**Binary Search Tree**

**Balanced Tree**

**AVL**

1. **Tree:**

-A tree <T> (Tree) is :

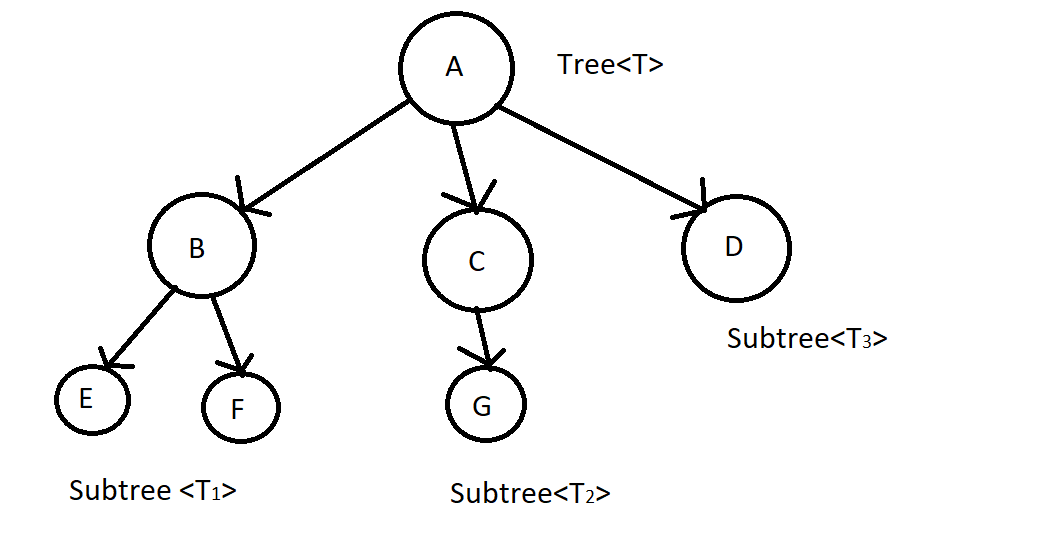
+A set of values ​​of the same type, elements of the same value are stored by a node, called node p1,p2,...

+ 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:

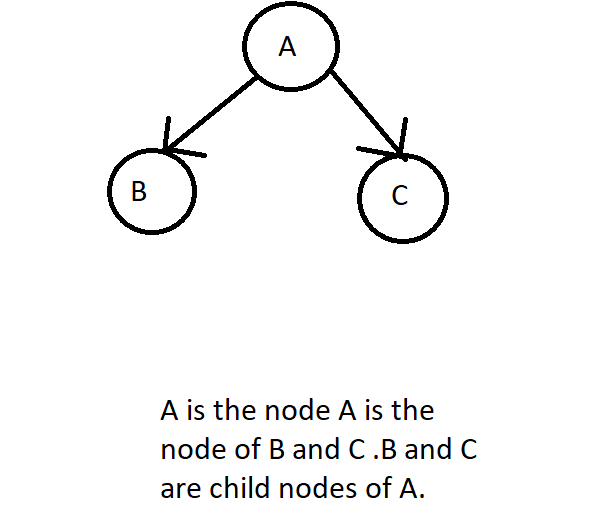


-Properties:

+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.

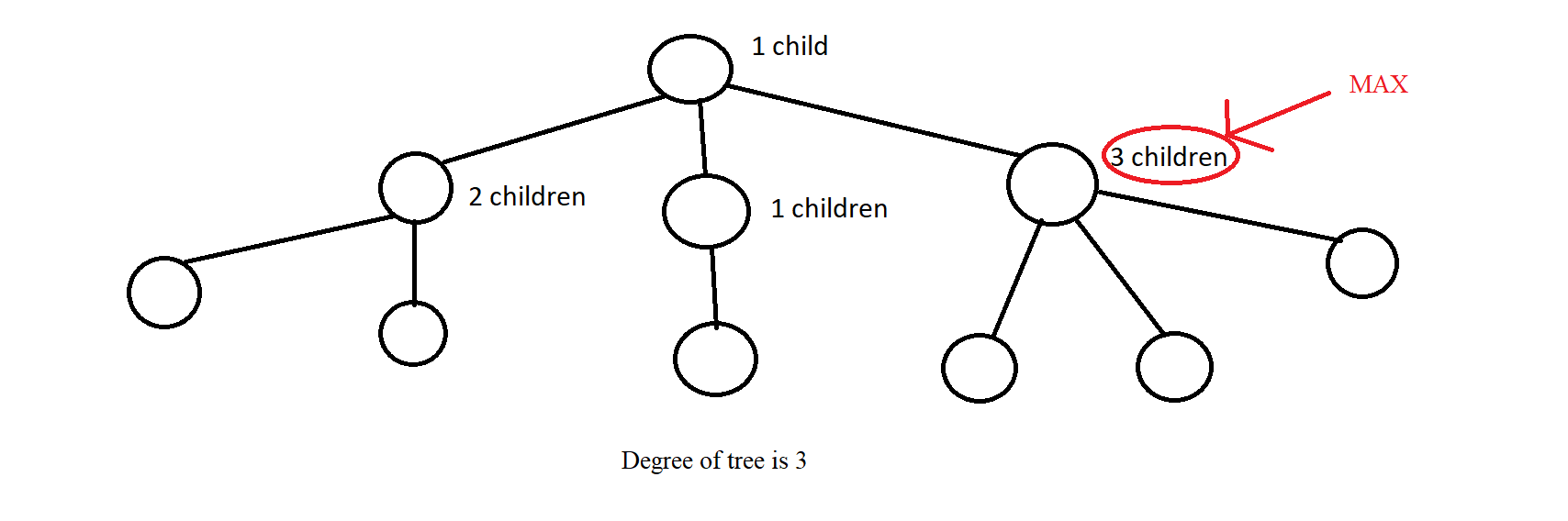
+Internal node: is a node that has both a parent node and a child node.

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

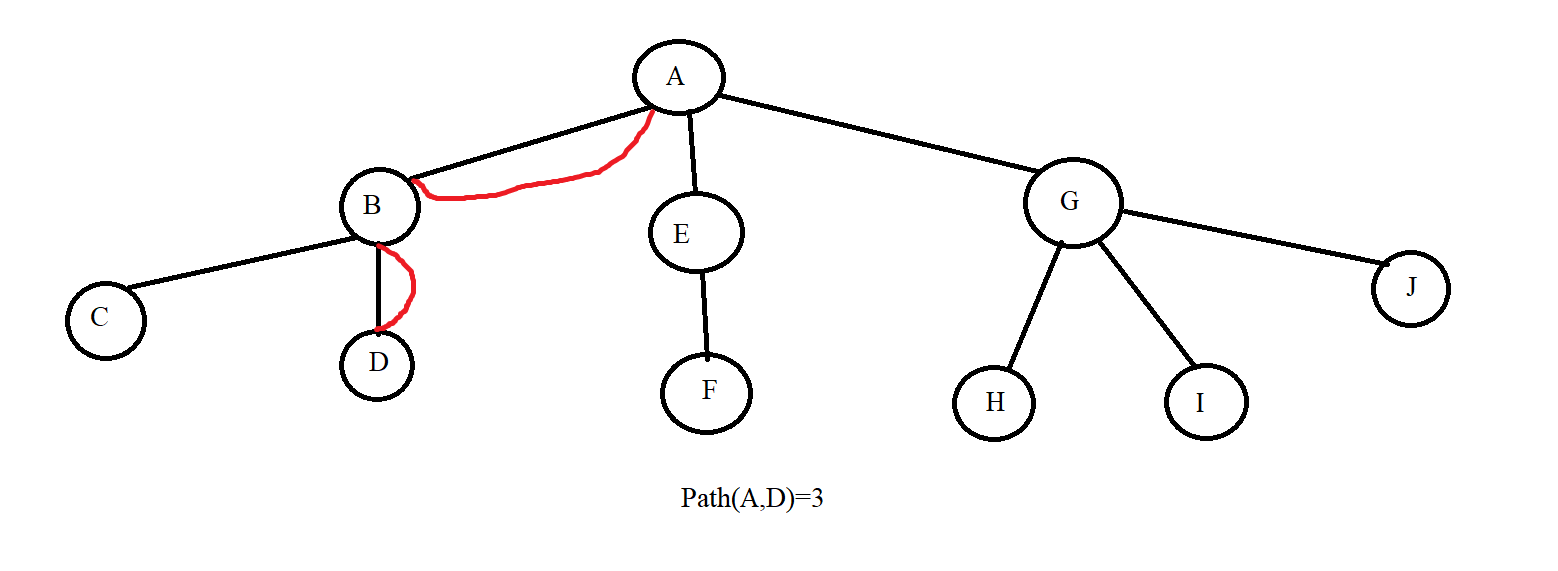
+Degree of tree: is the maximum value of the degree of node in the tree. When we interpret it with a formula, we can write:

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:

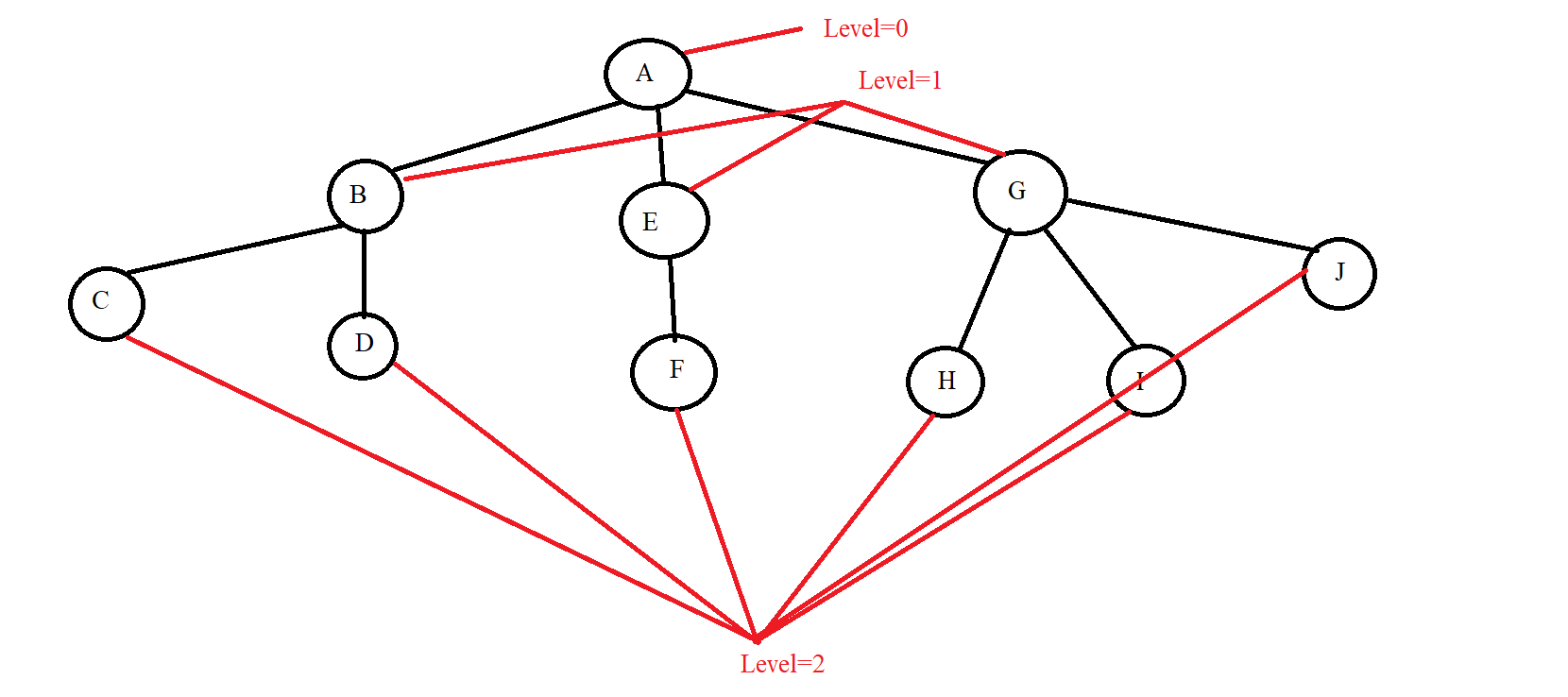


+Level: Just like how to calculate the floor of a whole tower, we apply it to trees as well. Where the ground floor is the root with level =0 and the upper floors are considered the bottom nodes of a node, the level is calculated as the parent node's level plus 1. From this we can deduce the formula:

Level(p) = 0 if p = root

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

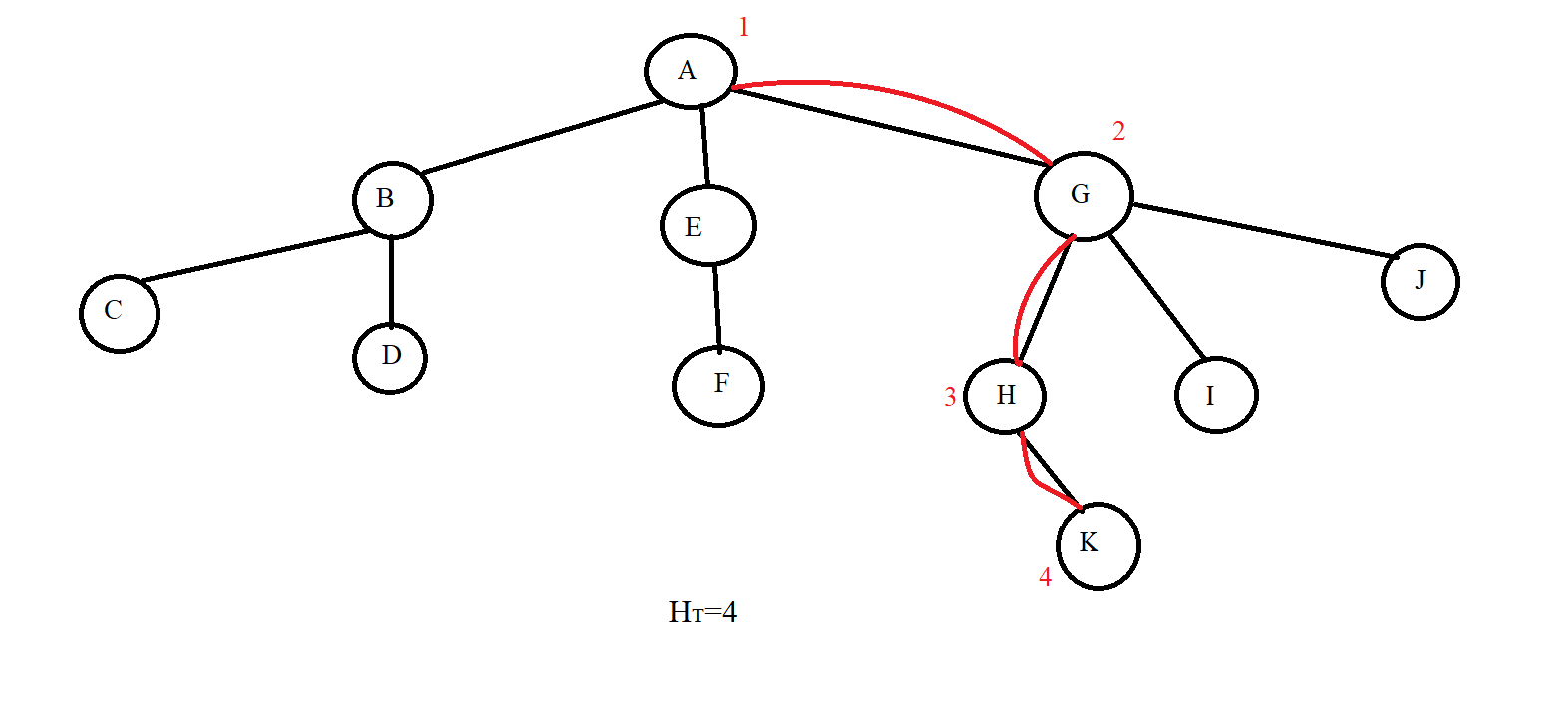
Example:



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

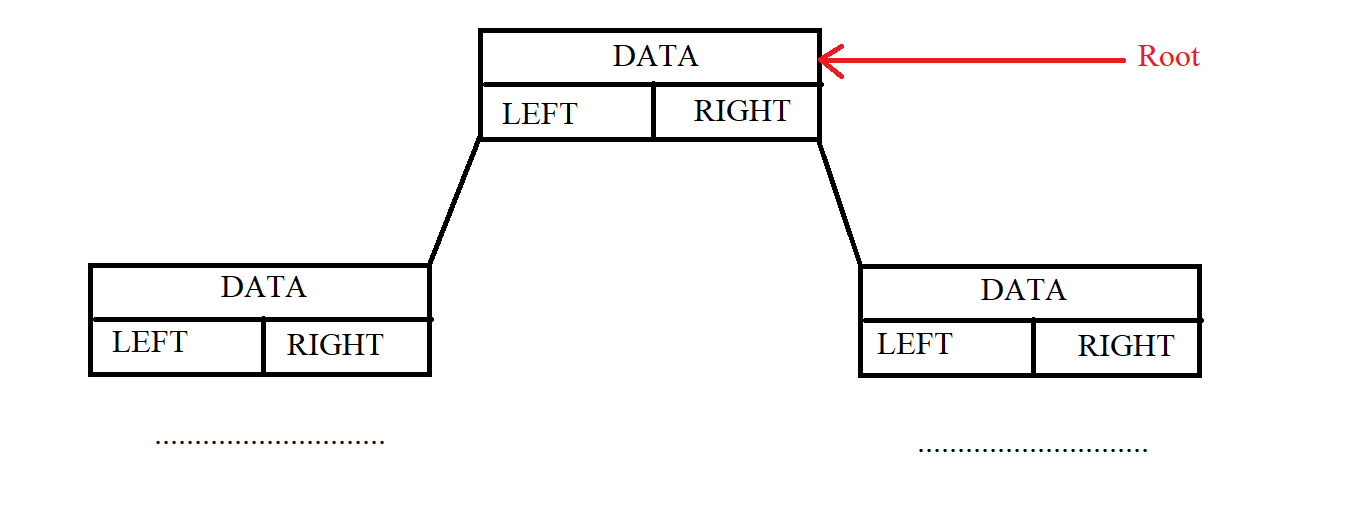
hT = max {Path(root, pi) | pi is the leaf node ∈ <T>}

Example:



1. **Binary tree :**

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

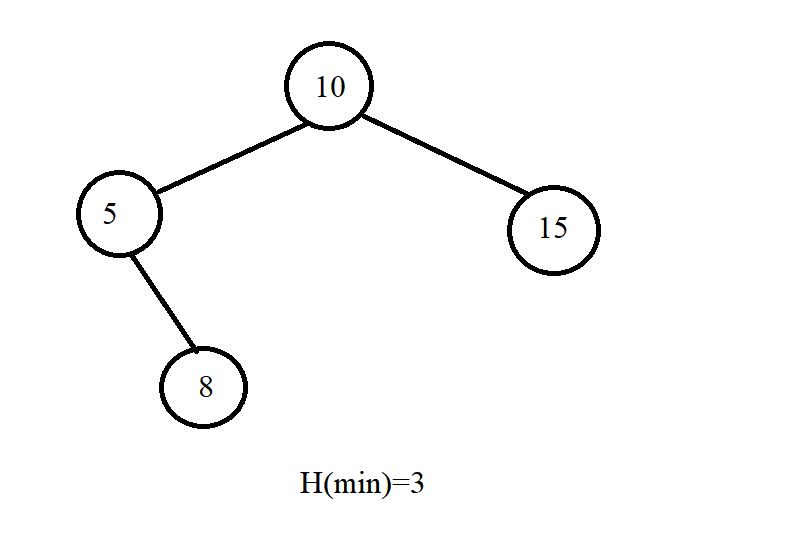


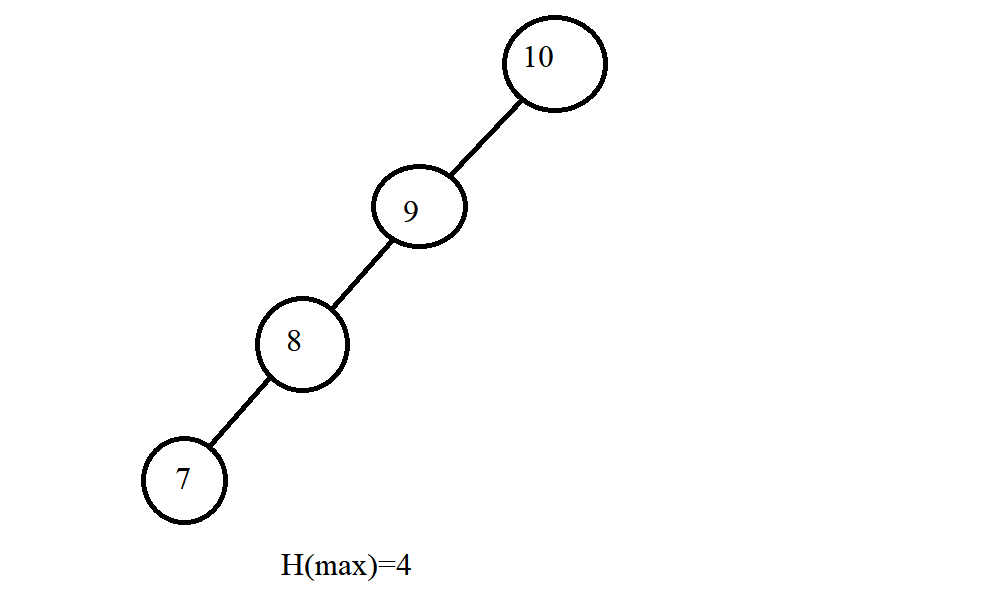
- The height of a binary tree has N nodes:

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.

