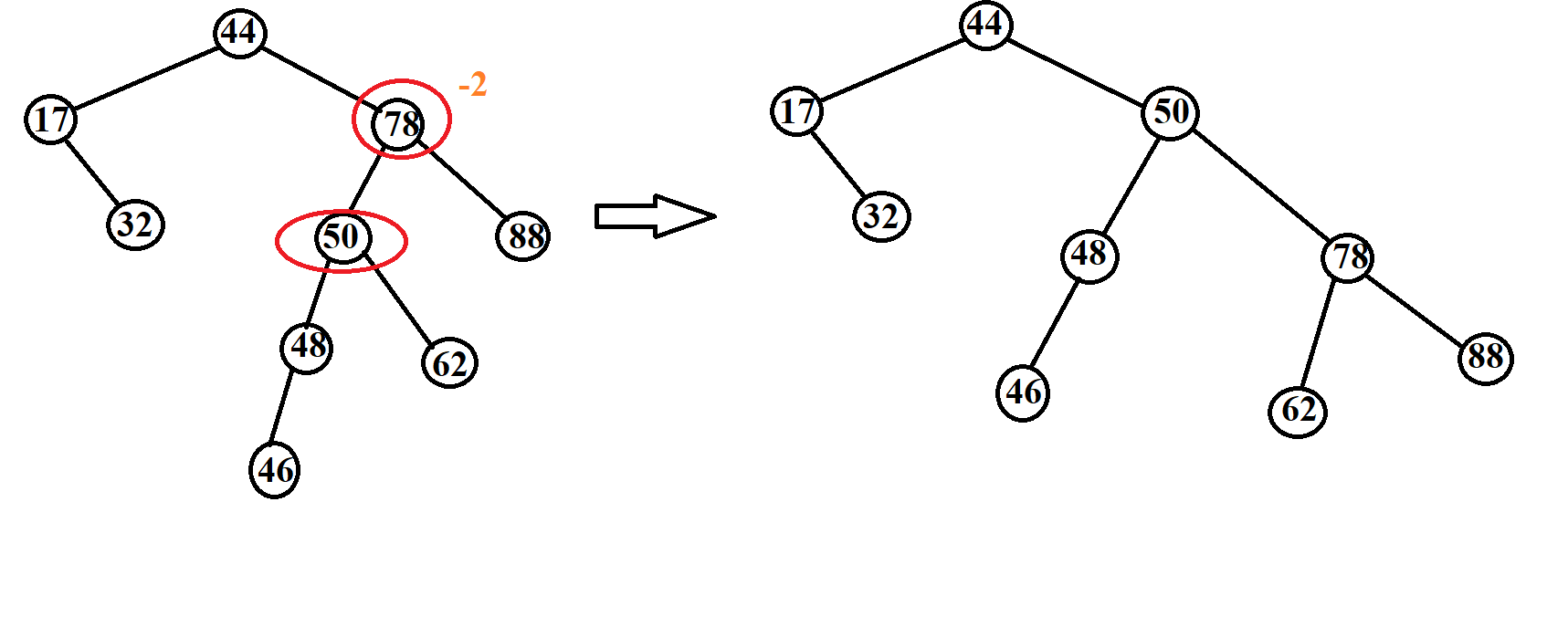
}

- We have 4 cases to edit the tree which are Left-Left Case, Right-Right Case, Left-Right Case and Right-Left Case. But we only need to know 2 cases, Left Left Case and Left Right Case because the other 2 cases we do the same but on different sides.

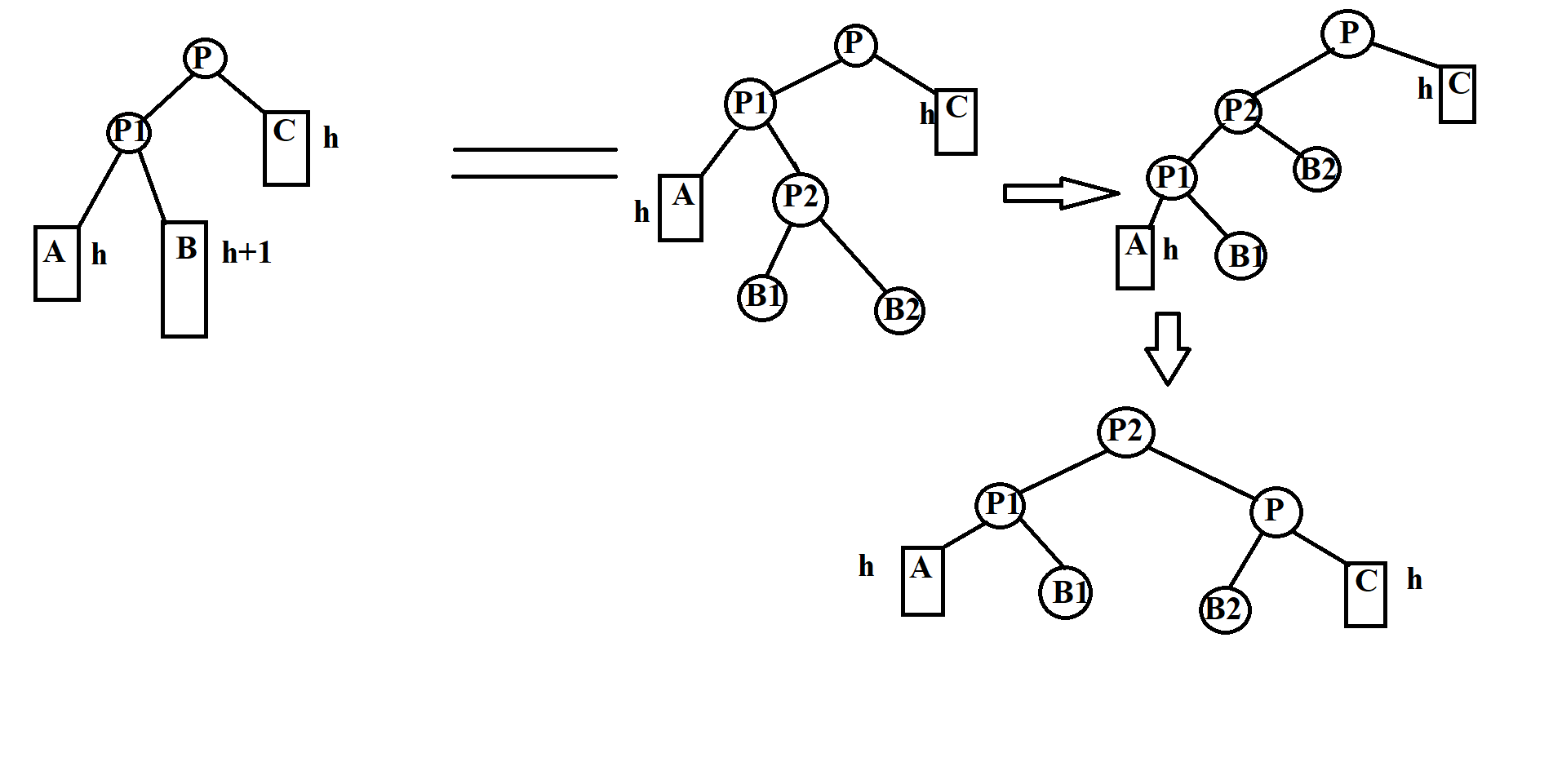
+Left-Left Case (L-L): In this case we remember that the node is left unbalanced, the left child of that node is also unbalanced to the left.



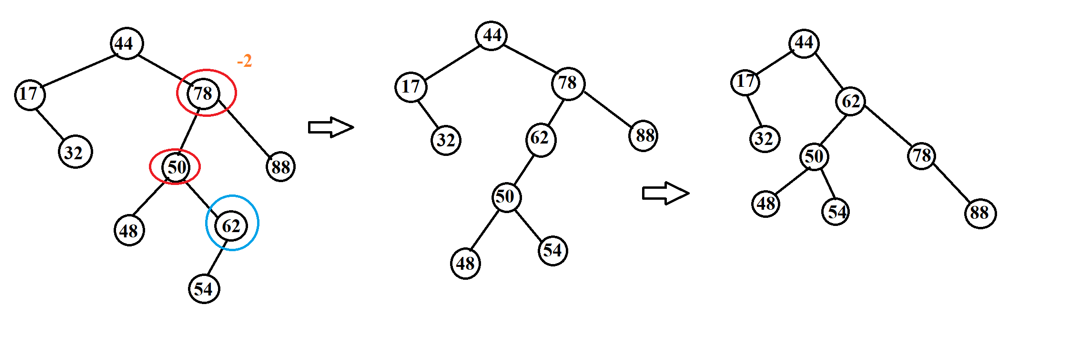
Example:



+Left\_Right Case (L-R): In this case we remember that the left node is unbalanced, the left child of that node is also unbalanced on the right.



Example:



-Insert, delete: How tree-like insert and delete BST but the other is in step when done, we must calibrate the balance tree (we only balance when the length of the left subtree minus the length of the right subtree and we take abs, the result is large more than 1).

+Insert: When it is finished, we will go back to the previously reviewed nodes to see if that node is unbalanced or not.

Node\* insert(Node\*node, int key)

{

/\* 1. Perform the normal BST rotation \*/

if (node == NULL)

{

//node = newNode(key);

return(newNode(key));

}

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else // Equal keys not allowed

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(height(node->left), height(node->right));

/\* 3. Get the balance factor of this

ancestor node to check whether

this node became unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced,

// then there are 4 cases

// Left Left Case

if (balance > 1 && key < node->left->key)

return rightRotate(node);

// Right Right Case

if (balance < -1 && key > node->right->key)

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node->left->key)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key)

{

node->right = rightRotate(node->right);

return leftRotate(node);

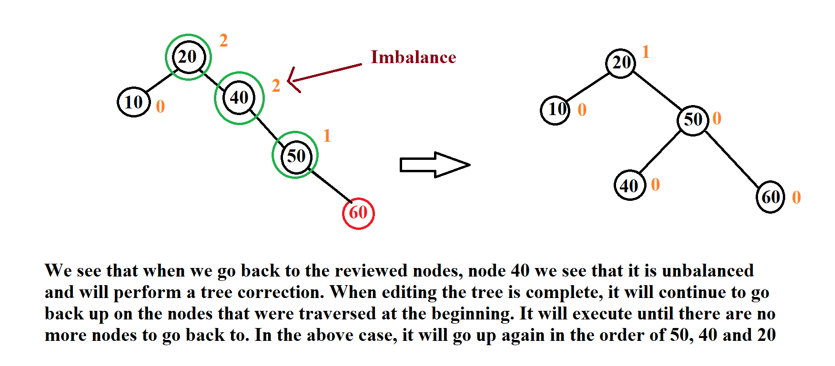
}

/\* return the (unchanged) node pointer \*/

return node;

}

Example: when we add 60



+Delete: When it is finished, we will go back to the previously reviewed nodes to see if that node is unbalanced or not.

Node\* deleteNode(Node\* root, int key)

{

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else

{

// node with only one child or no child

if ((root->left == NULL) ||

(root->right == NULL))

{

Node\* temp = root->left ? root->left : root->right;

// No child case

if (temp == NULL)

{

temp = root;

root = NULL;

}

else // One child case

\*root = \*temp; // Copy the contents of

// the non-empty child

free(temp);

}

else

{

Node\* temp = minValueNode(root->right);

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right,

temp->key);

}

}

if (root == NULL)

return root;

root->height = 1 + max(height(root->left),

height(root->right));

int balance = getBalance(root);

// Left Left Case

if (balance > 1 &&

getBalance(root->left) >= 0)

return rightRotate(root);

// Left Right Case

if (balance > 1 &&

getBalance(root->left) < 0)

{

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Right Case

if (balance < -1 &&

getBalance(root->right) <= 0)

return leftRotate(root);

// Right Left Case

if (balance < -1 &&

getBalance(root->right) > 0)

{

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

-Complexity and tree height:

+Tree height:

. hAVL <1.44log2(N + 1).

. AVL tree is 44% higher than optimal BST tree.

+ Complexity:

. Search: O().

. Adding an element: O(), including tree search O() and editing O().

. Deleting element O(log2N), including tree search O() and editing O().

***\*Illustration:***

*Example 1:*

**INPUT:**

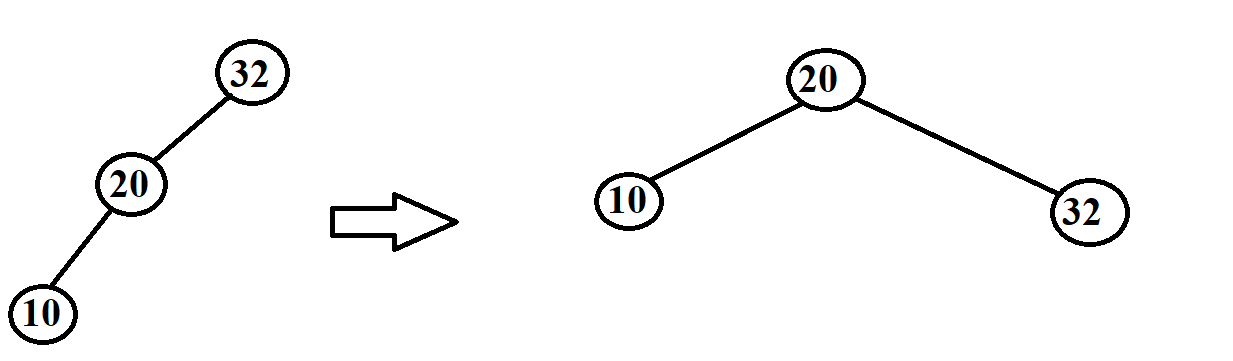
-Insert:32, 20, 10, 15, 40, 25, 27, 26, 13, 14.

