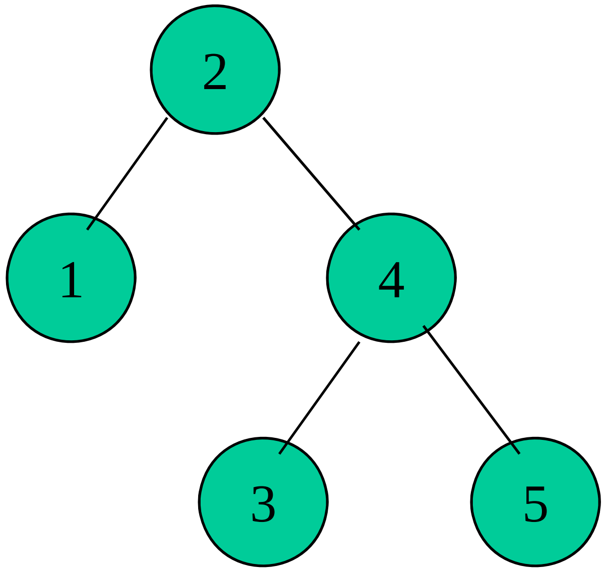
AVL Trees

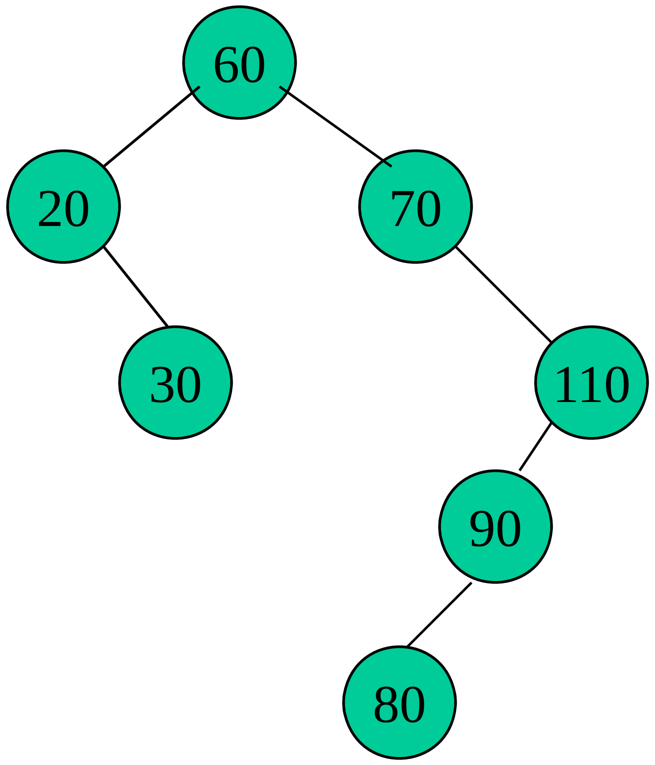
- A binary tree is a **height-balanced-p-tree** if for each node in the tree, the difference in height of its two subtrees is at the most p.
- **AVL tree** is a BST that is height-balanced-1-tree.



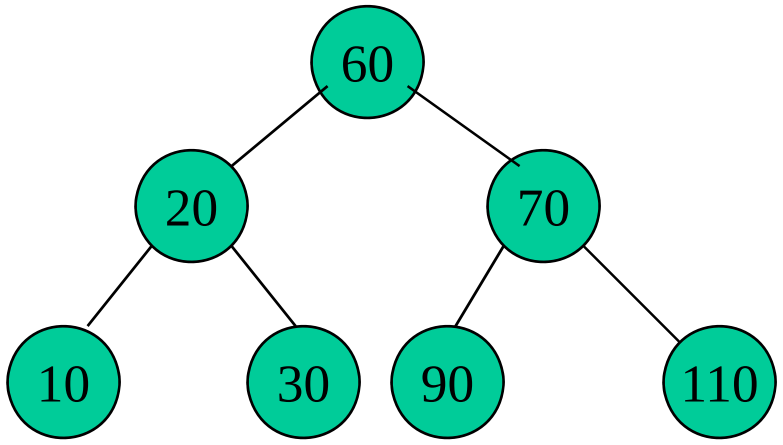
AVL Trees

A binary tree is an AVL tree if

- **Each node satisfies BST property** – key of the node is grater than the key of each node in its left subtree and is smaller than the key of each node in its right subtree
- **Each node satisfies height-balanced-1-tree property** – the difference between the heights of the left subtree and right subtree of the node does not exceed one.

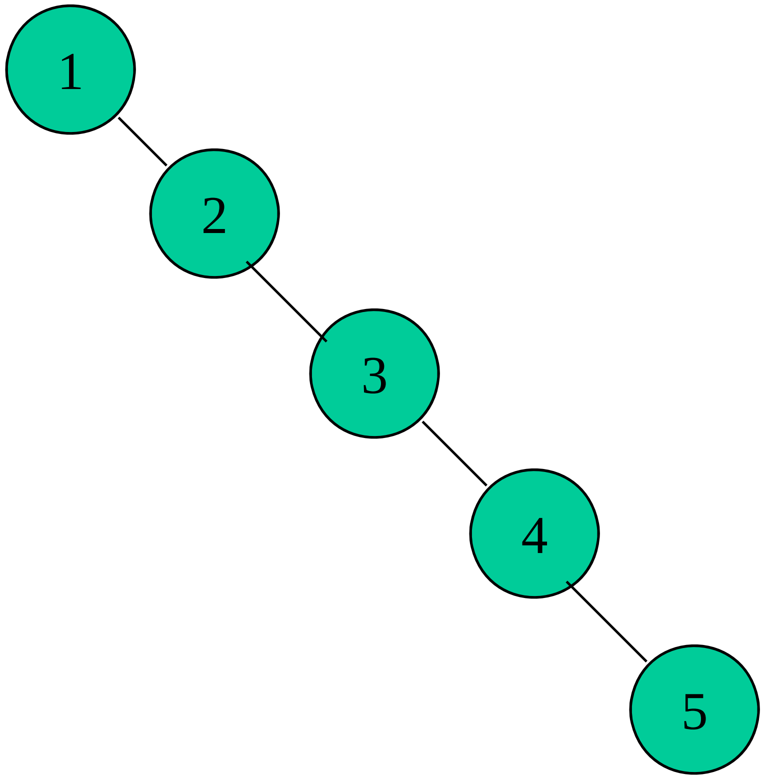


An Imbalanced Tree

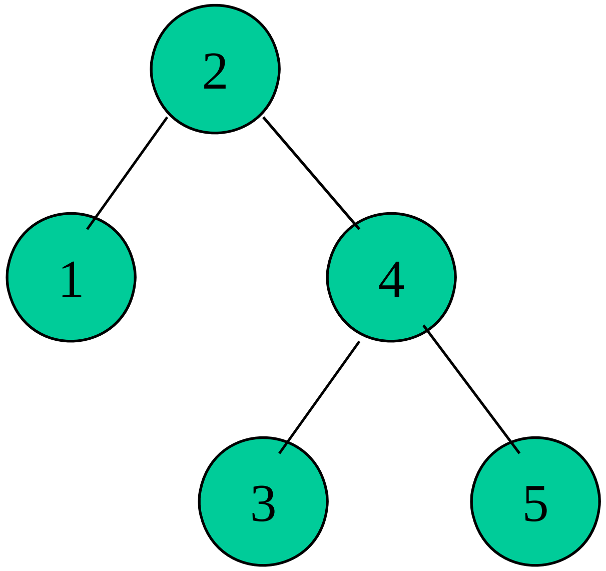


A Balanced Tree

Insert 1, 2, 3, 4 and 5 in the given order



BST after insertions



AVL Tree after insertions

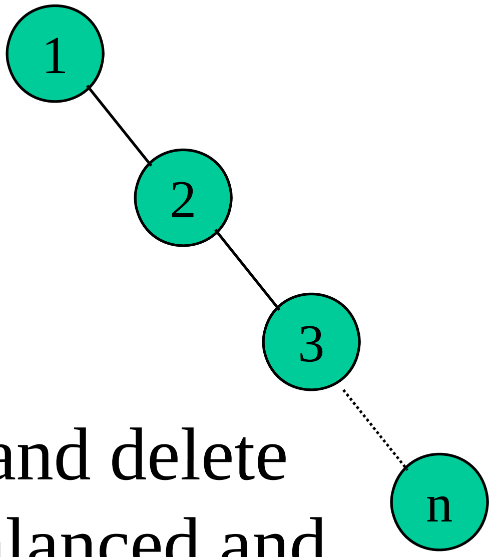
Why AVL Trees ?

- When data elements are inserted in a BST in sorted order: 1, 2, 3, …

  BST becomes a degenerate tree.

  Search operation FindKey takes O(n),

  which is as inefficient as in a list.



- It is possible that after a number of insert and delete operations a binary tree may become imbalanced and increase in height.
- Can we insert and delete elements from BST so that its height is guaranteed to be O(log n)?
- AVL tree ensures this.

Specification of AVL Tree ADT

Elements: Any data type

Structure: A binary tree such that if N is any node in the tree then all nodes in its left subtree are smaller than N, and all nodes in its right subtree are larger than N. The height difference of the two subtrees of any node is at the most one.

Note: A node N is larger than the node M if key value of N is larger than that of M and vice versa.

Domain: Number of elements is bounded

Operations:

Operation	**Specification**
void **empty**()	**Precondition:** none. **Process:** returns true if the AVL has no nodes.

Operation	**Specification**
void **findkey** (int k)	**Precondition:** none. **Process:** searches the AVL for a node whose key = k. If found then that node will be set as the current node and returns true. And if search failed, then (1) AVL empty then just return false; or (2) AVL not empty then current node is the node to which a node containing k would be attached as a child if it were added to AVL and return false.
void **insert**(int k, Type val)	**Precondition:** AVL not full. **Process:** (1) if we already have a node with key = k then current retain its old value (the value prior to calling this operation) and return false; or (2) insert a new node with the given key and data and setting it as current node, return true.
void **update**(Type)	**Precondition/Requires :** AVLis not empty. **Process:** update the value of data of the current node, key remains unchanged.
Type **retrieve**()	**Precondition:** AVL is not empty**.** **Process:** returns data of the current node..
bool **remove_key**(int k)	**Precondition:** none. **Process:** removes the node containing the key = k, and in case BST is not empty then sets the root as the current node. Returns true or false depending on whether the operation was successful or not.

Representation of AVL Tree ADT

Since AVL is a special binary tree, it can be represented using

- Linked List

Note : Array is not suitable for AVL

Implementation of AVL

```
public class AVLNode <T> {
    public int key, bal;
    public T data;
    public AVLNode<T> left, right;

    public AVLNode(int k, T val) {
        key = k; bal = 0;
        data = val;
        left = right = null;
    }
    public AVLNode (int k, AVLNode<T> l, AVLNode<T> r) {
        key = k; bal = 0;
        left = l;
        right = r;
    }
}
```

Implementation of AVL

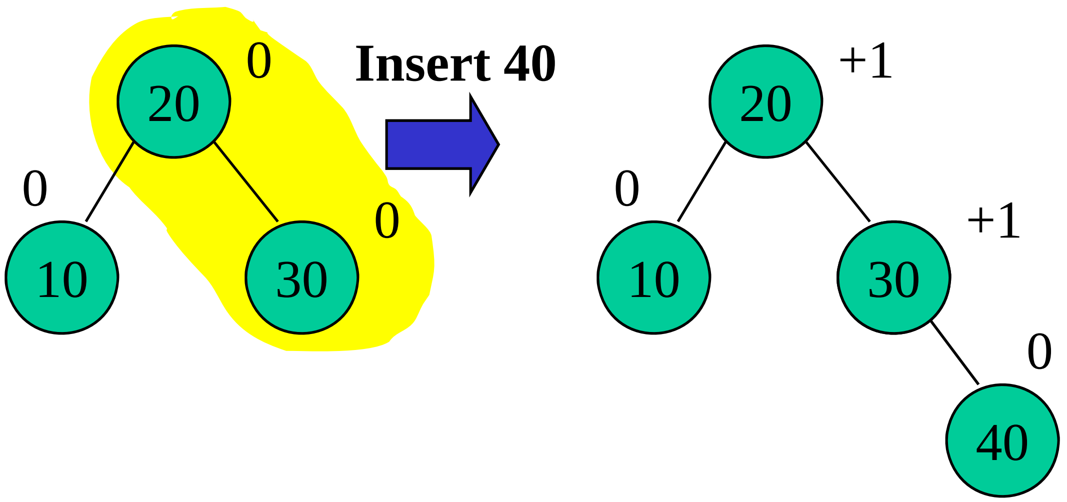
```
public class Flag {
    boolean value;
    /** Creates a new instance of Flag */
    public Flag() {
        value = false;
    }
    public Flag(boolean v){
        value = v;
    }
    public boolean get_value (){
        return value;
    }
    public void set_value(boolean v){
        value = v;
    }
}
```

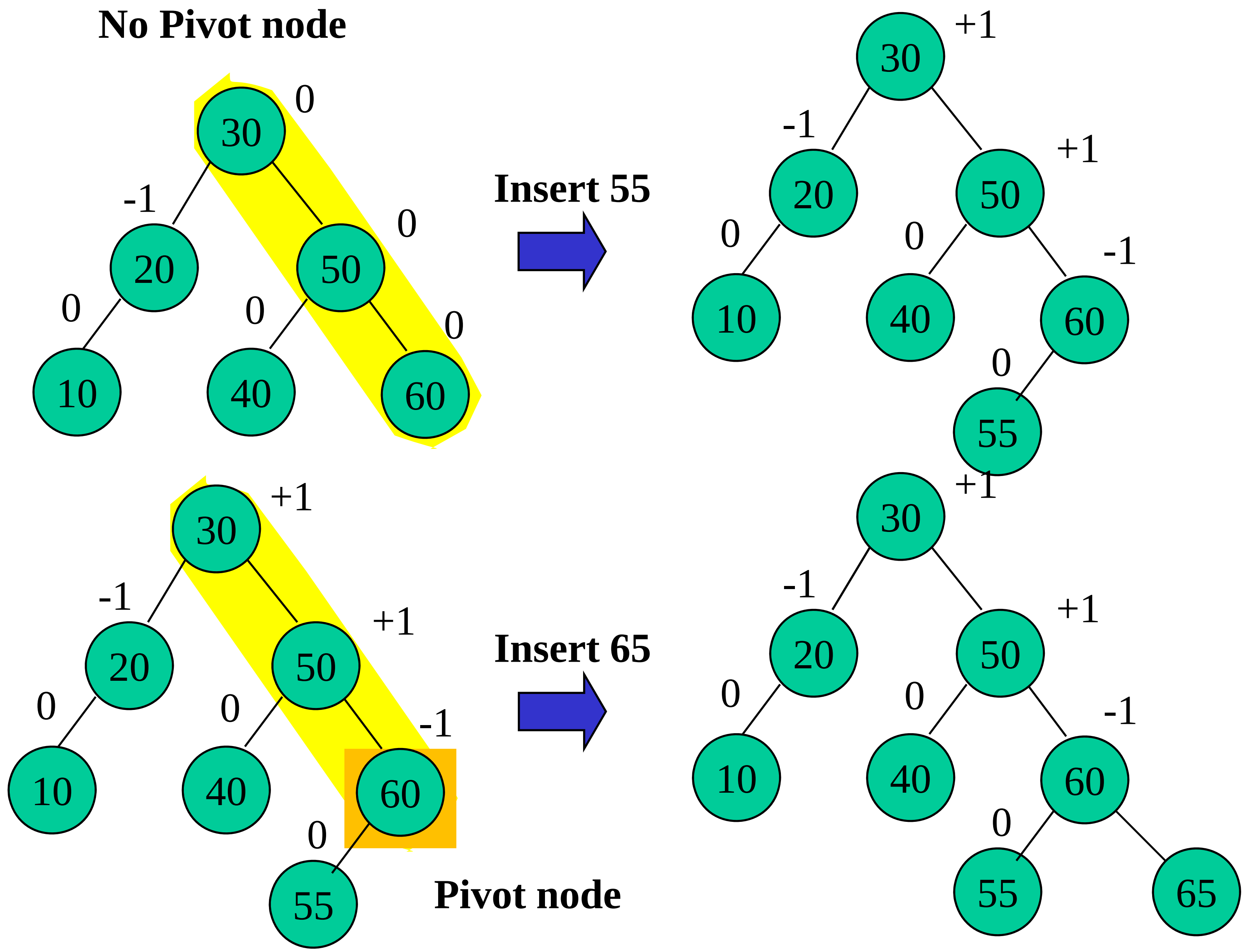
Implementation of AVL

```
public class AVL <T> {
    AVLNode<T> root, current;
    public BST()

    private AVLNode<T> findparent (AVLNode<T> p)
    private AVLNode<T> find_min(AVLNode<T> p)
    private AVLNode<T> remove_aux(int key, AVLNode<T> p, Flag
    flag)

    public boolean empty()
    public T retrieve ()
    public boolean findkey(int tkey)
    public boolean insert (int k, T val)
    public boolean remove_key (int tkey)
    public boolean update(int key, T data)
}
```

Note

- There is always a unique path from the root to the new node. It is called the **search path**.
- A **pivot node** is a node closest to the new node on the search path, whose balance factor is either –1 or +1.




No Pivot node
Insert 55
Insert 65
Pivot node
30
20
50
10
40
60
55
65
0
-1
+1

- If, after an insert operation or a delete operation, an AVL tree becomes imbalanced, adjustments must be made to the tree to change it back into an AVL tree.
- These adjustments are called **rotations**.
- Rotations are either **single** or **double** rotations.