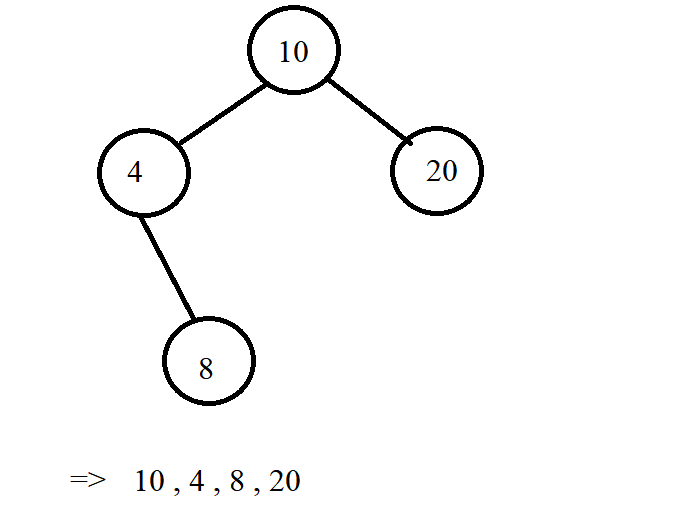


- Traverse in Tree: We have many different ways of traversing such as NLR, LRN, RNL, ... but we only consider 3 basic traversals: NLR, LRN and LRN.

+ 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



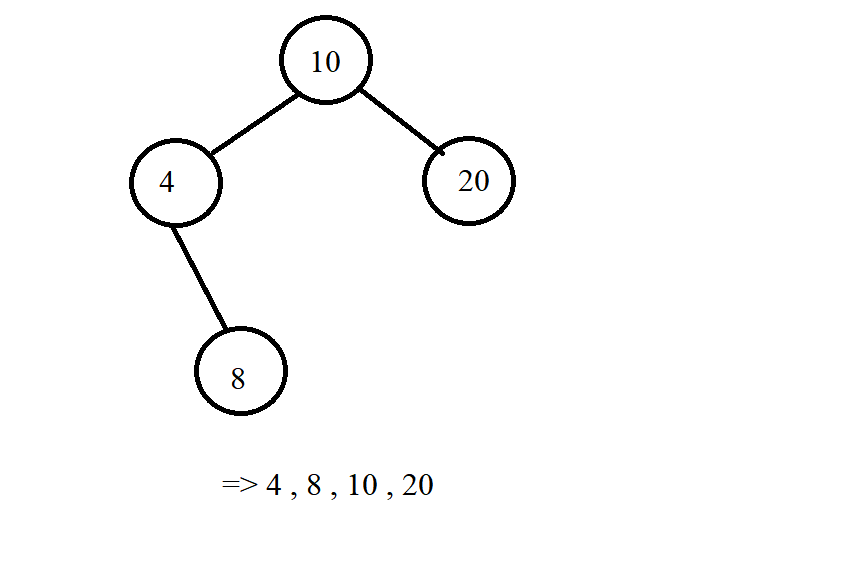
Example:



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



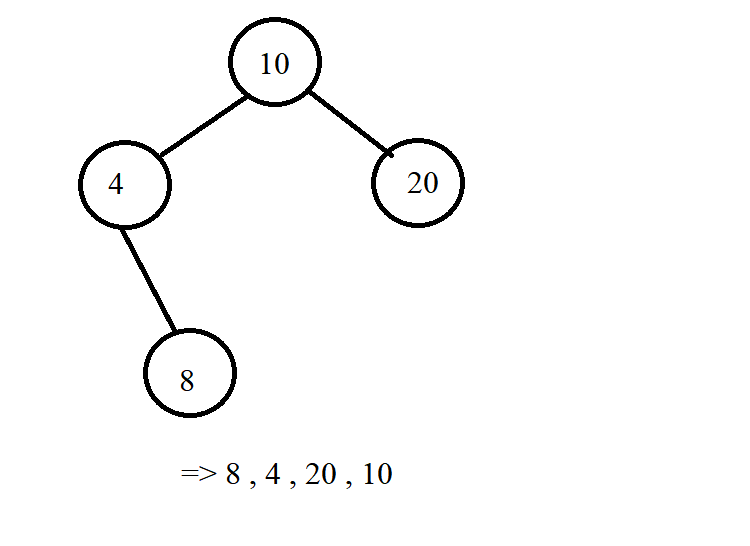
Example:



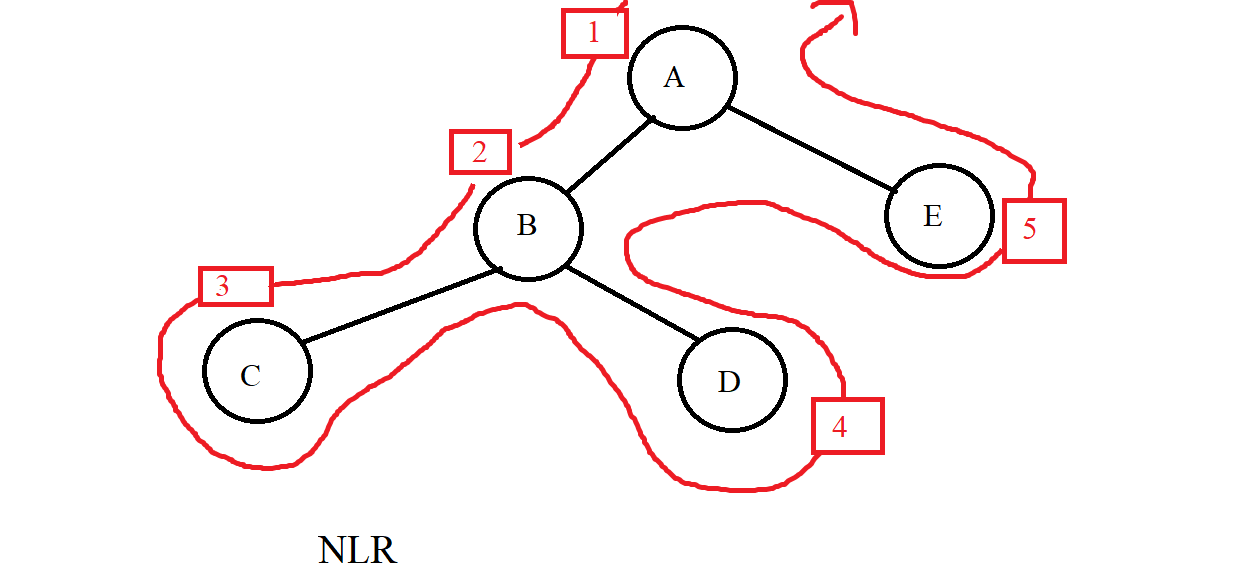
+Post-Order (LRN): We will recursively call the left node, go to the right node and finally process the current node.

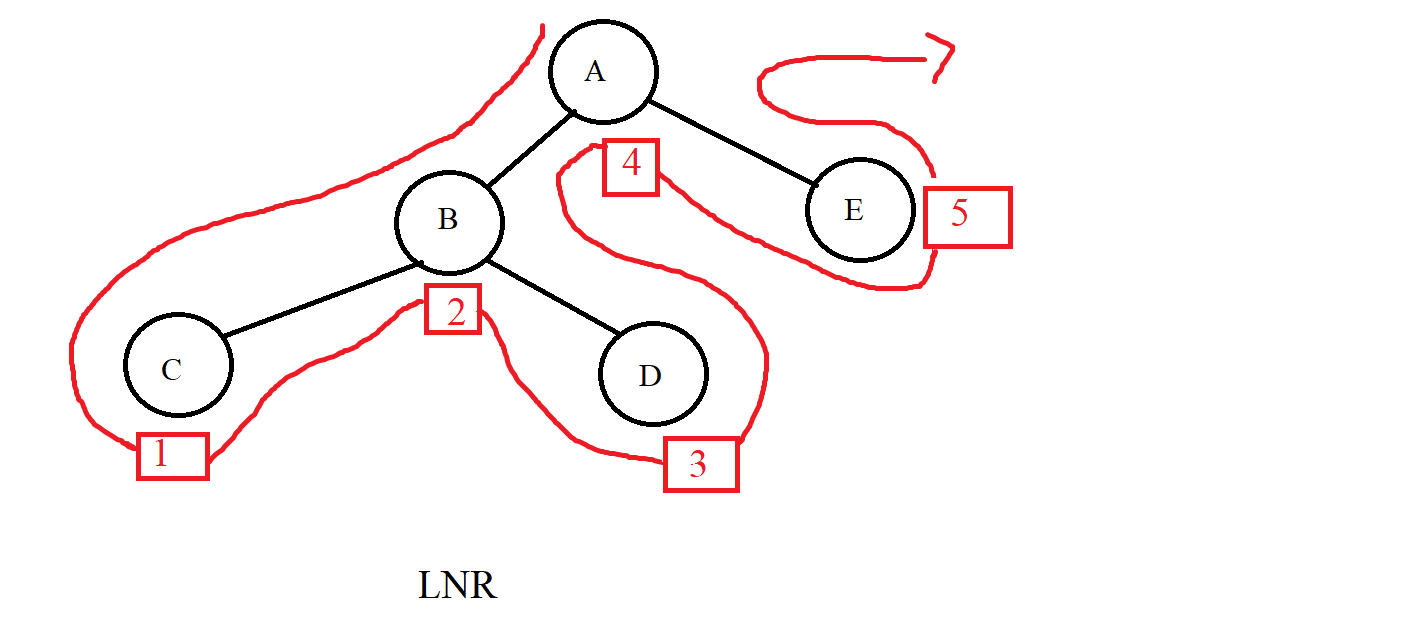


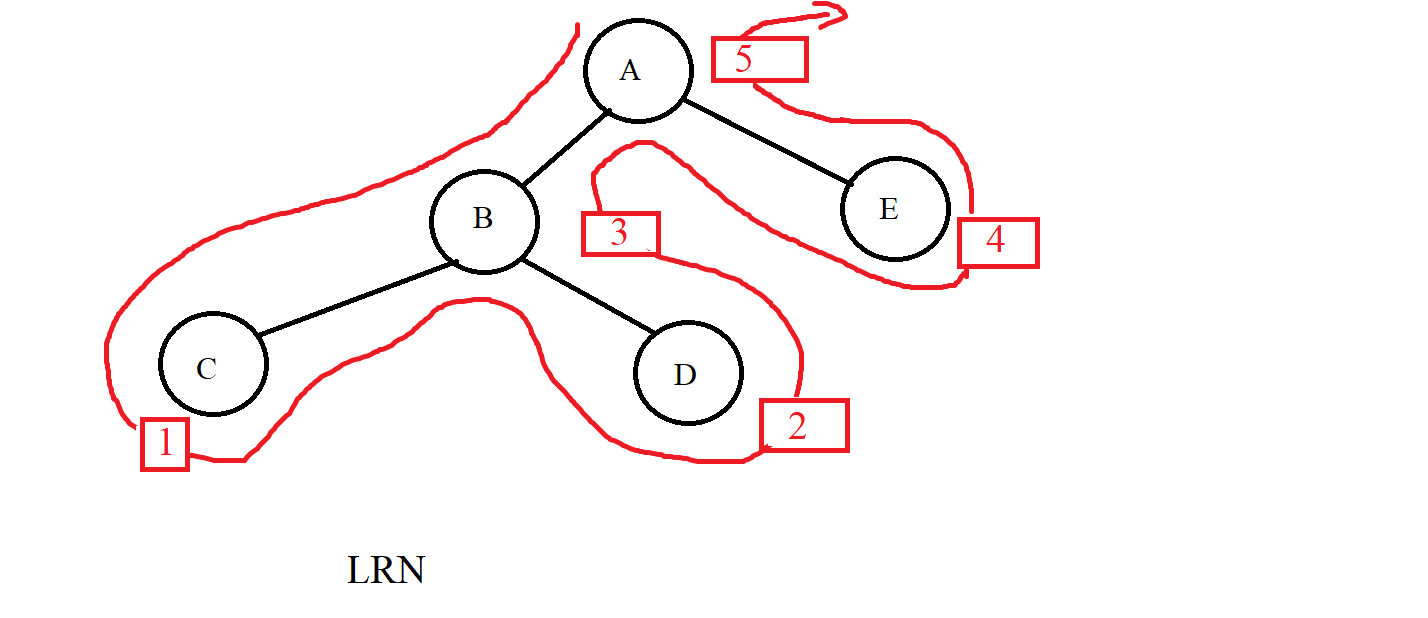
Example:



*\*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.







1. **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.

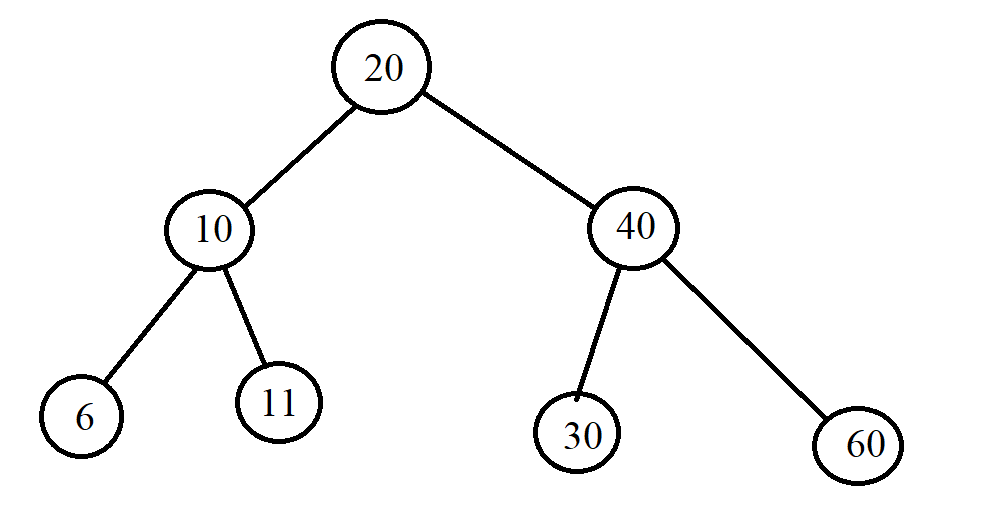
. All nodes of the right subtree must be less than the current node.

. 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:

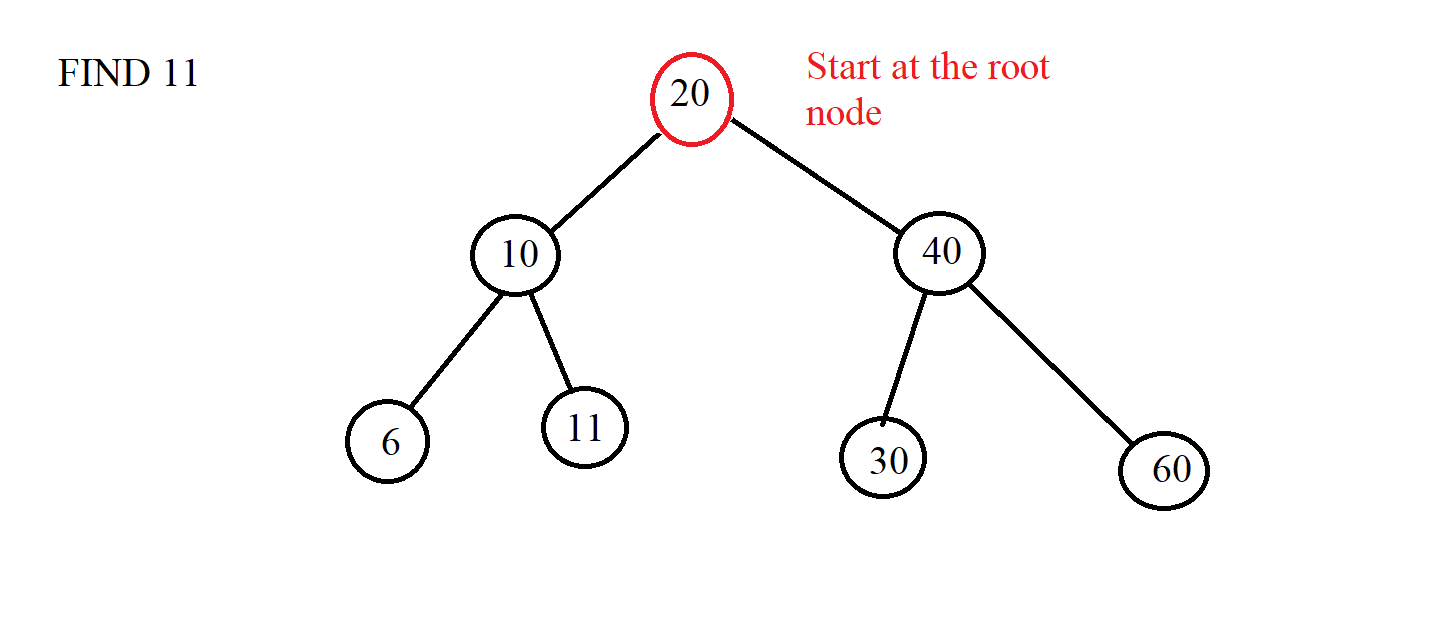
+ Create a empty tree.

