UNIT IV

Search Trees

Symbol Table

- In the world of computer science, compiler design is a complex field that bridges the gap between human-readable programming languages and machine-executable code.
- While compilers and assemblers are scanning a program, each identifier must be examined to determine if it is a keyword.
- This information concerning the keywords in a programming language is stored in a symbol table.
- The symbol table is a kind of a ‘keyed table’ which stores <key, information> pairs with no additional logical structure.
- A symbol table in compiler design is like a dictionary or a lookup table that keeps track of various pieces of information about identifiers (variables, functions, labels, etc.) used in the source code.

The operations performed on symbol tables are the following:

1. Inserting the <key, information> pairs into the collection.

2. Removing the <key, information> pairs by specifying the key.

3. Searching for a particular key.

4. Retrieving the information associated with a key.

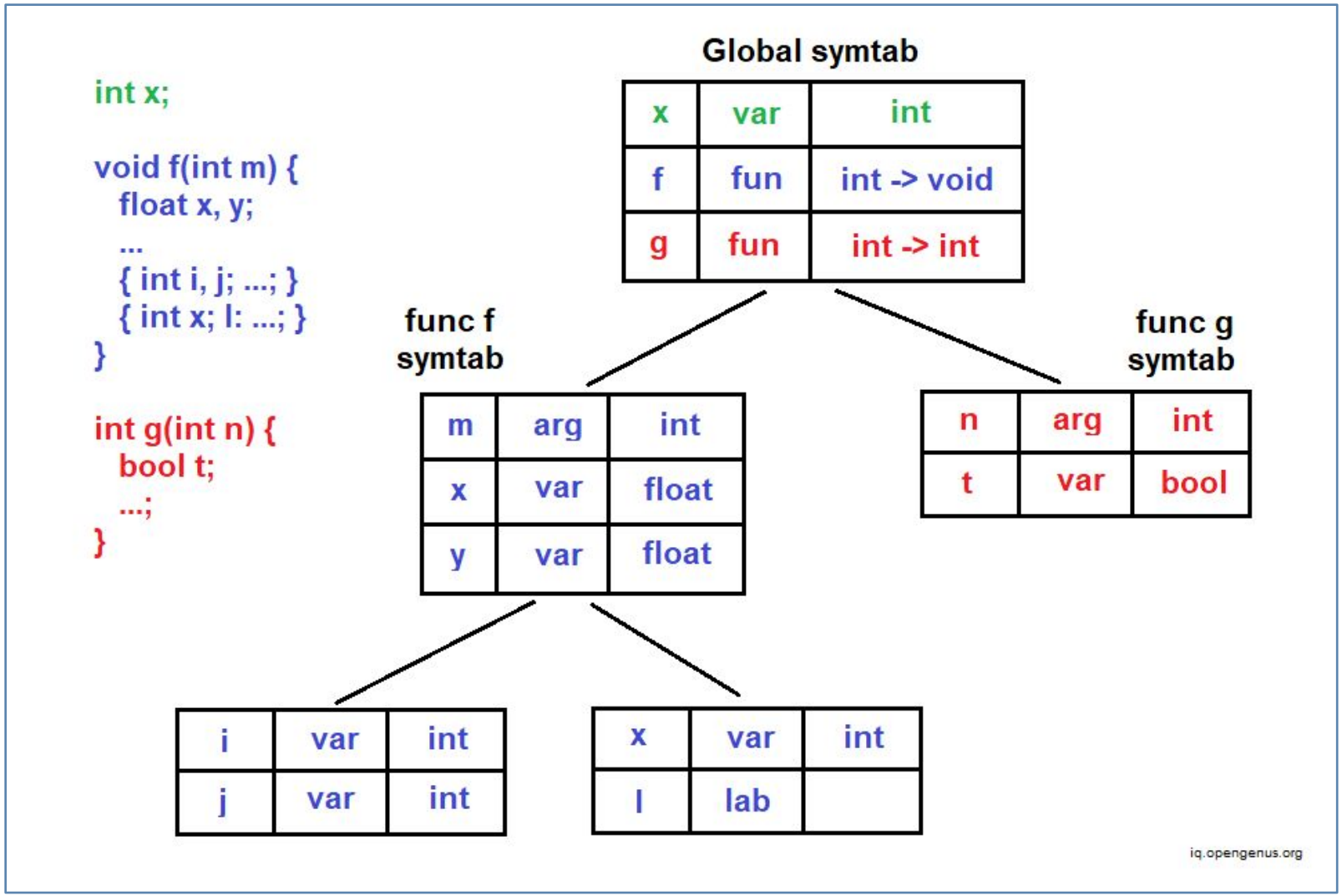
Consider the following C++ statement:

```
int limit;
```

- When a compiler processes this statement, it will identify that int is a keyword and limit is an identifier.
- Compiler classifies them as a keyword and a user-defined identifier.
- For identifying int as a keyword, the compiler is provided with a table of keywords.
- For faster search through a list of keywords, the symbol table is used as an efficient data structure.