Insert Operation

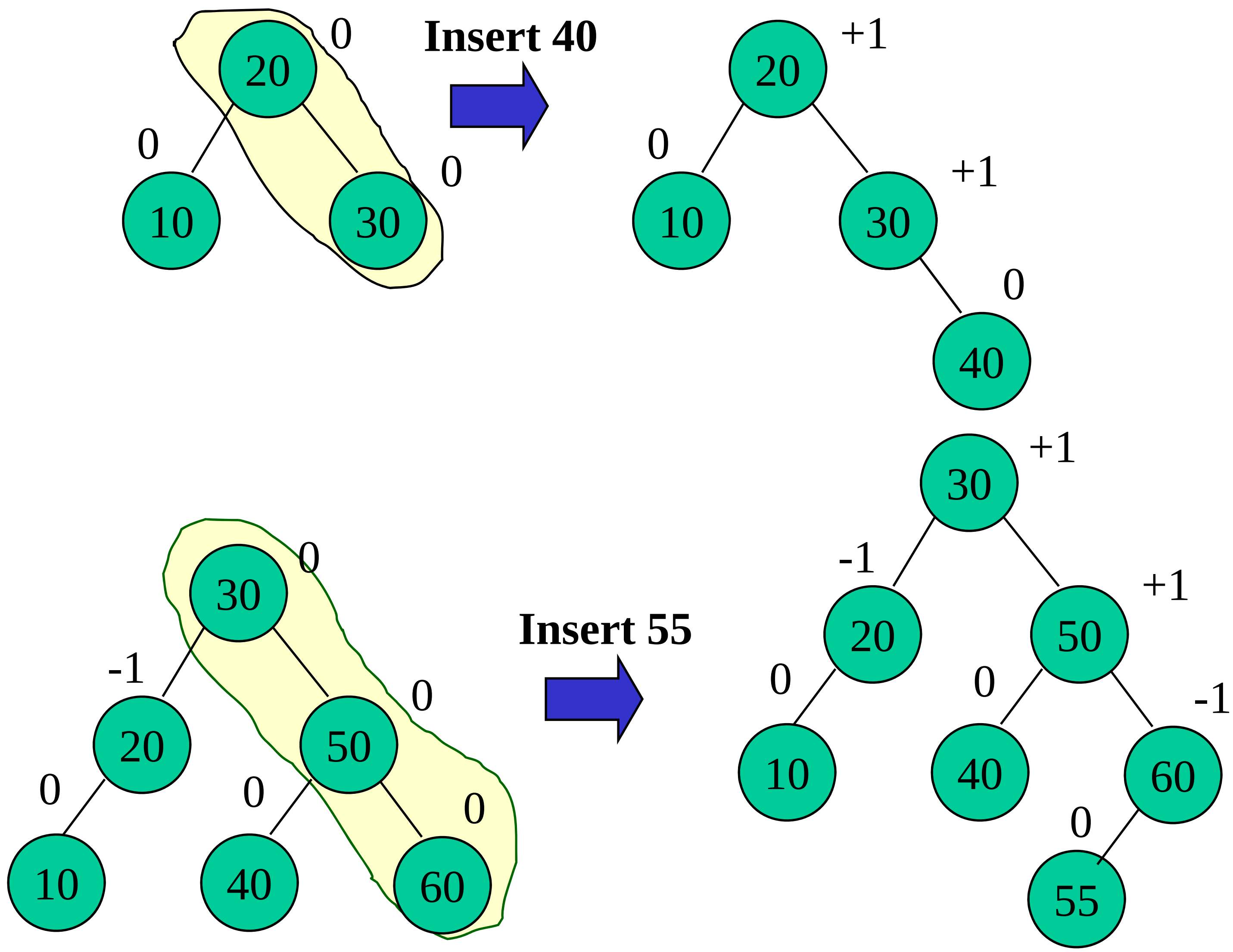
- **Step 1:** Insert a node into the AVL tree as it is inserted in a BST
- **Step 2:** Examine the search path to see if there is a pivot node. Three cases may arise.

Case 1: There is no pivot node. No adjustment required.

Case 2: The pivot node exists and the subtree of the pivot node to which the new node is added has smaller height. No adjustment required

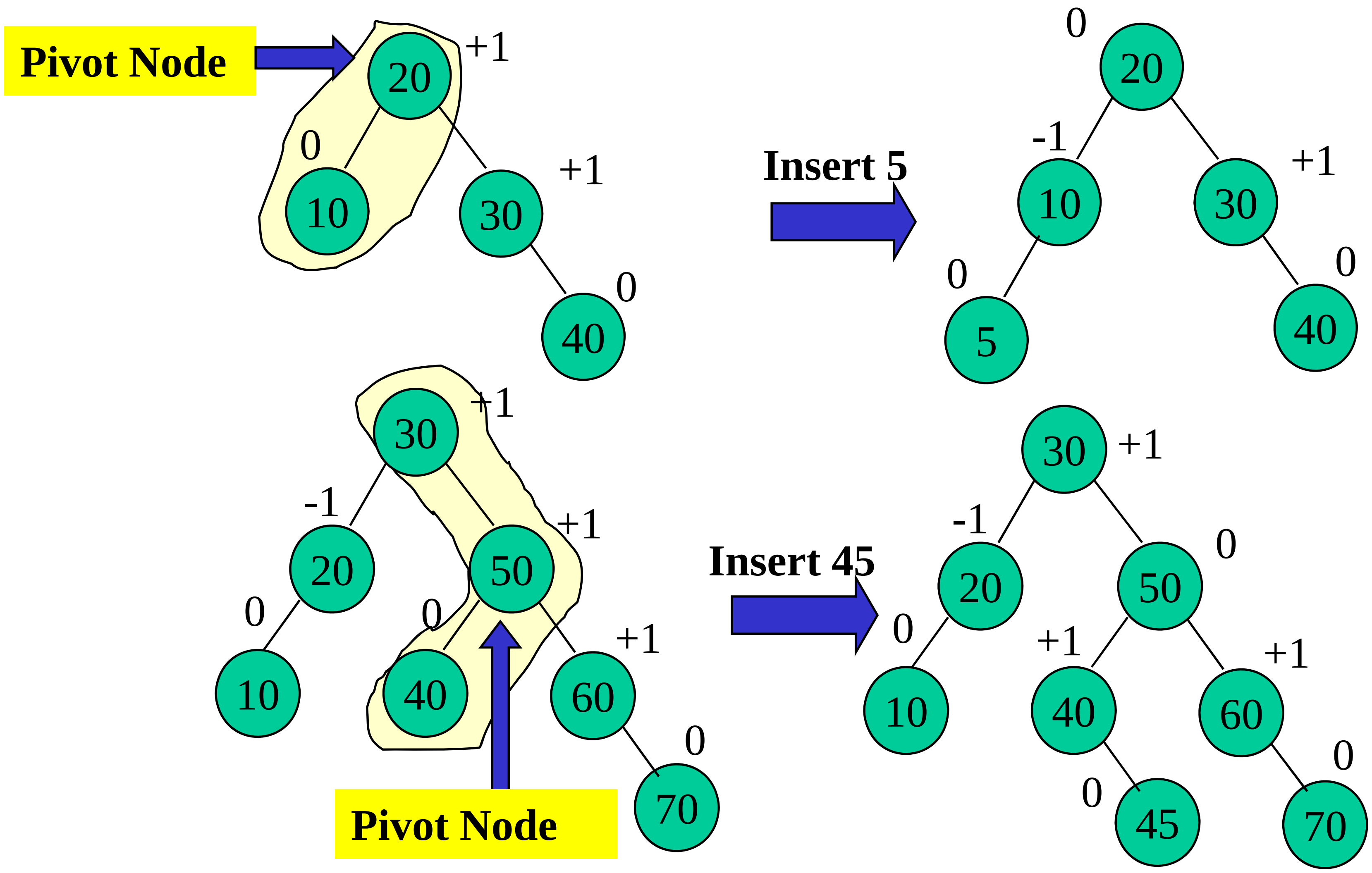
Case 3: The pivot node exists and the subtree to which the new node is added has the larger height. Adjustment required.

Insert: Case 1 – No pivot node



Insert: Case 2 – Pivot node exits

New node added to the shorter subtree of the Pivot node .



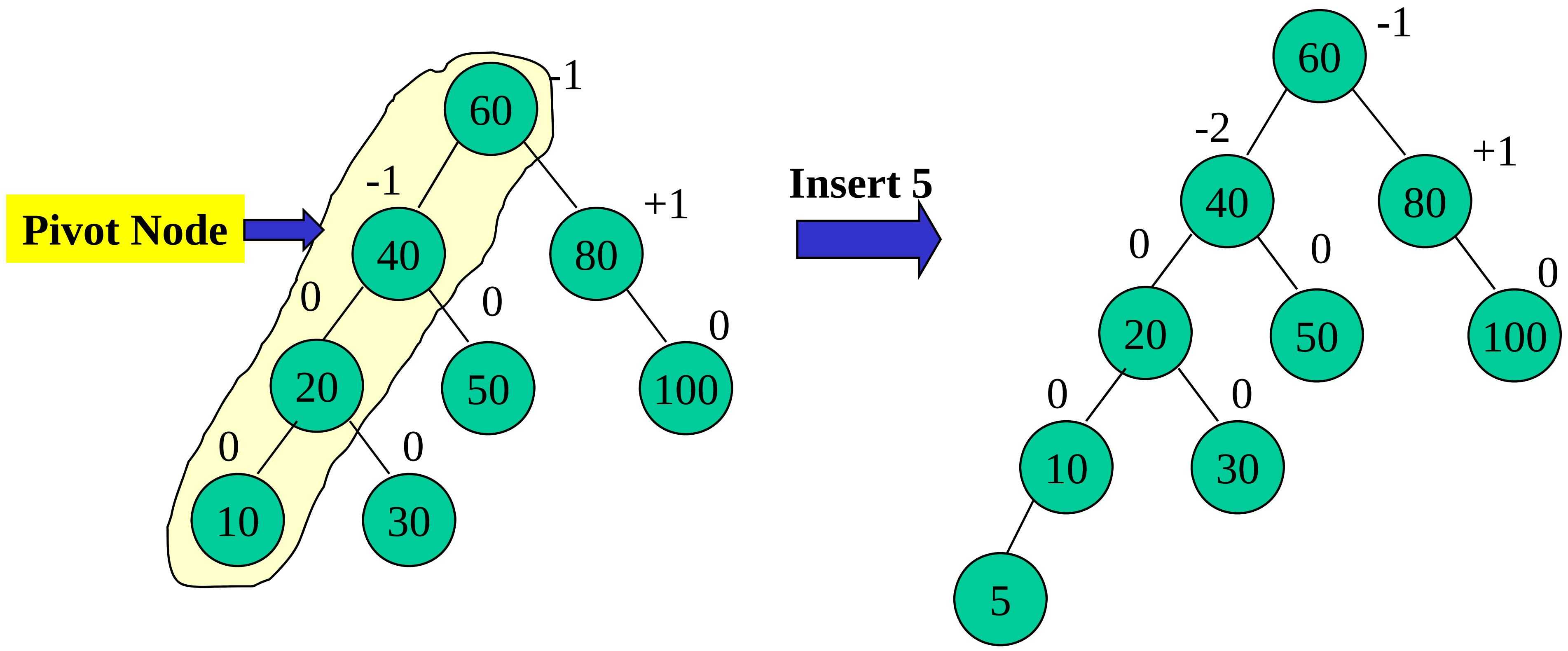
Note

In this case

- If balance factor of the pivot node is +1, the new node is inserted in the left subtree
- If balance factor of the pivot node is -1, the new node is inserted in the right subtree

Insert: Case 3 – Pivot node exits

New node added to the longer subtree of the Pivot node.



Note: AVL Tree is no more an AVL Tree after insertion.

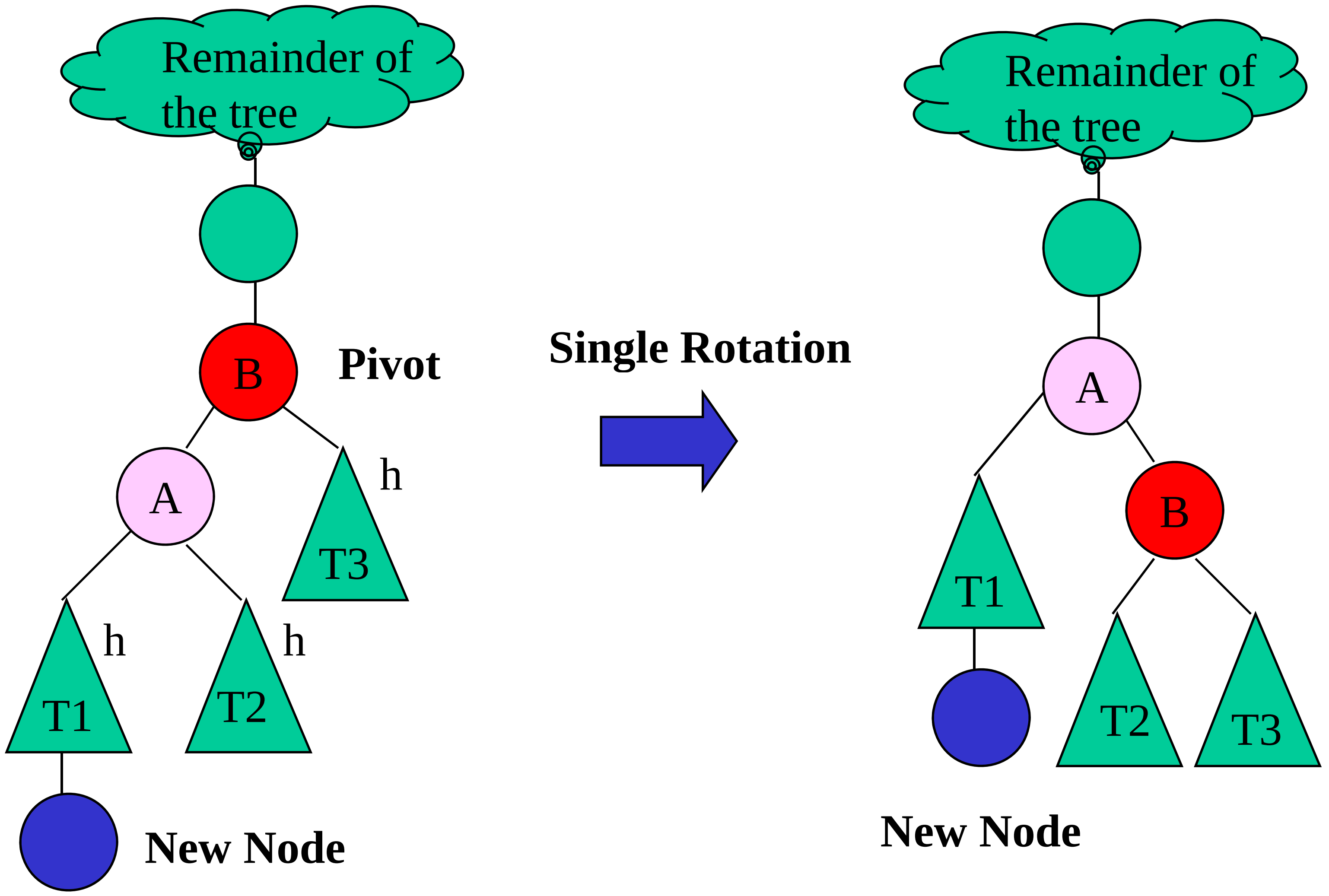
Note

Case 3 has further 4 cases:

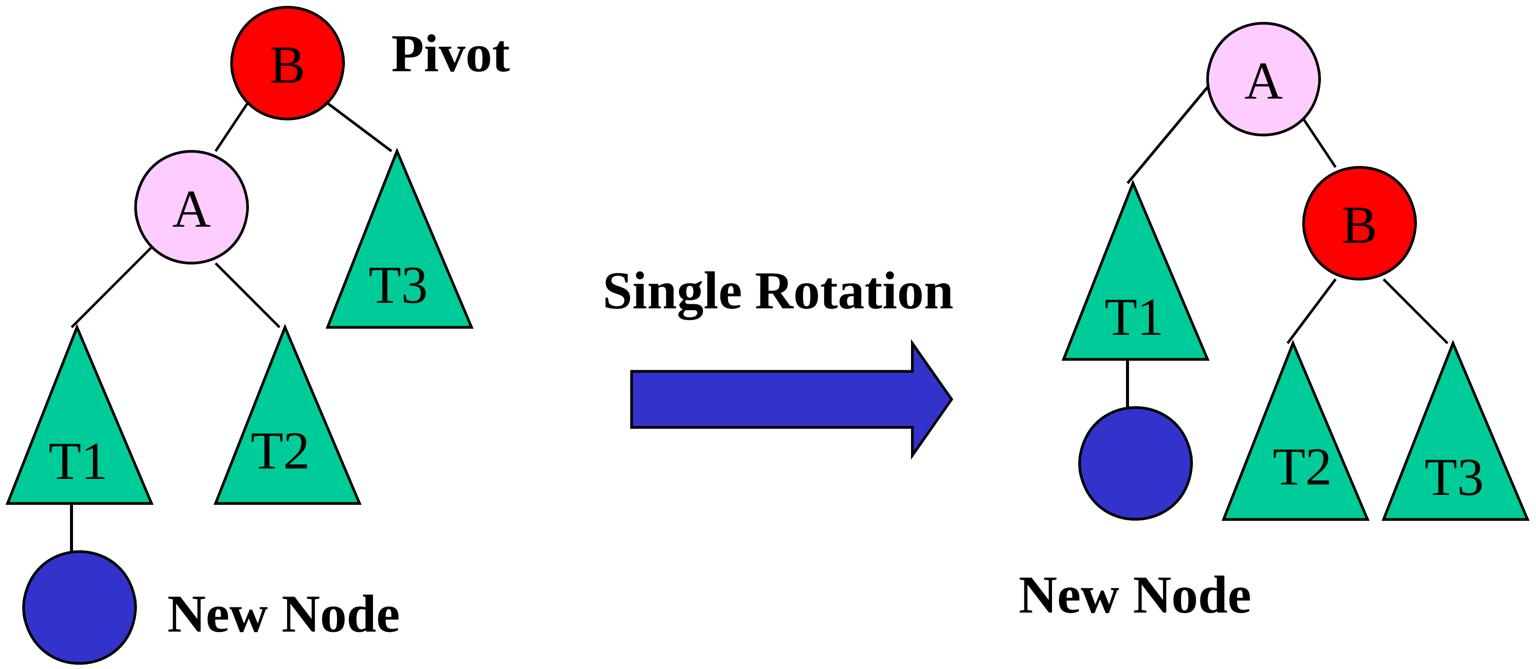
Left-left case:	An insertion in the left subtree of the left child of the pivot node
Right-left case:	An insertion in the right subtree of the left child of pivot node
Left-right case:	An insertion in the left subtree of the right child of pivot node
Right-right case:	An insertion in the right subtree of the right child of pivot node

- In the cases **left-left** and **right-right**, **single rotation** is applied after new node is inserted.
- In the cases **right-left** and **right-left**, **double rotation** is applied after new node is inserted.