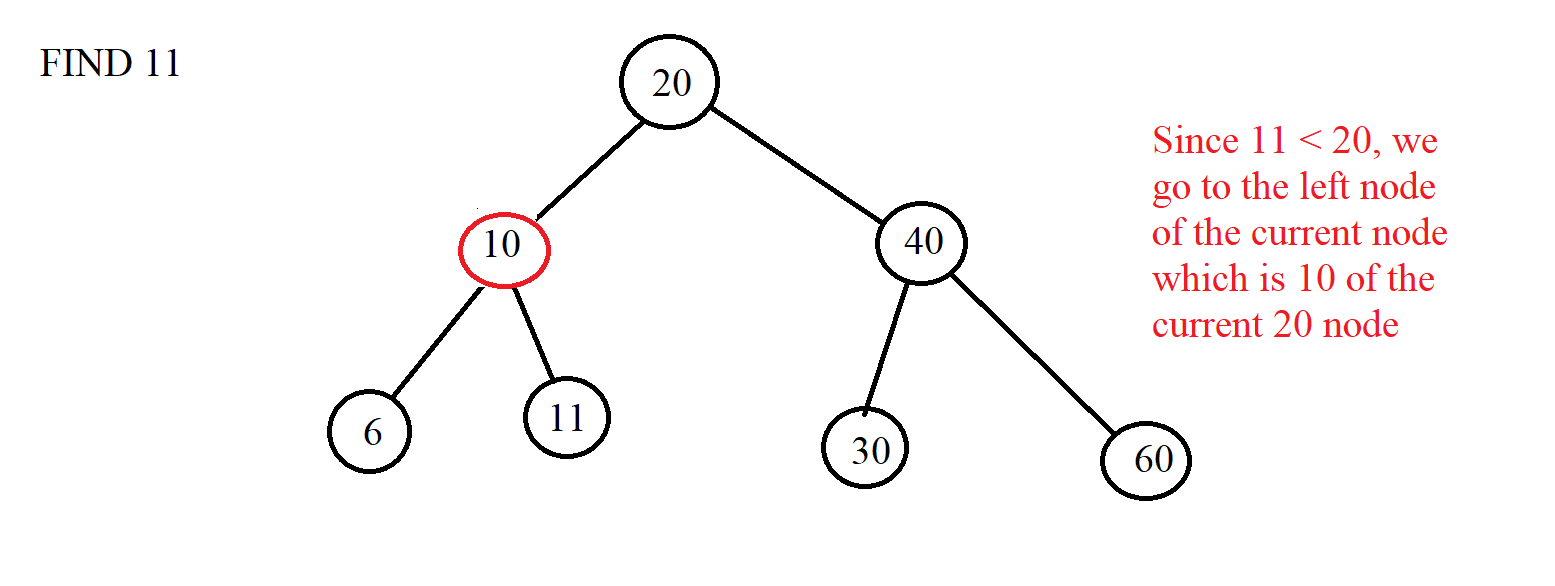


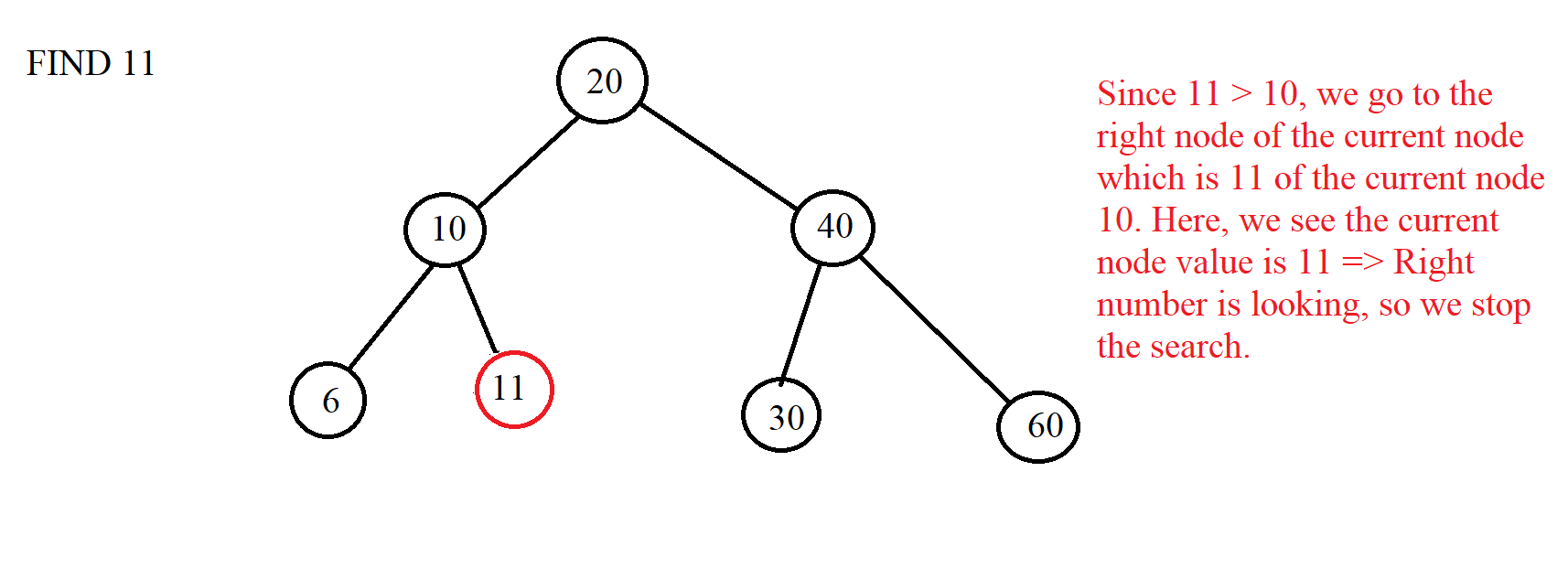
+ Check the empty tree.



+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:

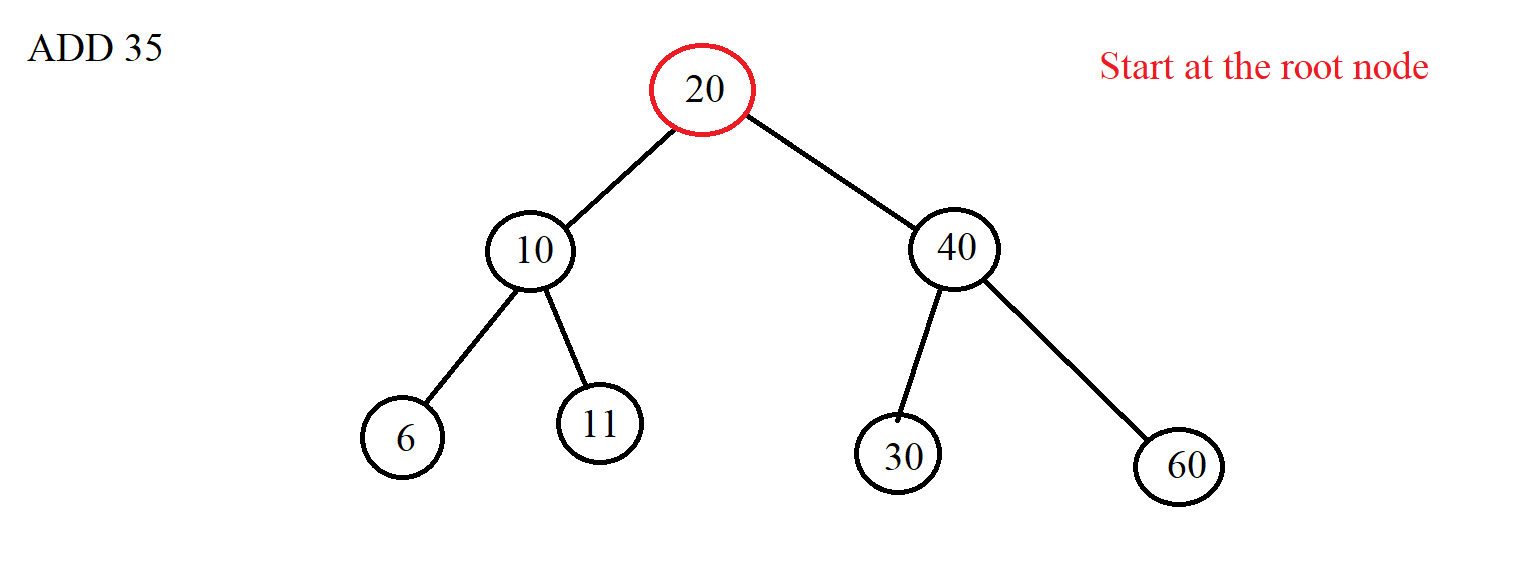


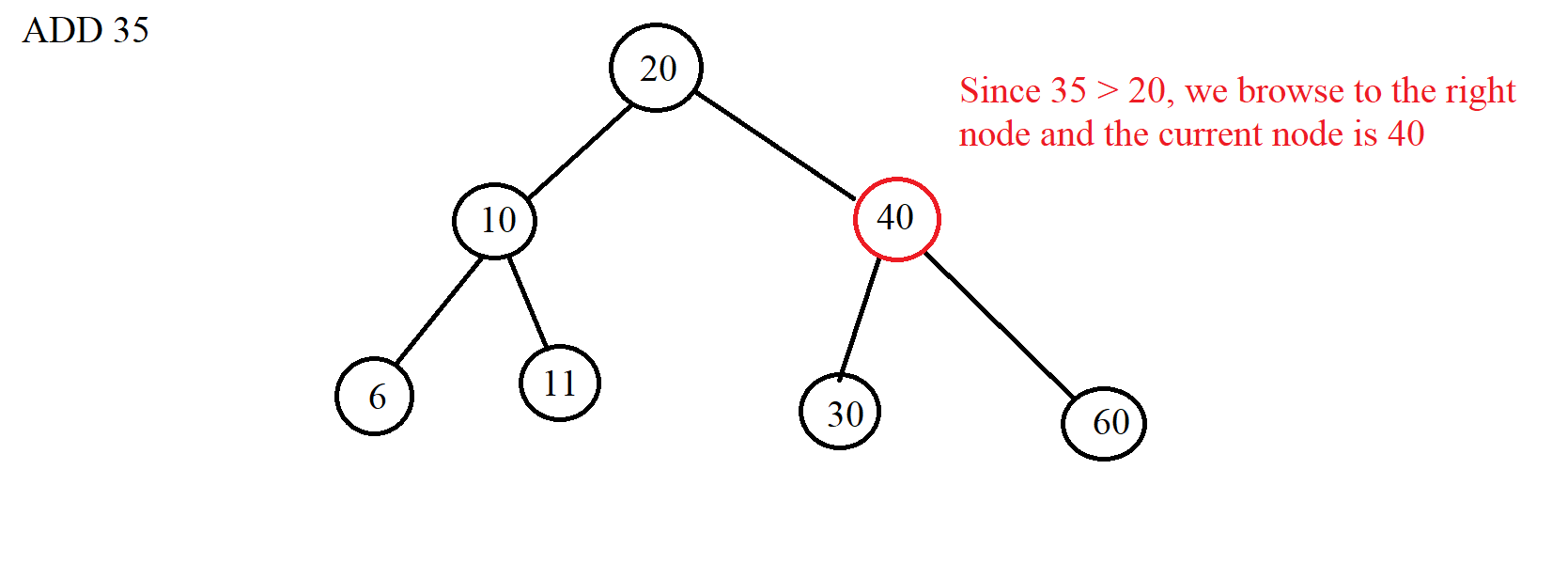


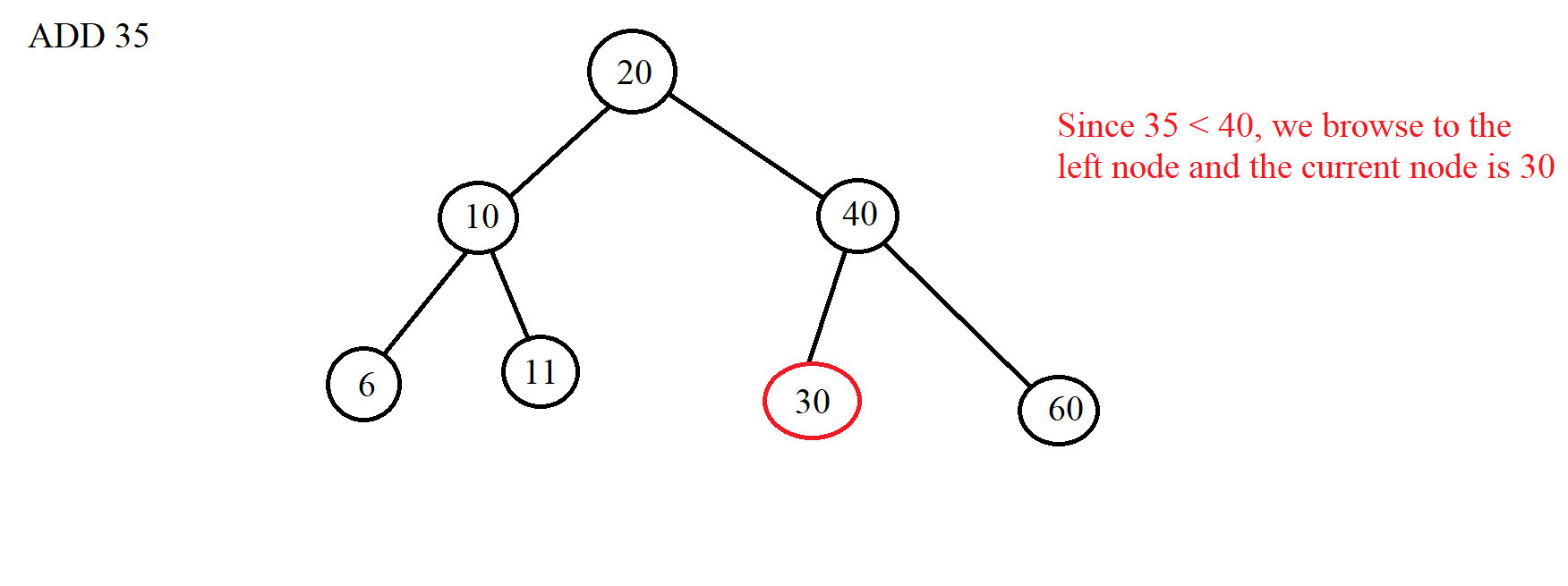


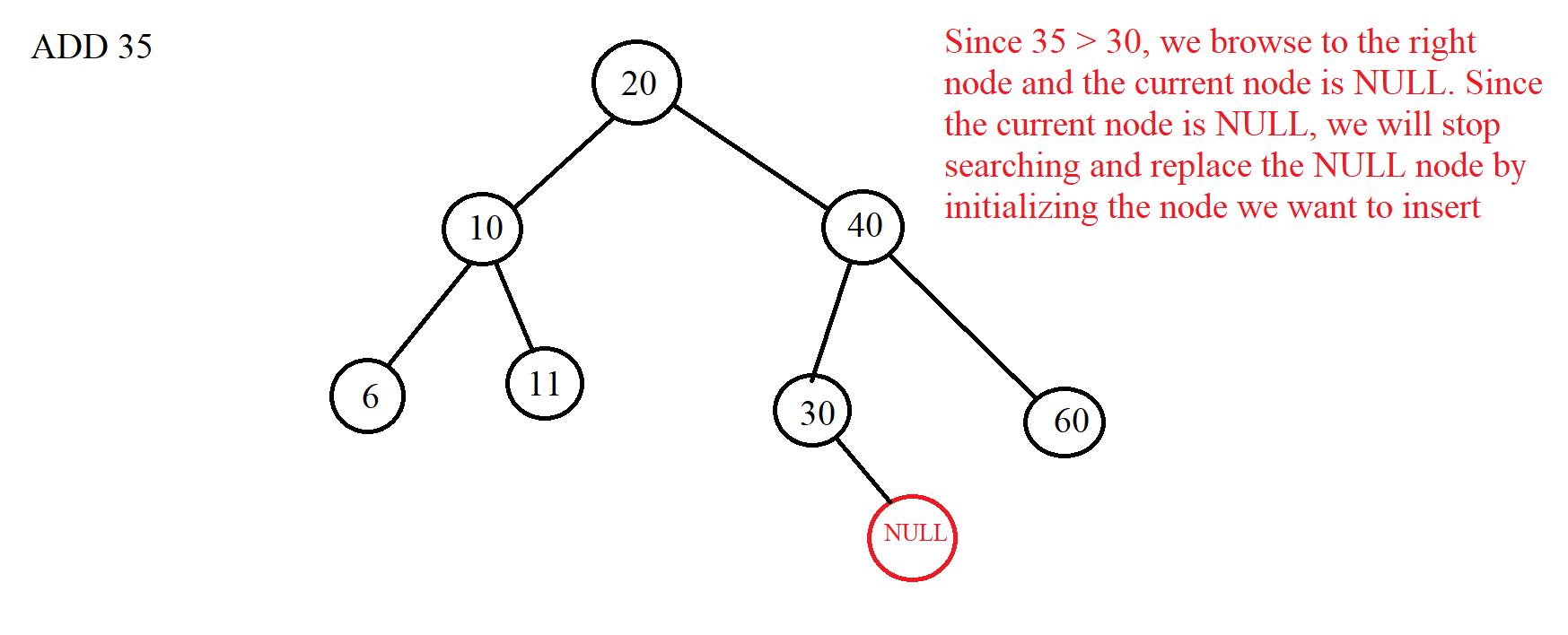


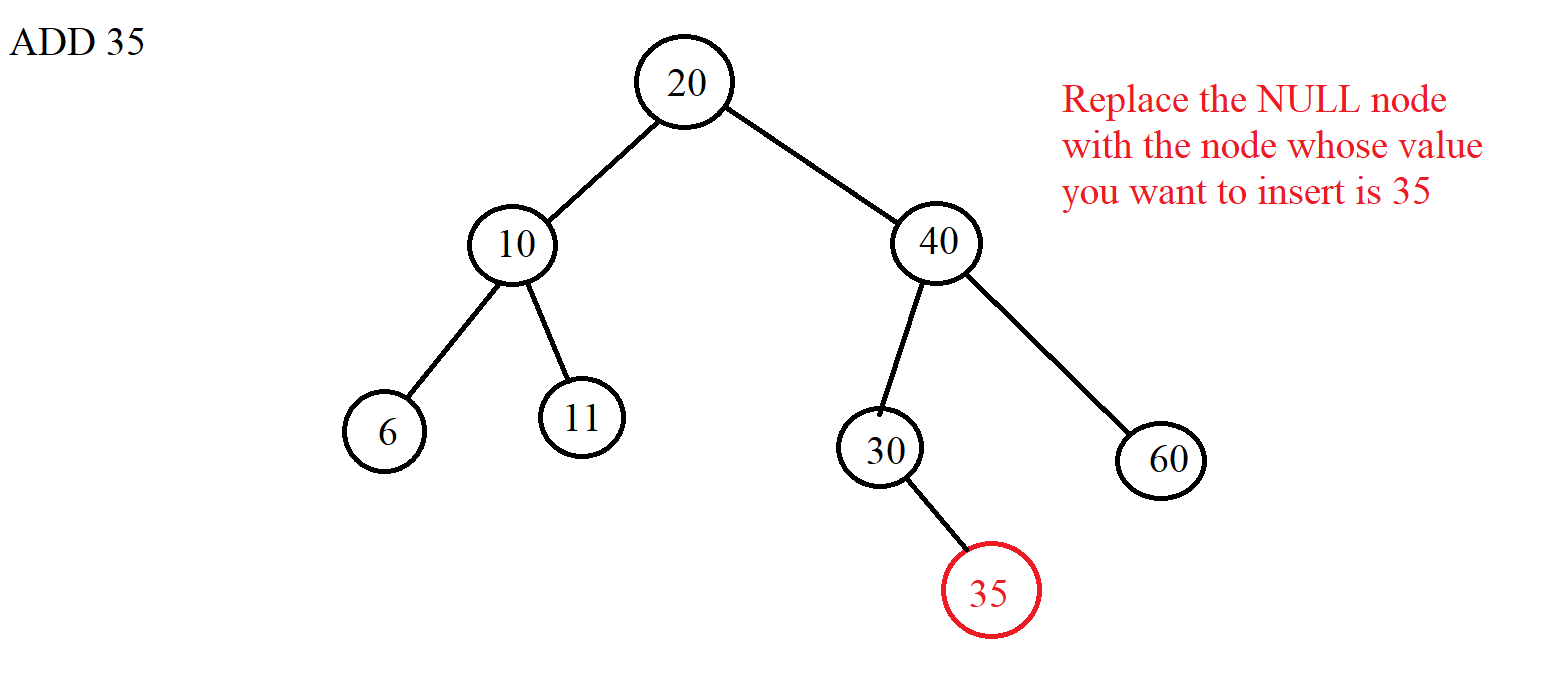
+ Add 1 element: Just like the search is based on conditions, we will start at the root node and will traverse left if the inserted element is greater than the current node and vice versa. We will do this until the current node is a NULL node and finally, we replace the NULL node with the node that we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