Trees implementation of Symbol table -

- It is a more efficient method of organizing symbol tables. Each record now has two link fields, LEFT and RIGHT.



Representation of Symbol Table

- There are two different techniques for implementing a keyed table, namely, the symbol table and the tree table.
- **Static Tree Tables –**
  - When symbols are known in advance and no insertion and deletion is allowed, such a structure is called a static tree table.
  - An example of this type of table is a reserved word table in a compiler.
  - This table is searched once for every occurrence of an identifier in a program.
  - If an identifier is not present in the reserved word table, then it is searched for in another table.
  - When we know the keys and their probable frequencies of being searched, we can optimize the search time by building an optimal binary search tree (OBST).
  - The keys have history associated with their use, which is referred to as their probability of occurrence.

Dynamic Tree Tables –

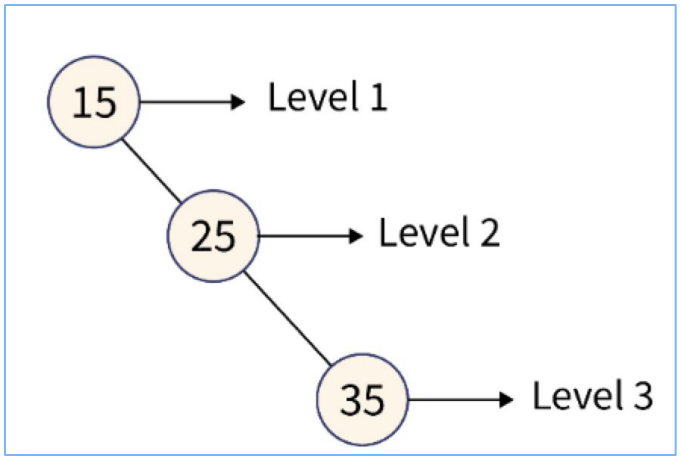
- A dynamic tree table is used when symbols are not known in advance but are inserted as they come and deleted if not required.
- Dynamic keyed tables are those that are built on the-fly.
- The keys have no history associated with their use.
- The dynamically built tree that is a balanced BST is the best choice.

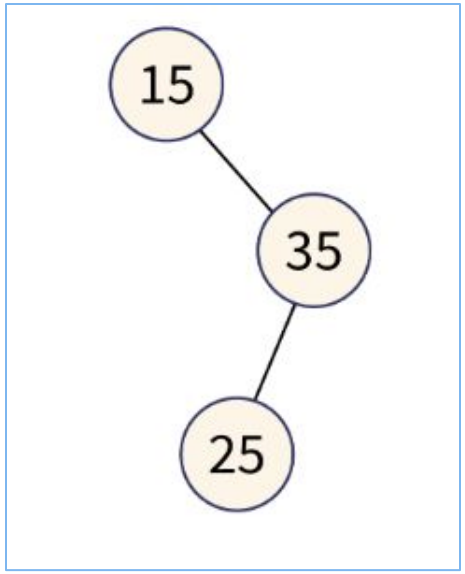
OPTIMAL BINARY SEARCH TREE

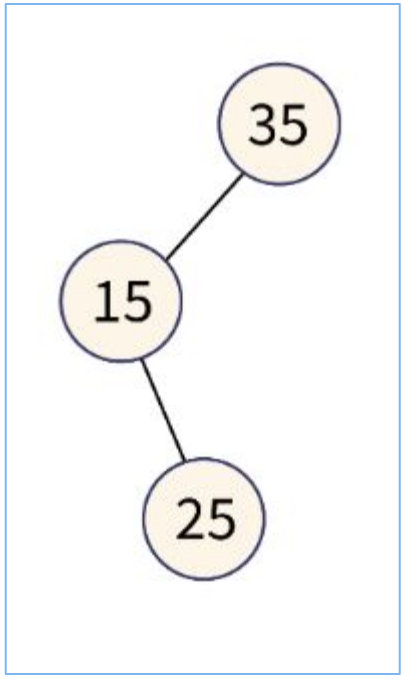
- The optimal binary search tree (Optimal BST) is also known as a weight-balanced search tree.
- It is a binary search tree that provides the shortest possible search time or expected search time.
- An Optimal Binary Search Tree (OBST), is a binary search tree that minimizes the expected search cost.
- In a binary search tree, the search cost is the number of comparisons required to search for a given key.
- In an OBST, each node is assigned a weight that represents the probability of the key being searched for.
- The sum of all the weights in the tree is 1.0.
- **The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.**