Insert: Left-left subcase

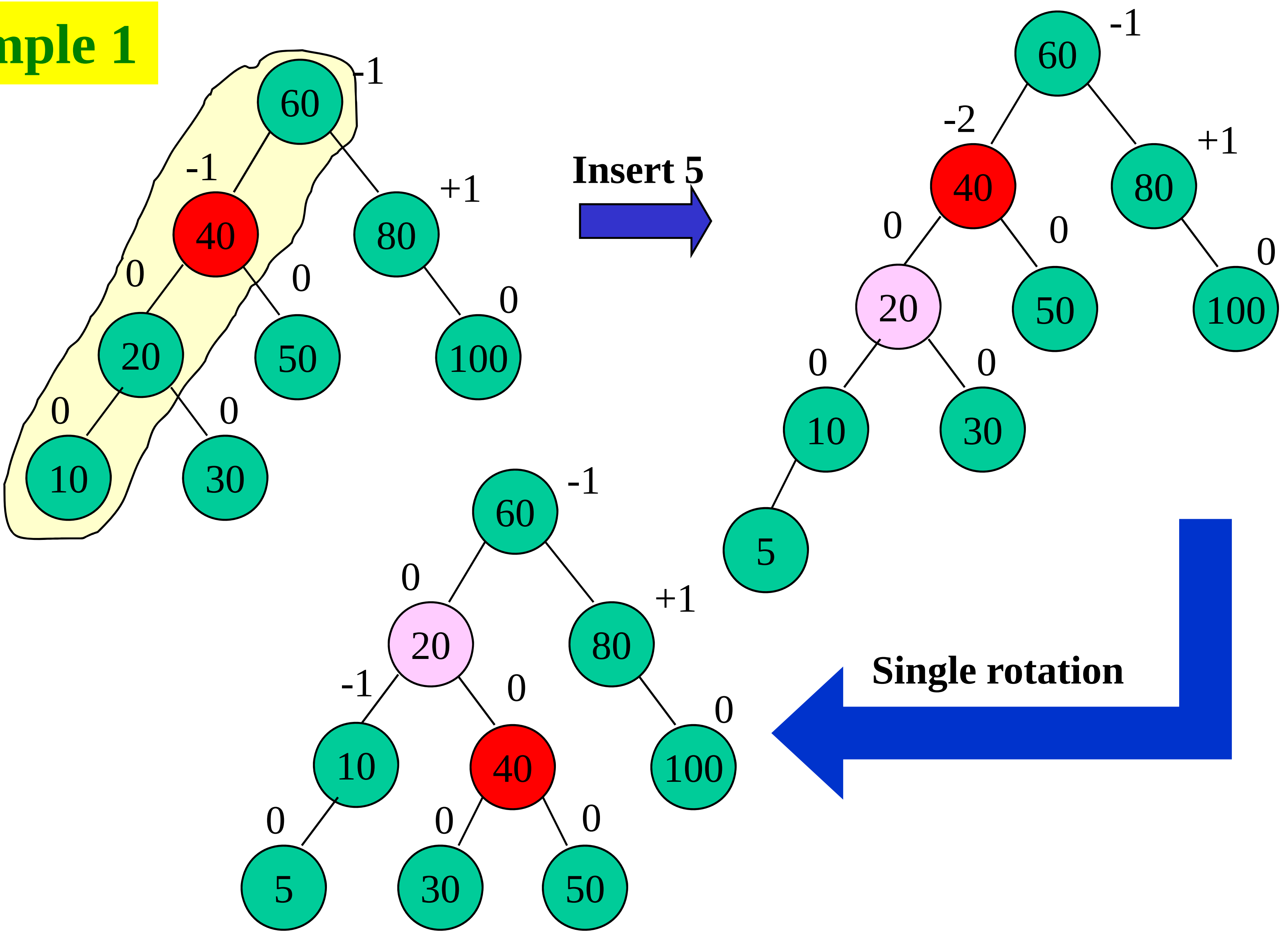


Note: Rotation is applied about pivot node.

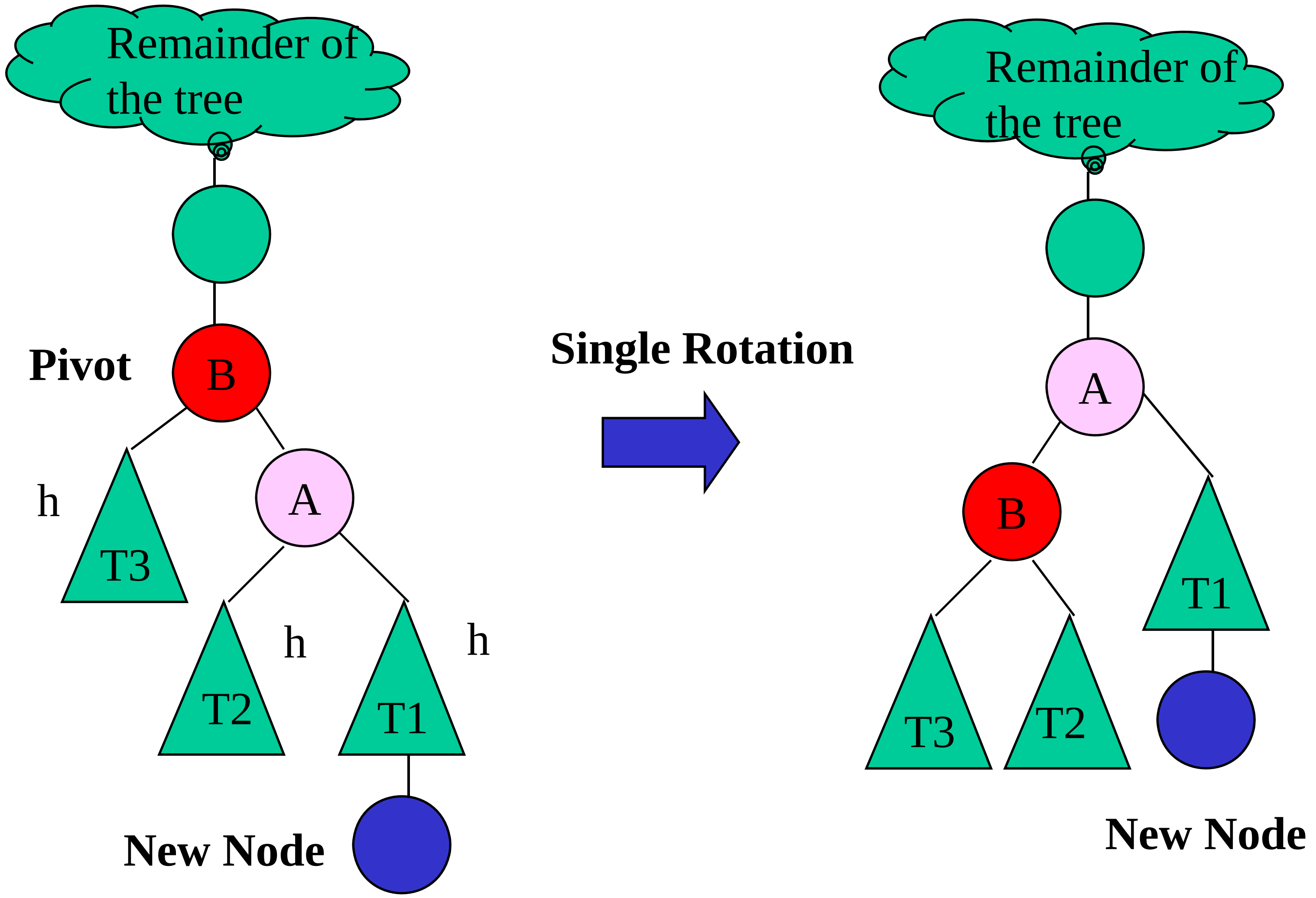


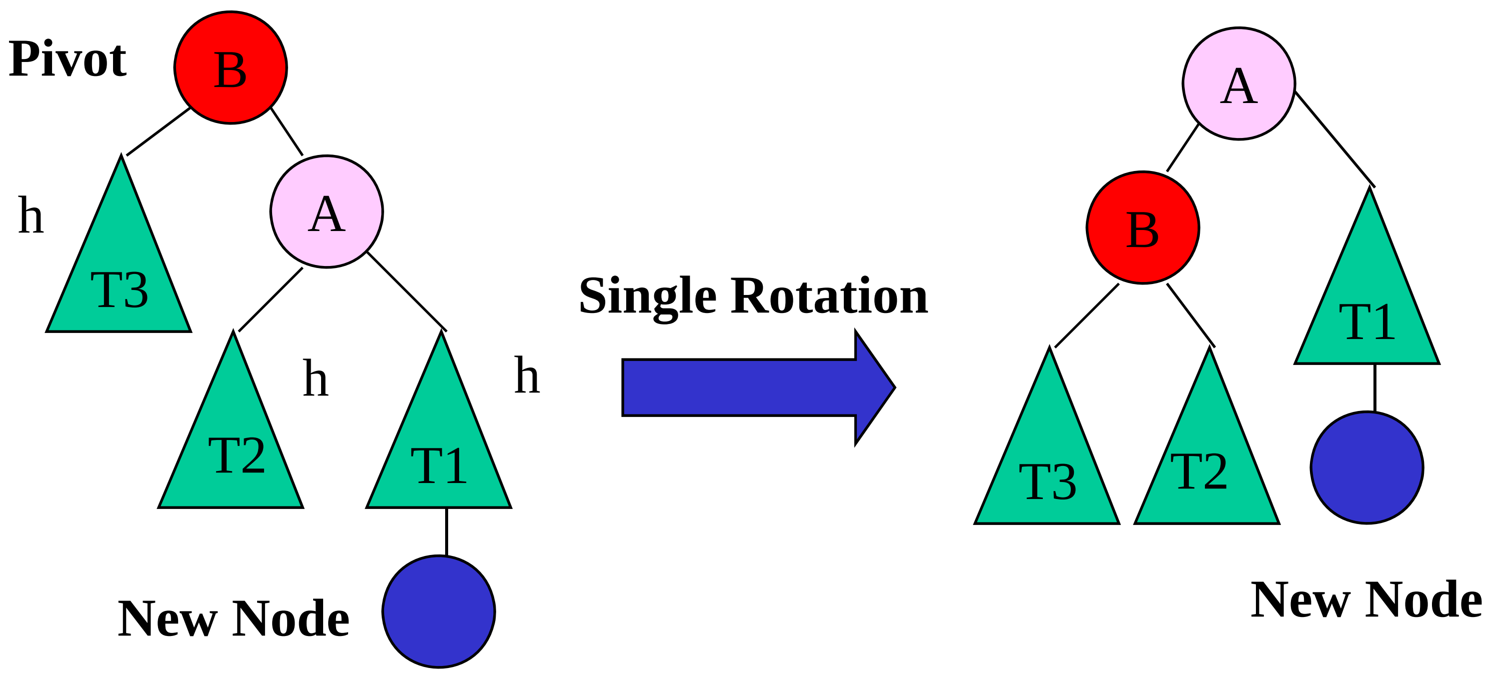
```
public void rotateLeftChild(AVLNode<T> B)
{
       AVLNode<T> A = B.left;
       B.left = A.right;
       A.right = B;
       B = A;
}
```

Example 1


60
-1
40
-1
80
+1
20
0
50
0
100
0
10
0
30
0
Insert 5
60
-1
40
-2
80
+1
20
0
50
0
100
0
10
0
30
0
5
Single rotation
60
-1
20
0
80
+1
10
-1
40
0
100
0
5
0
30
0
50
0