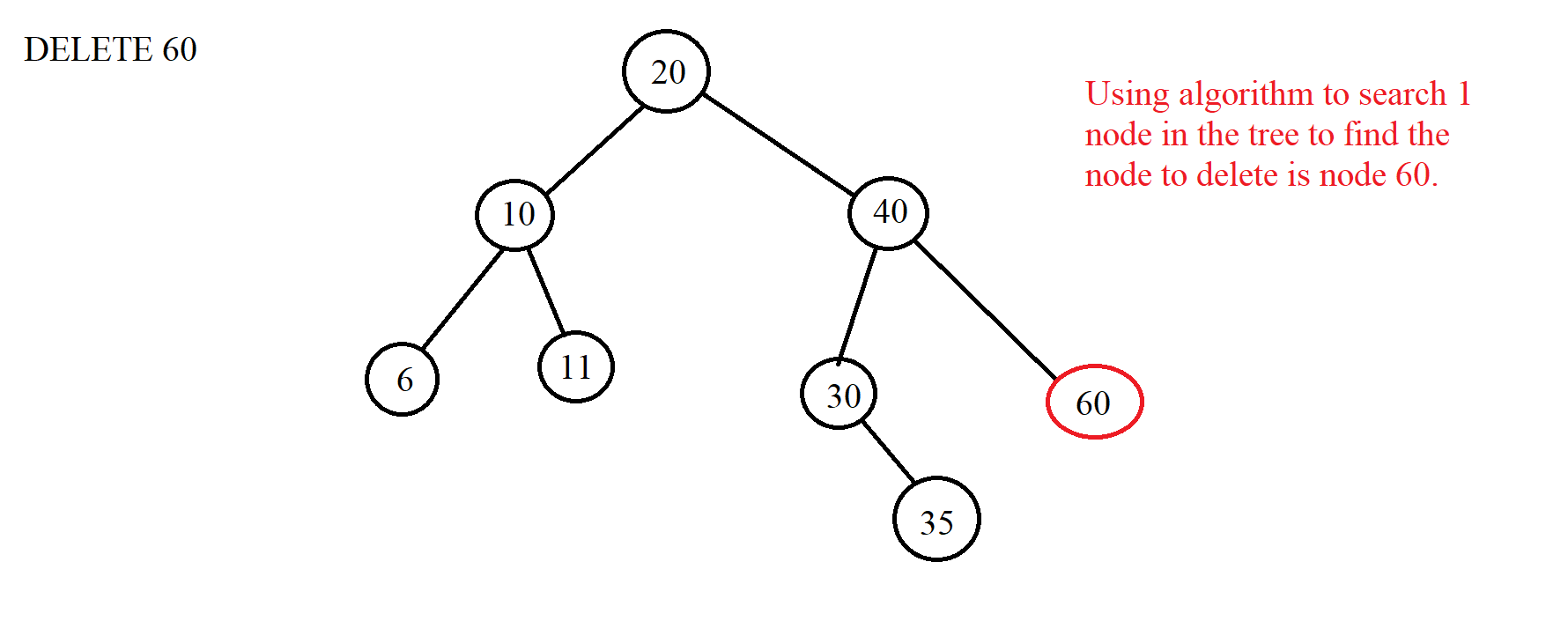


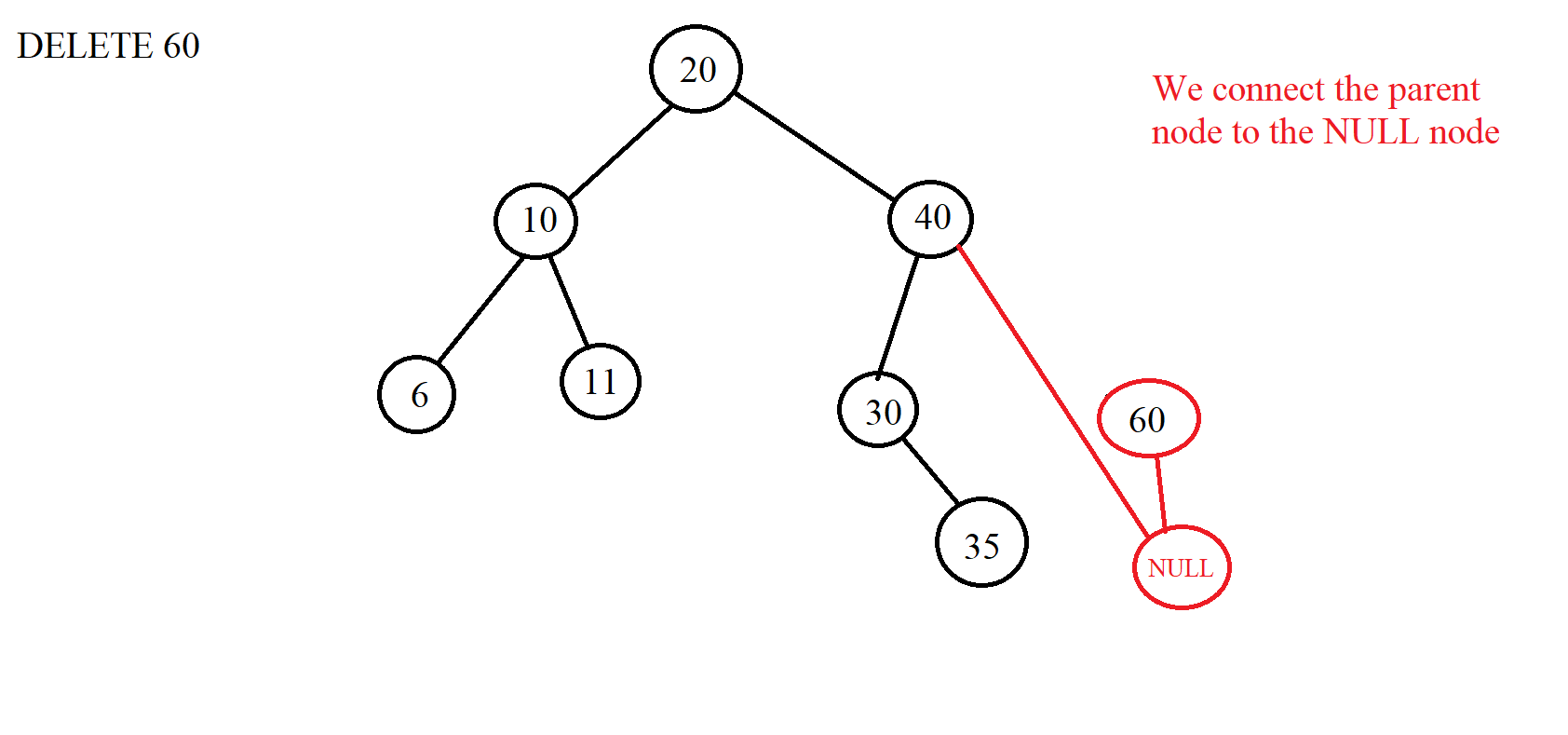


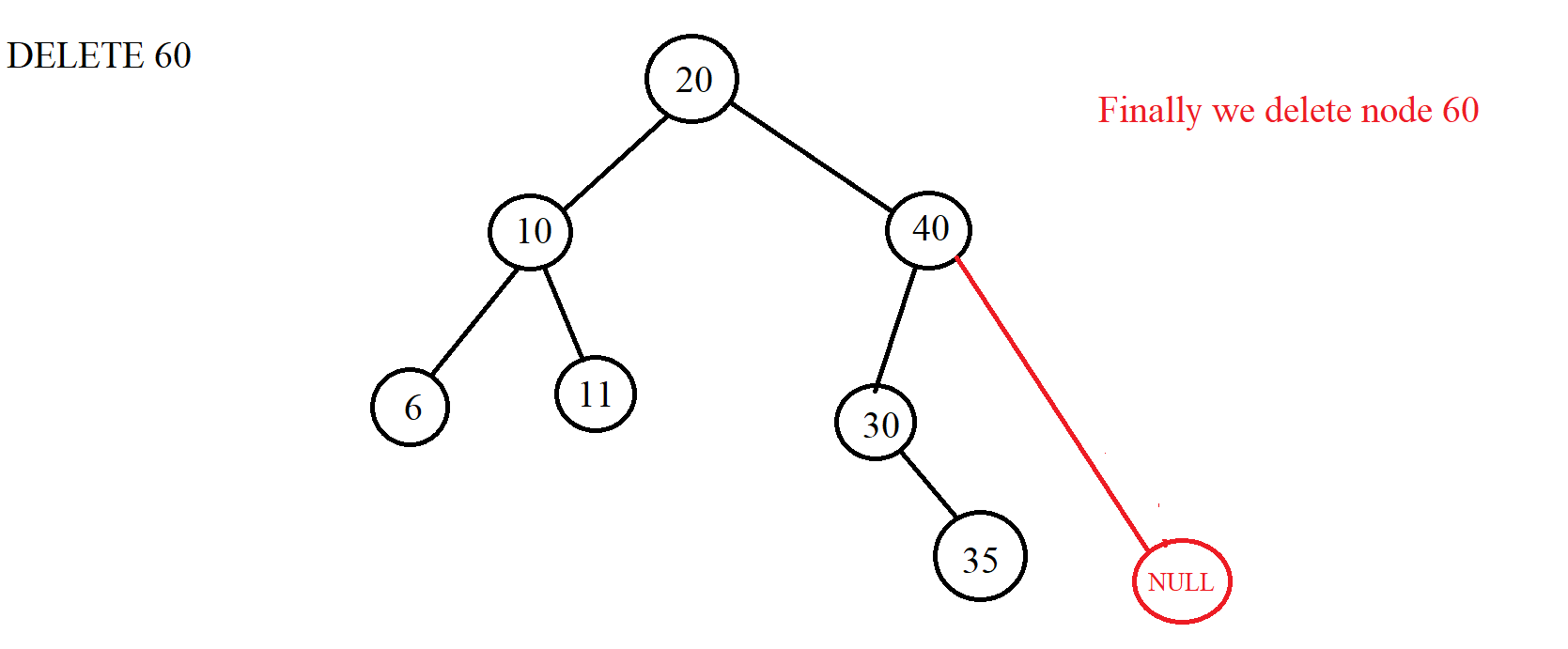


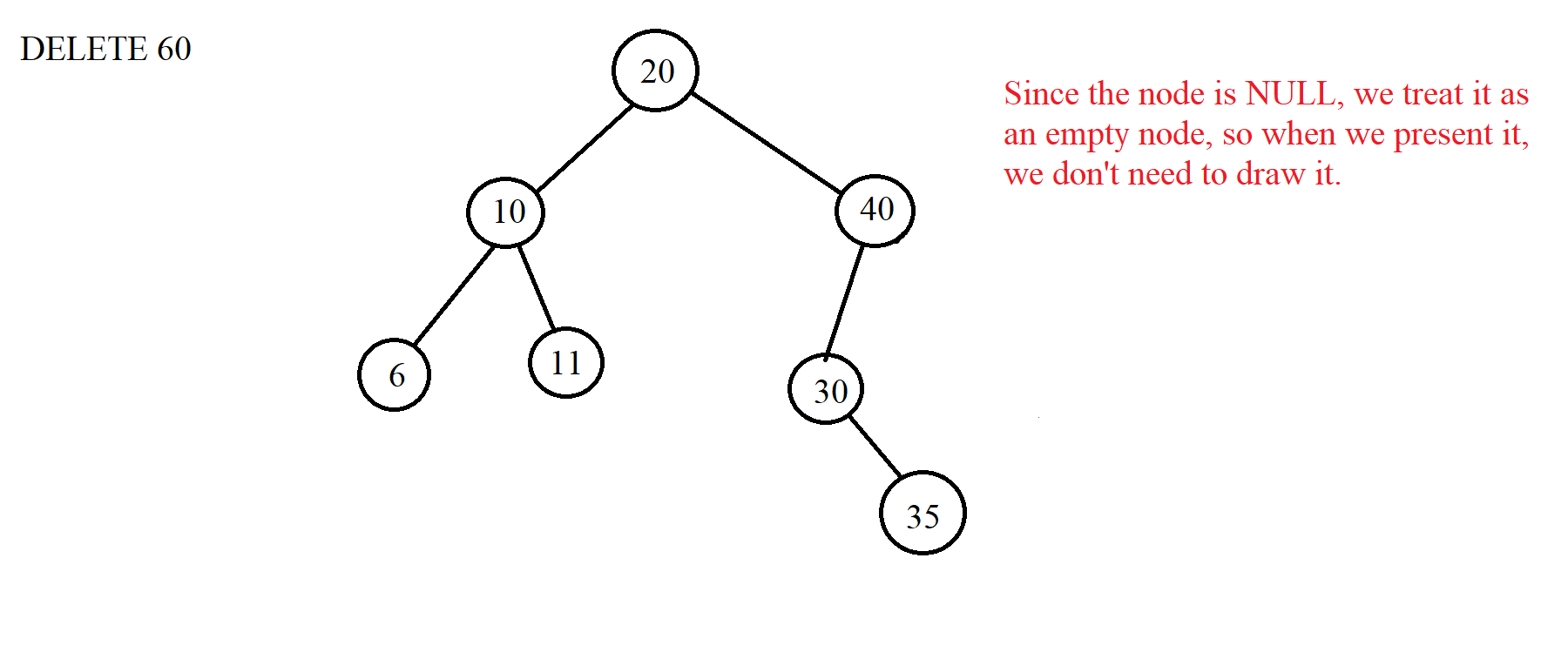
+ Delete 1 element: We use the above tree search algorithm to find the element we want to delete. If you can't find it, it means there is no element in the tree. And if there is, then we divide into 3 cases to delete.

*.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:

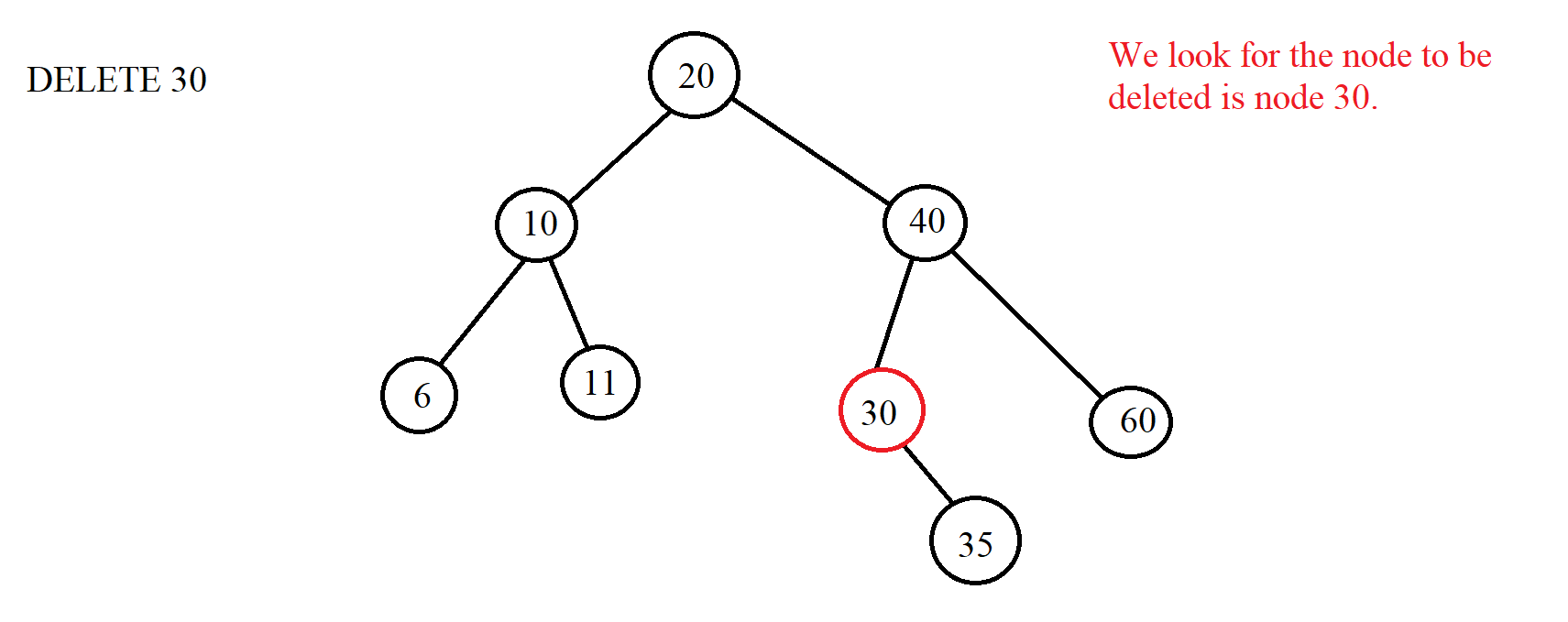


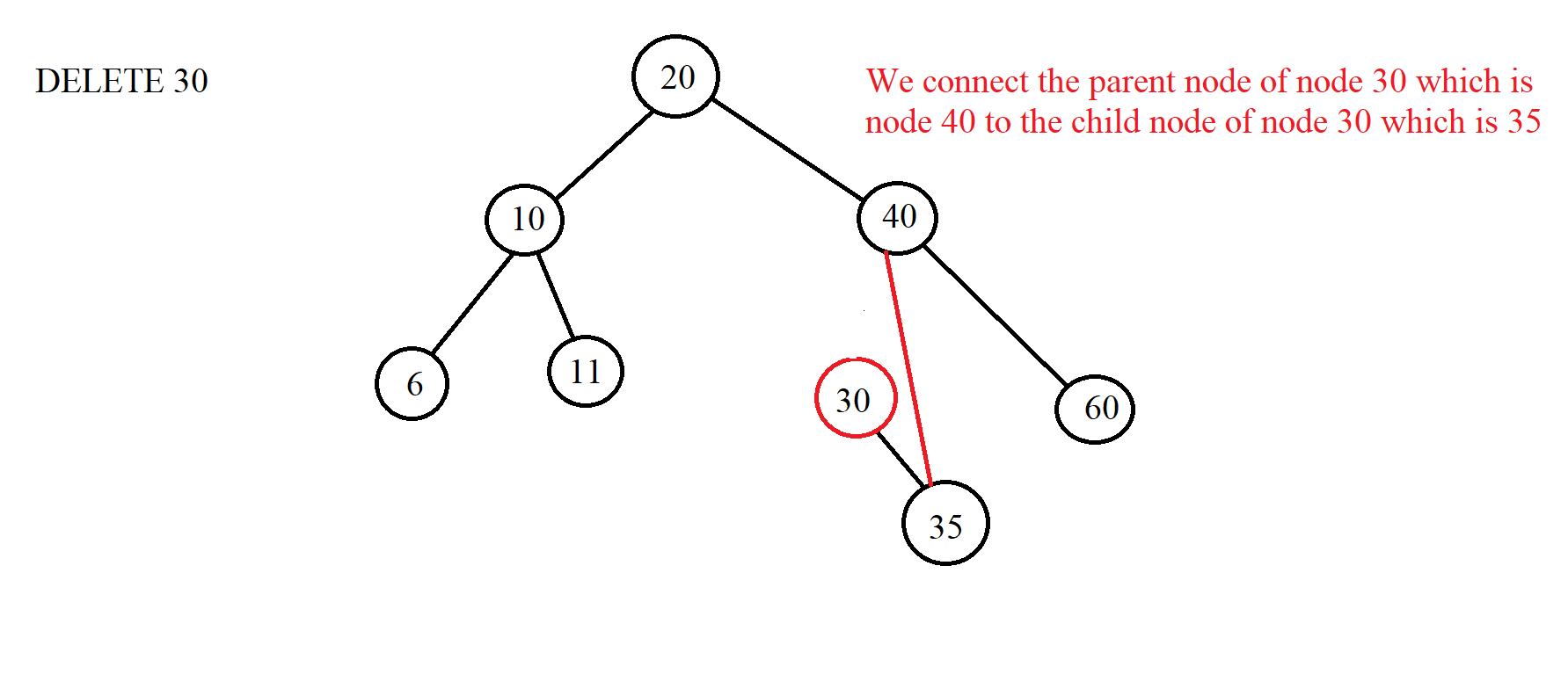


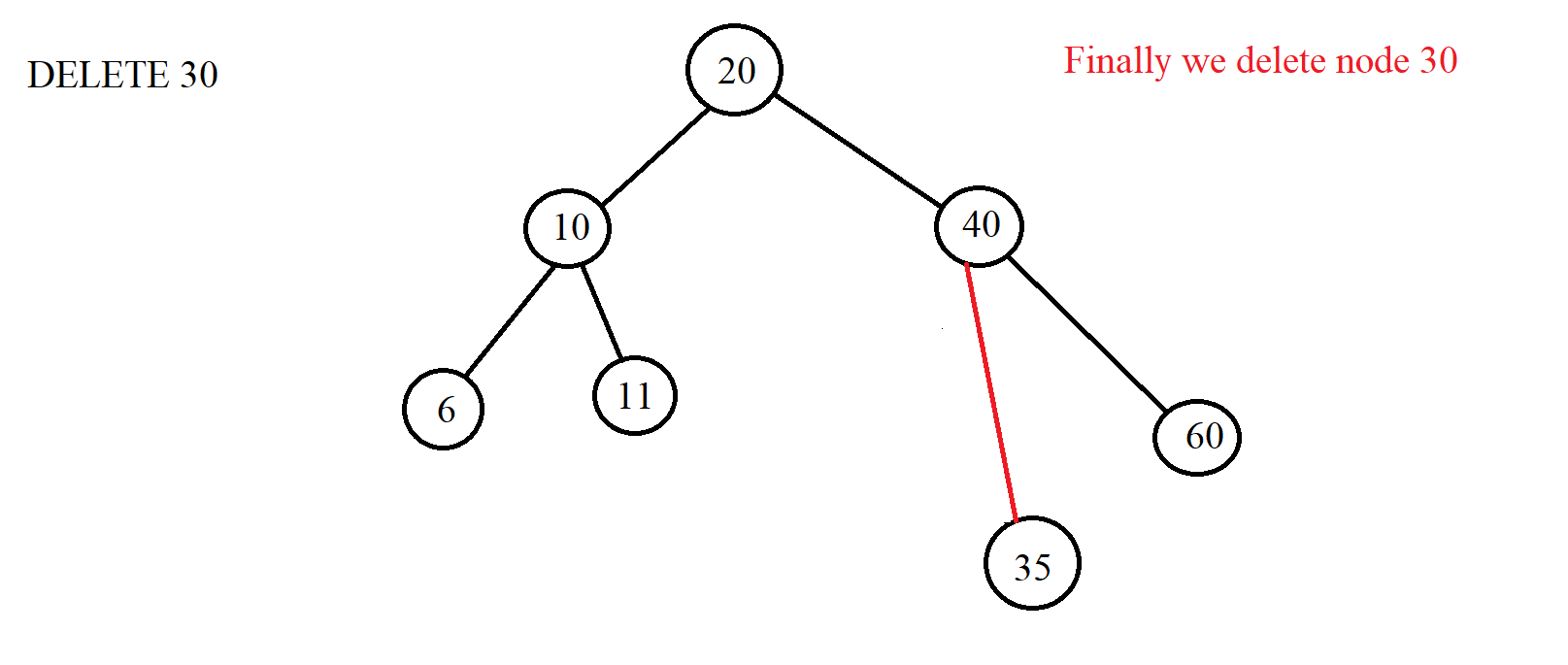




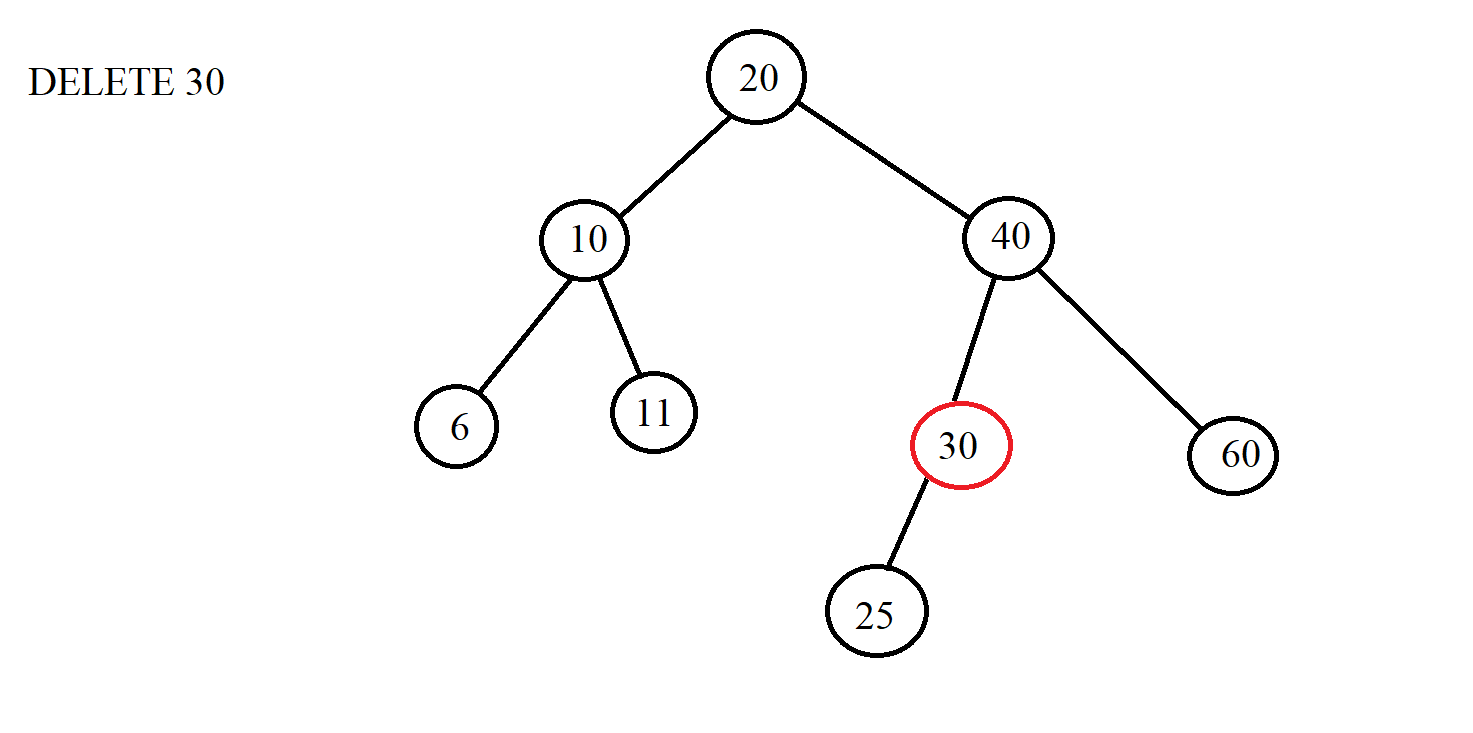
*.Case 2*: Delete node with 1 child node => When we want to delete the element to be deleted in this case, we must connect the parent node of the node to be deleted with the child node of that node because that way the new tree will not be broken. When the connection is complete, we just need to delete 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:

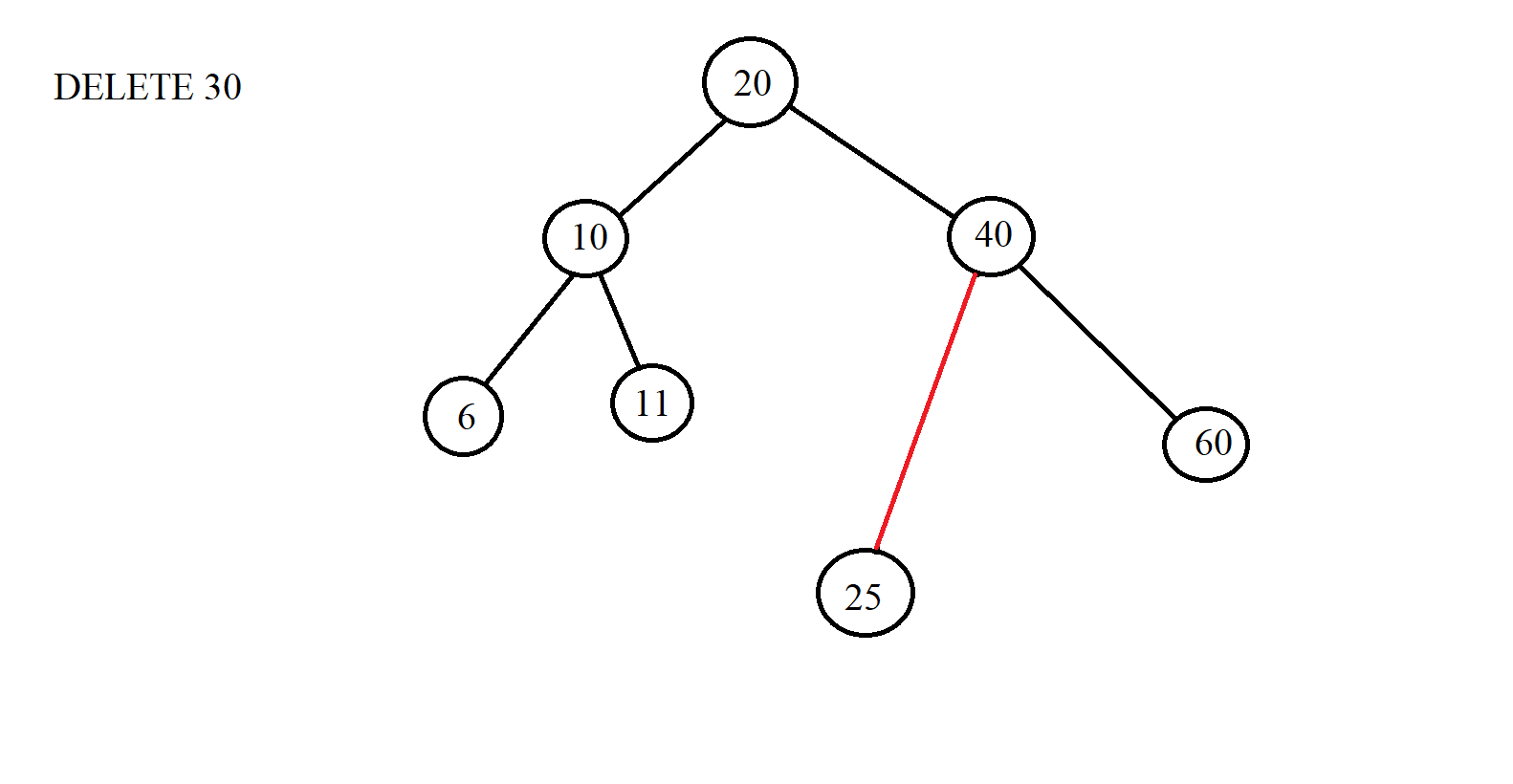






Similarly when we delete 30 and 25 are left child nodes:



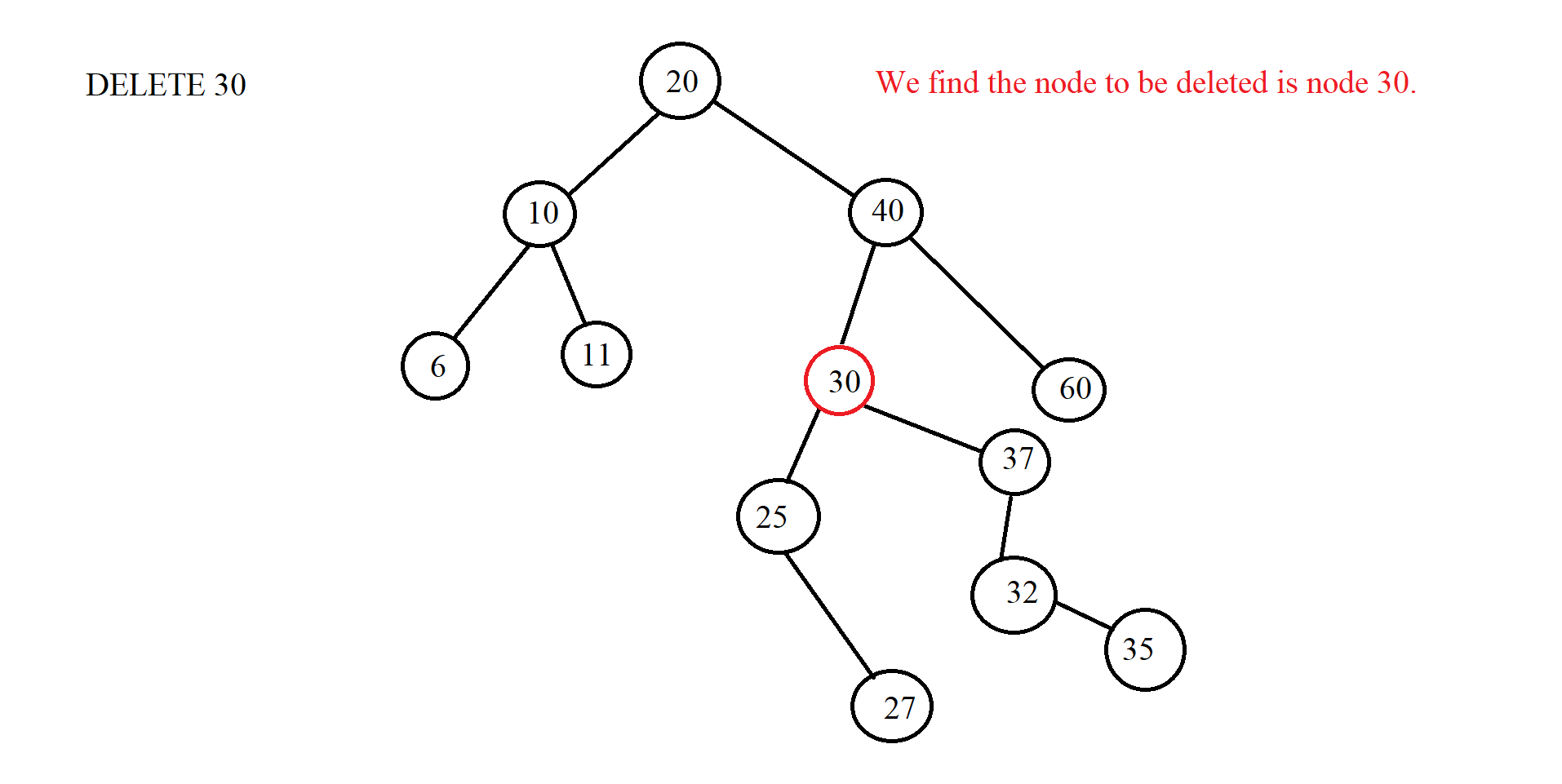


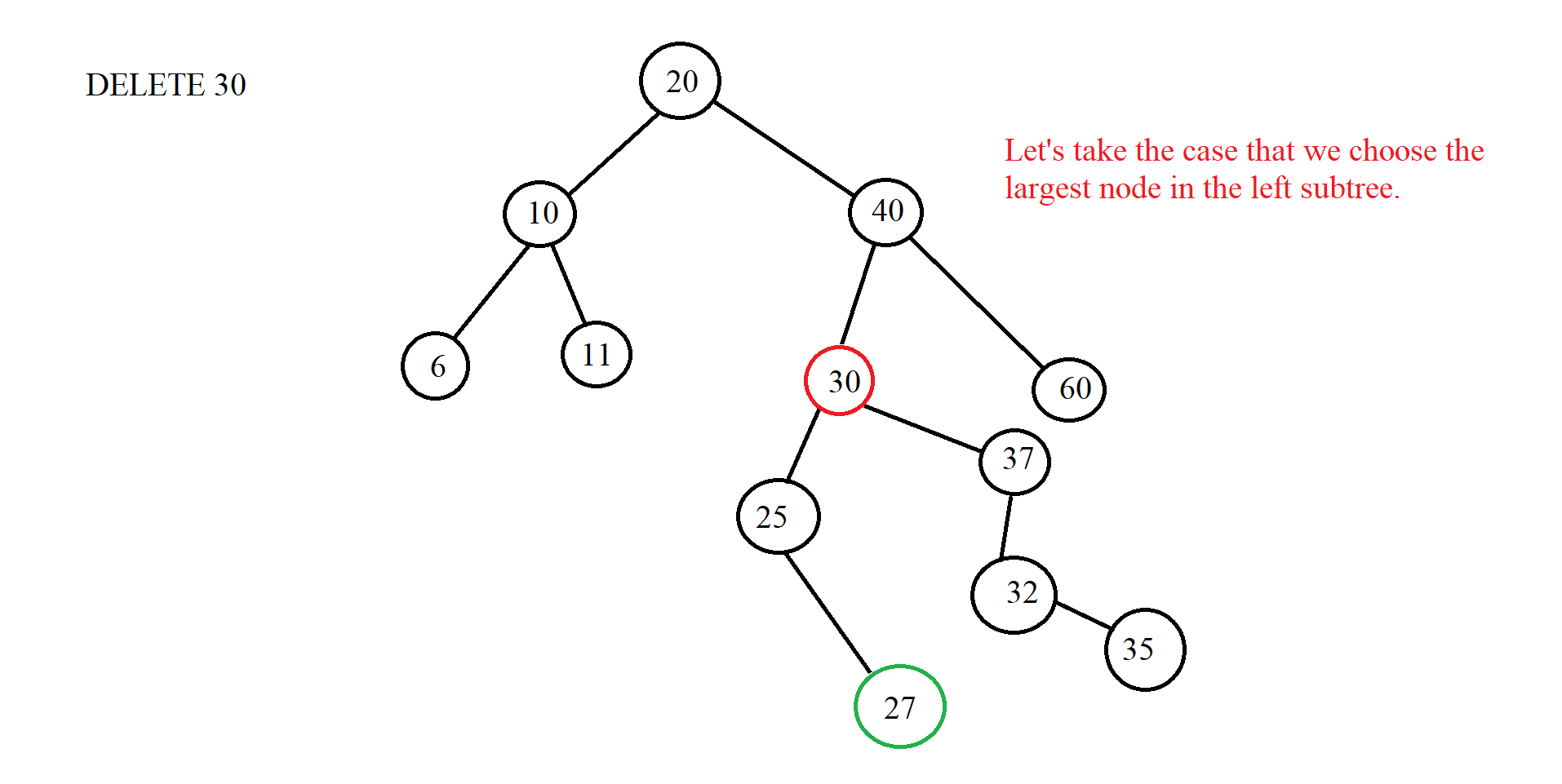
*.Case 3*: Delete node with 2 children => Instead of deleting that node, we delete the node for which we get the value to replace the node that needs to be deleted. Then the node that we want to delete will be replaced with another value. Therefore, the node so that we can substitute the value for the 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:

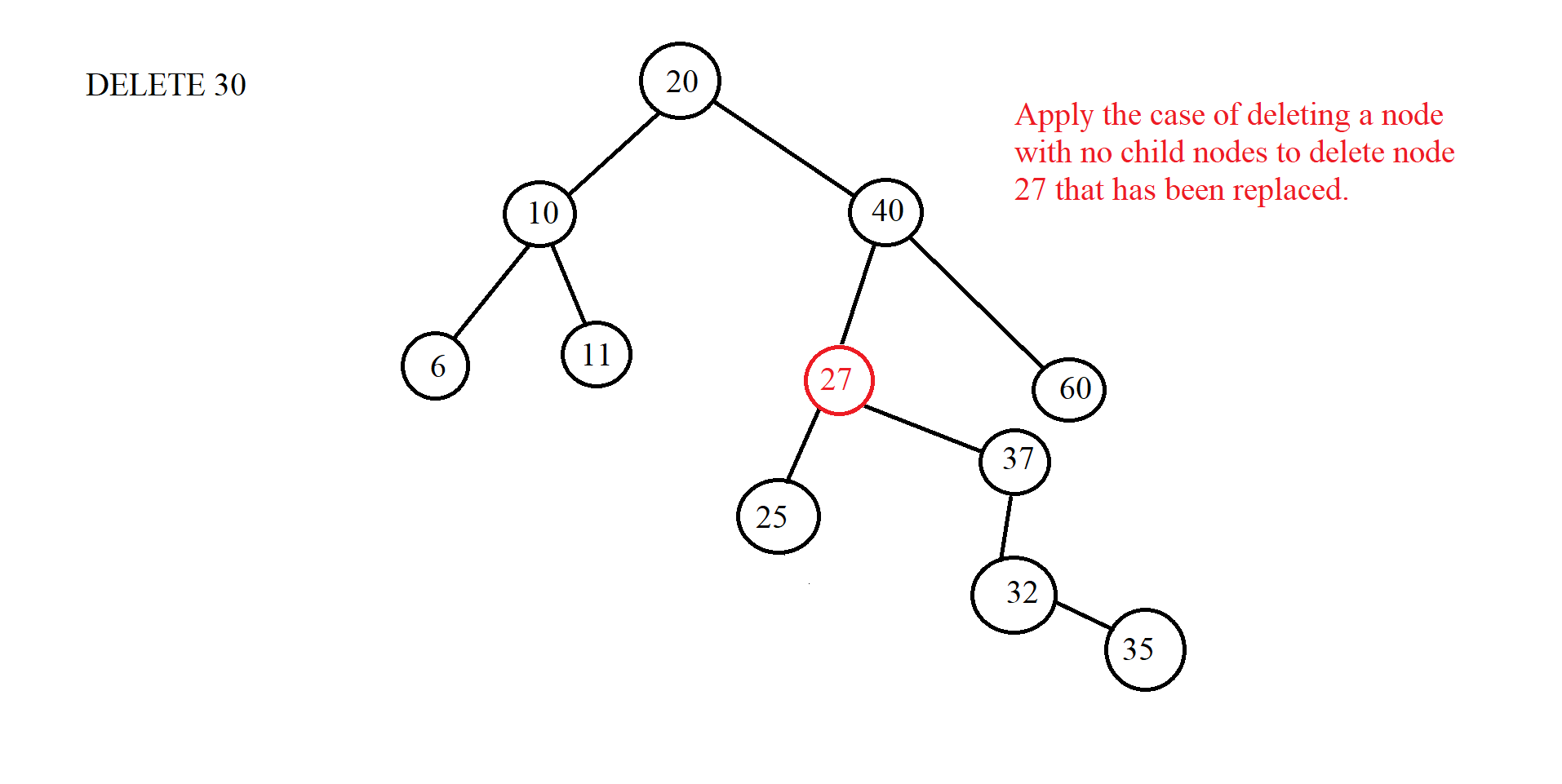
\*Node 1: We will choose the node with the largest value of the left subtree of the replaced node.