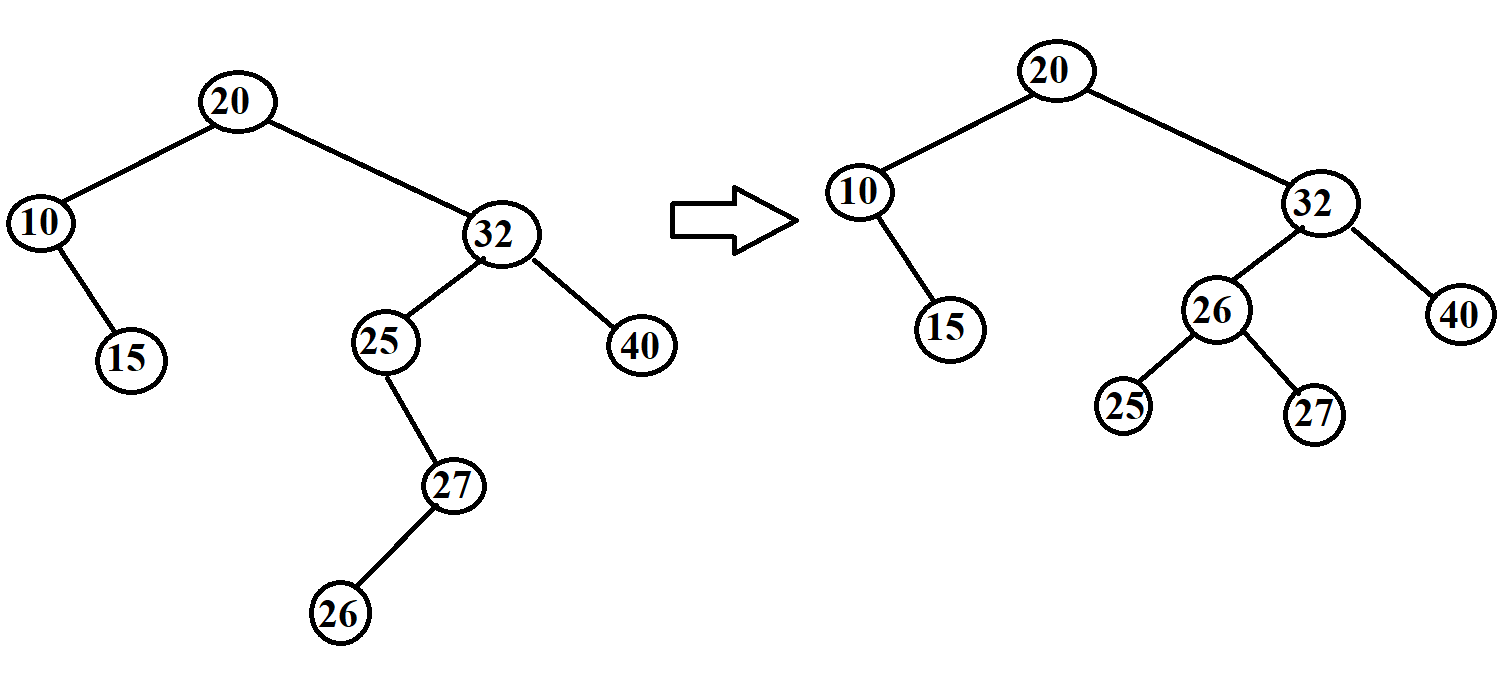
-Find: 15, 29

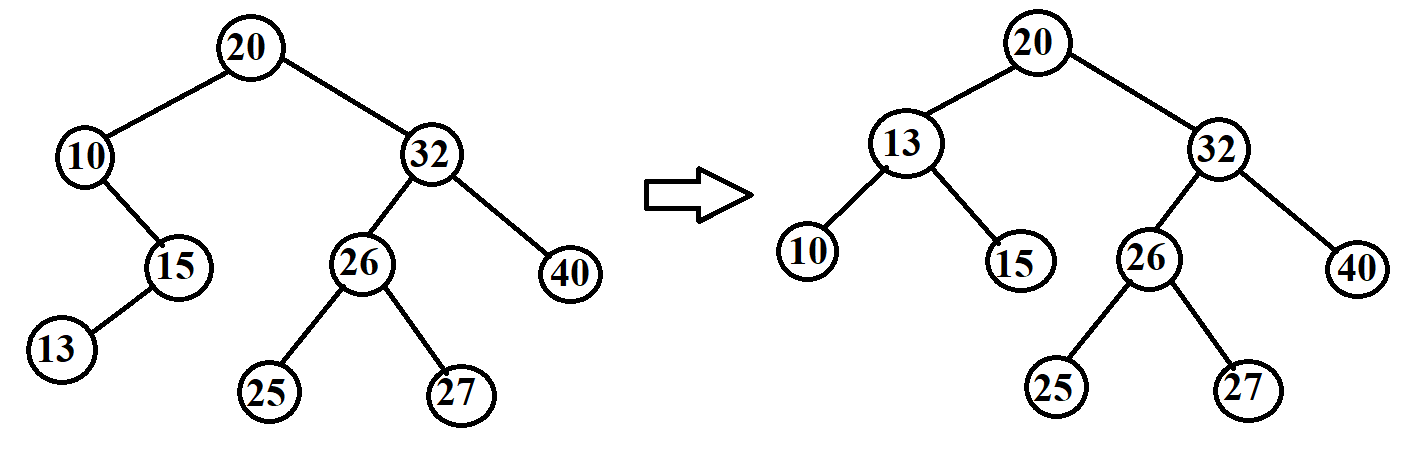
-Delete: 25, 40 , 20

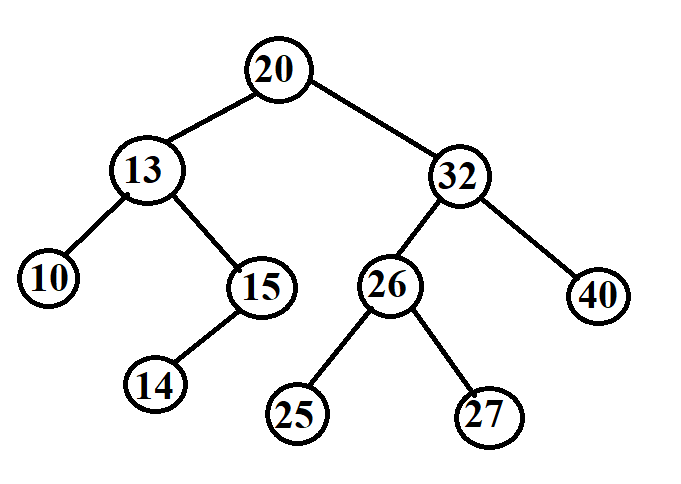
**OUTPUT:**

-Insert:



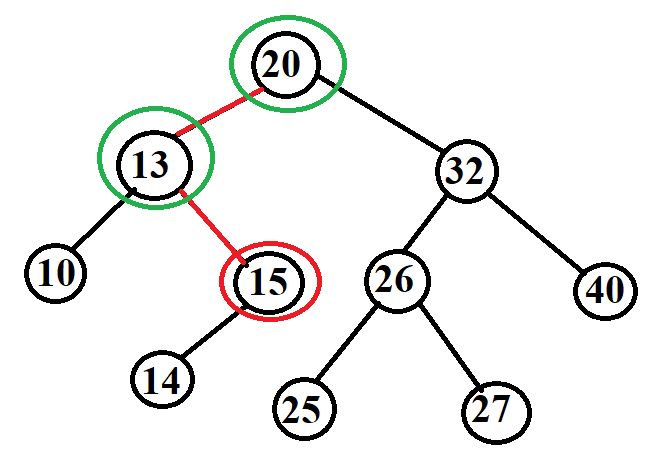




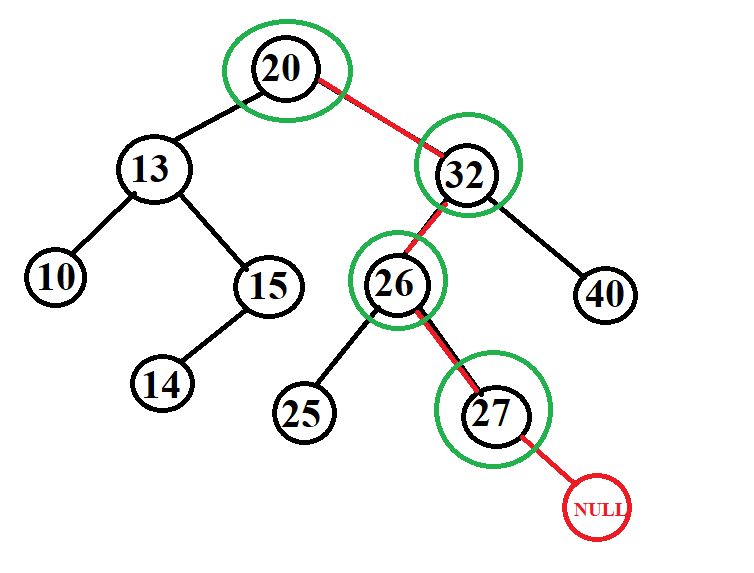


-Find:

+Find 15

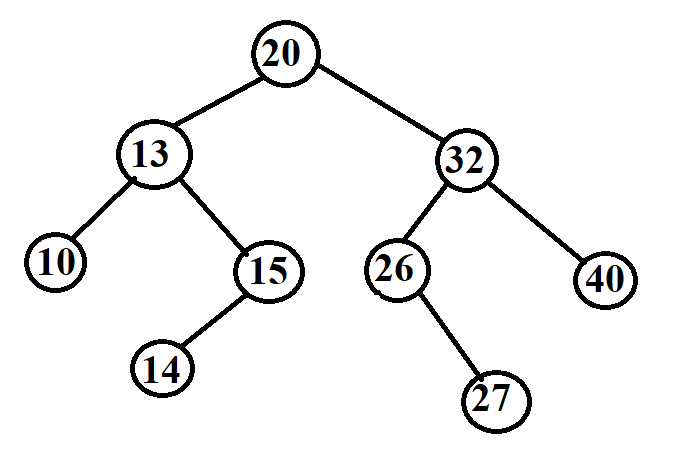


+Find 29

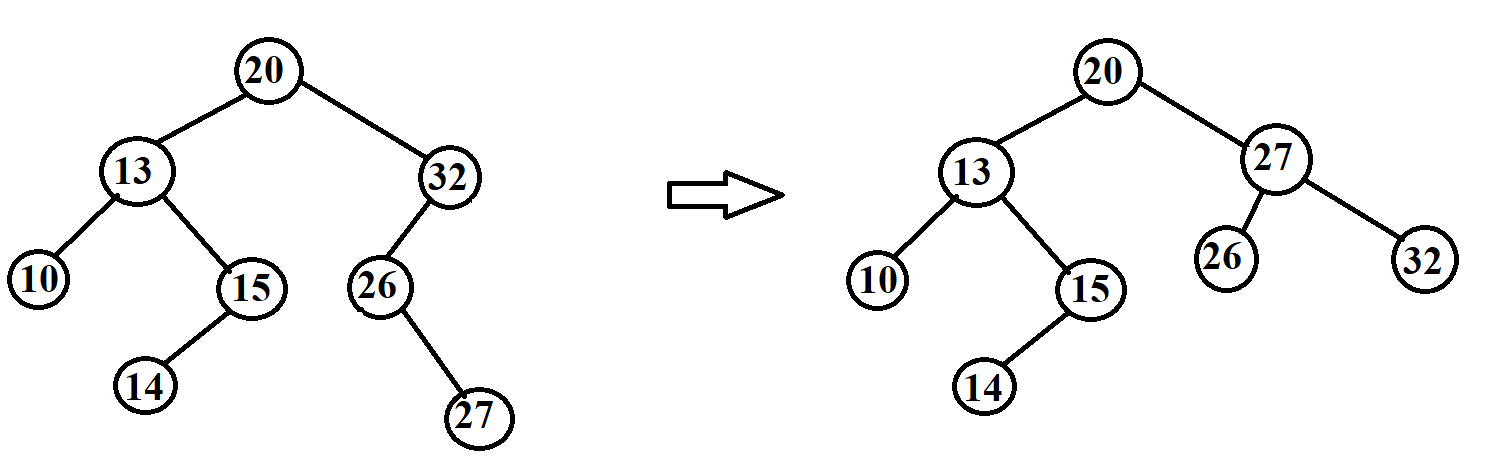


-Delete:

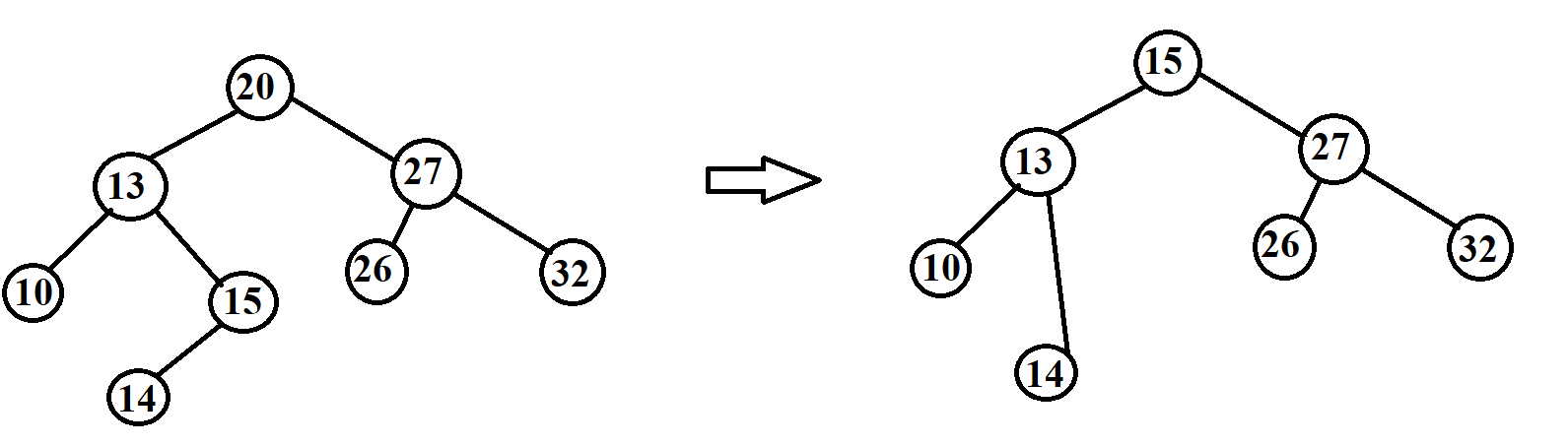
+Delete 25



+Delete 40



+Delete 20



*Example 2:*

**INPUT:**

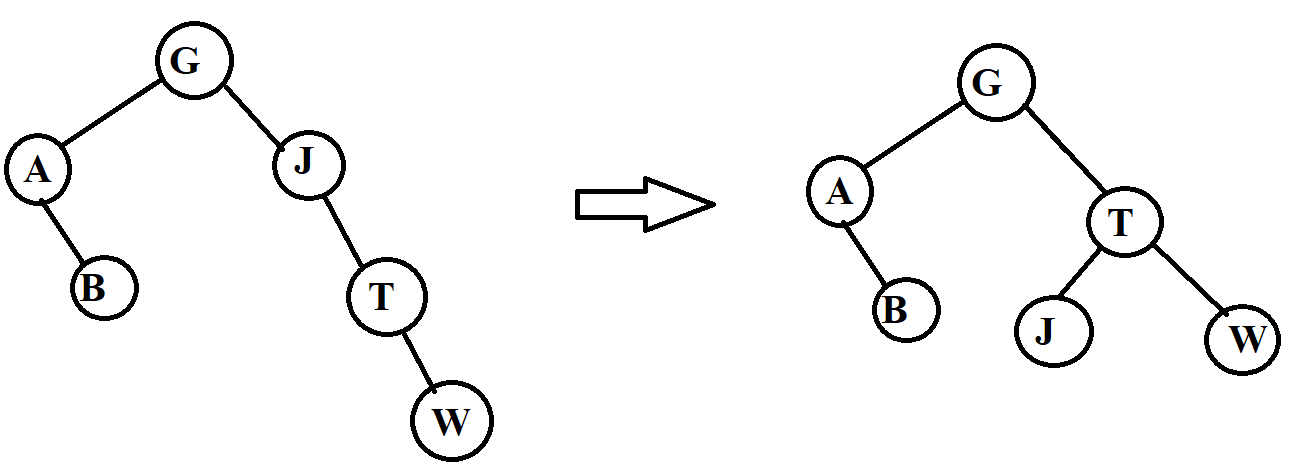
-Insert: G, J, A, T, B, W, C, K.