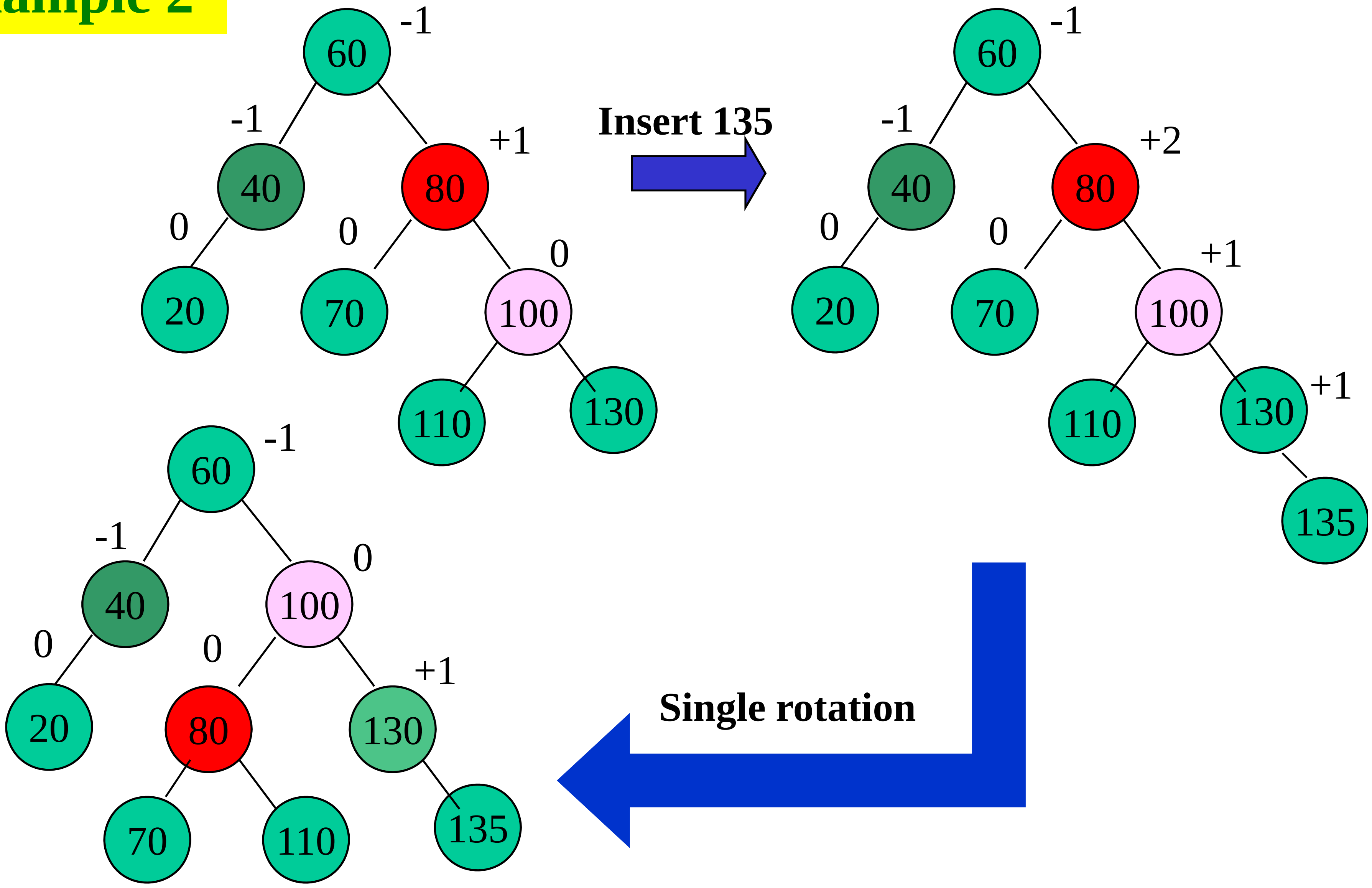
Insert: Right-right subcase



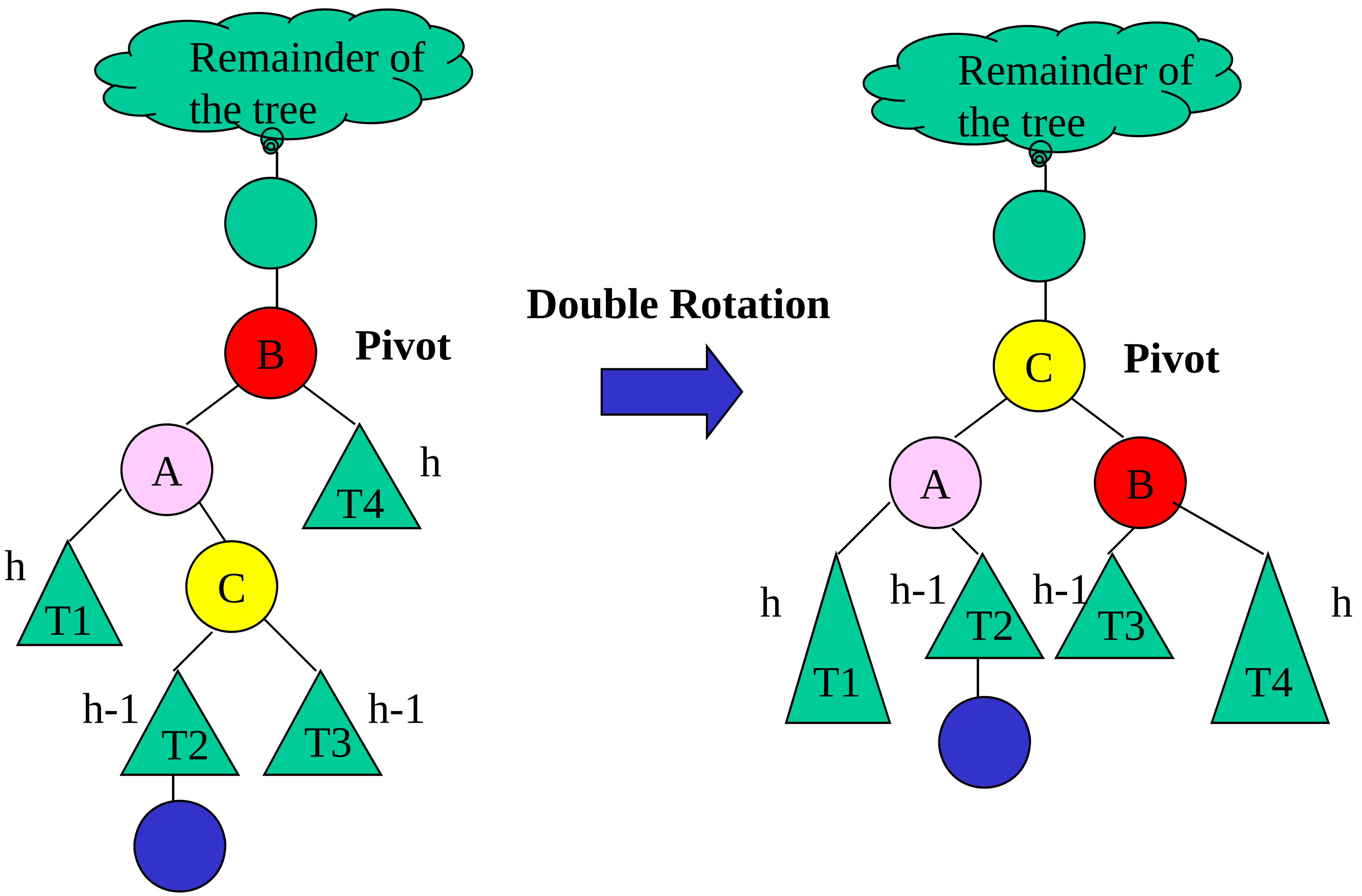


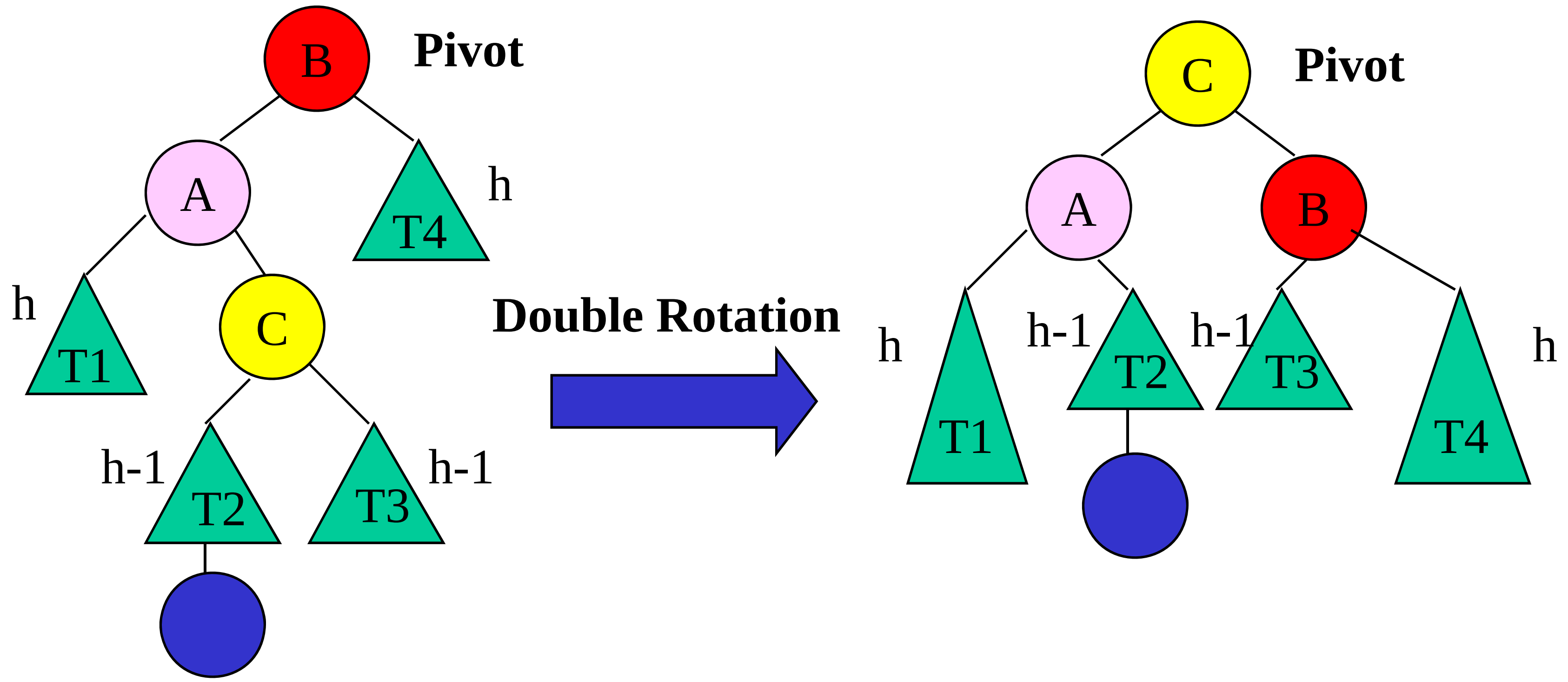
```
public void rotateRightChild(AVLNode<T> B)
{
       AVLNode<T> A = B.right;
       B.right = A.left;
       A.left = B;
       B = A;
}
```

Example 2


60
-1
40
-1
80
+1
20
0
70
0
100
0
110
130
Insert 135
60
-1
40
-1
80
+2
20
0
70
0
100
+1
110
130
+1
135
Single rotation
60
-1
40
-1
100
0
20
0
80
0
130
+1
70
110
135

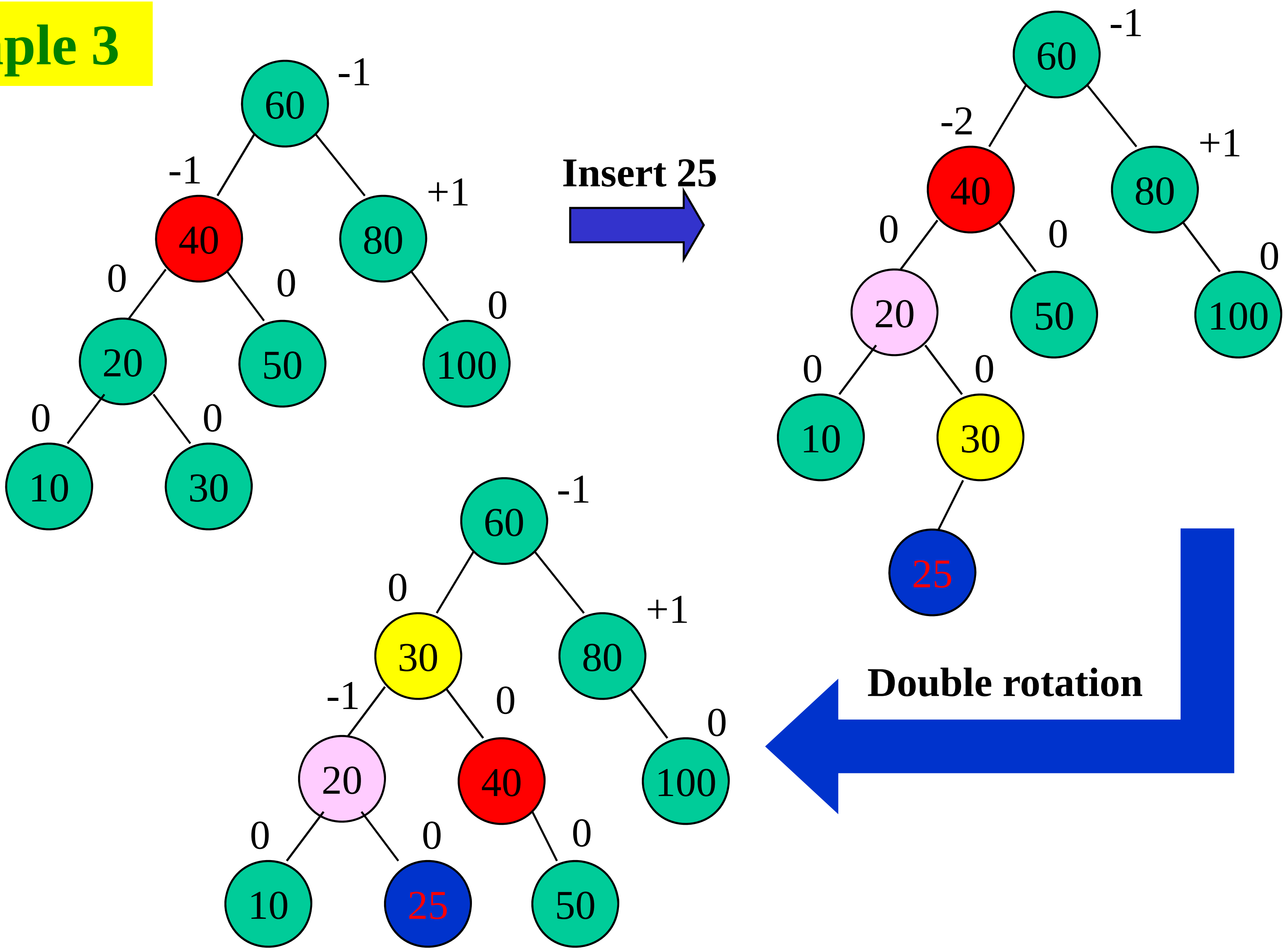
Insert: Left-right subcase



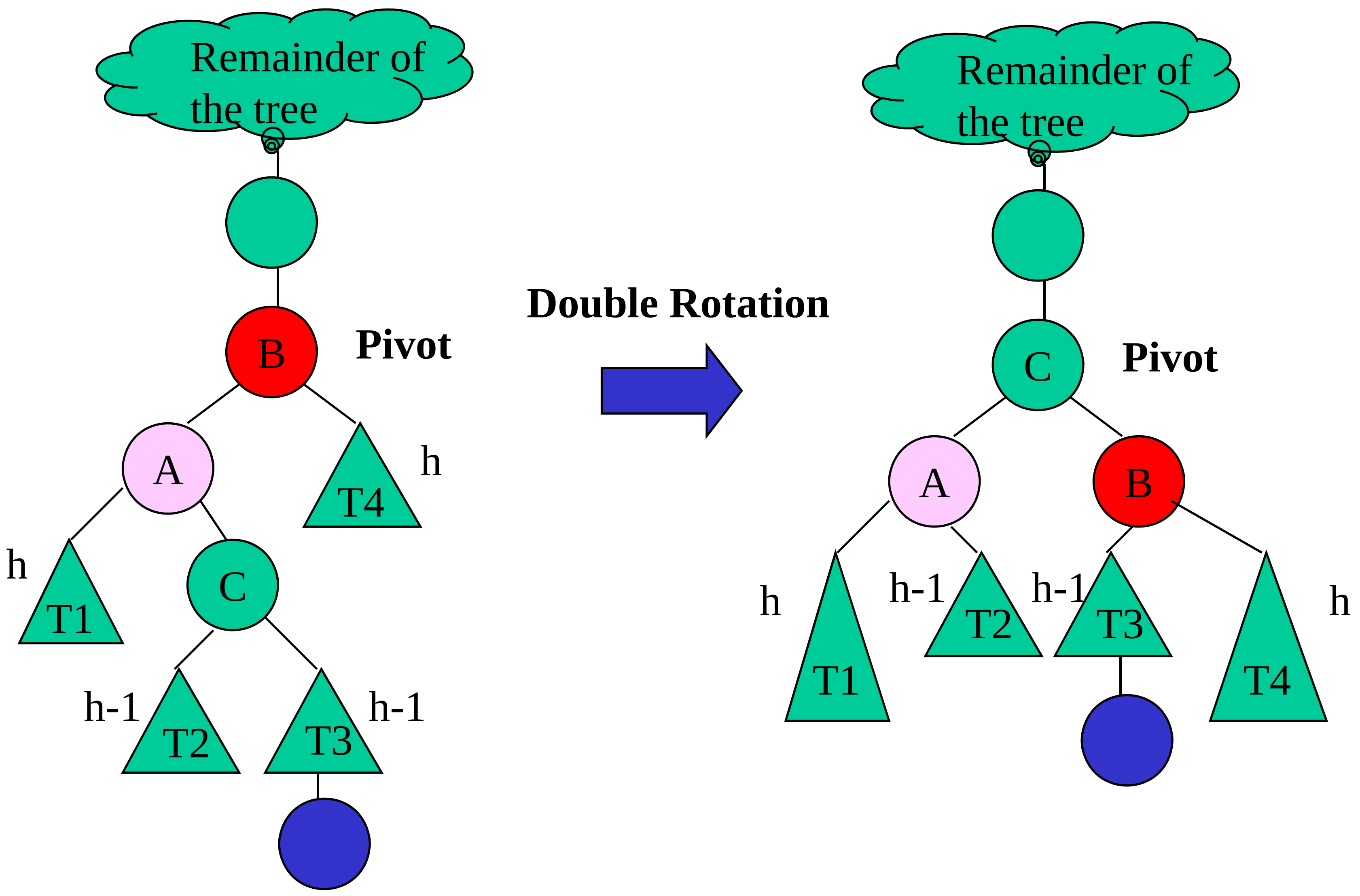


```
public void doubleRotateleftChild(AVLNode<T> B)
{
        rotateRightChild(B.left);
        rotateleftChild(B);
}
```

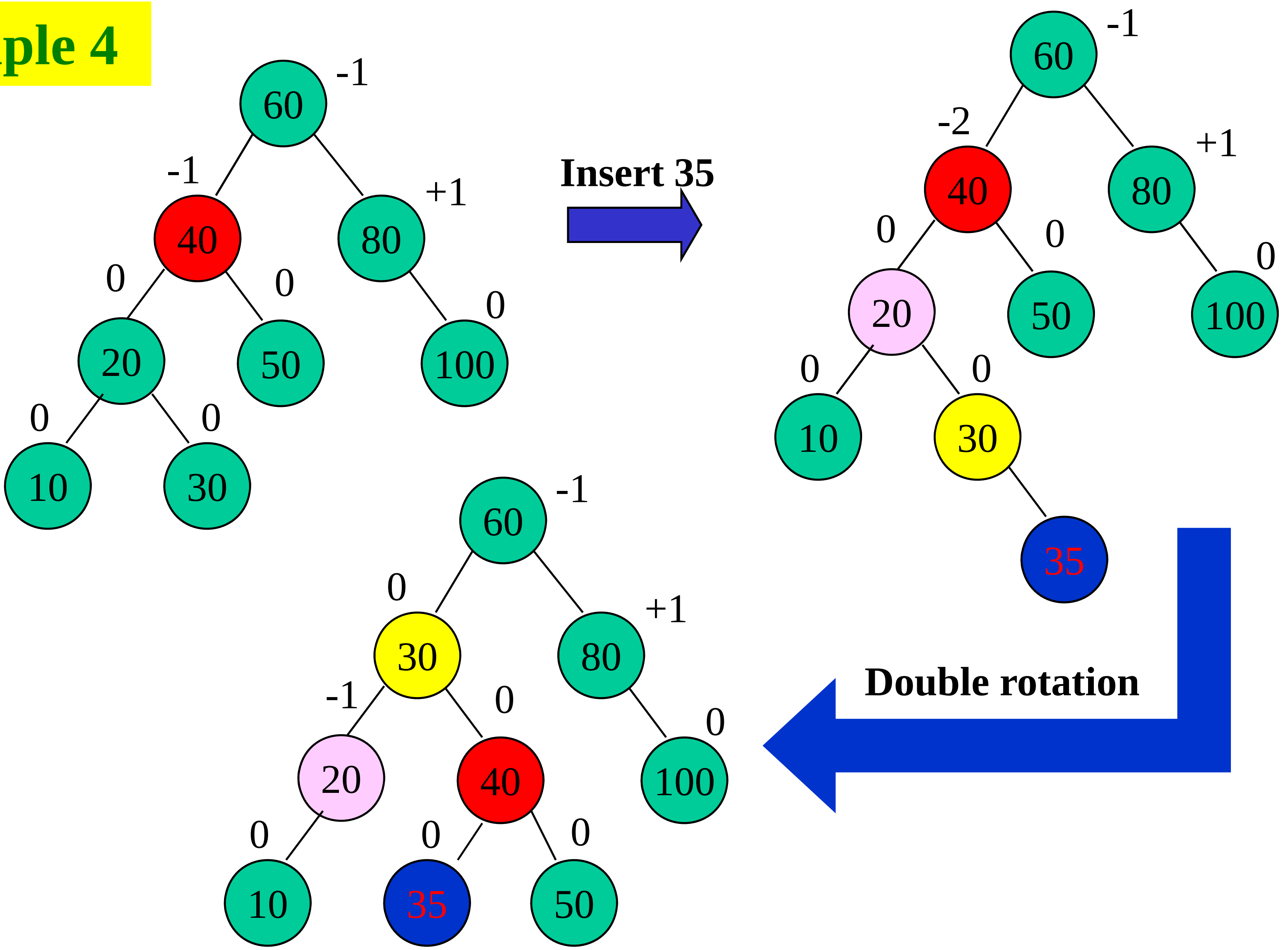
Example 3


60
-1
40
-1
80
+1
20
0
50
0
100
0
10
0
30
0
Insert 25
60
-1
40
-2
80
+1
20
0
50
0
100
0
10
0
30
0
25
Double rotation
60
-1
30
0
80
+1
20
-1
40
0
100
0
10
0
25
0
50
0