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

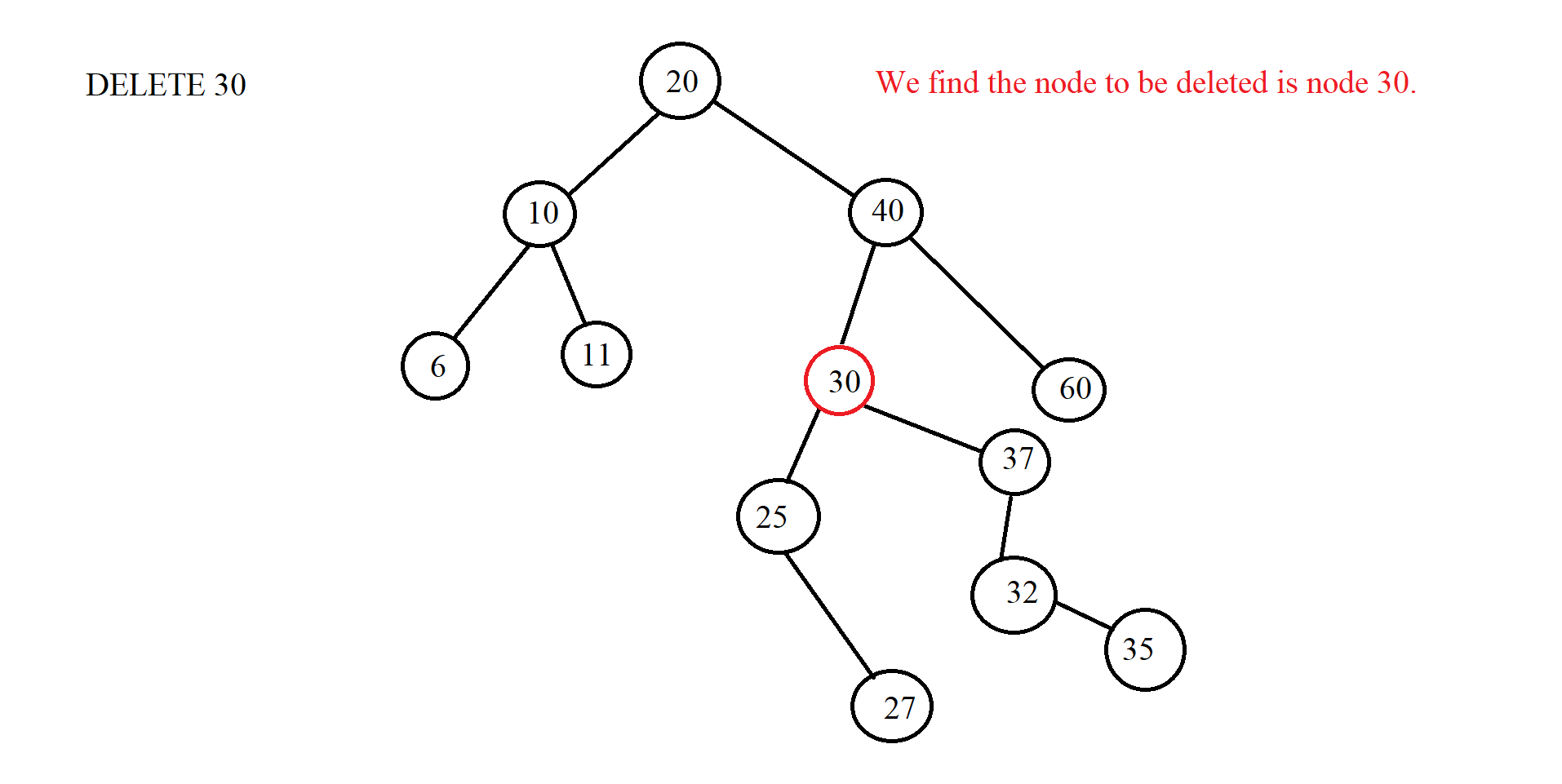
When the replacement is complete, we delete the node that has been taken to replace the node we want to delete. Here is the largest node of the left subtree or the smallest node of the right subtree. If the replacement node is deleted, it will fall into one of the two cases 1 or 2 above. Example:

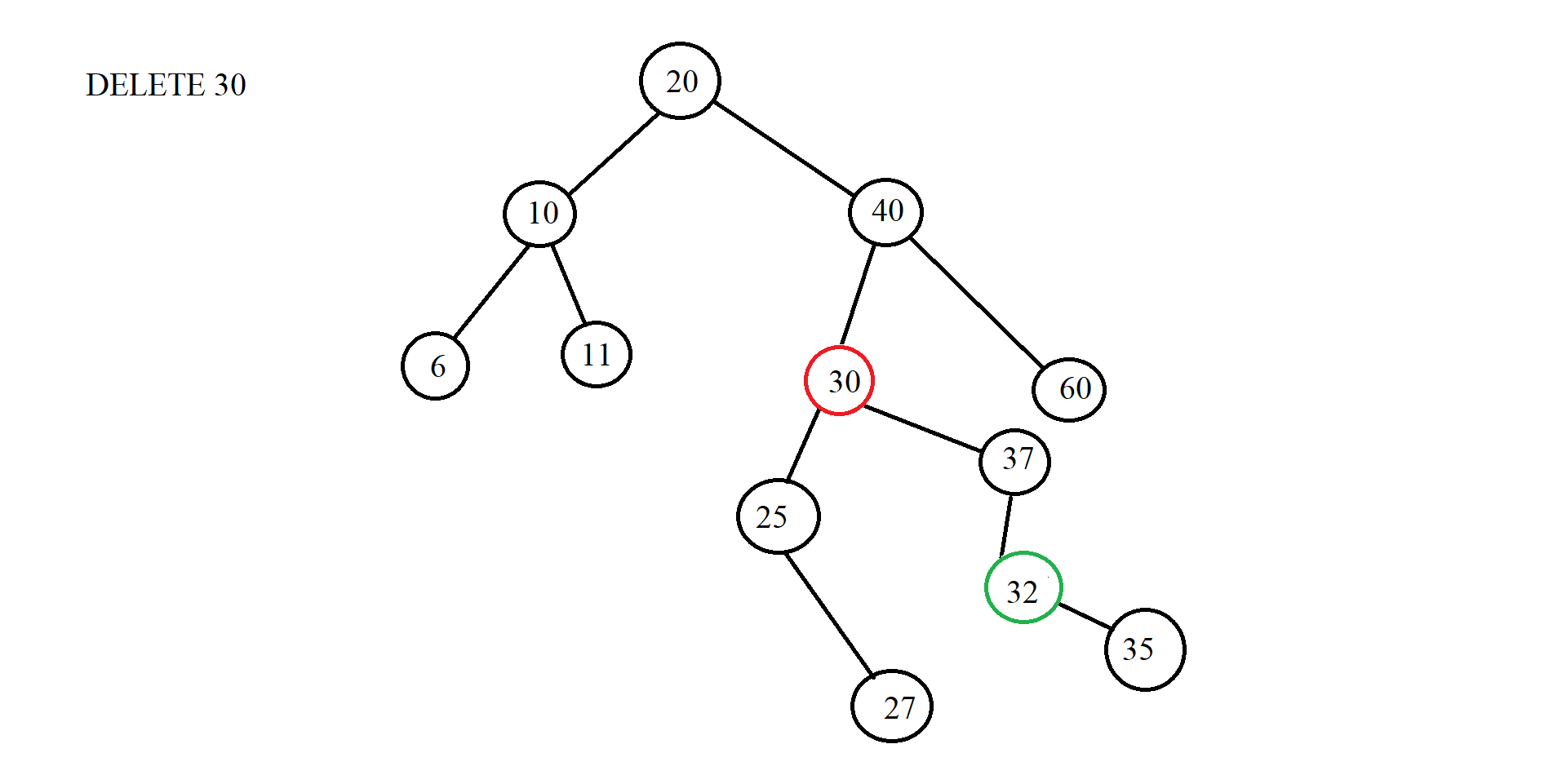


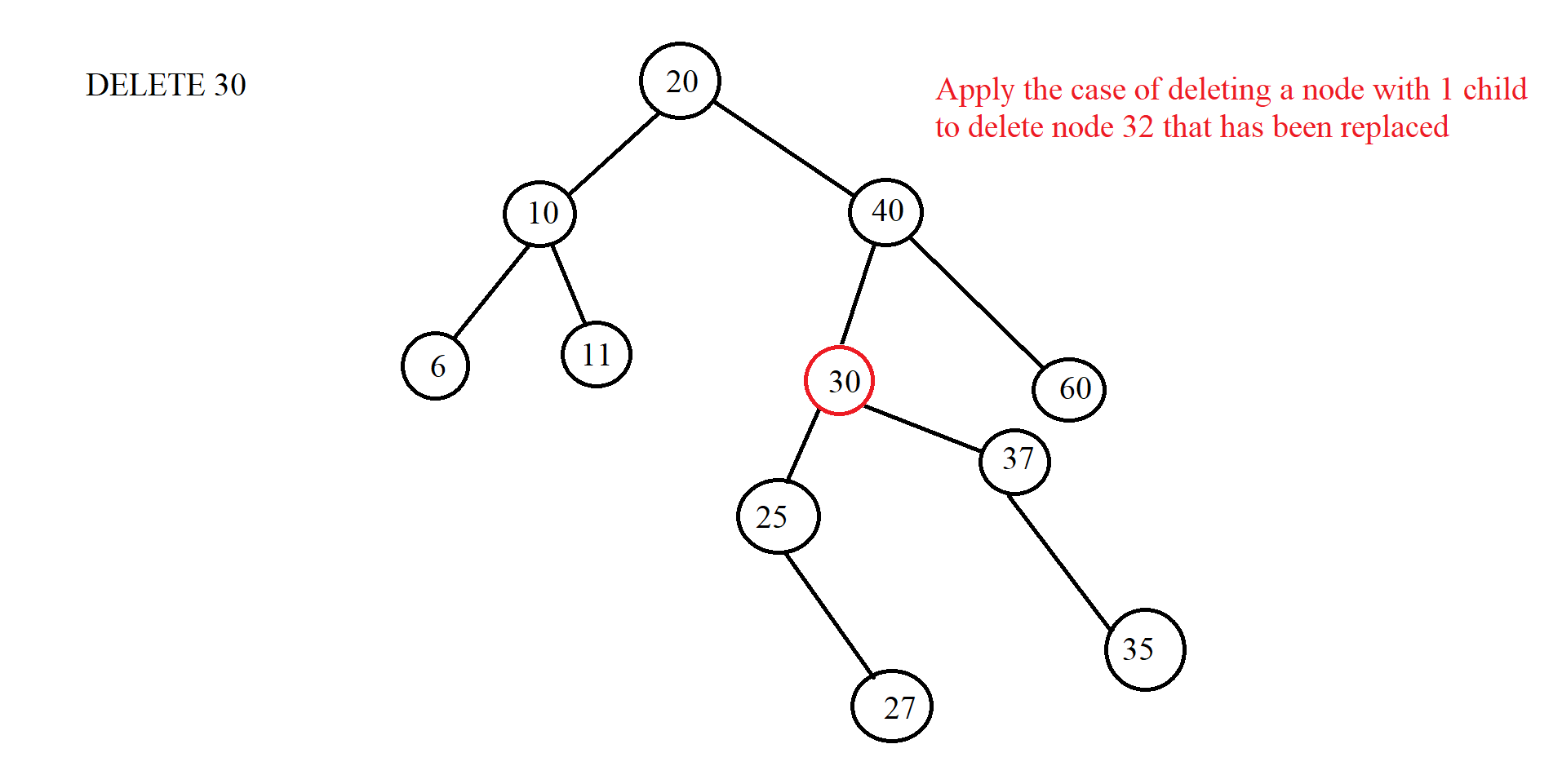




(Let's say we want to get the smallest node of the right subtree instead)











***\*Illustration:***

1. *Example 1:*

* **INPUT:**

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

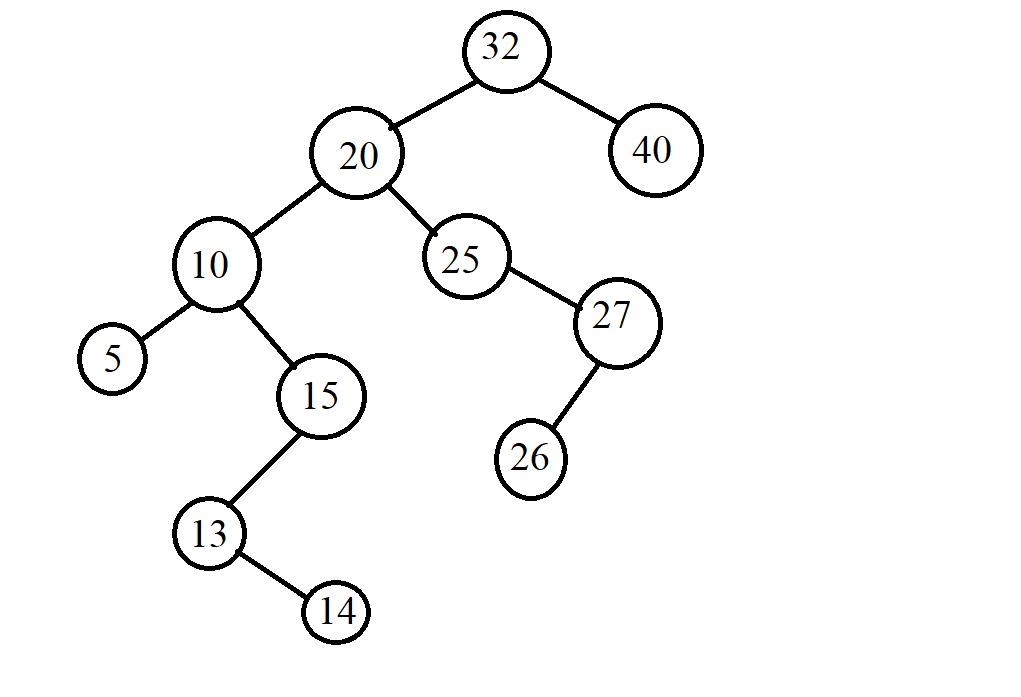
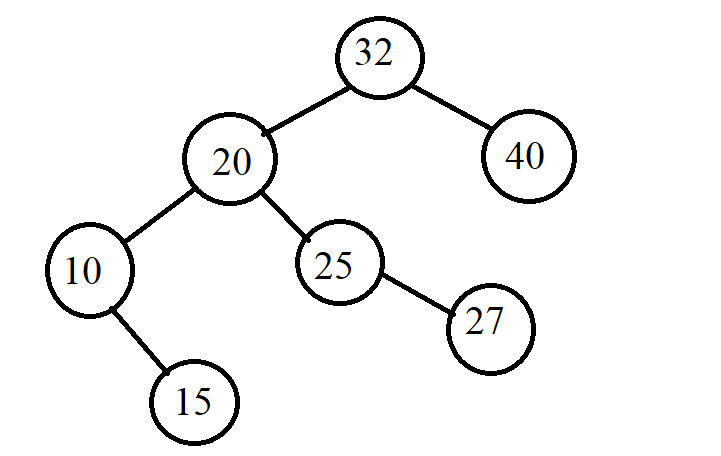
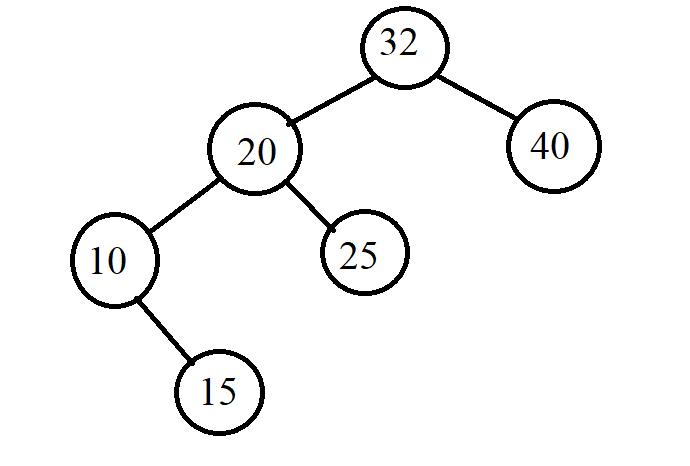
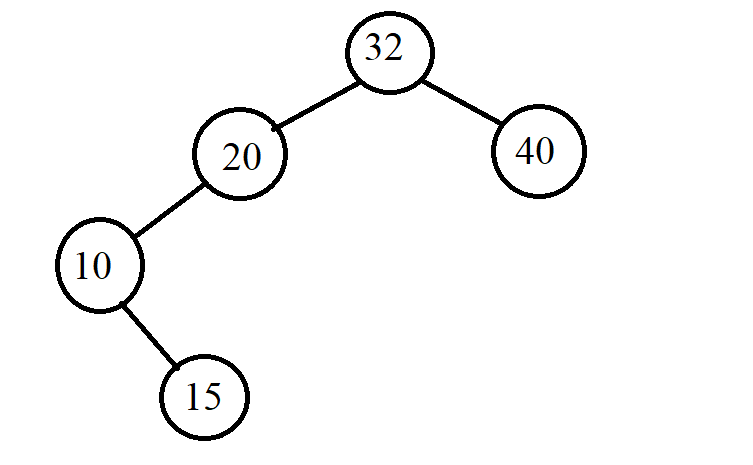
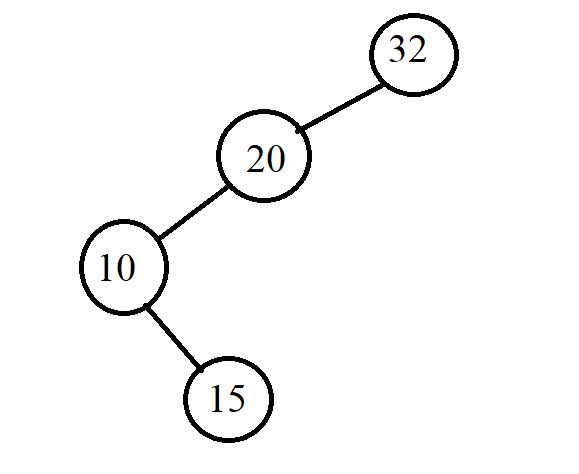
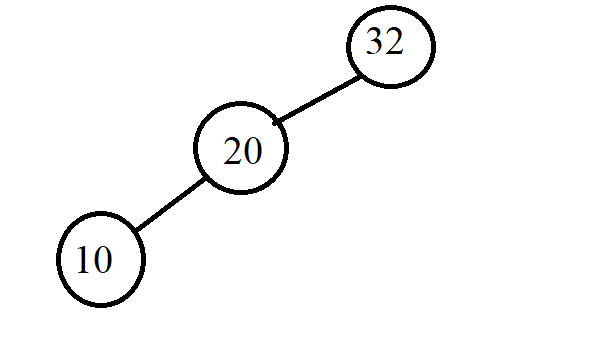
-Find: 15, 29