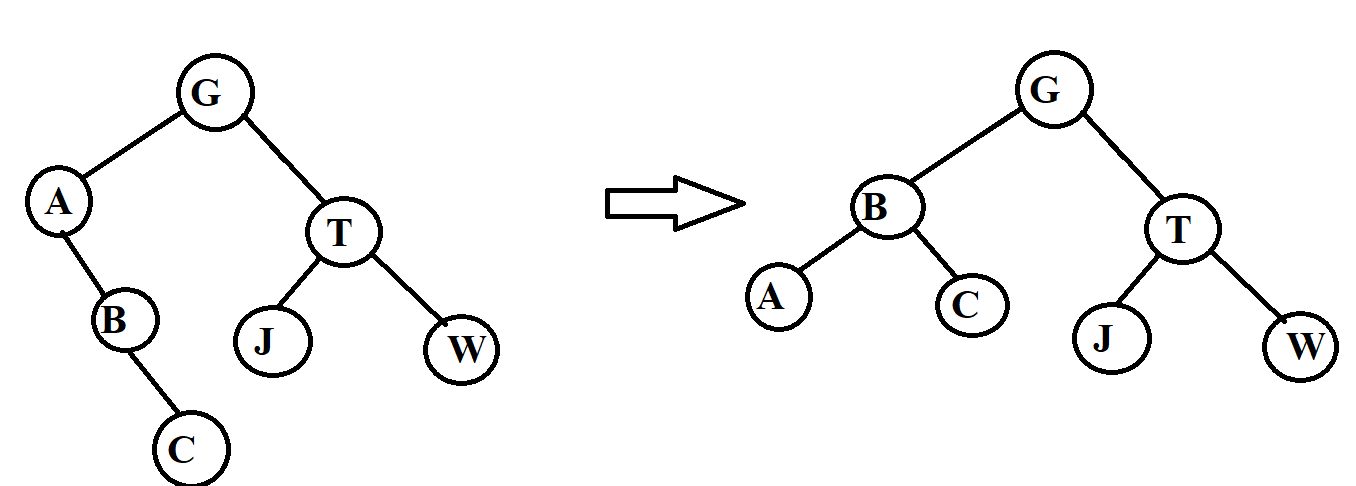
-Find: A, Z

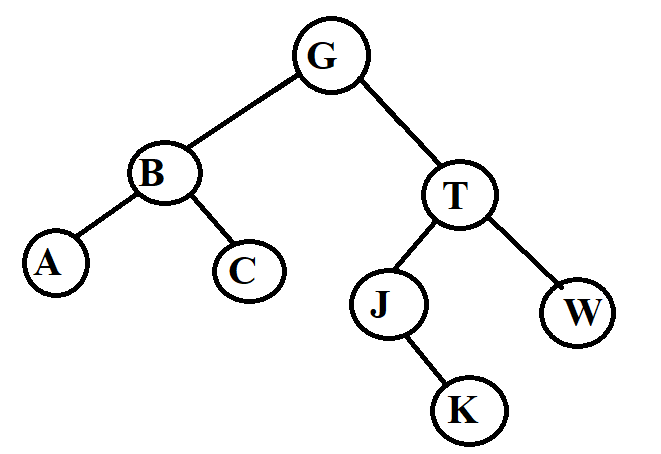
-Delete: J, A

**OUTPUT:**

-Insert:

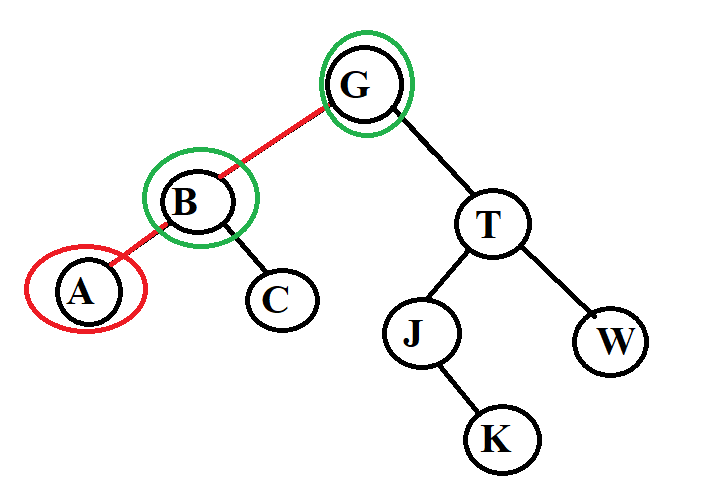




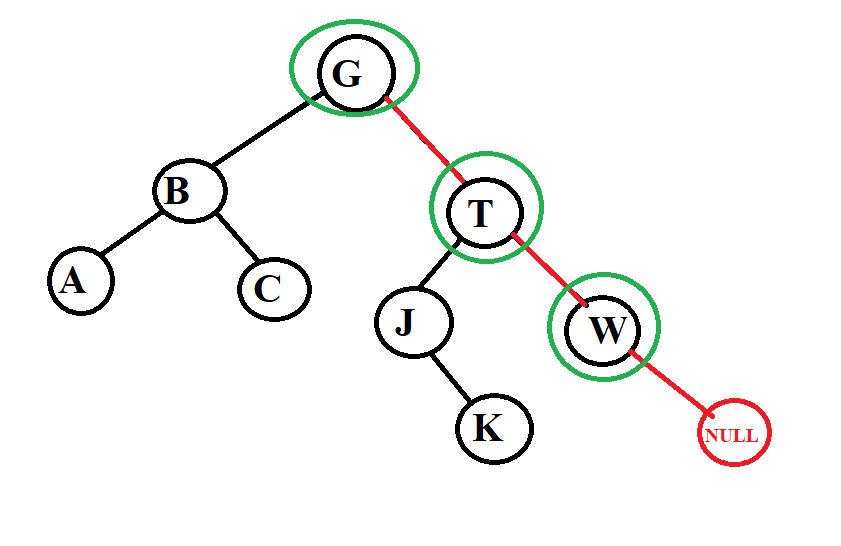


-Find:

+Find A.

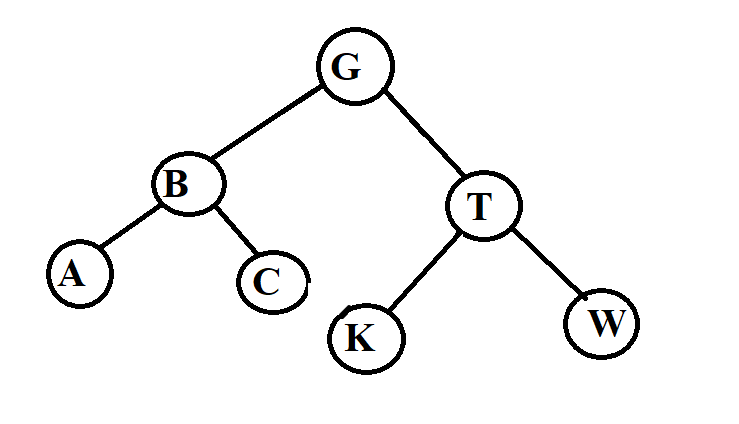


+Find Z.

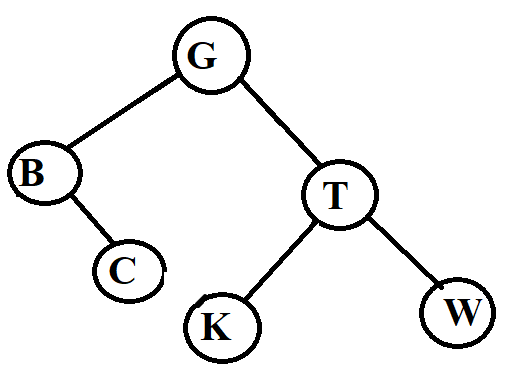


-Delete:

+Delete J



+Delete A



*Example 3:*

**INPUT:** input is a data pair of the form (x,y) where x is an integer and y is a letter. We will compare the number first if the number is equal, we compare the letter.

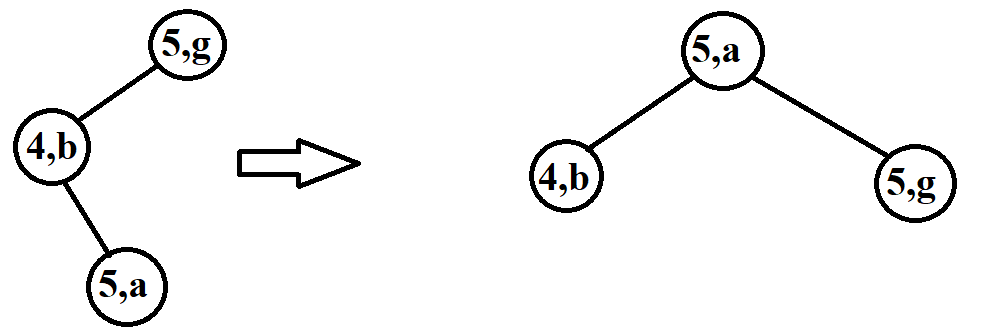
**-**Insert: (5,g), (4,b), (5,a), (9,e), (2,p), (1,z)

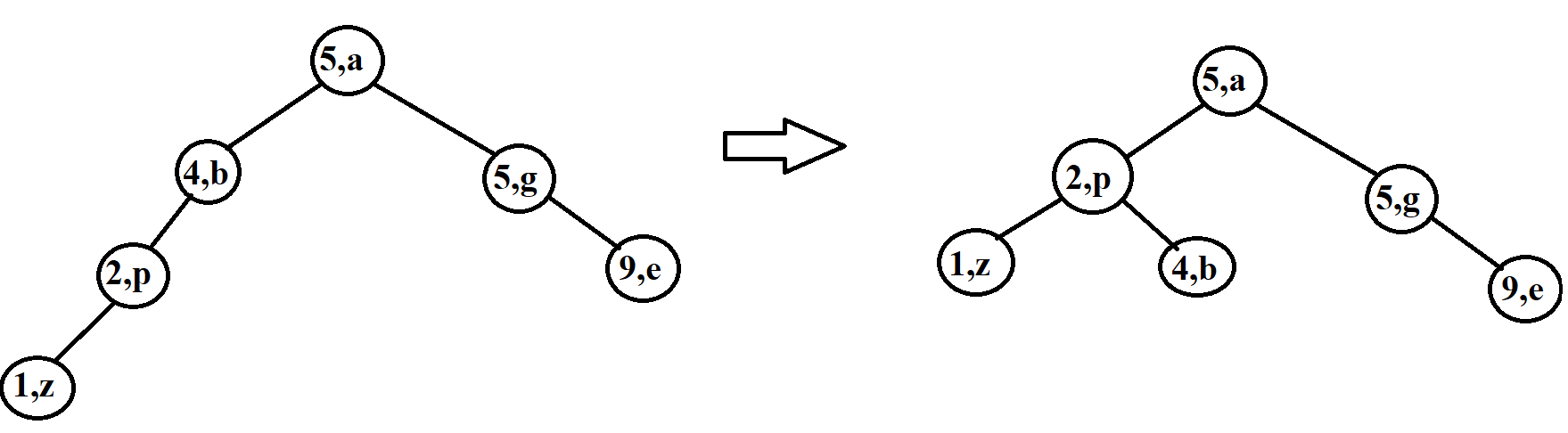
**-**Find: (9,e), (7,a)

**-**Delete: (5,g), (2,p)

**OUTPUT:**

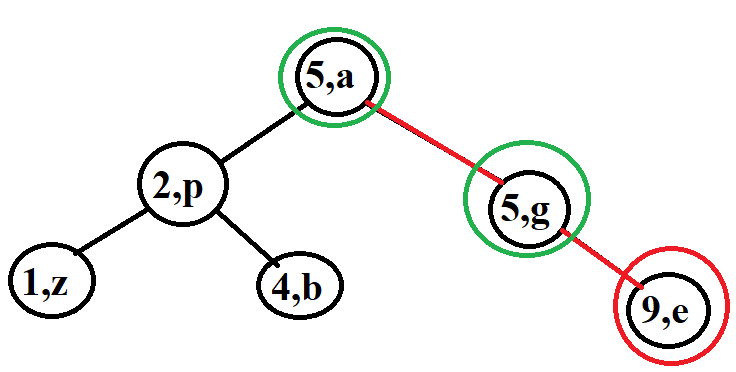
-Insert:



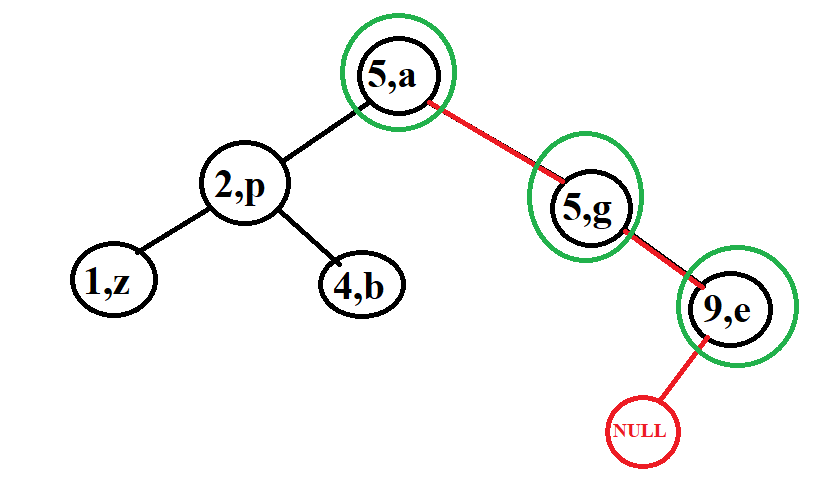


**-**Find:

**+**Find (9,e)

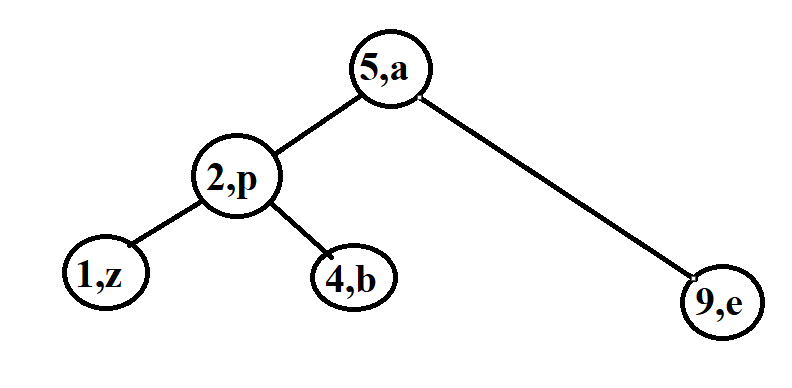


**+**Find (7,a)

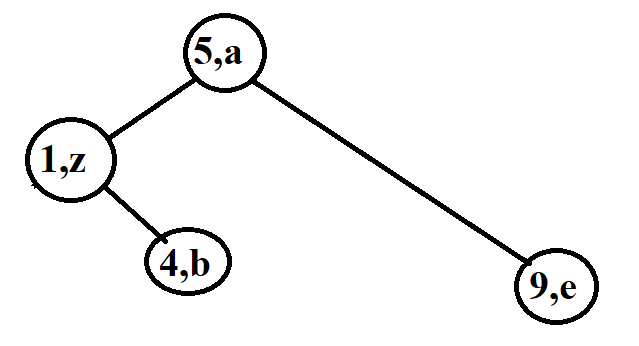


-Delete:

+Delete (5,g)



+Delete (2,p)



**IV.Red-Black Tree (RB Tree)**

**\***Question:

-Is there a balanced search tree that has fewer structural changes in the tree but is faster to find, delete and insert than an AVL tree in practice? The answer is yes, it is a red black tree as fewer rotations are done due to relatively relaxed balancing.

\*The main content of the Red-Back tree:

-Red-Black Tree is a binary search tree.

-It must satisfy 5 principles:

[1] Each node is red or black.

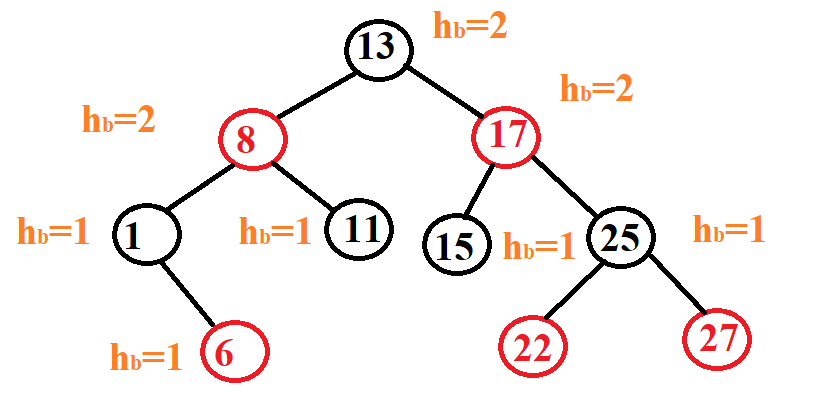
[2] The root node is always black.

[3] Node NULL is always black. In a B Tree, we will call the NULL node NIL because it has the unique properties of the RB Tree such as color.

[4] If that node is red then its child node is black. That means there is no case of 2 red nodes in a row.

[5] If all patches originate from any 1 node (including the root node), the number of black nodes in its path to the NULL node must be equal. From there, we will denote the number of black nodes as hb(x). This black node calculation does not count node x if that node x is a black node.