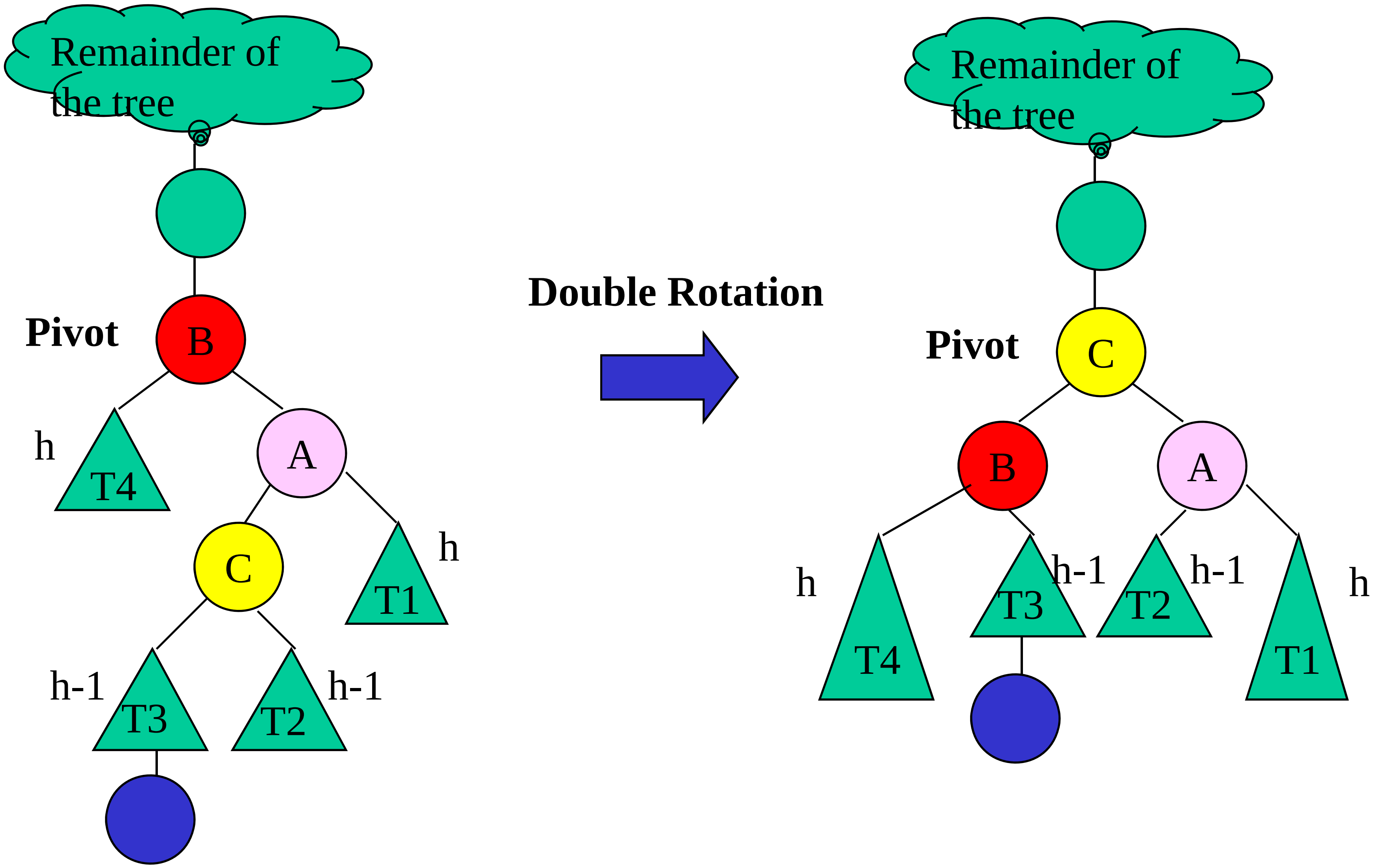
Insert: Left-right subcase

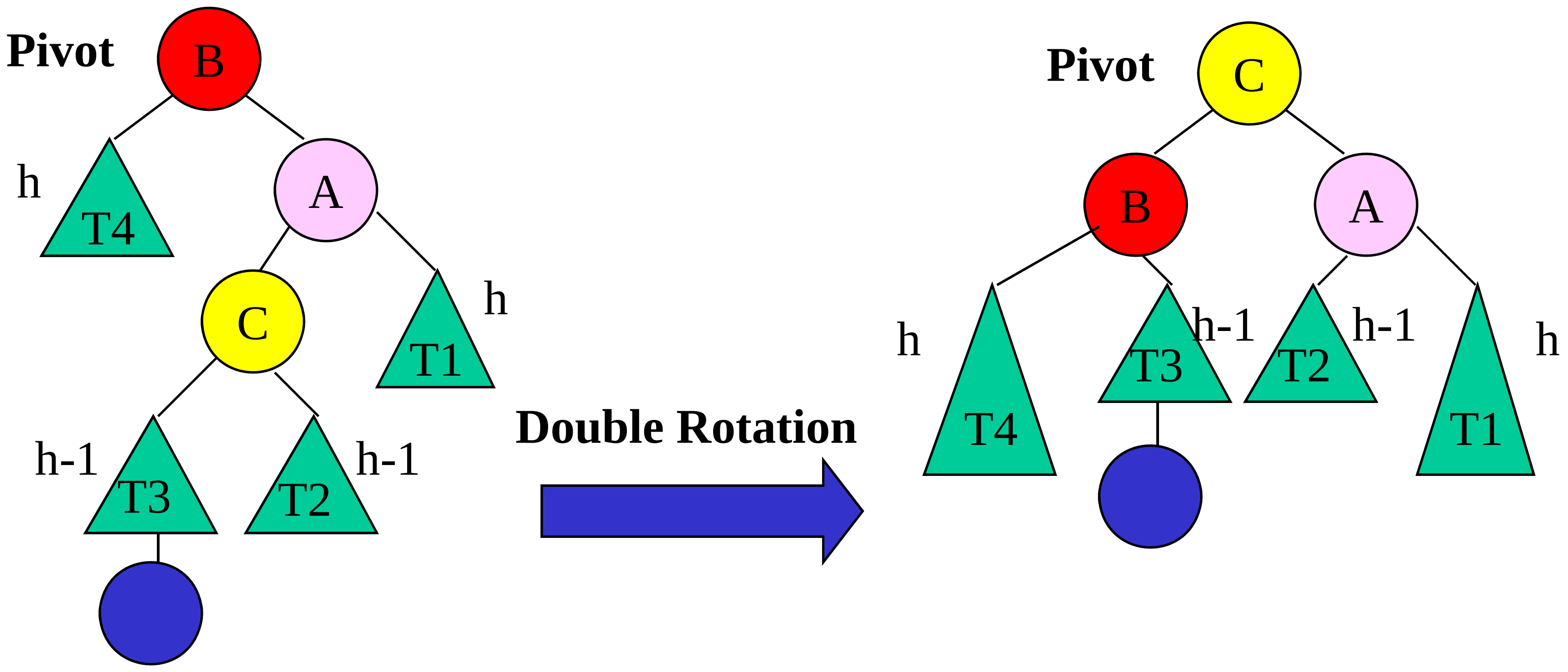


Example 4


60
-1
40
-1
80
+1
20
0
50
0
100
0
10
0
30
0
Insert 35
60
-1
40
-2
80
+1
20
0
50
0
100
0
10
0
30
0
35
Double rotation
60
-1
30
0
80
+1
20
-1
40
0
100
0
10
0
35
0
50
0

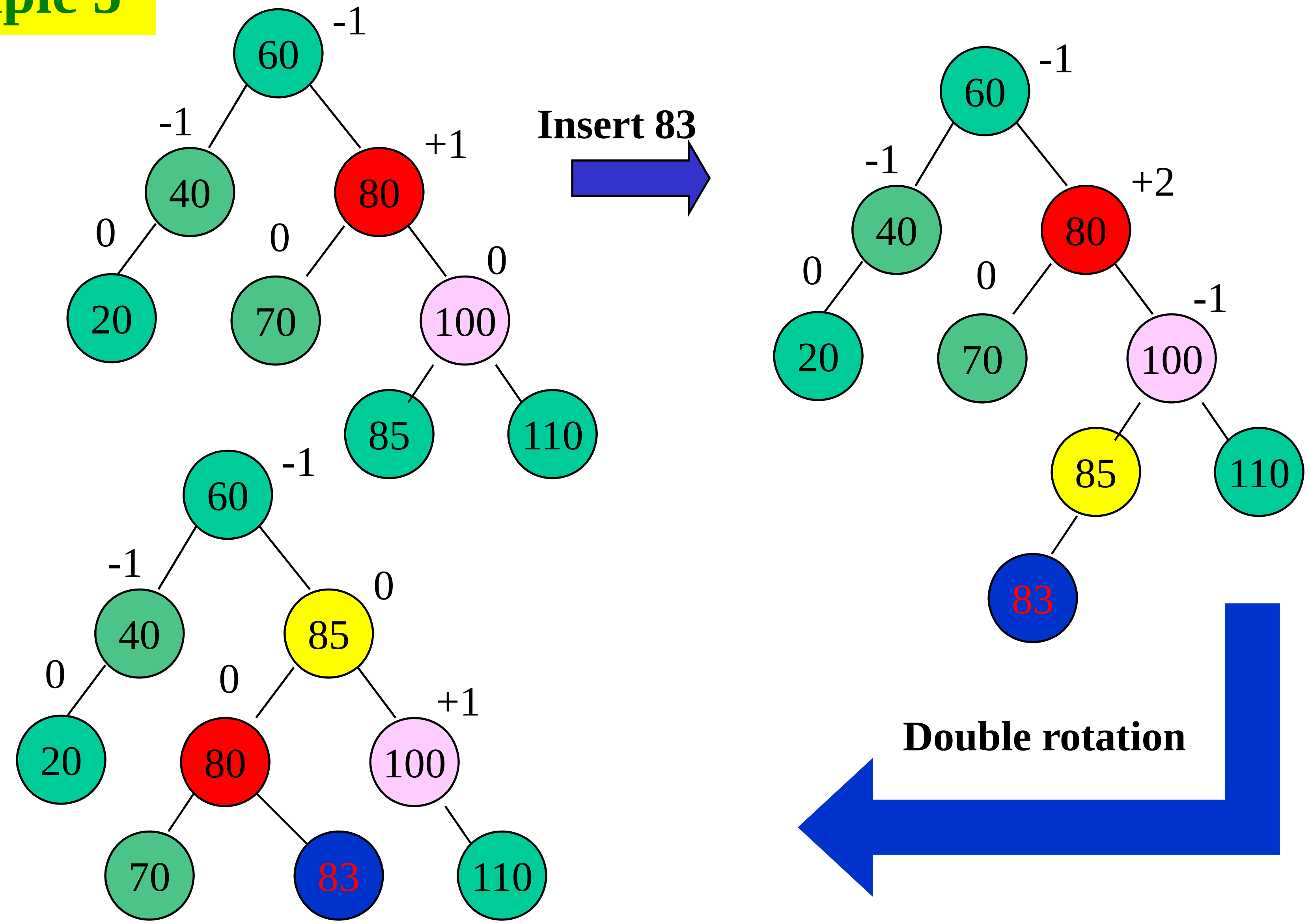
Insert: Right- Left subcase



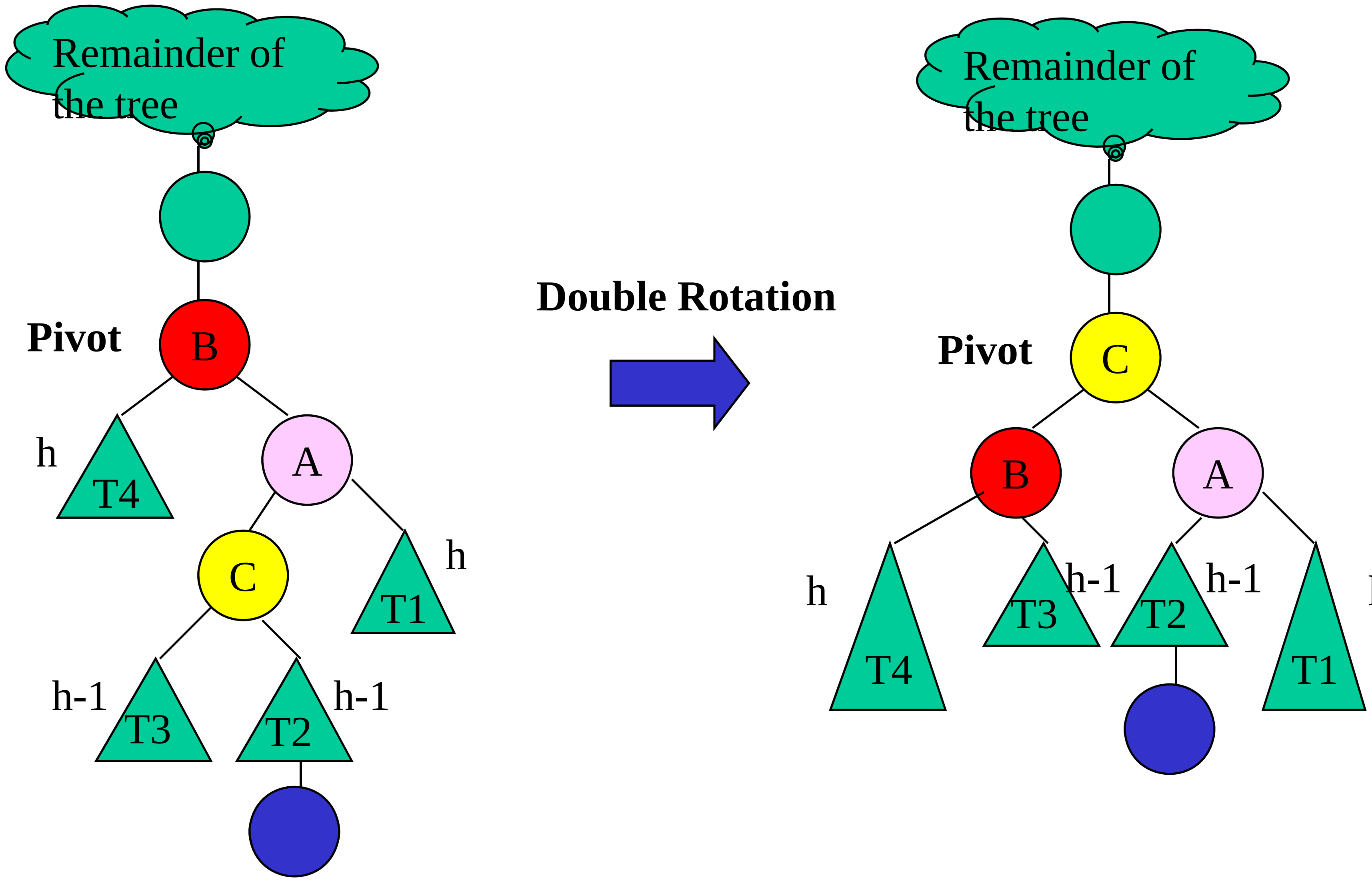


```
public void doubleRotateRightChild(AVLNode<T> B)
{
       rotateLeftChild(B.rihgt);
       rotatRightChild(B);
}
```