-Delete: 25, 40 , 20

* **OUTPUT:**

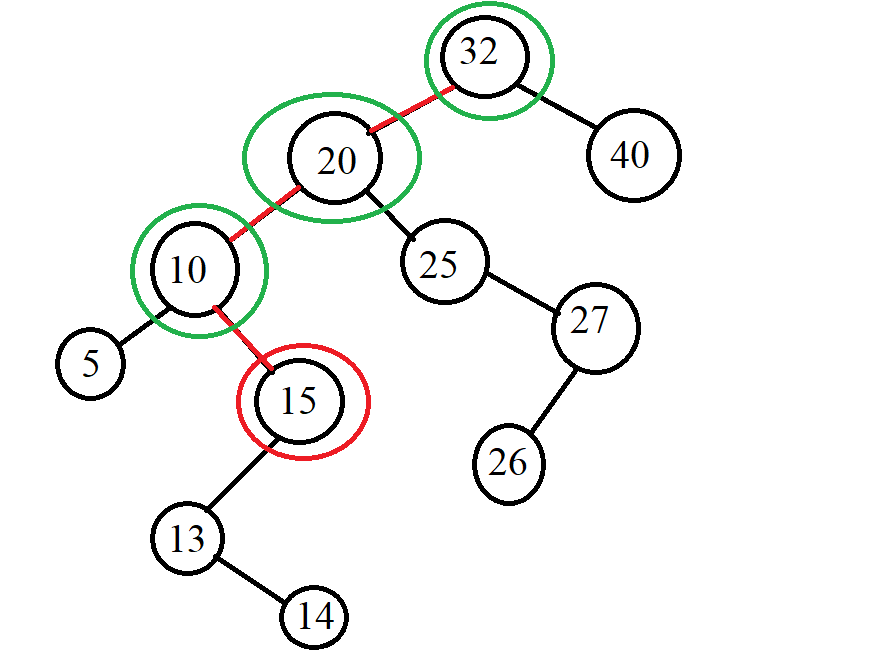
-Insert:

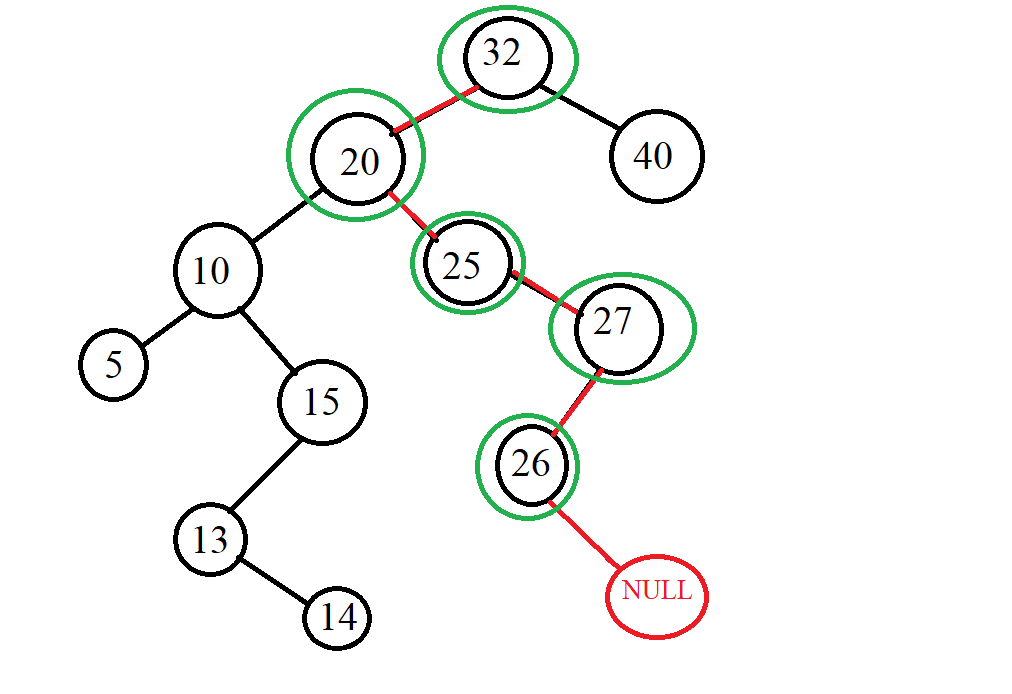




-Find:

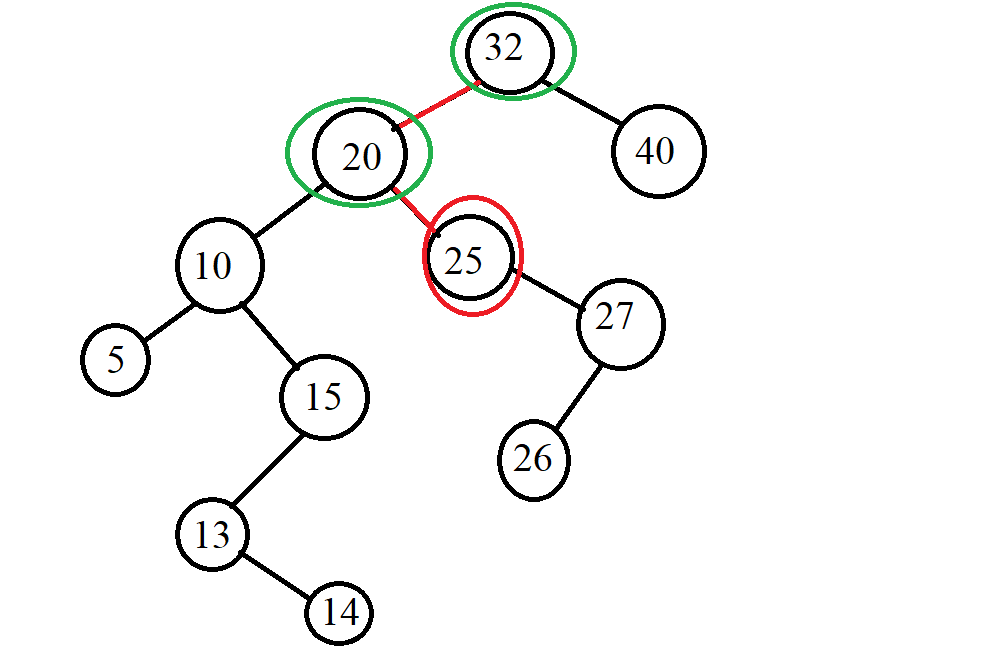
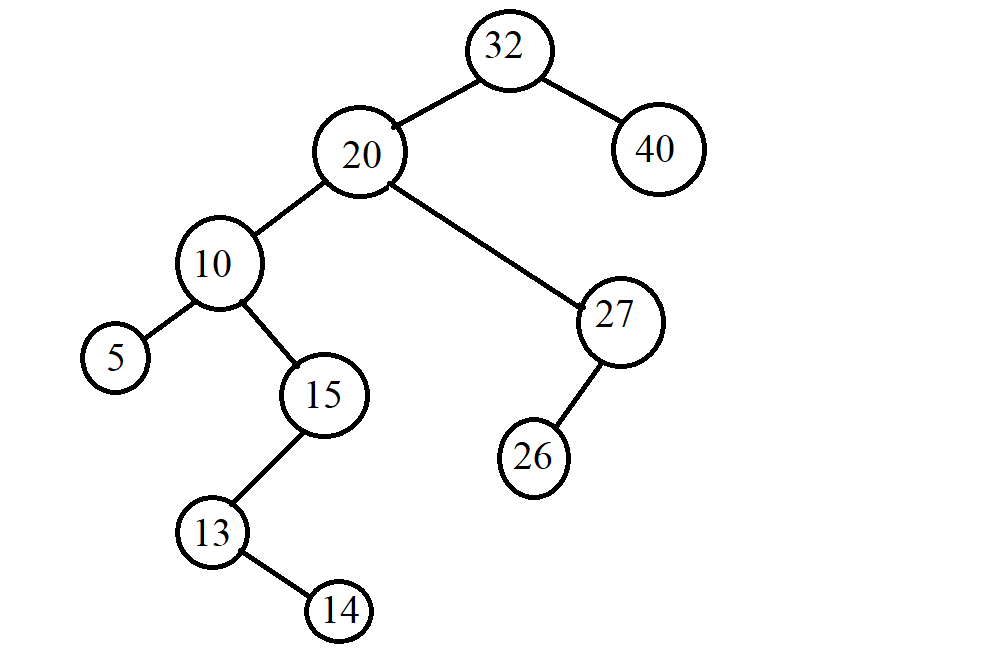
.Find 15:

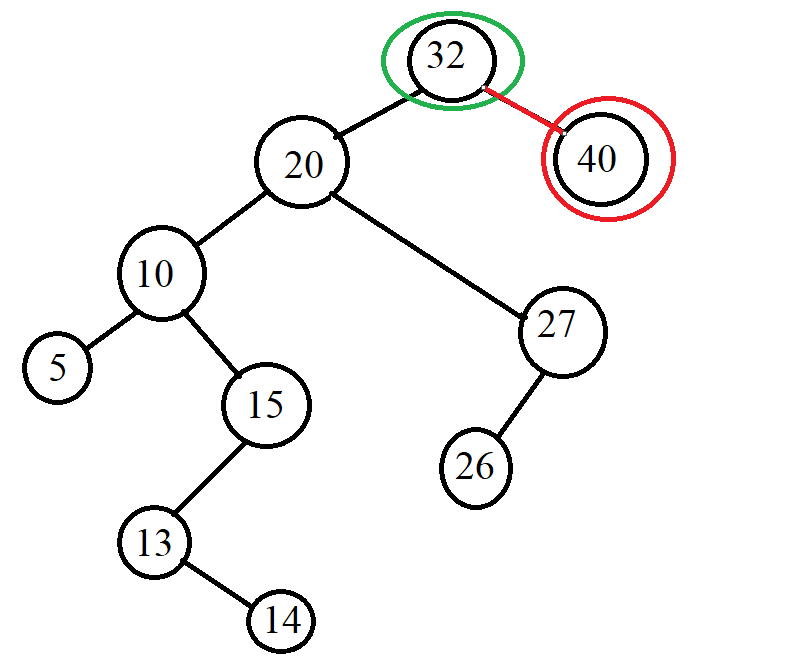


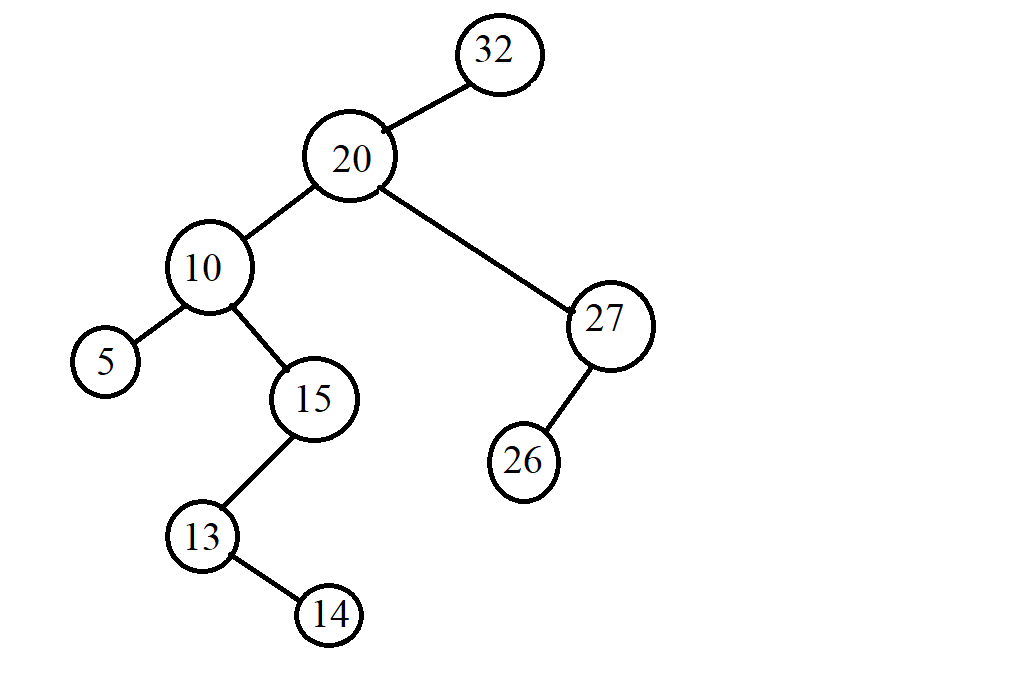
.Find 29: 

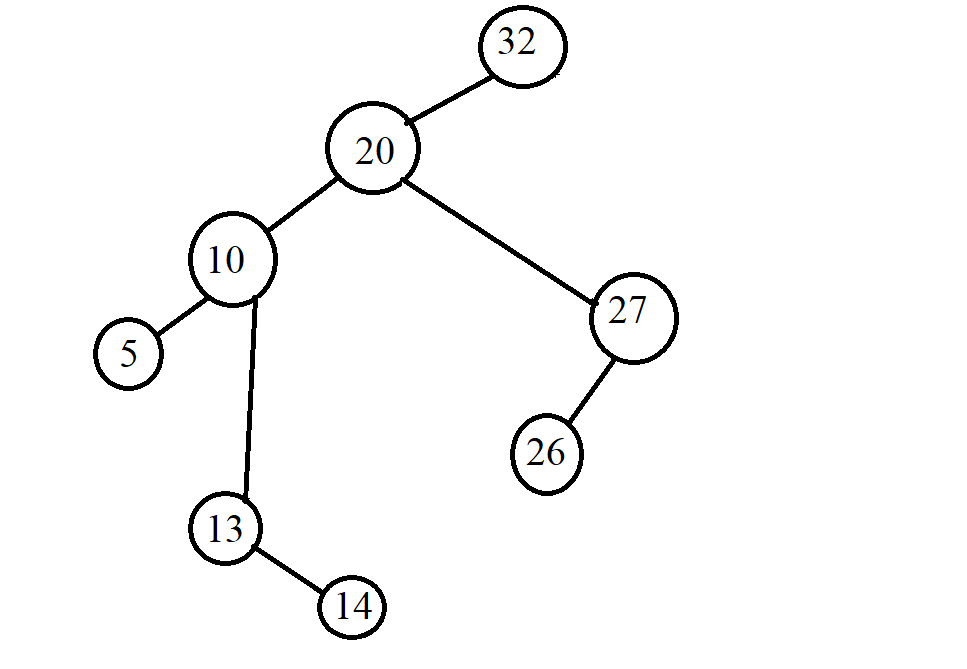
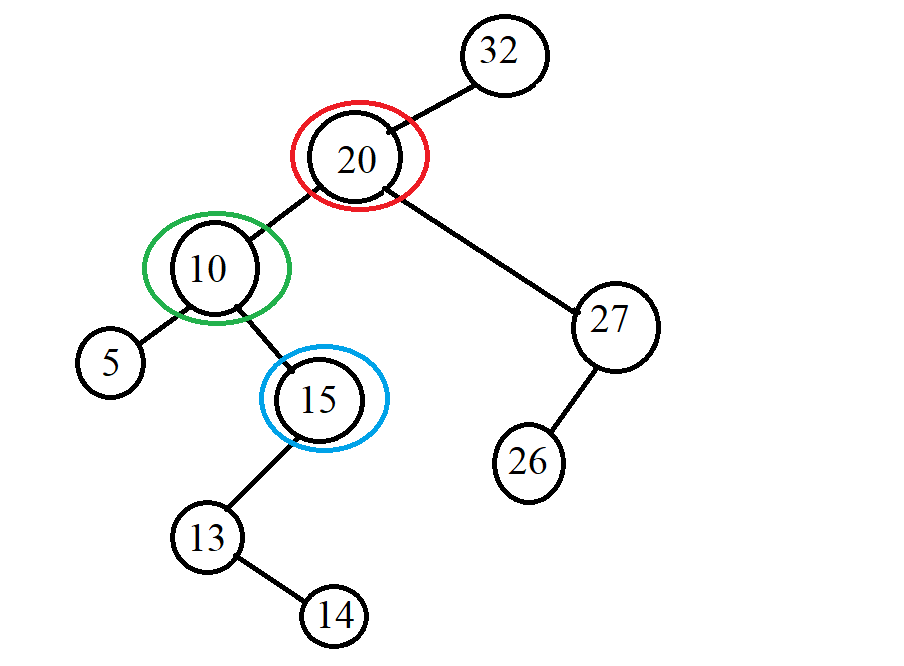
-Delete:

.Delete 25

.Delete 40: 



.Delete 20: 

1. *Example 2:*

* **INPUT:**

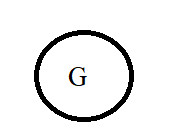
-Insert: G, J, A, T, B, W, C, K.