Example:



-How to declare node in RB tree:

#define RED 1

#define BLACK 0

typedef bool colour;

struct node

{

node \*parent,\*left,\*right;

int key;

colour color;

};

#define NIL &sentinel /\* all leafs are sentinels \*/

node sentinel = {NIL,NIL, NIL, 0, BLACK};

-Insert a new element:

+The inserted node is always red because then our rule number 5 will not be normalized but agree that rule number 4 can be violated but then it will be easier for us to modify the tree.

+We also do the same as the BST tree in the first step, find the insertion location.

+The next step we will check to see if it is in any of the cases after inserting:

. If the parent node is Black => it doesn't matter.

. If the parent node is Red => violates parent-child are Red => adjust to balance tree. The tree correction here only takes place when there are 2 consecutive red nodes, not like the AVL tree, which has to check the length of 2 subtrees of the node. After done, if the tree meets the equilibrium condition, then it is correct.

void insert(int key)

{

if(root==NIL)

{

root=new node;

root->left=NIL;

root->right=NIL;

root->parent=NULL;

root->color=BLACK;

root->key=key;

return;

}

node \*parent,\*current,\*x;

parent=NULL;

current=root;

while(current!=NIL)

{

if(key==current->key) return;

parent=current;

if(key>current->key) current=current->right;

else current=current->left;

}

x=new node;

x->parent=parent;

x->left=NIL;

x->right=NIL;

x->color=RED;

x->key=key;

if(key>parent->key) parent->right=x;

else parent->left=x;

fixinsert(x);

return;}

Before going into how to edit the tree, let's talk about rotation in RB Tree. Basically, the rotation is no different from the rotation of a node in AVL, but in the code we have to modify the parent node of that node because then the parent node of that node is already connected to the child node of the rotated node.

void rotateLeft(node \*x)

{

if(x->right==NIL) return;

node \*y=x->right;

x->right=y->left;

if(y->left!=NIL) y->left->parent=x;

y->parent = x->parent;

if(x!=root)

{

if(x==x->parent->left)

x->parent->left=y;

else x->parent->right=y;

}

else root=y;

y->left=x;

x->parent=y;

}

void rotateRight(node \*x)

{

if(x->left==NIL) return;

node \*y=x->left;

x->left=y->right;

if(y->right!=NIL) y->right->parent=x;

y->parent = x->parent;

if(x!=root)

{

if(x==x->parent->left)

x->parent->left=y;

else x->parent->right=y;

}

else root=y;

y->right=x;

x->parent=y;

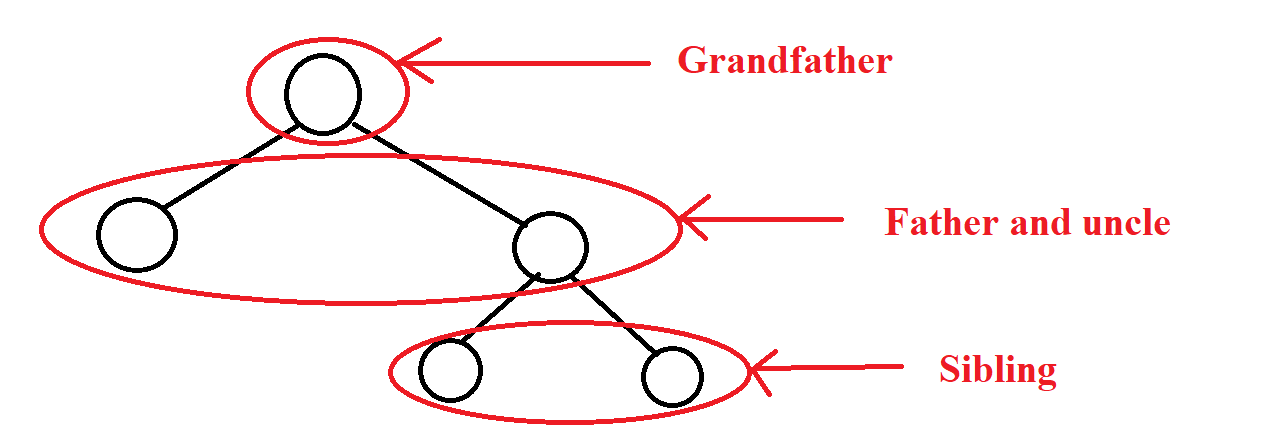
}

+ How to edit a tree when the parent node is a red node: The algorithms have mainly two cases.

. If the uncle is red, we do recolour.

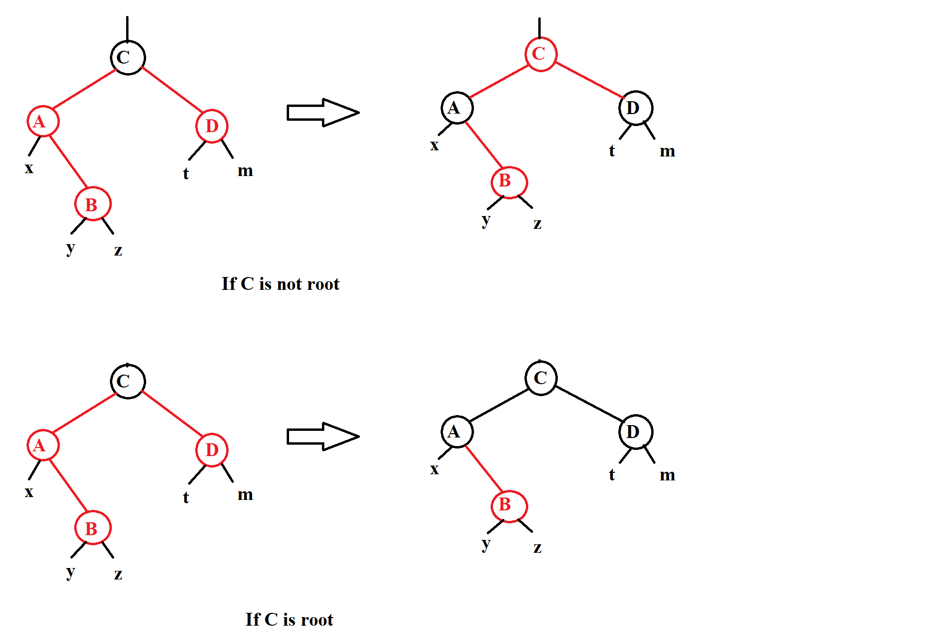
. If the uncle is black, we do rotations and /or recolouring.

*Note:* uncle is the same node as the parent node of the node under consideration.

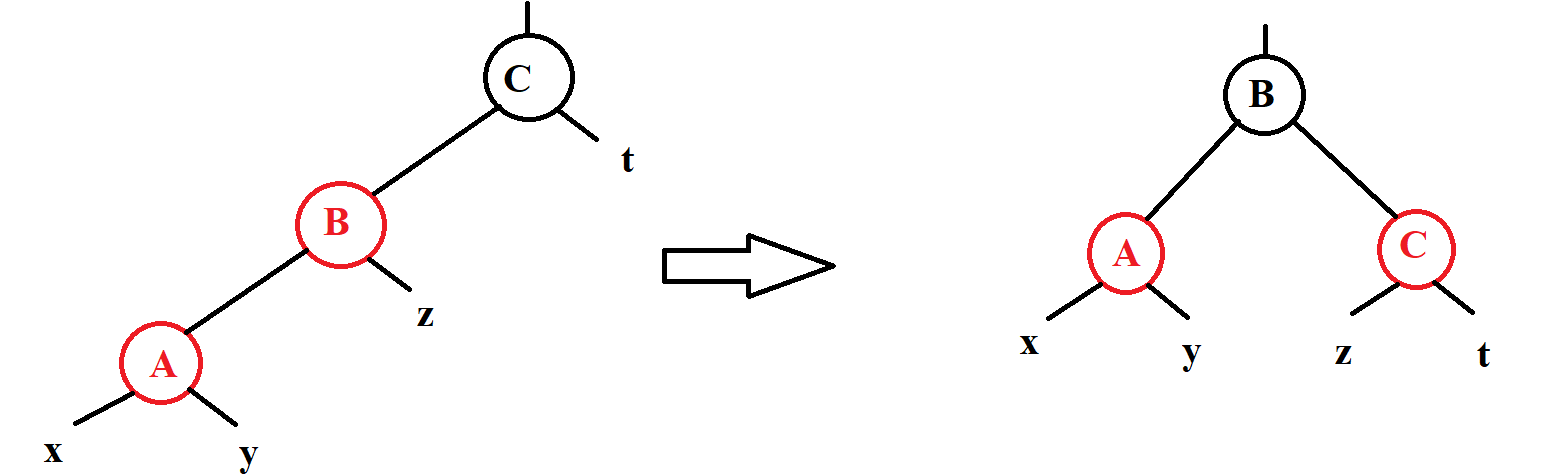


We list 6 smaller cases for easy control. But we only need to find the first 3 cases where the newly inserted node is located in the left subtree of the grandfather node and the last 3 cases do the same as above but only differ in direction.

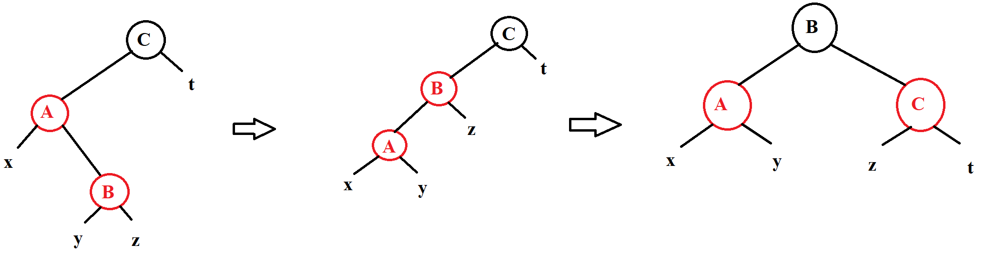
***Case 1****:* If the parent node is red, the uncle is red, then we only need to change the color of the parent node, uncle and grandfather node. But notice that if he is a root node, we won't change the color but keep it black.



***Case 2****:* If the parent node is red, the uncle is black and the inserted node is the left child of the parent node, we change the color of the parent node and the grandfather node, then we will rotate right at the grandfather node.



***Case 3****:* If the red parent node, the black uncle node and the newly inserted node are the right child nodes of the parent node, then we take the parent node to turn left and right rotate the grandfather node. Finally, we change the color of the parent node and the grandfather node.



***Case 4, Case 5, Case 6****:* We also do the same operation as above, but only in the opposite direction, this time the node is added it is in the right subtree of his node.

+ At the last step of the insert, we just do the same as the AVL tree, which is to go against the above nodes that have been approved to see if there is a violation of the attribute of 2 consecutive red nodes.

void fixinsert(node\* x)

{

while(x!=root&&x->parent->color==RED)

{

if(x->parent==x->parent->parent->left)

{

node \*y=x->parent->parent->right;

if(y->color==RED)

{

x->parent->color=BLACK;

y->color=BLACK;

x->parent->parent->color=RED;

x=x->parent->parent;

}

else{

if(x==x->parent->right)

{

x=x->parent;

rotateLeft(x);

}

x->parent->color=BLACK;

x->parent->parent->color=RED;

rotateRight(x->parent->parent);

}

}

else

{

node \*y=x->parent->parent->left;

if(y->color==RED)

{

x->parent->color=BLACK;

y->color=BLACK;

x->parent->parent->color=RED;

x=x->parent->parent;

}

else

{

if(x==x->parent->left)

{

x=x->parent;

rotateRight(x);

}

x->parent->color=BLACK;

x->parent->parent->color=RED;

rotateLeft(x->parent->parent);

}

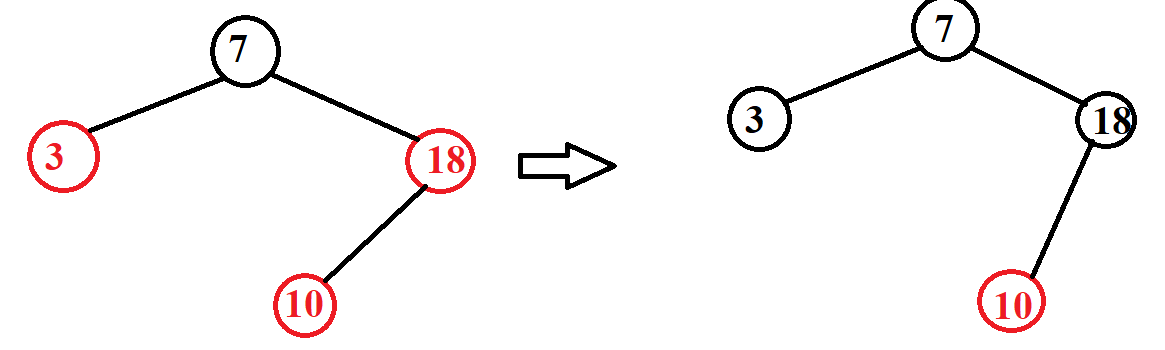
}

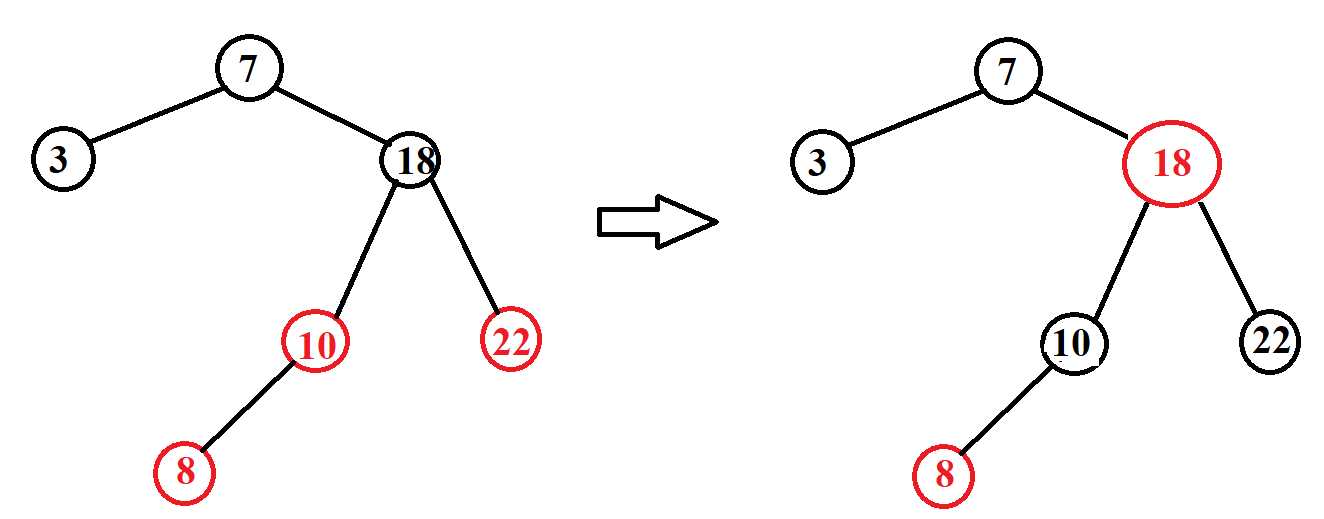
}

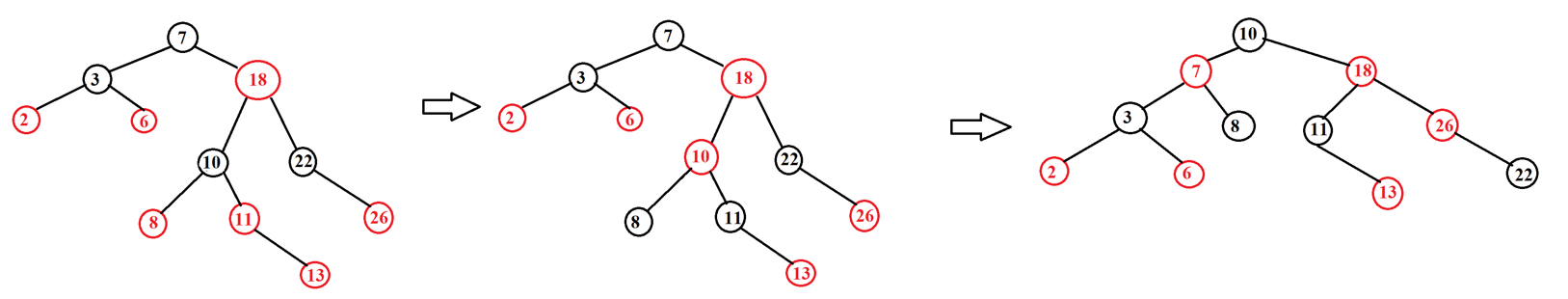
root->color=BLACK;

}

Example: Add 7, 3, 18, 10, 22, 8, 11, 26, 2, 6, 13







-Delete an element:

+The first step we do the same as BST is to find and delete the node to be deleted. Note when in the case of deleting node 2 children, we only replace the lice without changing the color.