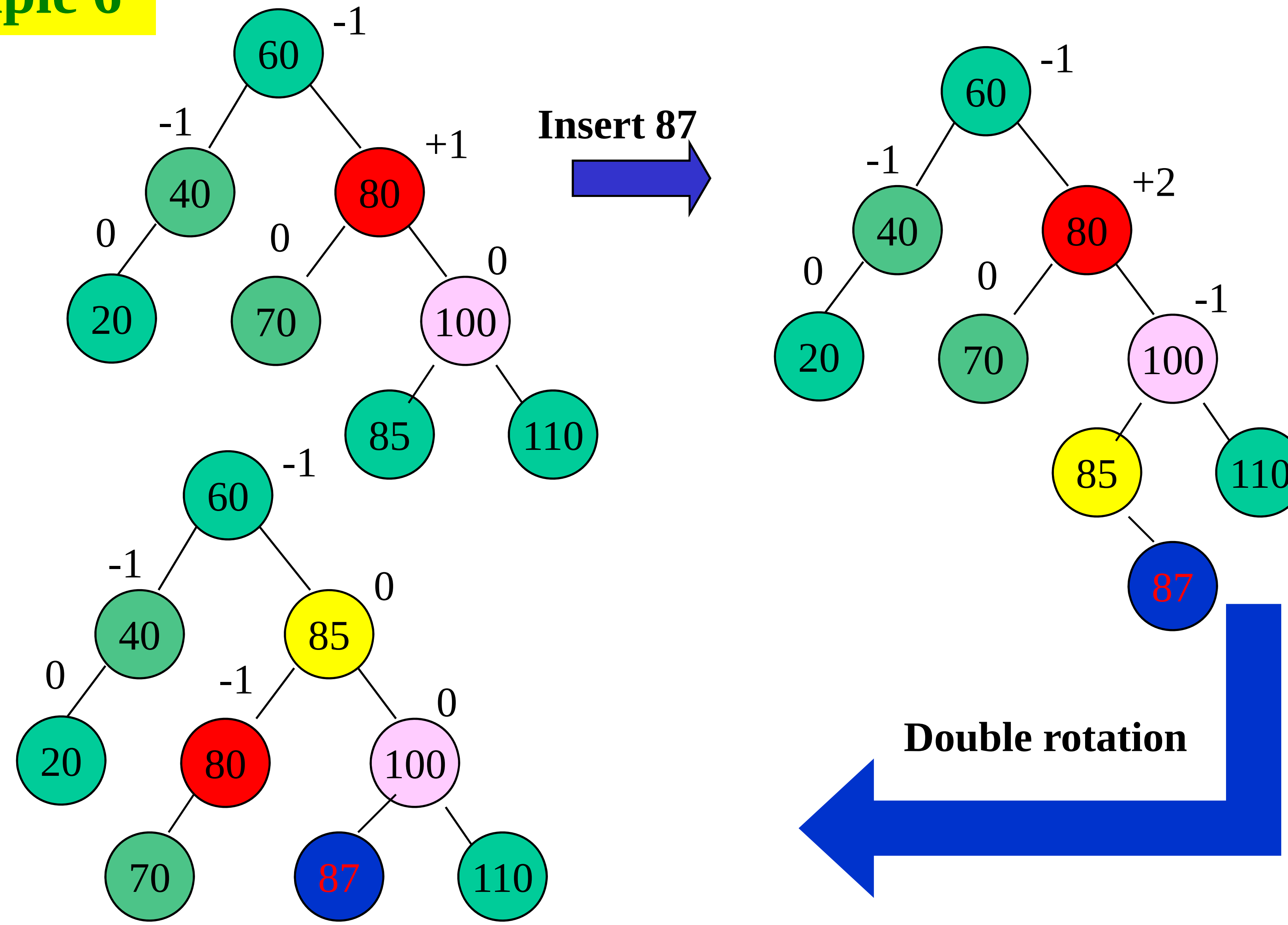
Example 5


Insert 83
Double rotation
60 -1
40 -1
80 +1
20 0
70 0
100 0
85
110
60 -1
40 -1
80 +2
20 0
70 0
100 -1
85
110
83
60 -1
40 -1
85 0
20 0
80 0
100 +1
70
83
110

Insert: Right- Left subcase



Example 6



60
-1
40
-1
80
+1
20
0
70
0
100
0
85
110
Insert 87
60
-1
40
-1
80
+2
20
0
70
0
100
-1
85
110
87
Double rotation
60
-1
40
-1
85
0
20
0
80
-1
100
0
70
87
110

Delete Operation

- **Step 1**: Delete the node as in BST. Leaf node will always be deleted.
- **Step 2**: Check each node **p** on the path from the root to the parent node **W** of the deleted node. Start from the node on the path closest to **W.** Three cases are possible.

Case 1: Node **p** has balance factor 0. No rotation needed.

Case 2: Node **p** has balance factor of +1 (or –1) and a node was deleted from right sub-tree (or left subtree). No rotation needed.

Case 3: Node p has balance factor of +1 (or –1) and a node was deleted from left sub-tree (right sub-tree). Rotation needed. Four sub-cases are possible.

Let

- **q** be the child of **p** with larger height
- **r** be the child of **q** with larger height

Sub-case -1 (left-left)

- **r** is left child of **q**, and **q** is left child of **p**
- Apply Single rotation

Sub-case -2 (right-right)

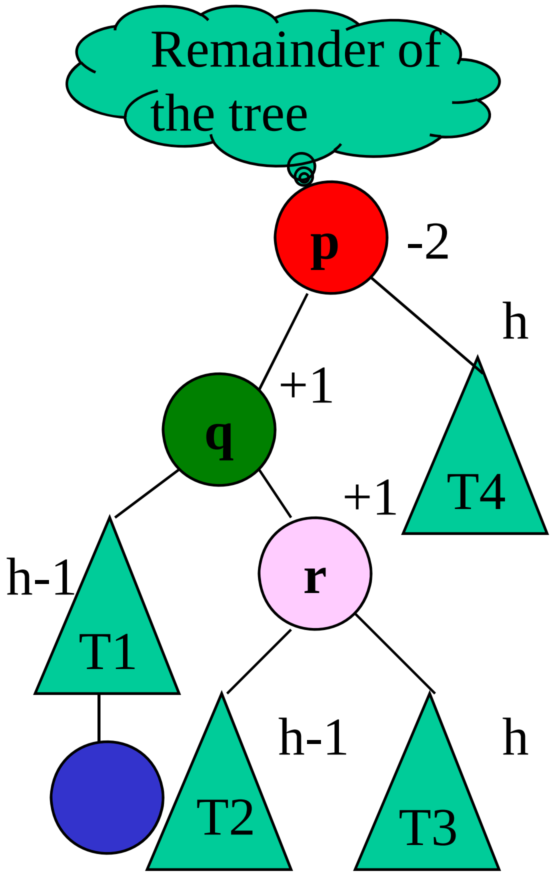
- **r** is right child of **q**, and **q** is right child of **p**
- Apply Single rotation

Sub-case -3 (left-right)

- **r** is left child of **q**, and **q** is right child of **p**
- Apply double rotation

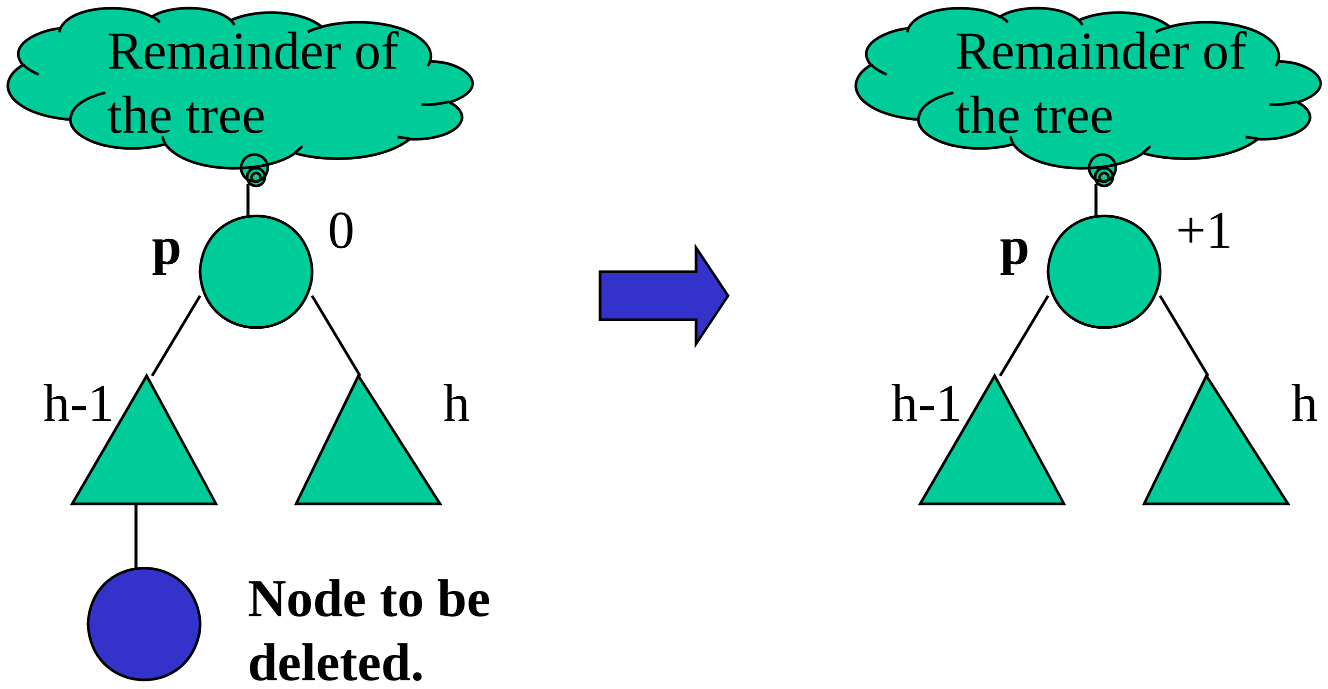
Sub-case -4 (right-left)

- **r** is right child of **q**, and **q** is left child of **p**
- Apply Single rotation