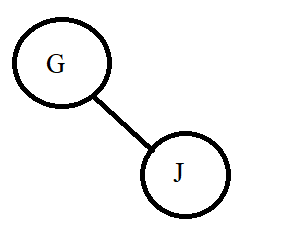
-Find: A, Z

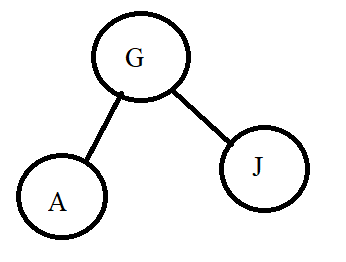
-Delete: J, A

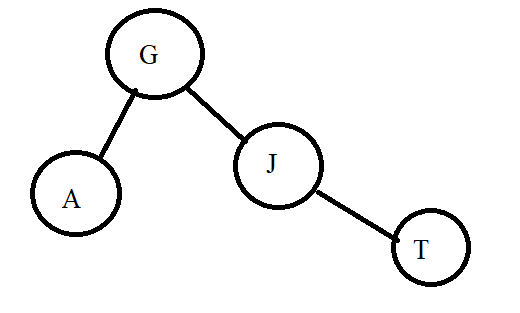
* **OUTPUT:**

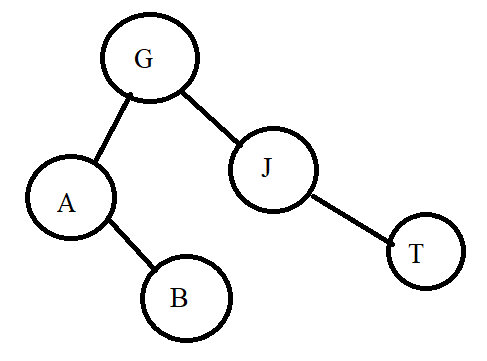
-Insert:

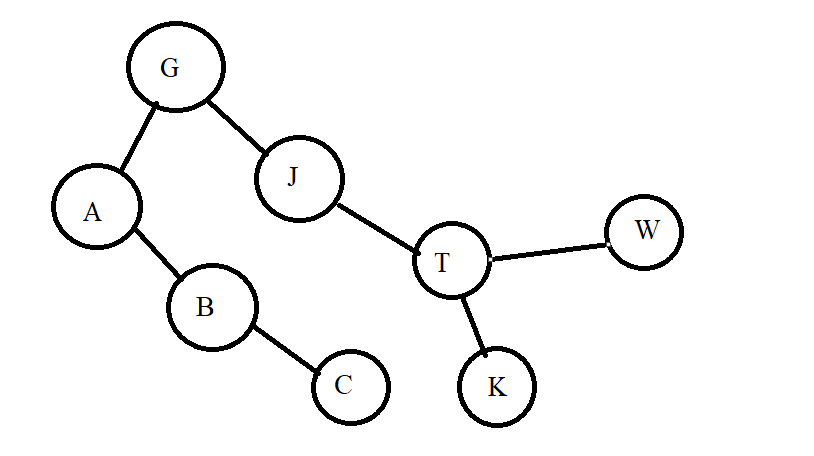






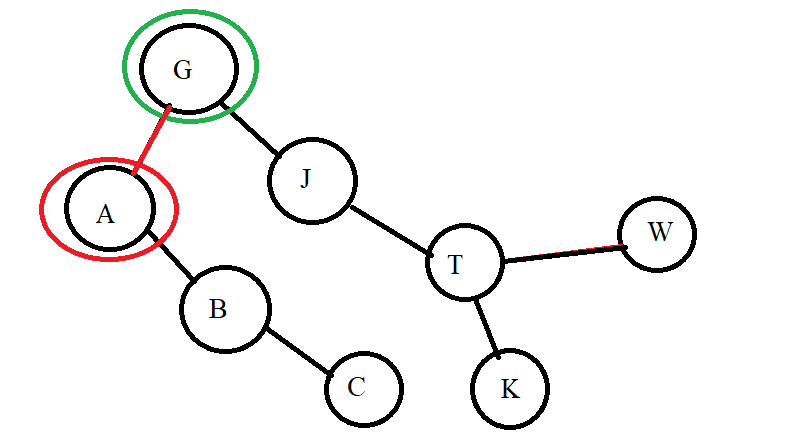




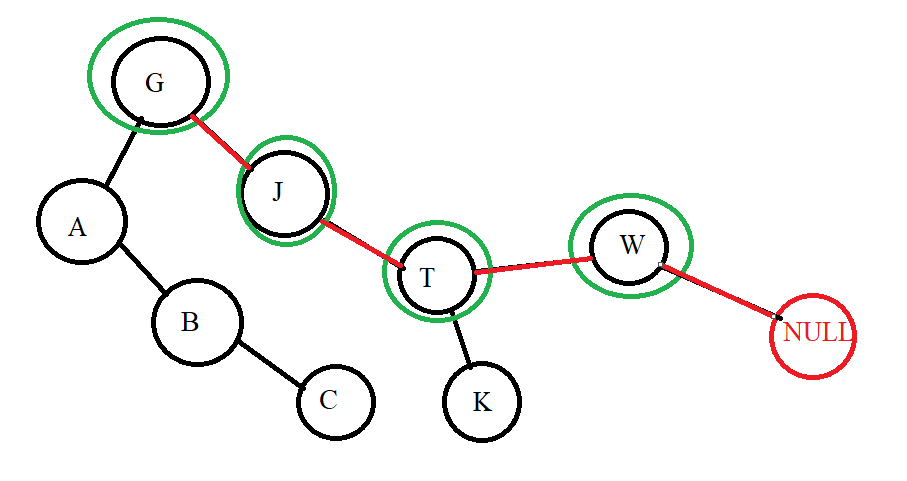


-Find:

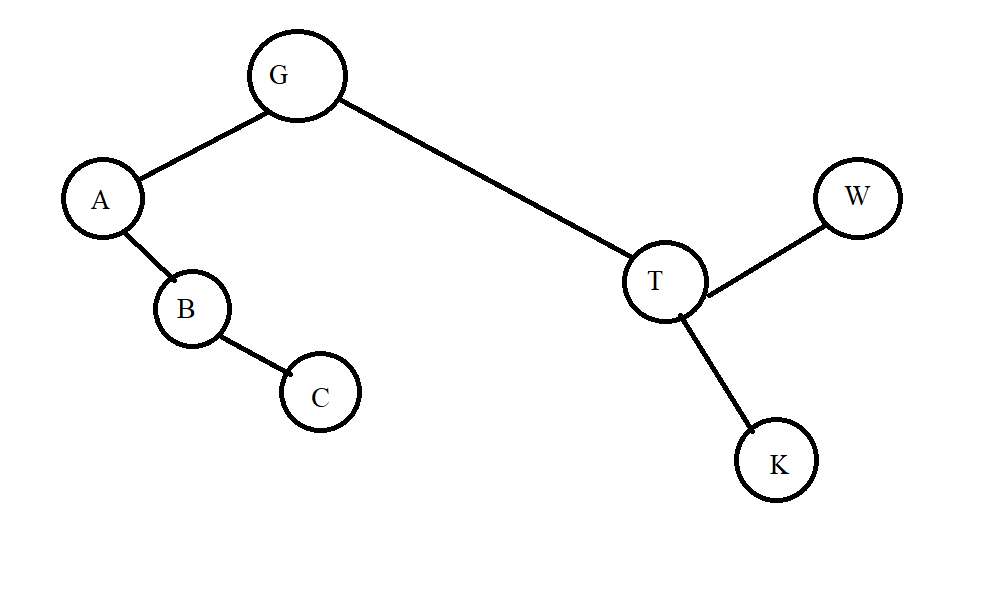
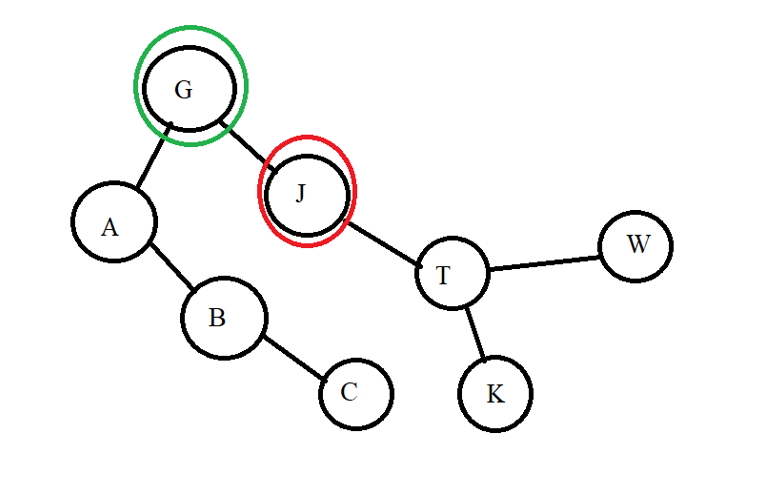
.Find A

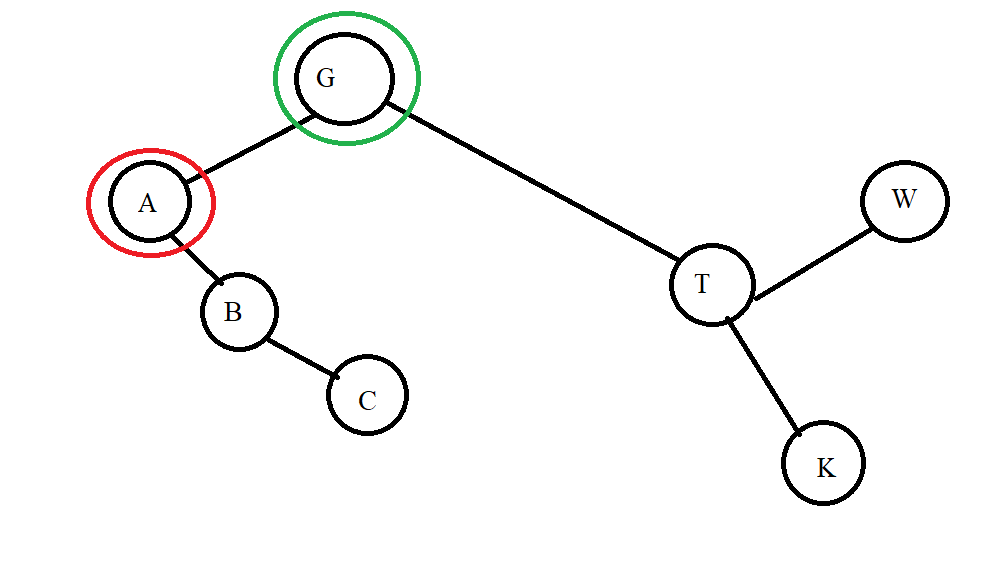


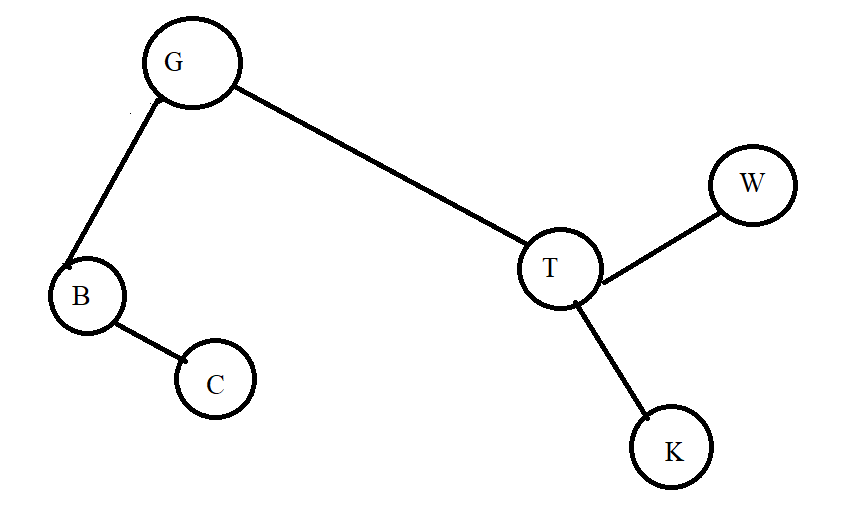
.Find Z



-Delete:

.Delete J: 

.Delete A: 



1. *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)

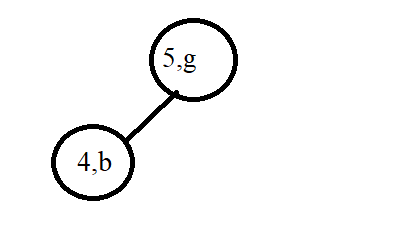
**-**Find: (9,e), (7,a)

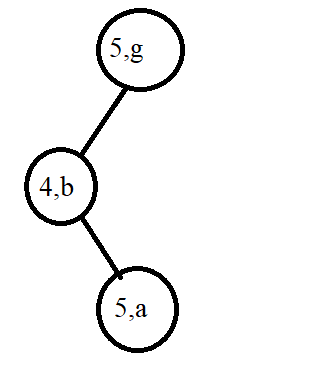
**-**Delete: (5,g), (2,p)

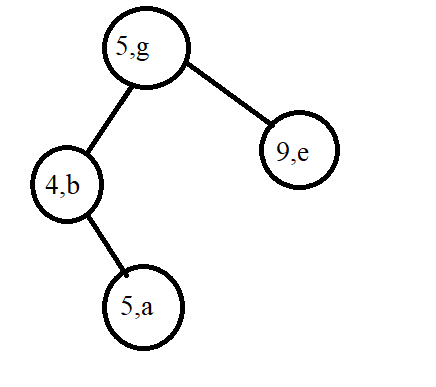
* **OUTPUT:**

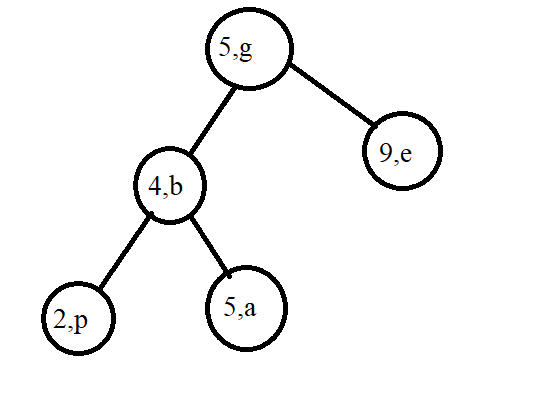
-Insert:

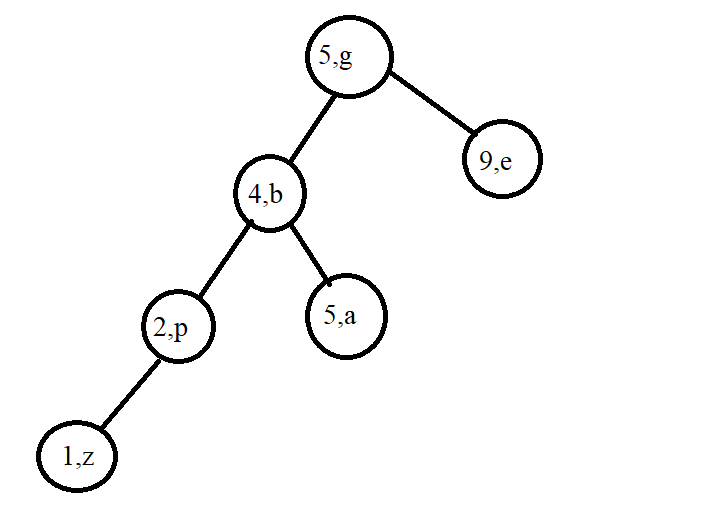






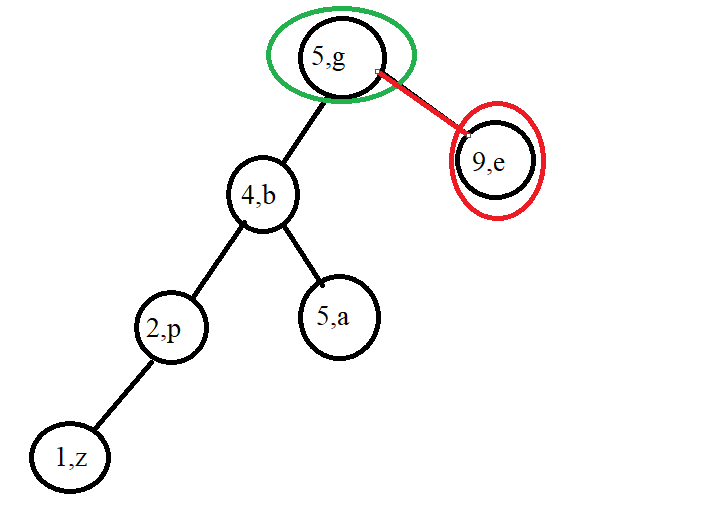




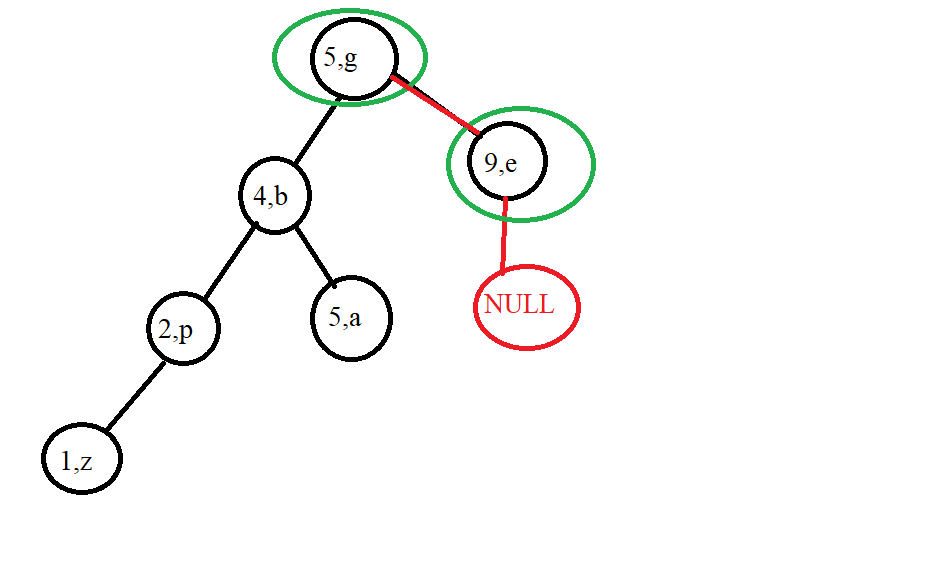


-Find:

.Find (9,e)

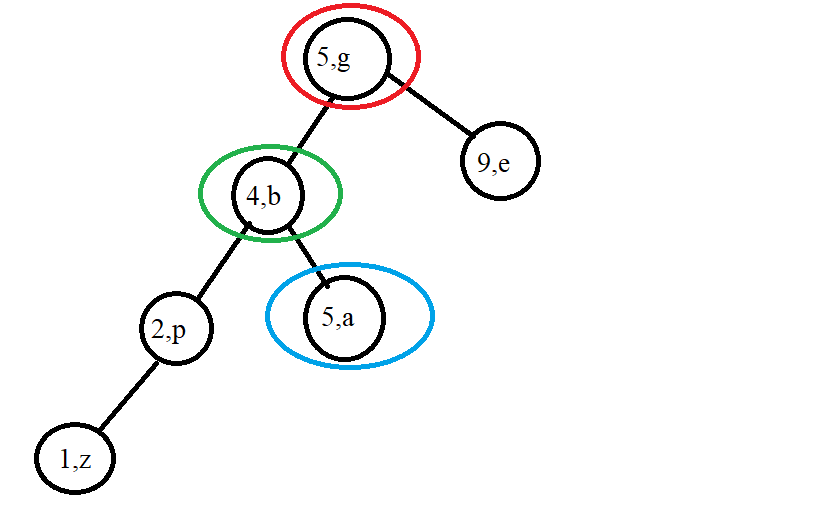


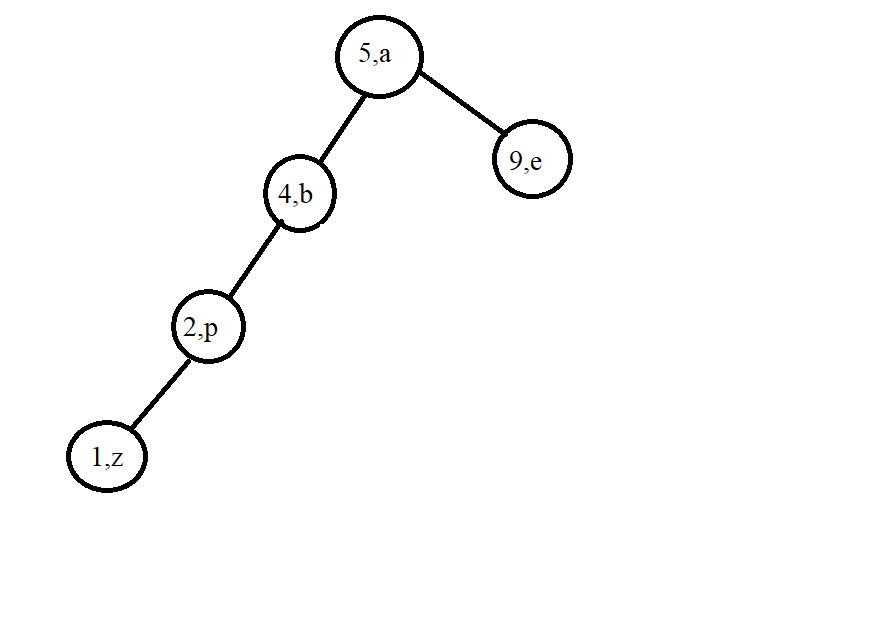
.Find (7,a)



Delete:

.Delete (5,g)





.Delete (2,p)