+Just like inserting we also have 2 cases:

. The actually deleted node is the red node. In this case, we just need to delete the node normally or in other words, treat it as a BST tree to delete. Because it doesn't violate any rules in the RB tree.

. The actually deleted node is the black node. This is the most difficult case because deleting it will cause many cases of violating the rules of the tree. For example, we will be able to make the root node turn red because it can delete root and replace it with a red node. It may violate the rule that 2 consecutive child nodes cannot be red because deleting that node may cause the child node and parent node of the deleted node to be red and connected. Finally, we can change the number of black nodes of a node to the NIL node because deleting can reduce the black node.

+As mentioned above, the case of deleting a black node is the most difficult case we need to consider.

. After finding the node that needs to be deleted, we don't need to delete it, but we have to do an action called removing that node from the tree. That is, if the node is less than 2 children, then we connect the parent node with the child and then we will edit the tree immediately the child node of the node is deleted, and in the case of deleting a node with 2 child nodes, we will immediately deal with the child node at the node whose replacement price is taken.

. After editing the tree, we will delete the node to delete.

void erase(int key)

{

node \*x,\*y,\*current;

current=root;

while(current!=NIL)

{

if(current->key==key) break;

if(key>current->key) current=current->right;

else current=current->left;

}

if(current==NIL) return;

if(current->left==NIL||current->right==NIL)

{

y=current;

}

else

{

y=current->left;

while(y->right!=NIL) y=y->right;

}

if(y->left!=NIL) x=y->left;

else x=y->right;

x->parent=y->parent;

if(y->parent!=NIL)

{

if(y==y->parent->left)

y->parent->left=x;

else y->parent->right=x;

}

else

{

root = x;

}

current->key=y->key;

if(y->color==BLACK)

fixerase(x);

delete y;

return;

}

\*There are 2 cases in case of deleting the black node:

***Case 1****:* The child of the delete black node is in red: we just need to change the color of child node to black.

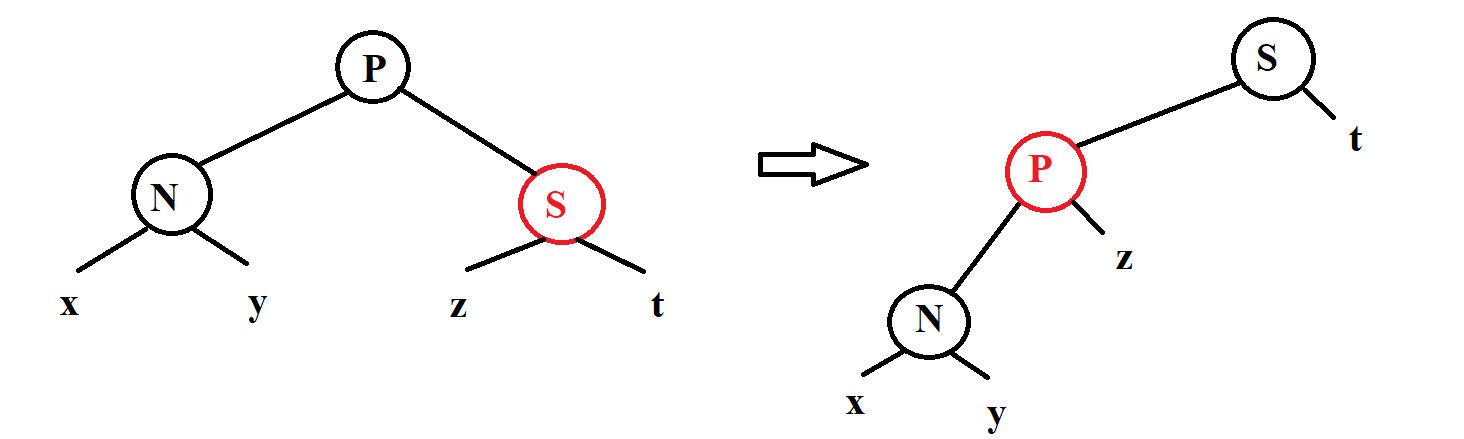
***Case 2****:* the child of the deleted black node is in black: we will do the steps below(We only need to consider the case that the node to be deleted is in the left subtree of its parent node, in the other case we do the same but only do the opposite.) We will call the node that is deleted as node n, the sibling node as node s and the parent node is p.

* Step 1: If n has no parent (n is the root): end. Otherwise we go to step 2.
* Step 2: n is black, s is red.

-We change color and rotate left at parent node.

-We change the color of node p.

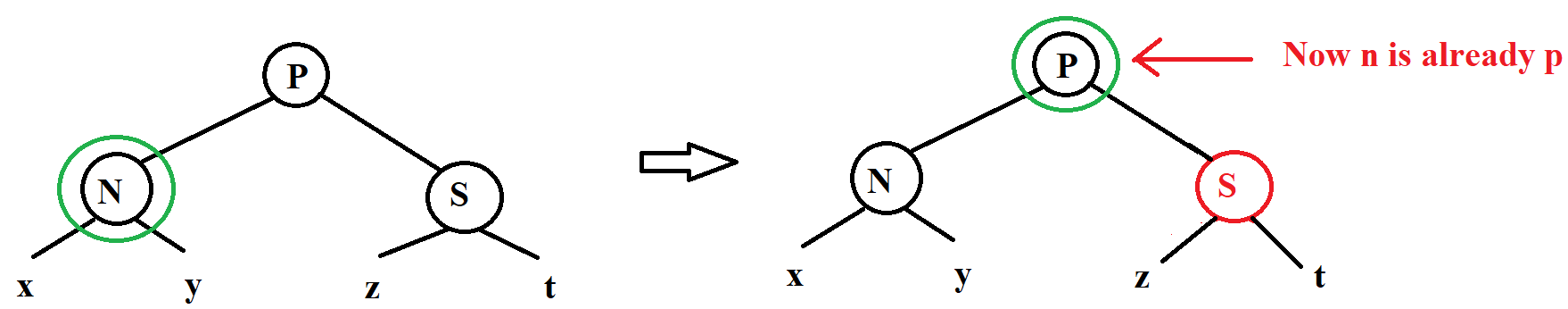
-Go to step 3.



* Step 3: n is black, s is black, children of s are black, p is black.

-Change node s color to red.

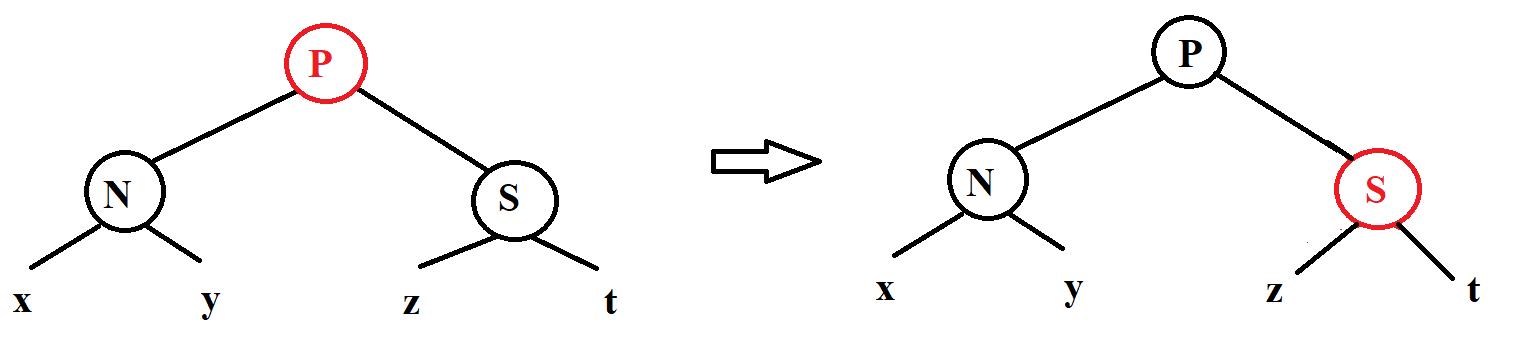
-Go back to step 1 with n being its parent node.



* Step 4: n is black, s is black, children of s is black, p is red.

-Change node p to black and node s to red.

-End the process of editing the tree when deleting here. Otherwise, if we do not fall into this step, we will go to step 5.

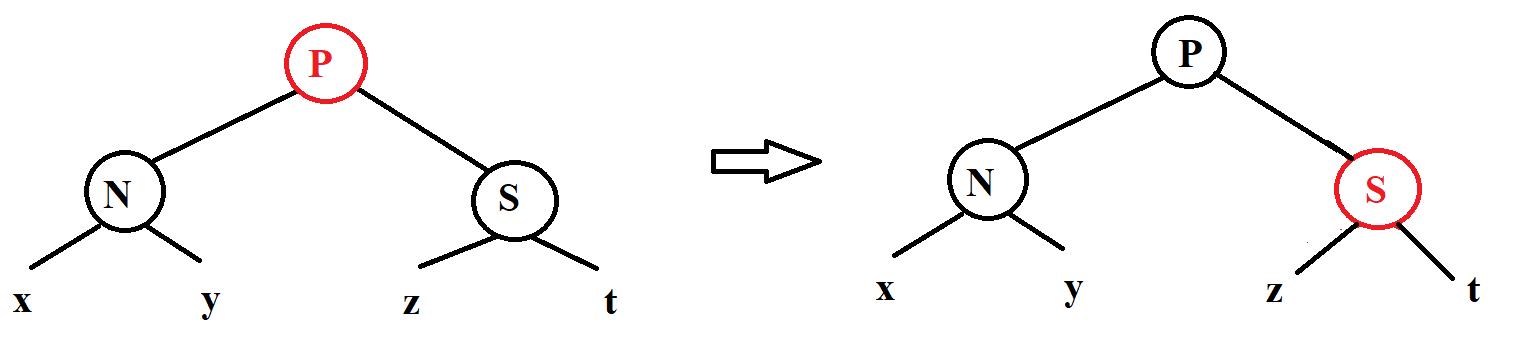


* Step 5: n is black, s is black, left child of s is red, right child of s is black.

-We change the color of node s and the left child of node s.

-We turn right at node s.

-Go to step 6.



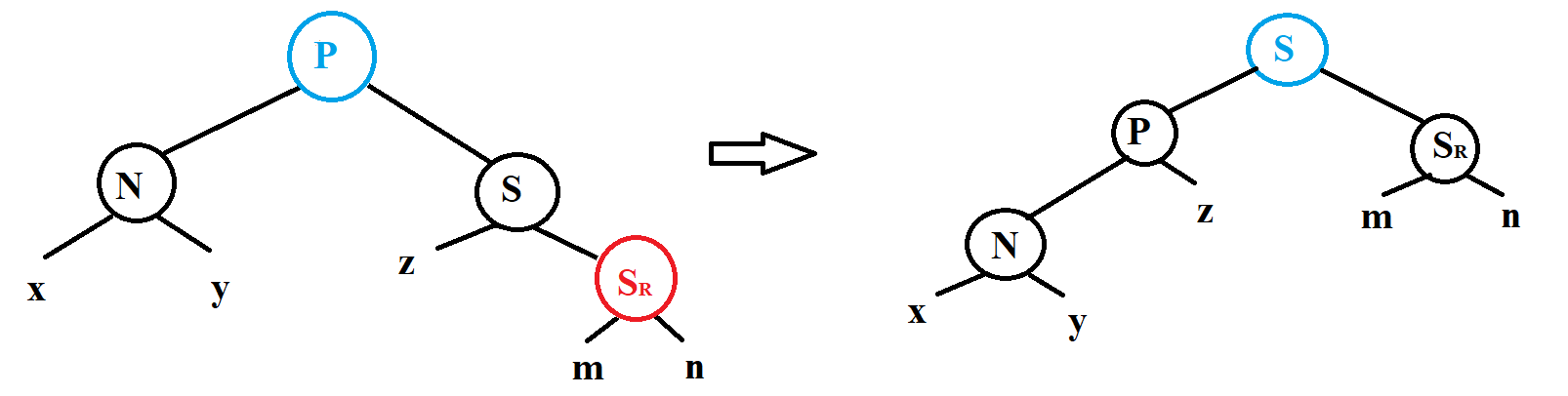
* Step 6: n is black, s is black, right child of s is red.

-We change the color of node p and s for each other.

-We change the color of the right child node of node s to black.

-Turn left at node p.

-We end the tree editing process here.



void fixerase(node \*x)

{

while (x != root && x->color == BLACK)

{

if (x == x->parent->left)

{

node\* w = x->parent->right;

if (w->color == RED)

{

w->color = BLACK;

x->parent->color = RED;

rotateLeft (x->parent);

w = x->parent->right

}

if (w->left->color == BLACK && w->right->color == BLACK)

{

w->color = RED;

x = x->parent;

}

else

{

if (w->right->color == BLACK)

{

w->left->color = BLACK;

w->color = RED;

rotateRight(w);

w = x->parent->right;

}

w->color = x->parent->color;

x->parent->color = BLACK;

w->right->color = BLACK;

rotateLeft(x->parent);

x = root;

}

}

else

{

node \*w = x->parent->left;

if (w->color == RED)

{

w->color = BLACK;

x->parent->color = RED;

rotateRight (x->parent);

w = x->parent->left;

}

if (w->right->color == BLACK && w->left->color == BLACK)

{

w->color = RED;

x = x->parent;

}

else

{

if (w->left->color == BLACK)

{

w->right->color = BLACK;

w->color = RED;

rotateLeft (w);

w = x->parent->left;

}

w->color = x->parent->color;

x->parent->color = BLACK;

w->left->color = BLACK;

rotateRight (x->parent);

x = root;

}

}

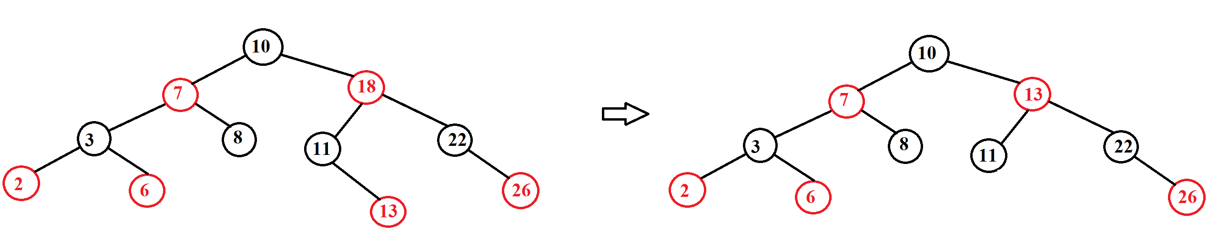
}

x->color = BLACK;

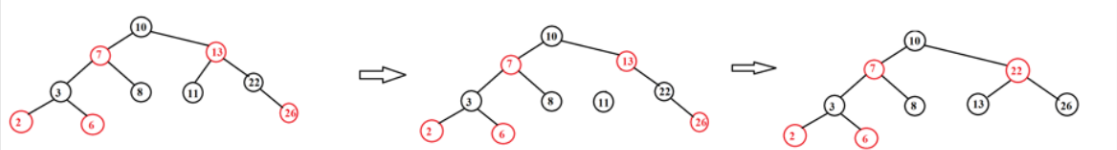
}

Example: We have an RB tree 7, 3, 18, 10, 22, 8, 11, 26, 2, 6, 13 . We will delete the elements 18, 11.