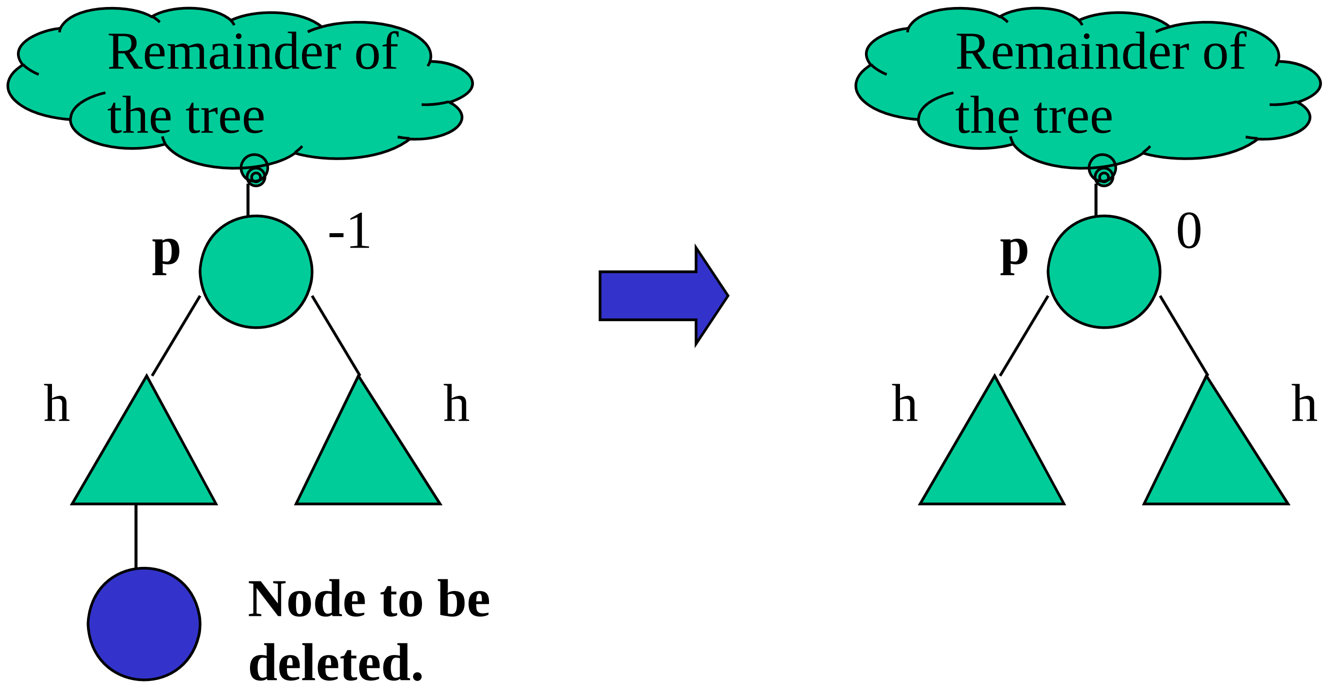
Case 1

Balance factor of **p** is 0



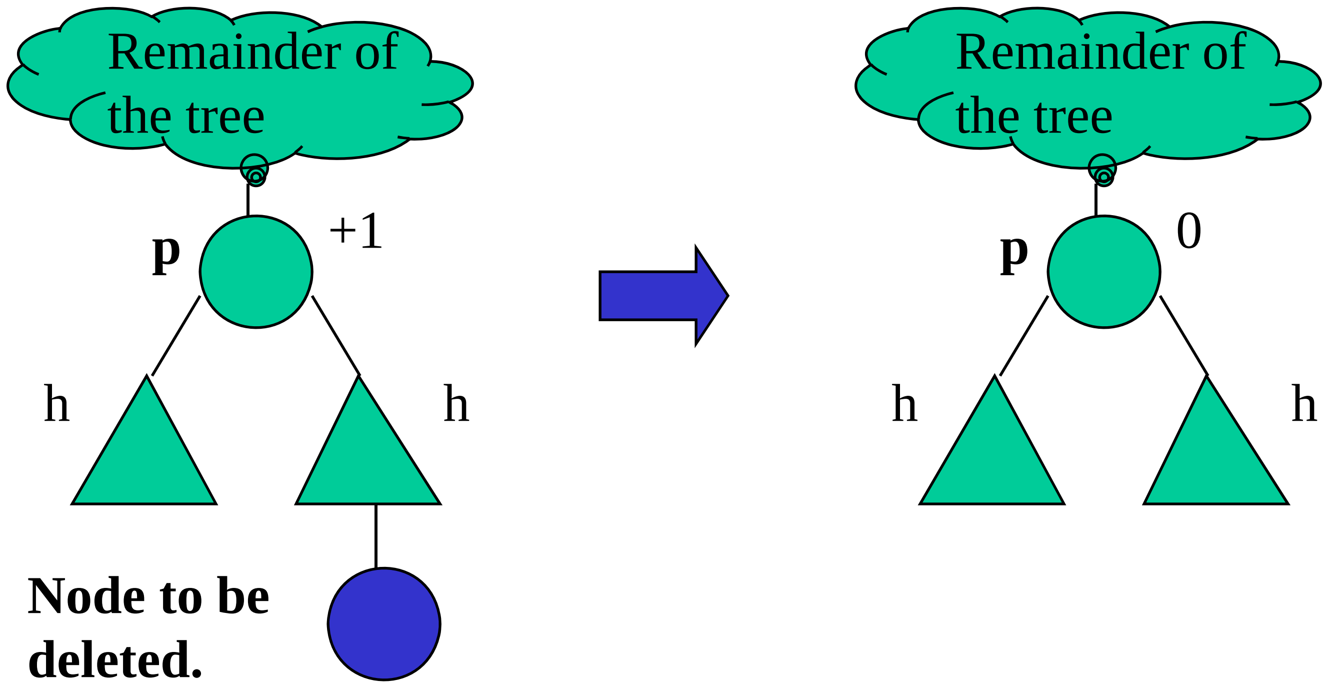
Case 2

Balance factor of p is -1 and node is deleted from left subtree



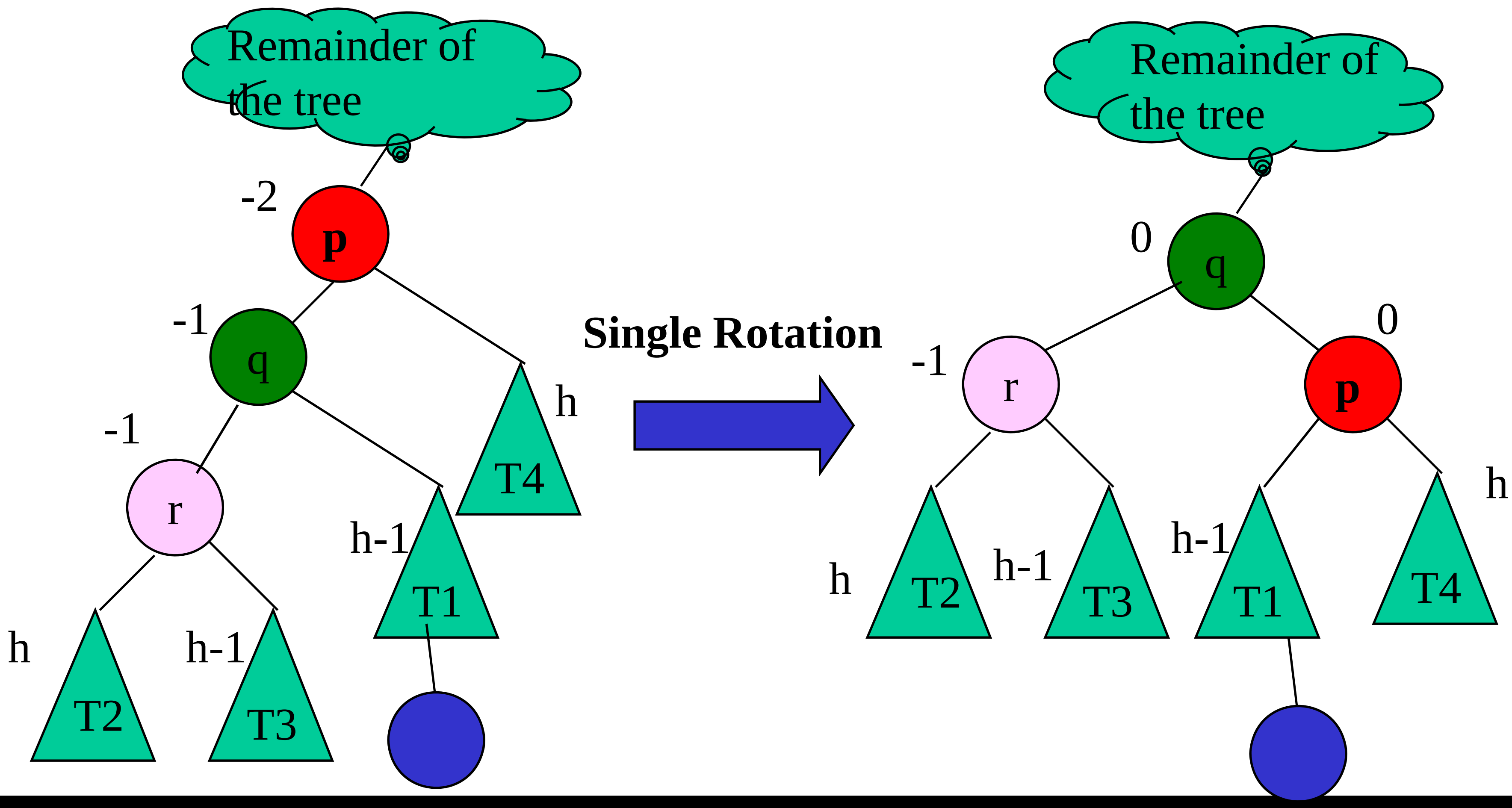
Case 2

Balance factor of p is +1 and node is deleted from right subtree



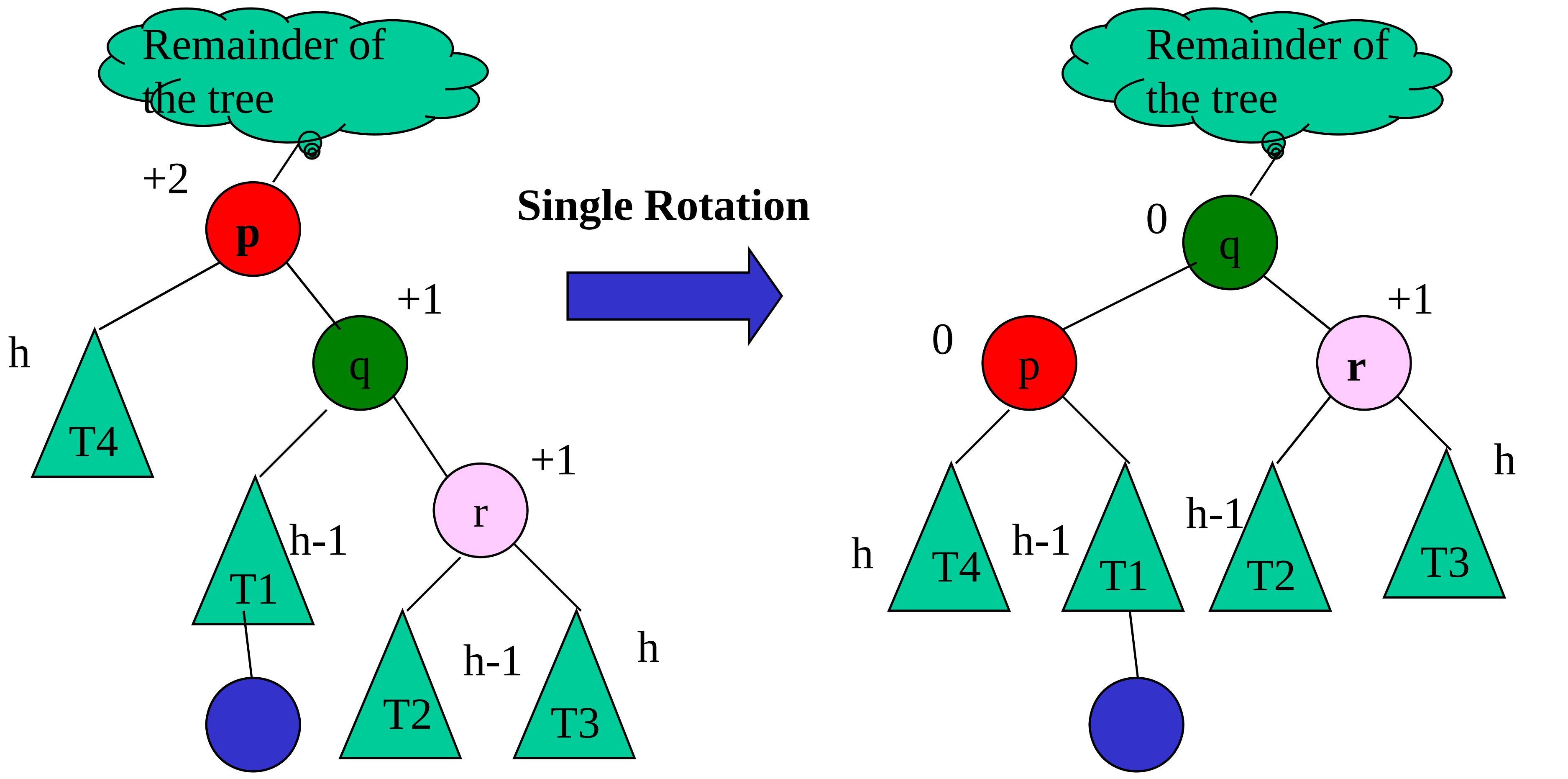
Case 3 – Subcase 1 (left-left)

r is left child of **q**, and **q** is left child of **p**



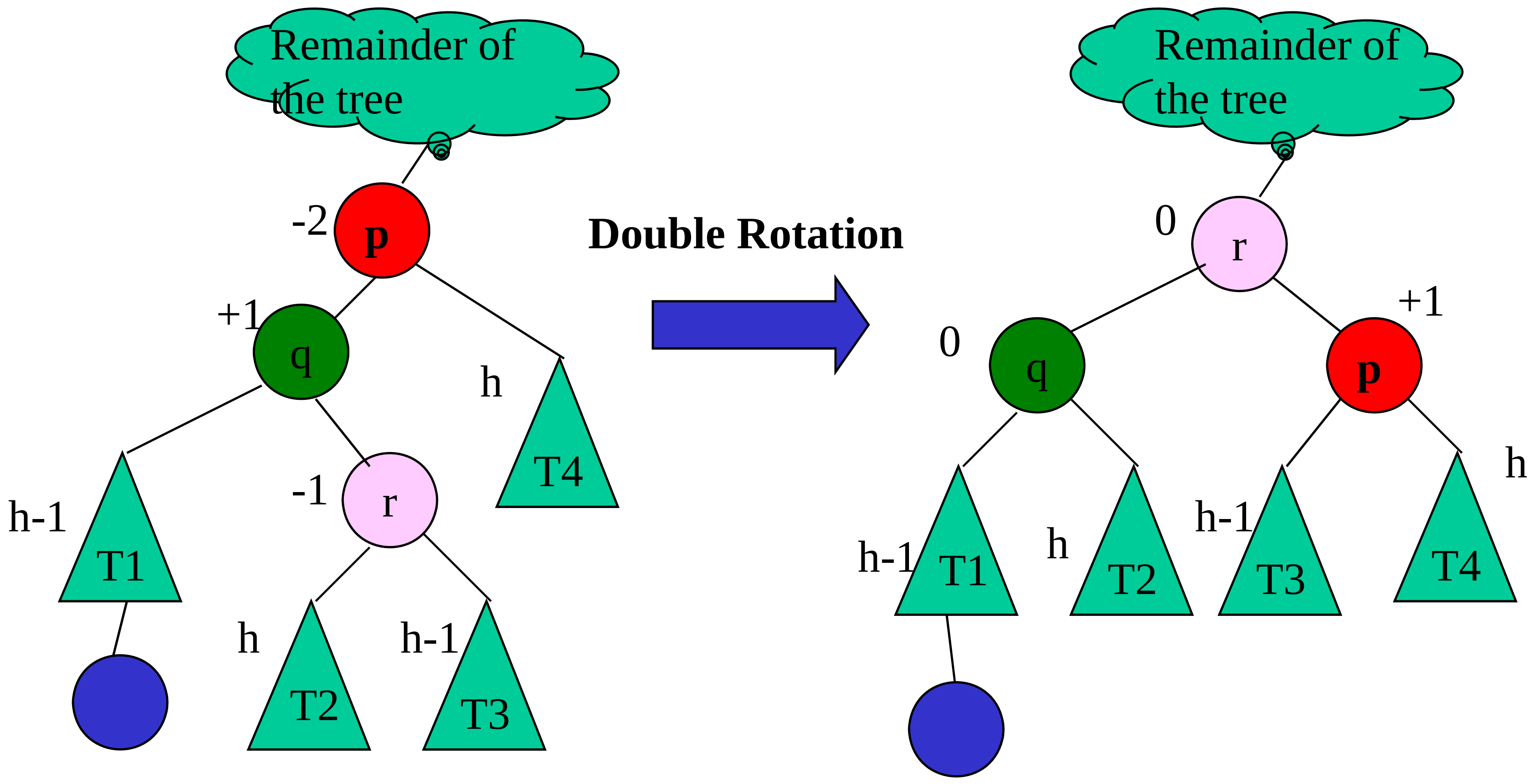
Case 3 – Subcase 2 (right-right)

r is right child of **q**, and **q** is right child of **p**



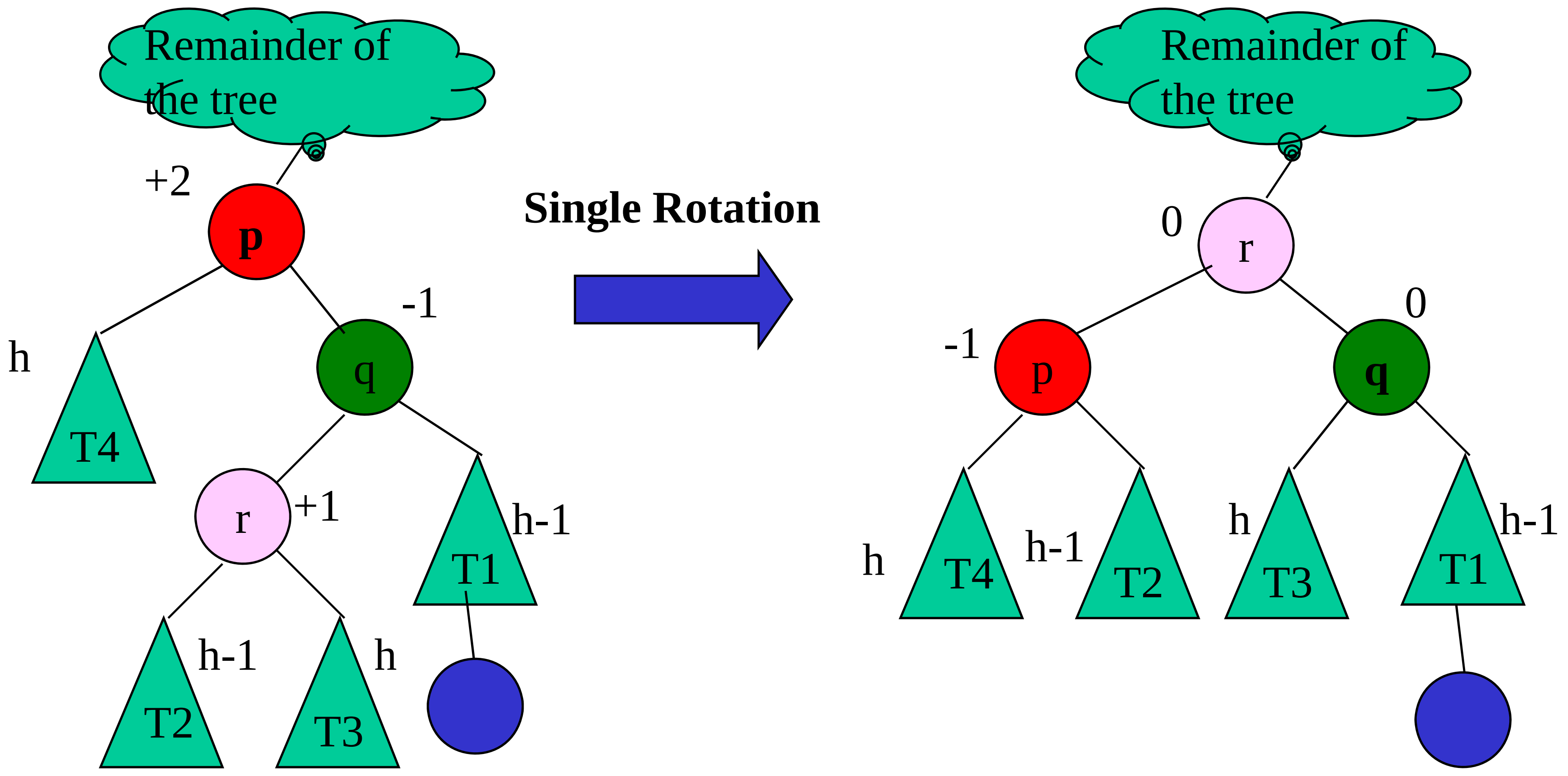
Case 3 – Subcase 3 (left-right)

r is right child of **q**, and **q** is left child of **p**

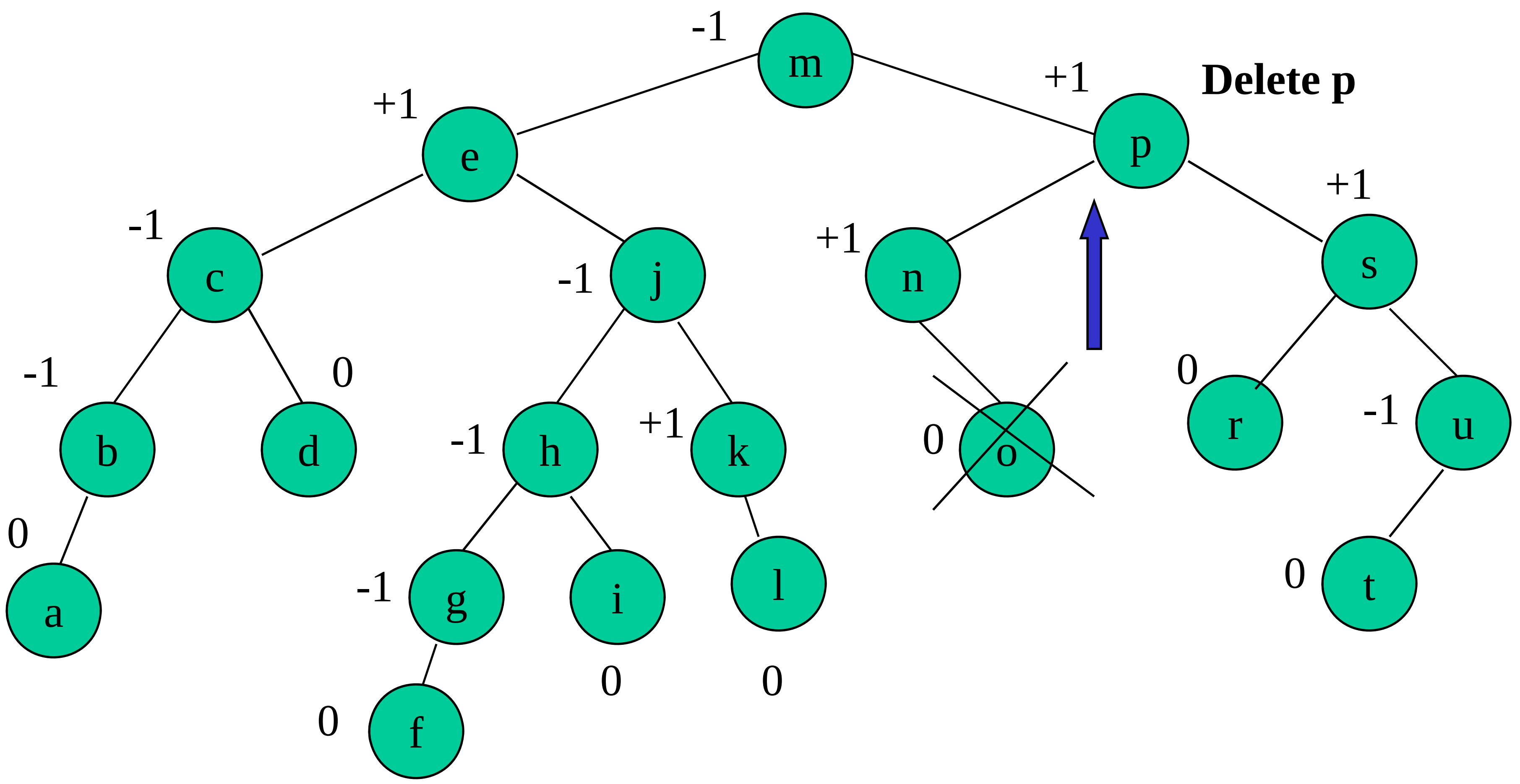


Case 3 – Subcase 4 (right-left)