+Delete 18:



+Delete 11



-Complexity and tree height:

+ Complexity:

. Search O()

. Insert O()

. Delete O()

. Minimum O()

. Maximum O()

***\*Illustration:***

*Example 1:*

* **INPUT:**

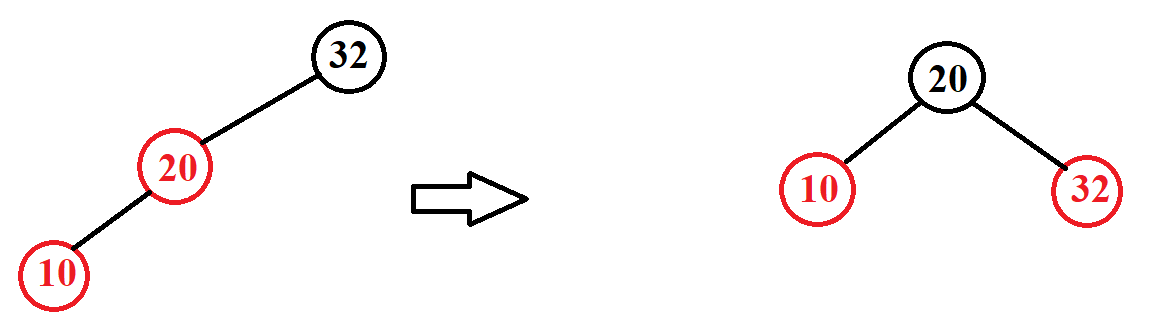
-Insert:32, 20, 10, 15, 40, 25, 27, 26, 13, 14.

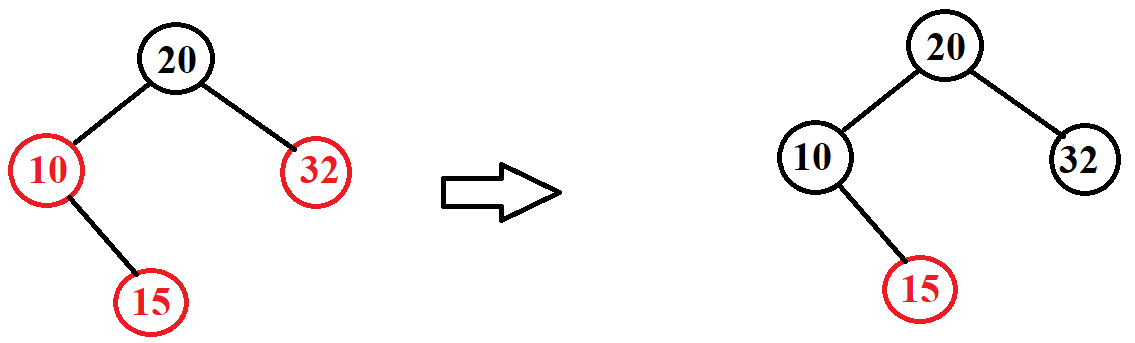
-Find: 15, 29

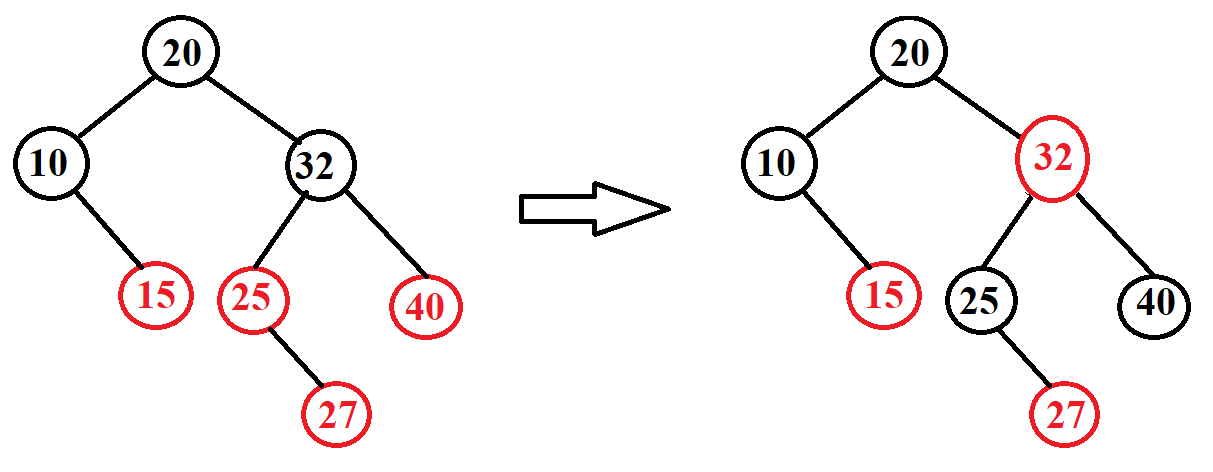
-Delete: 25, 40, 20

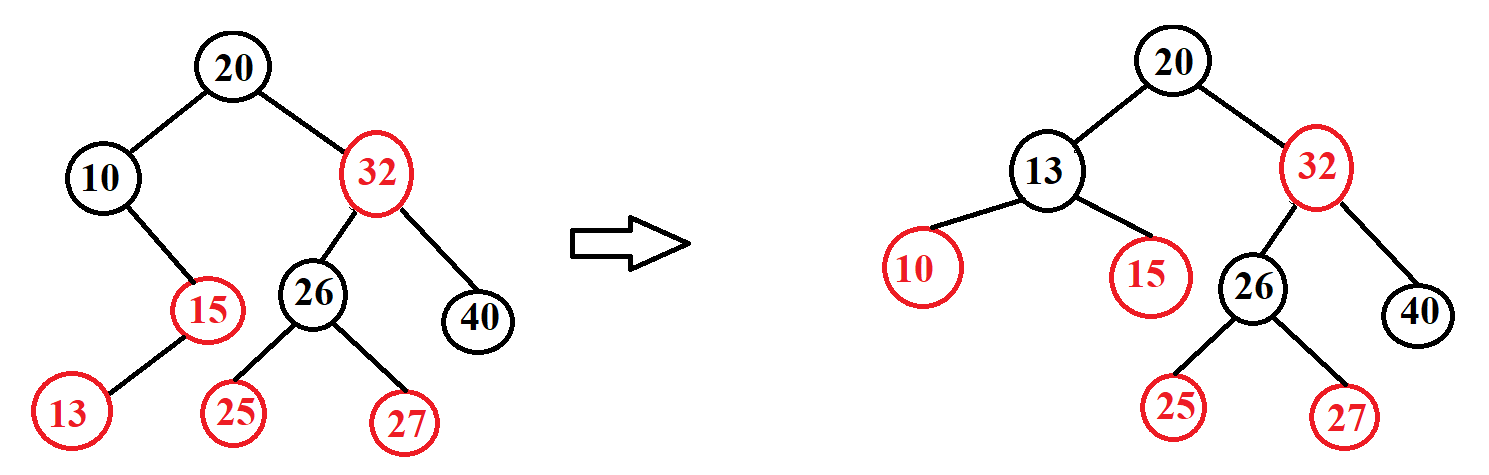
* **OUTPUT:**

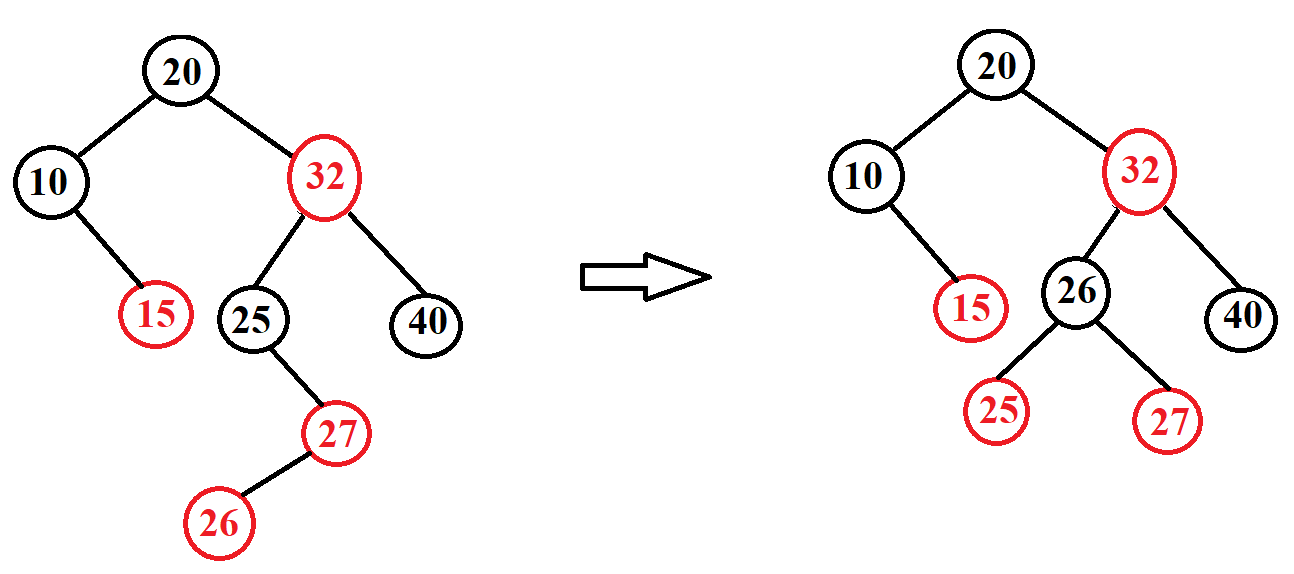
-Insert:

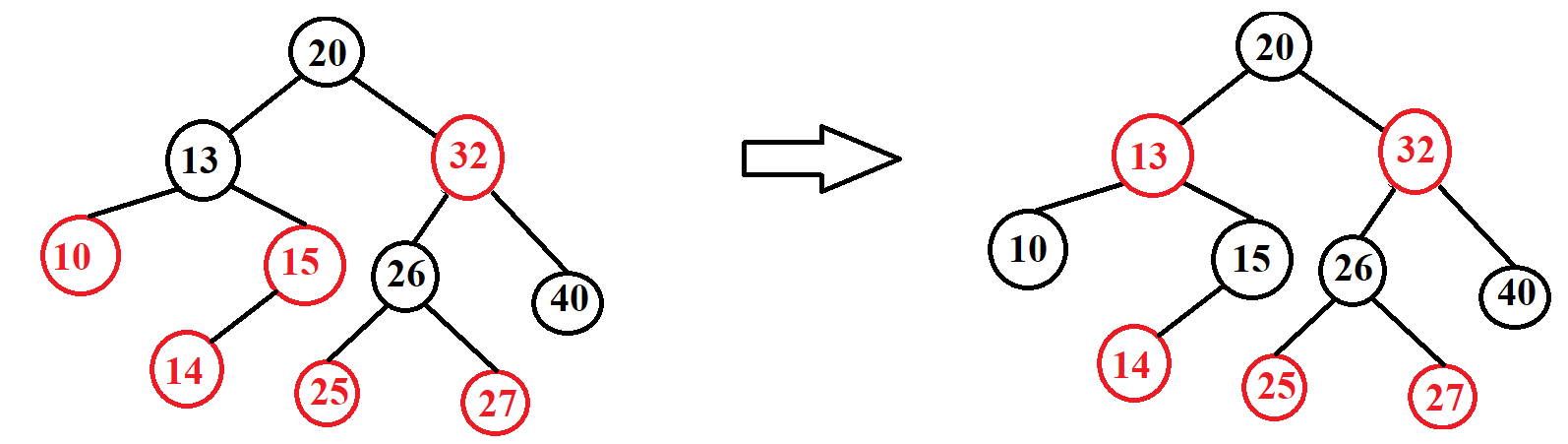






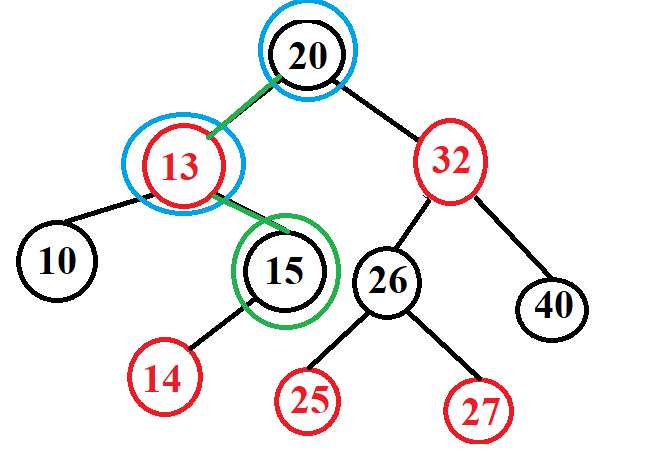




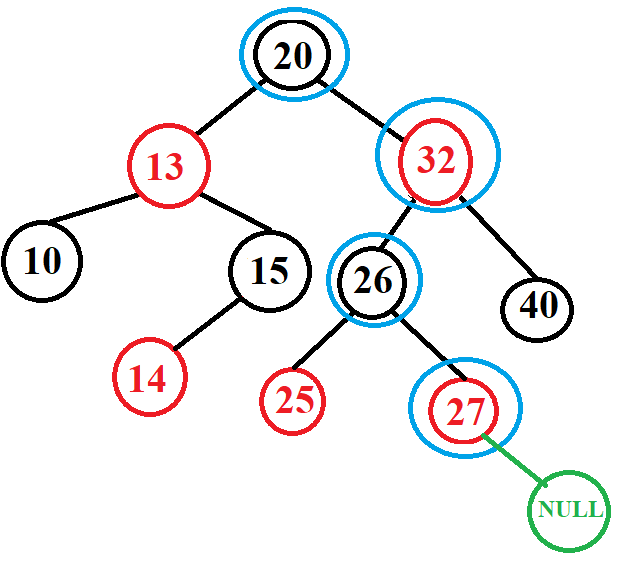


-Find:

+Find 15

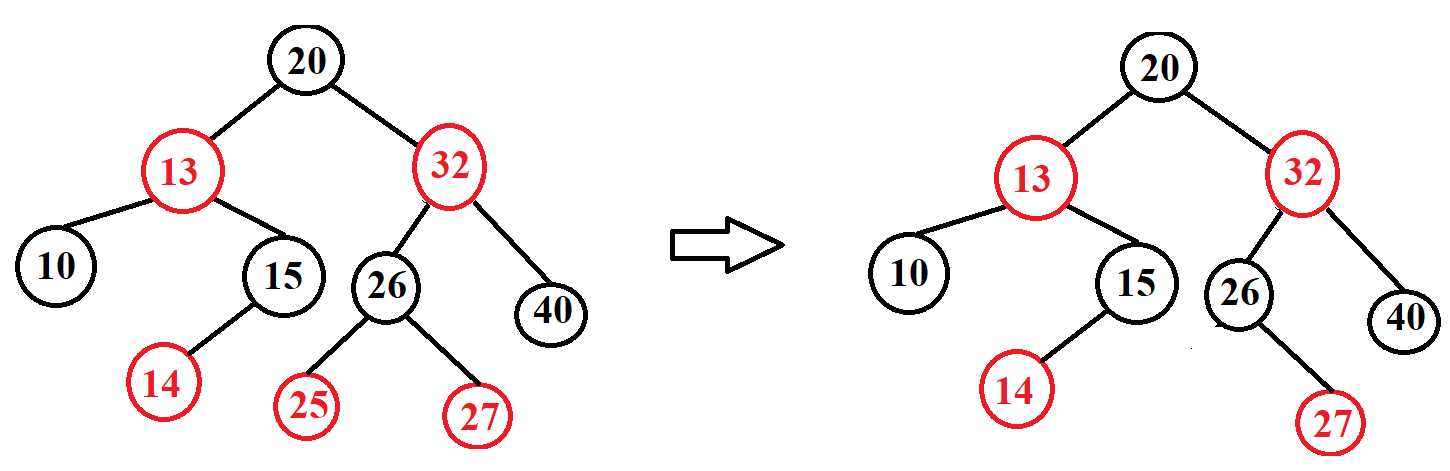


+Find 29

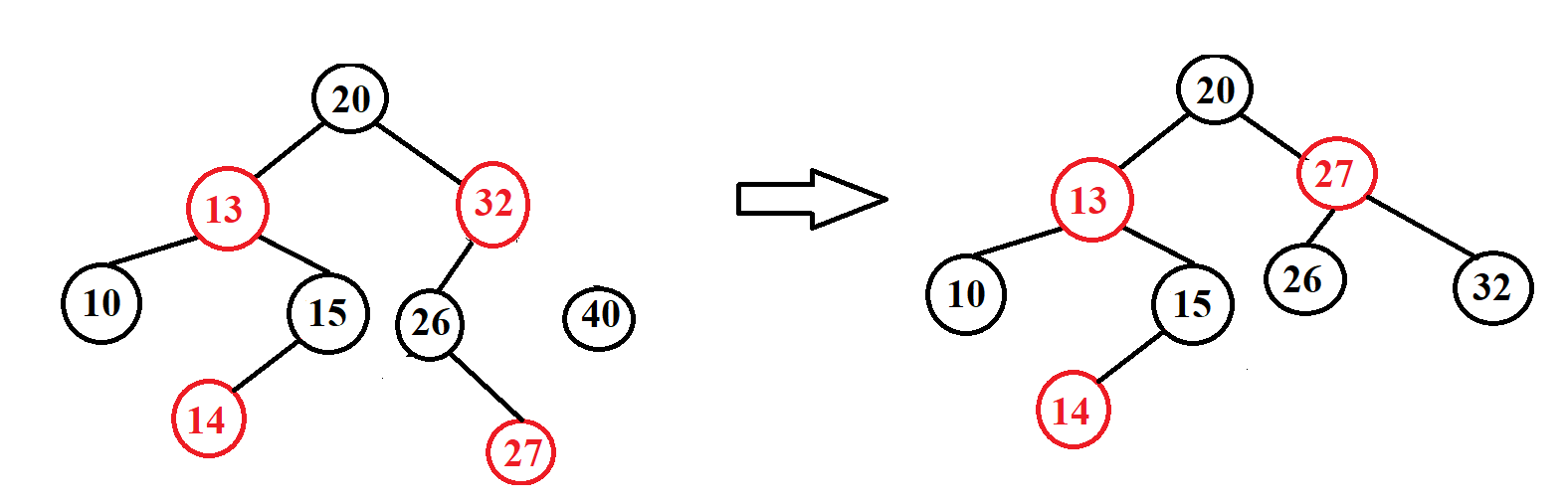


-Delete:

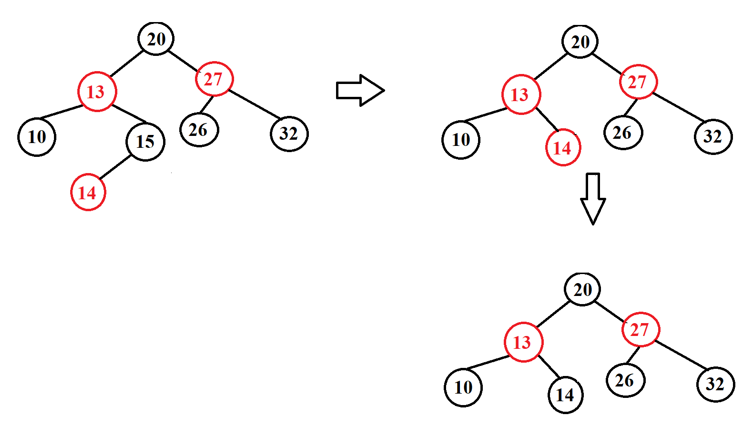
+Delete 25.



+Delete 40.



+Delete 20.



*Example 2:*

* **INPUT:**

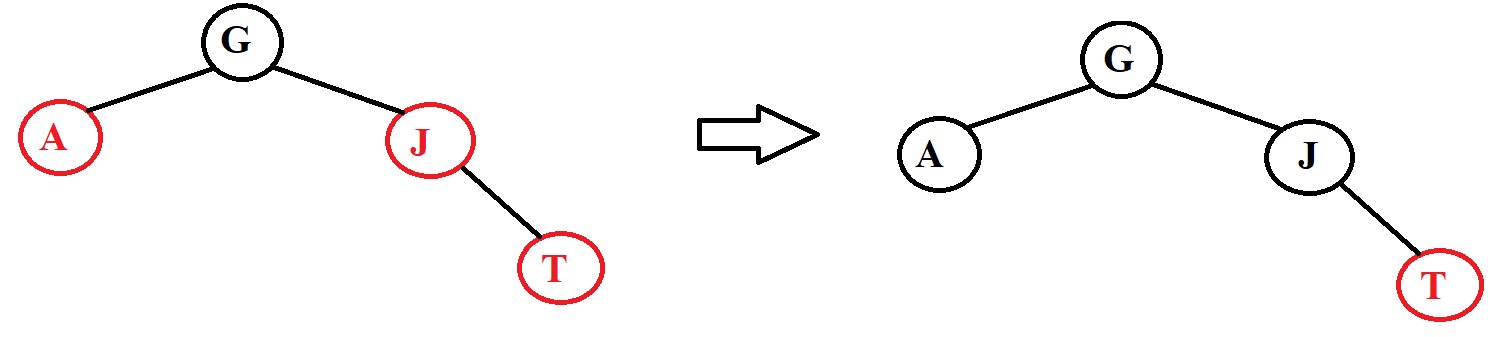
-Insert: G, J, A, T, B, W, C, K.

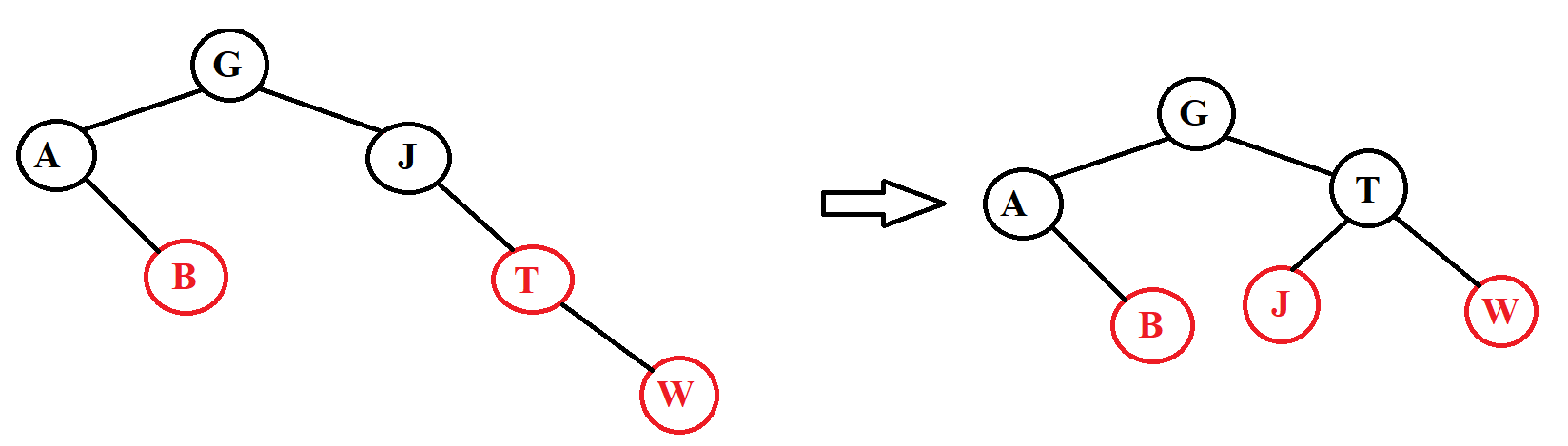
-Find: A, Z

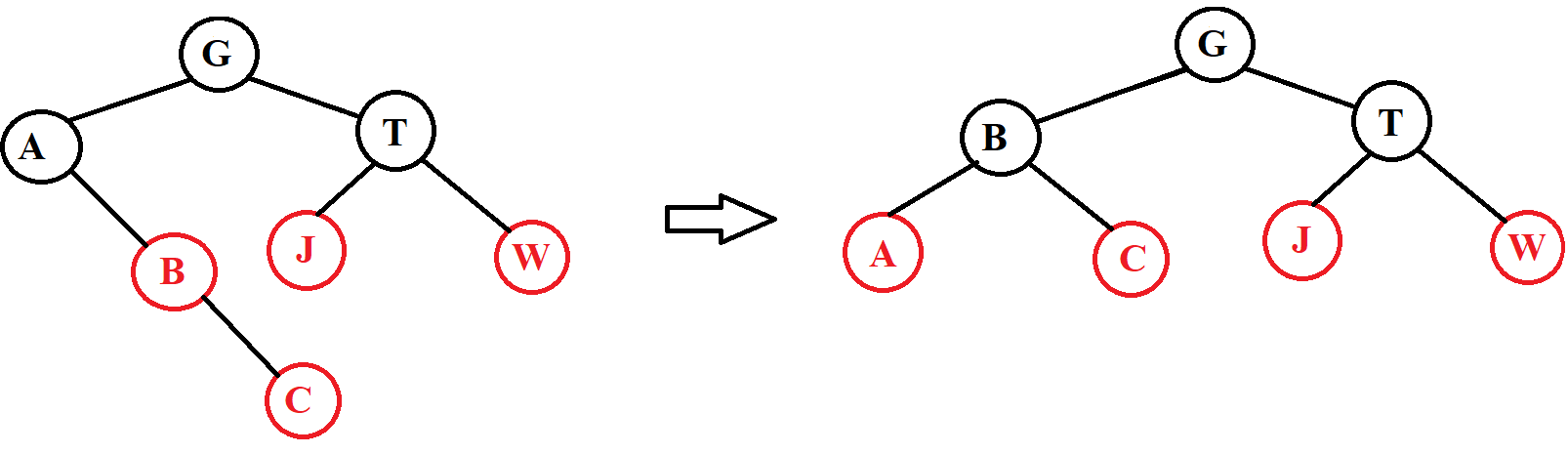
-Delete: J, A

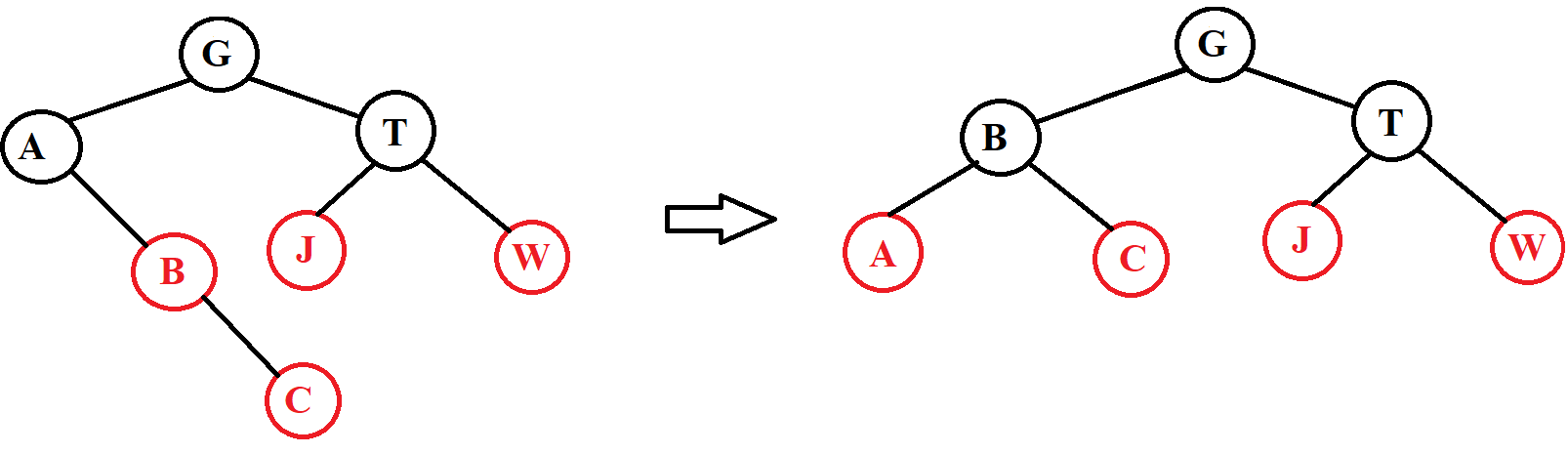
* **OUTPUT:**

-Insert:



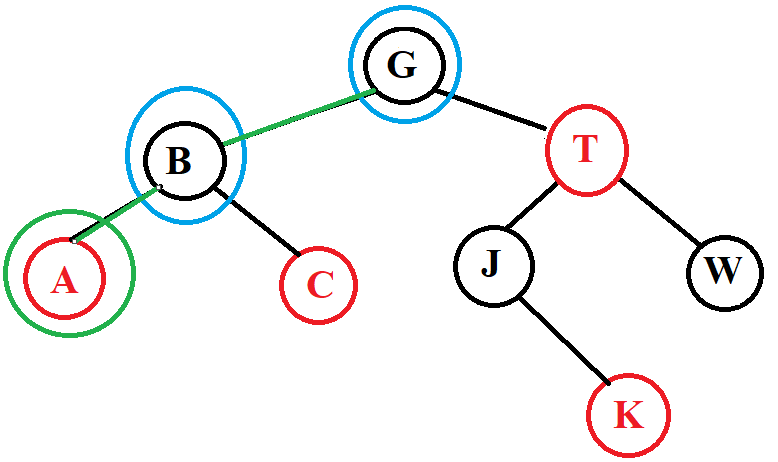




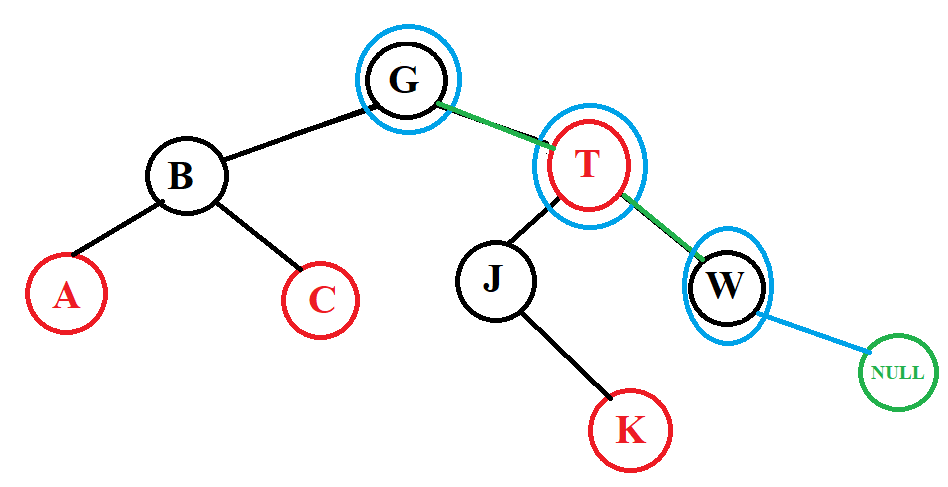


-Find:

+Find A

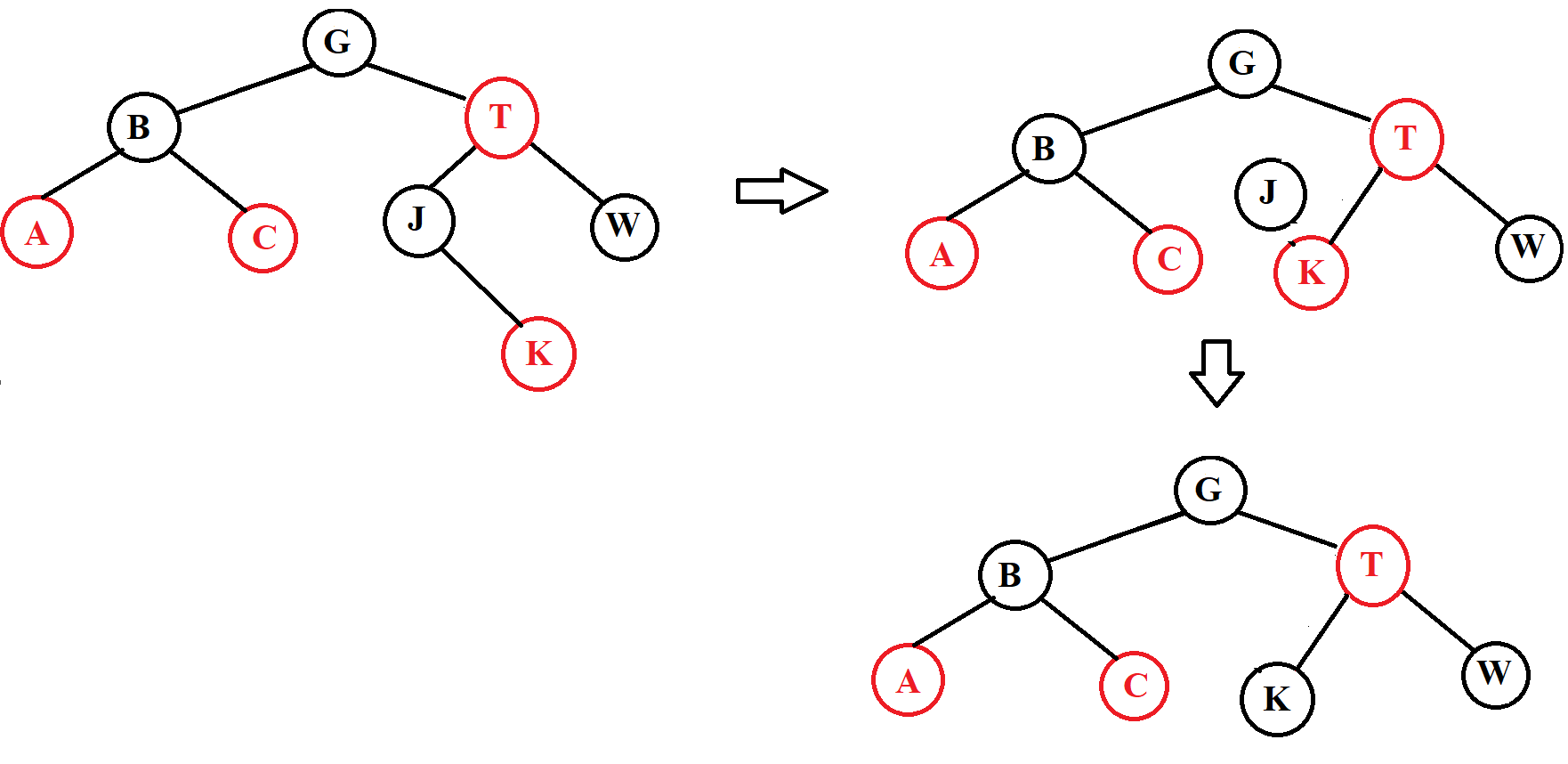


+Find Z

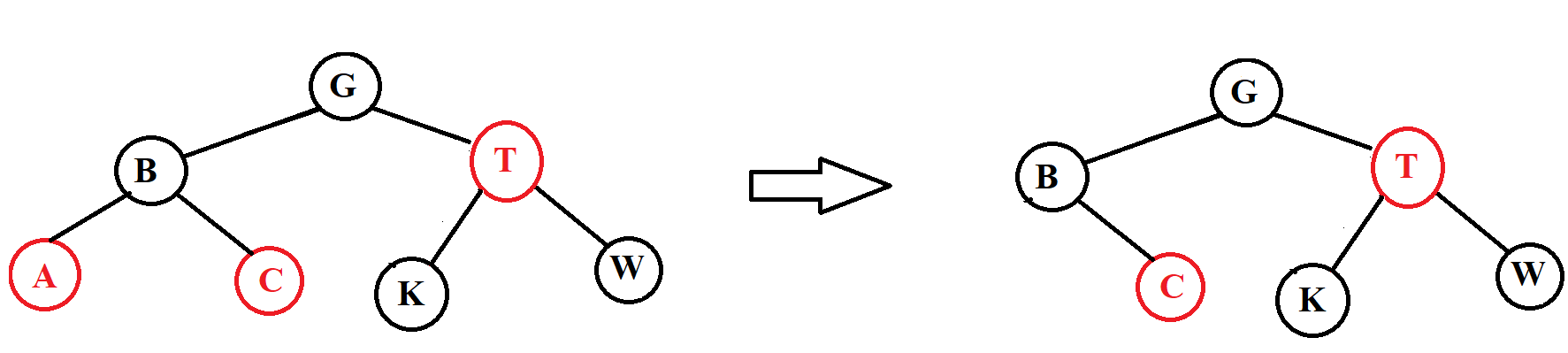


-Delete:

+Delete J:



+Delete A



*Example 3:*

* **INPUT:** input is a data pair of the form (x,y) where x is an integer and y is a letter. We will compare the number first if the number is equal, we compare the letter.

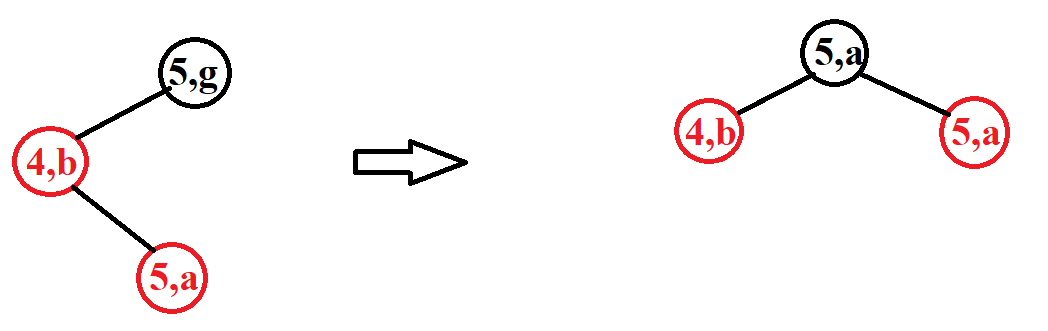
**-**Insert: (5,g), (4,b), (5,a), (9,e), (2,p), (1,z)

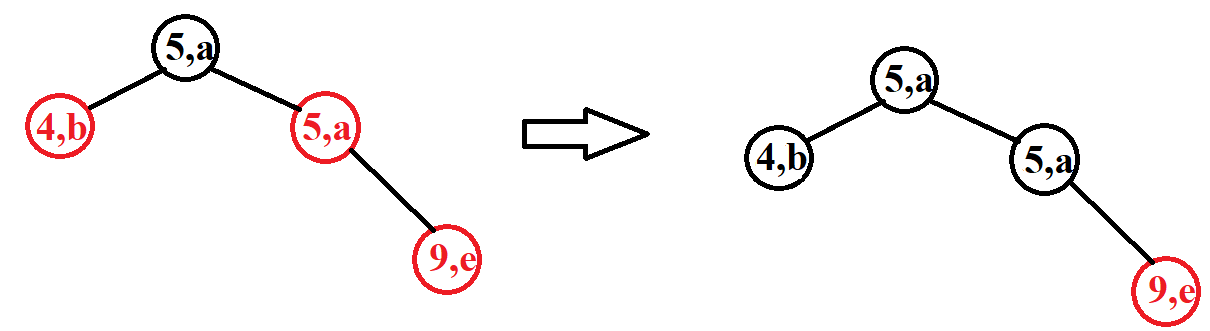
**-**Find: (9,e), (7,a)

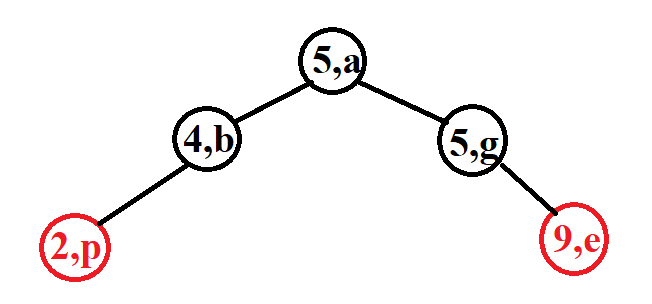
**-**Delete: (5,g), (2,p)

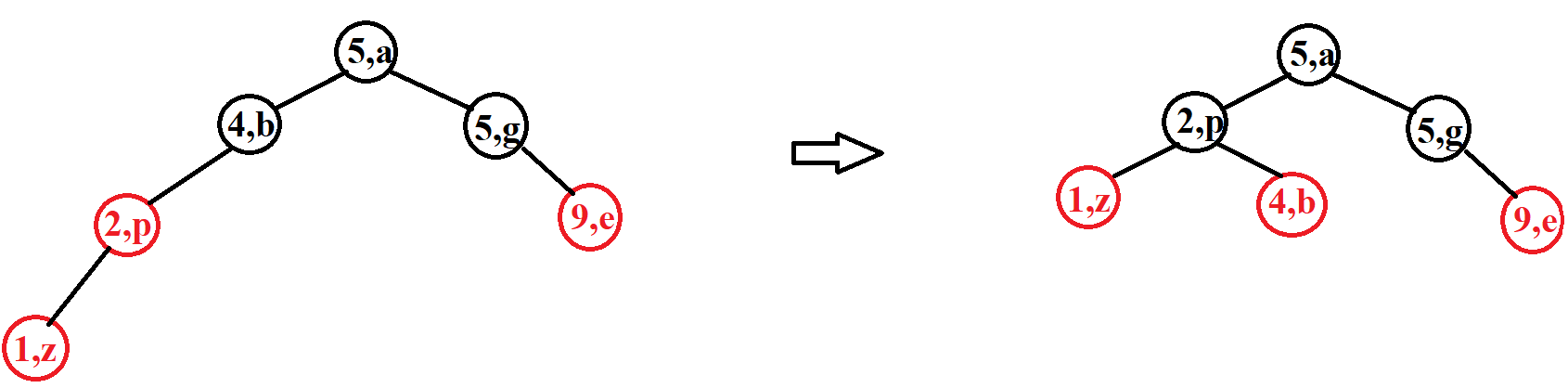
* **OUTPUT:**

-Insert:



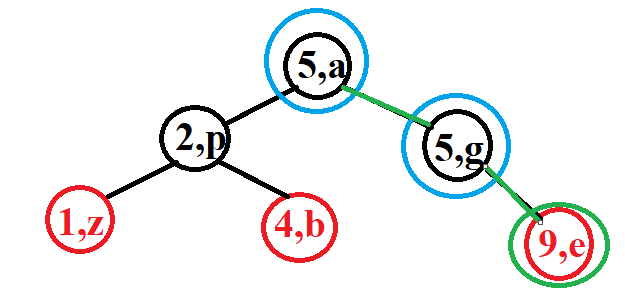




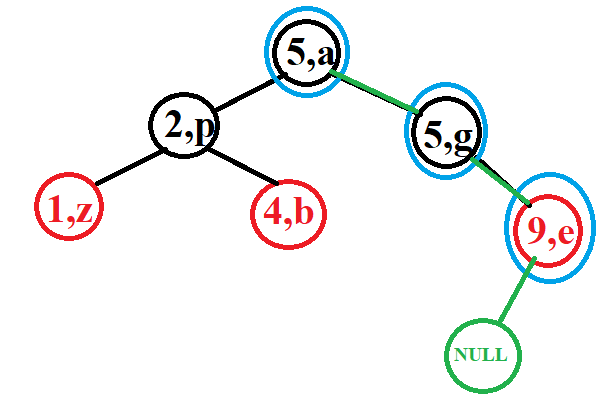


-Find:

+Find (9,e)

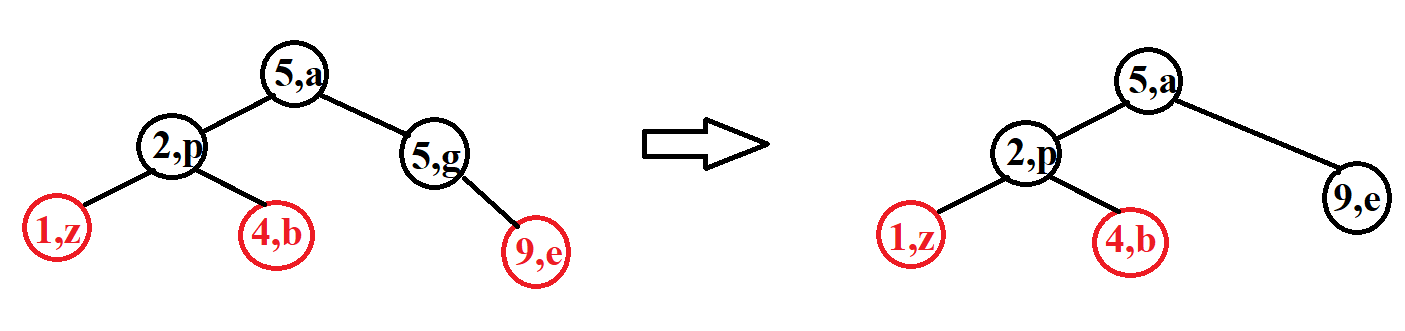


+Find (7,e)

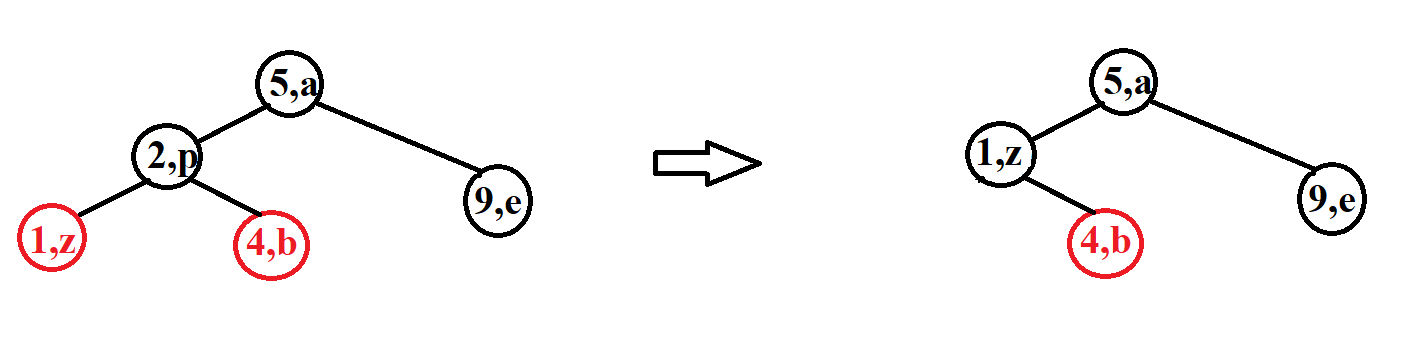


-Delete:

+Delete (5,g)



+Delete (2,p)



**IV.Compare AVL and RB tree:**

- Search operations of AVL trees are faster than those of RB trees because they are more tightly balanced. Because it checks for it to correct the tree occurs with more frequency than AVL, for example in insert we see it will check 2 child nodes in a row so it will rotate more, leading to a tighter equalization tree.

- Red-black tree has faster insertion and deletion than AVL tree because RB tree has a tighter balance, so those two operations make RB tree faster than AVL.

- Storing in 1 node of AVL tree is less than RB tree because it will store 1 more data which is color.

- the rotations for an AVL tree are harder to implement and debug than that for a Red-Black tree.

- Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++.

**\*Reference source:**

**-**Geeksforgeeks.

-kc97ble.

-Lesson Slides.