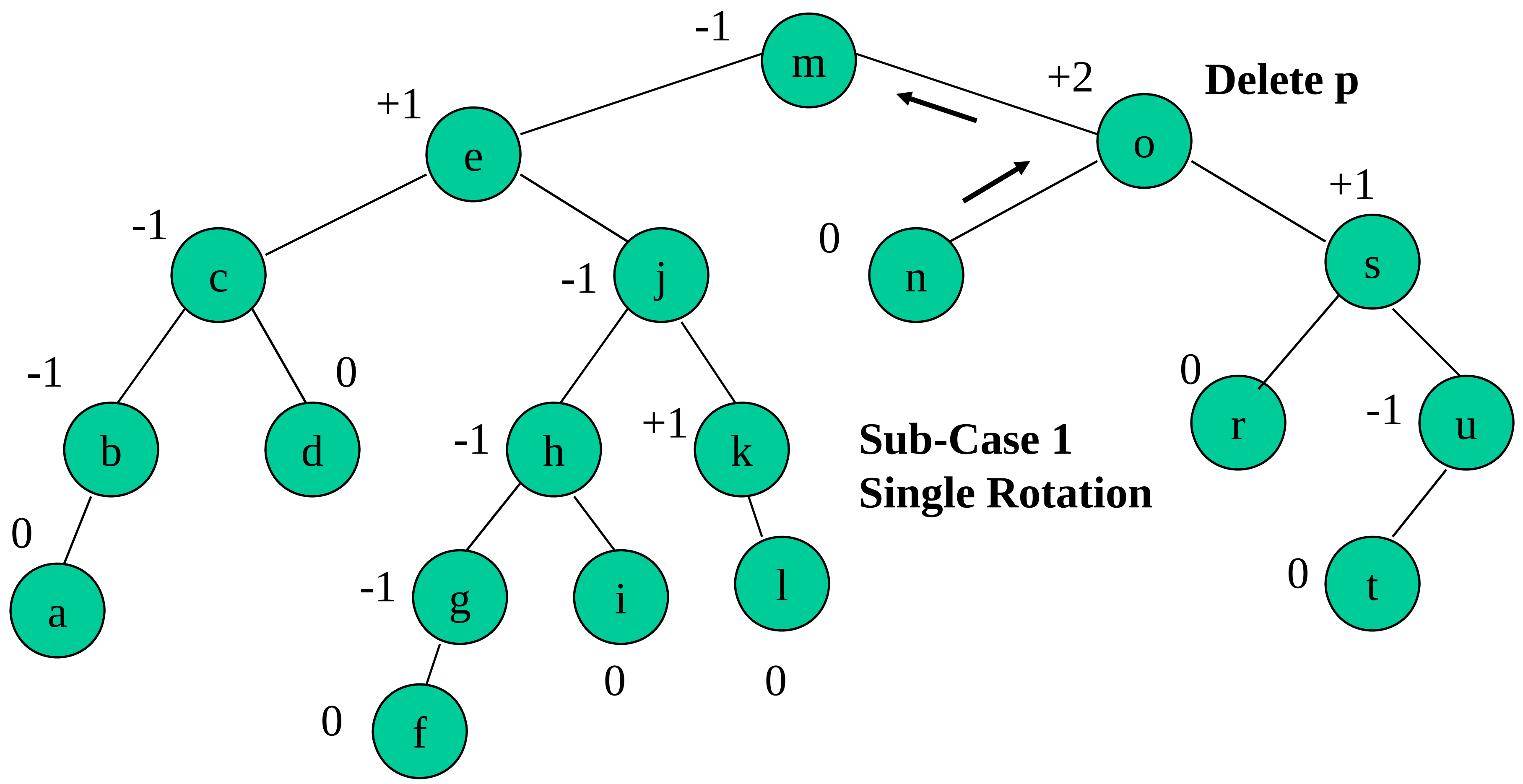
r is left child of **q**, and **q** is right child of **p**



Deletion: Example



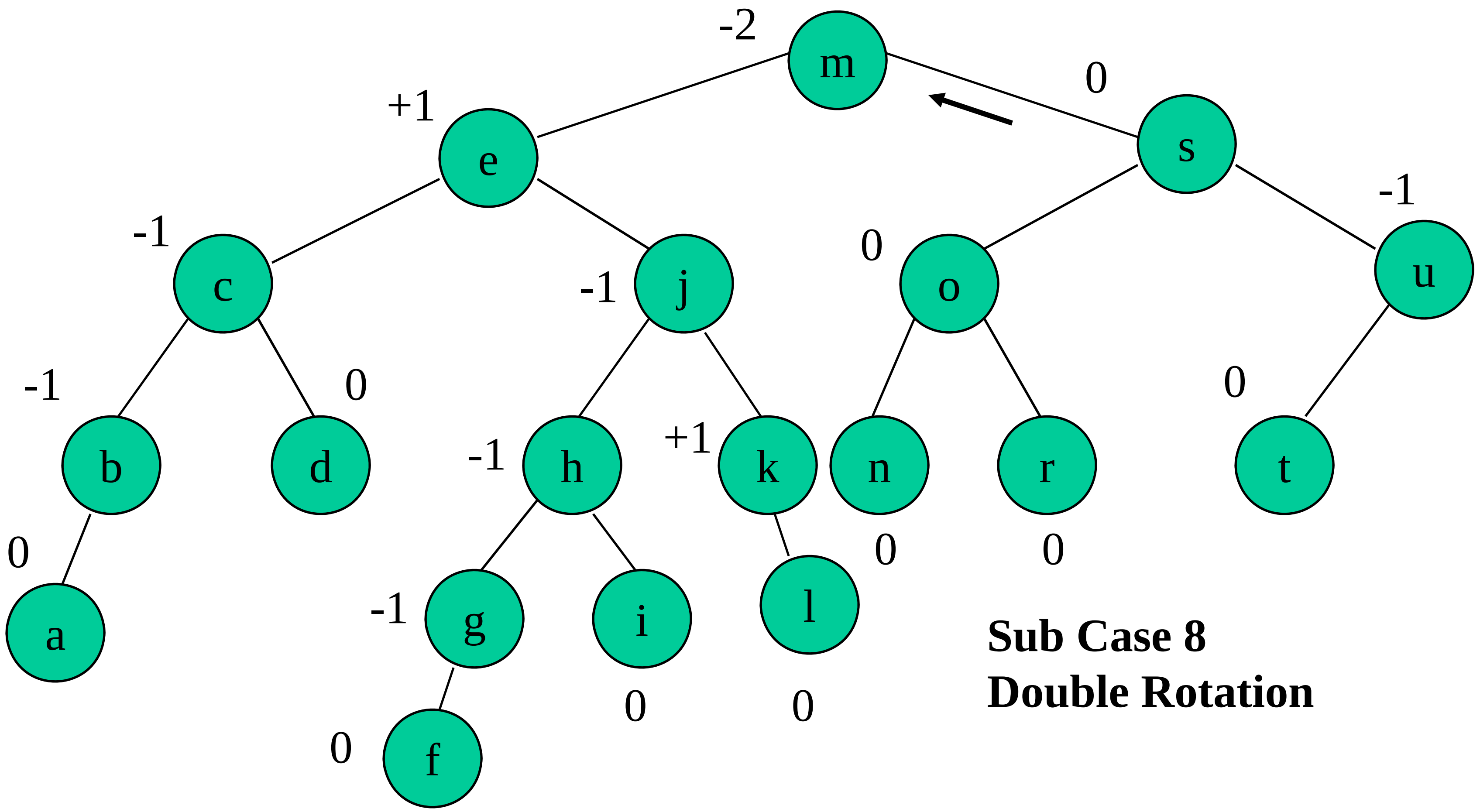
Deletion: Example



Delete p

Sub-Case 1
Single Rotation

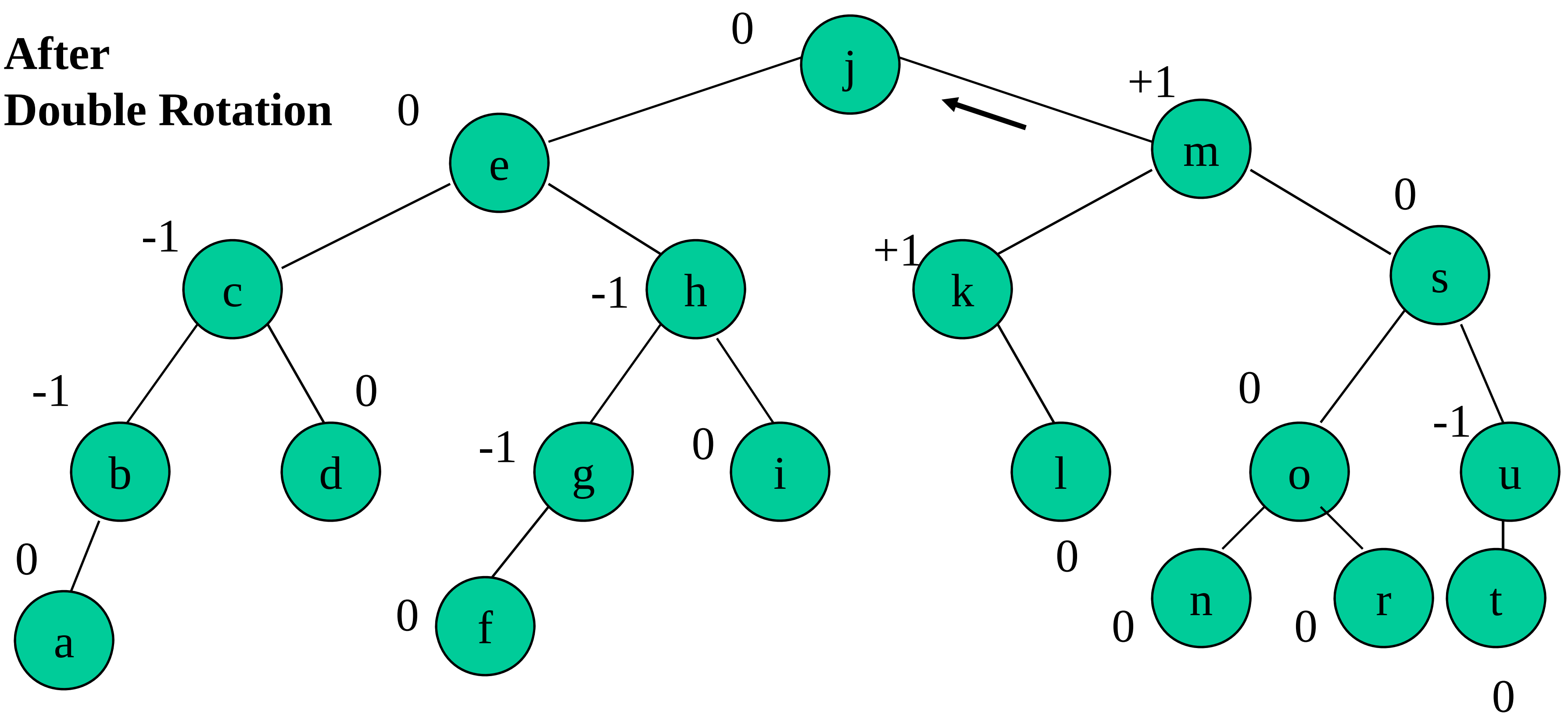
Deletion: Example



Sub Case 8
Double Rotation

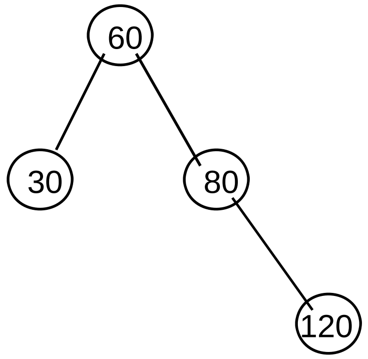
Deletion: Example

After
Double Rotation



Exercise 1

(*i*) *Is the following tree an AVL tree? If not, convert it into an AVL tree.*

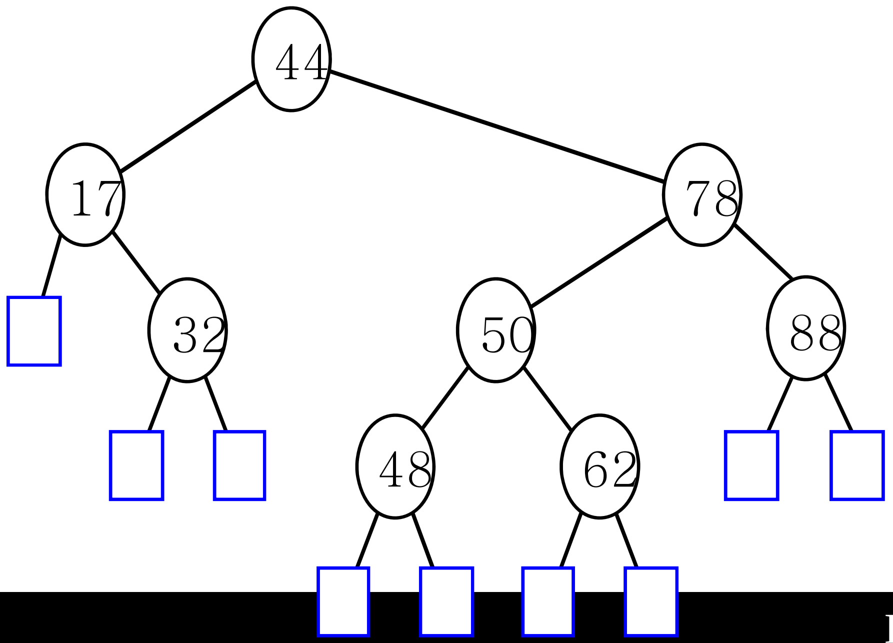


(*ii*) *Insert the nodes 70, 125, 65, 72, and 61 in the tree given in part (i), one at a time, and show the result of each insertion in a separate diagram. Ensure that the resulting tree satisfies AVL condition.*

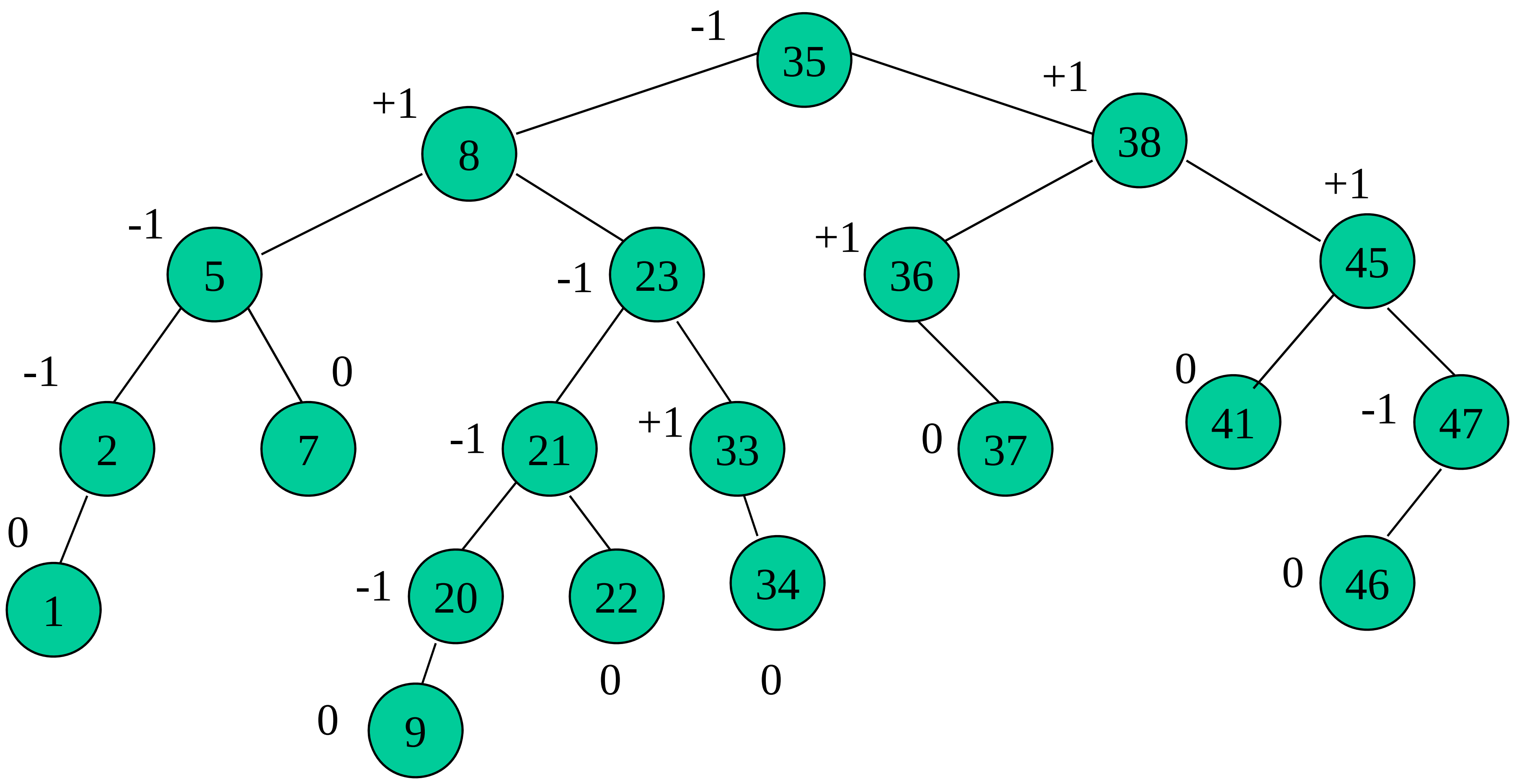
(*iii*) *Delete node 120 from the AVL tree you get in part (ii) and show the result in a separate diagram. Make sure the resulting tree is an AVL tree.*

Exercise 1

1. *What is an AVL tree and when is it used?*
2. *What is the time complexity for inserting N keys into an AVL tree?*
3. *Insert the following keys into an empty AVL tree 5, 1, 2, 6, 8, 3, 9*
4. *Delete 88 and then 33.*



6. Delete 38 from the following AVL tree


35
-1
8
+1
38
+1
5
-1
23
-1
36
+1
45
+1
2
-1
7
0
21
-1
33
+1
37
0
41
0
47
-1
1
0
20
-1
22
0
34
0
46
0
9
0