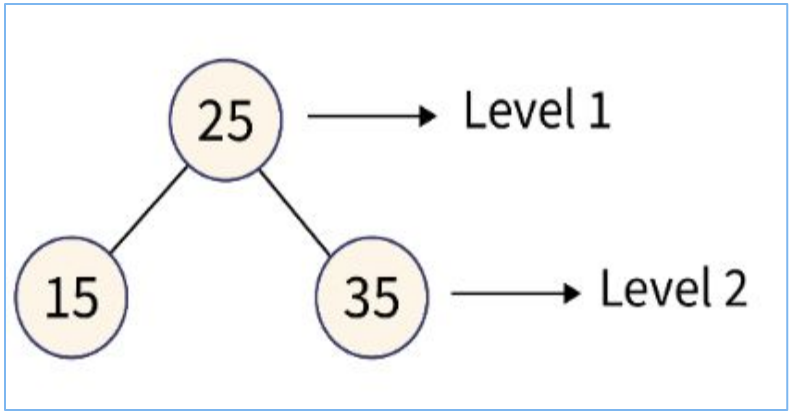
- If we have keys: 15, 25, 35, then see how many BST can be formed. We will use the formula given below:

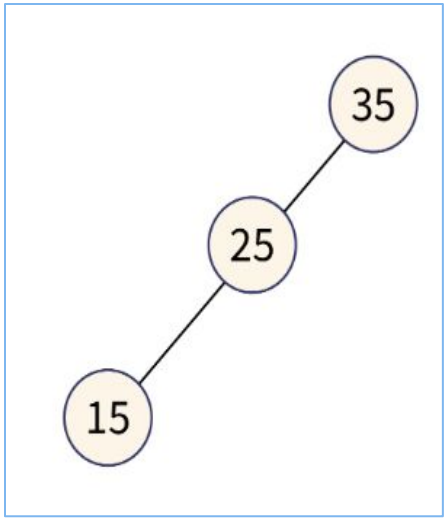
$$= \frac{^{2n}C_n}{n+1}$$











- Cost of a tree can be defined as follows

$$\text{Cost(T)} = \sum_{i=1}^{n} l(a_i)$$

- Here, the total number of nodes are n, and $l(a_i)$ is the length of the i^{th} key, a.
- Here, we assume that all the keys are searched with equal probabilities.
- However, in reality, the keys are searched with different probabilities, and it should be taken care of while constructing the tree so that the keys searched more often should require less time as compared to those searched rarely.
- This can be achieved by placing the more frequently searched key nodes closer to the root as compared to those that are searched rarely, to reduce the total number of average searches.
- A node is said to be closer to the root when its path length is lesser than that of the other nodes.

- Then, cost of a tree is computed with respect to its node's probability of search and path length. Hence,

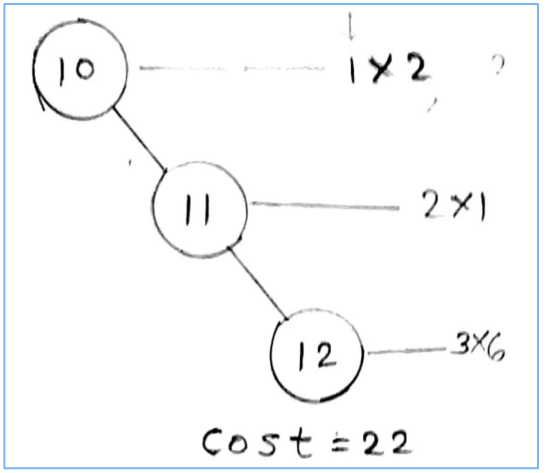
$$\text{Cost(T)} = \sum_{i=1}^{n} W_i \times L_i$$

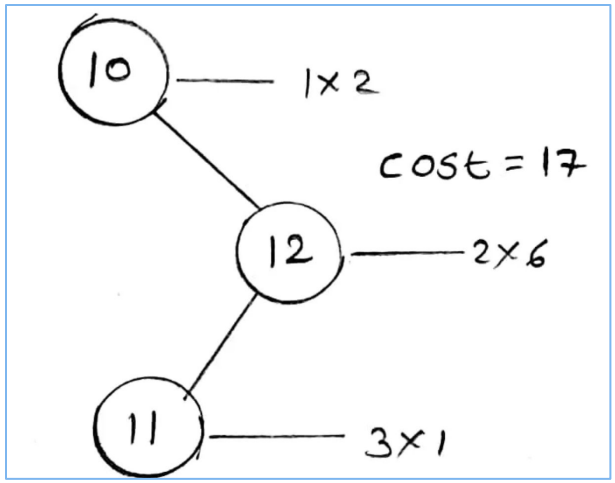
where,

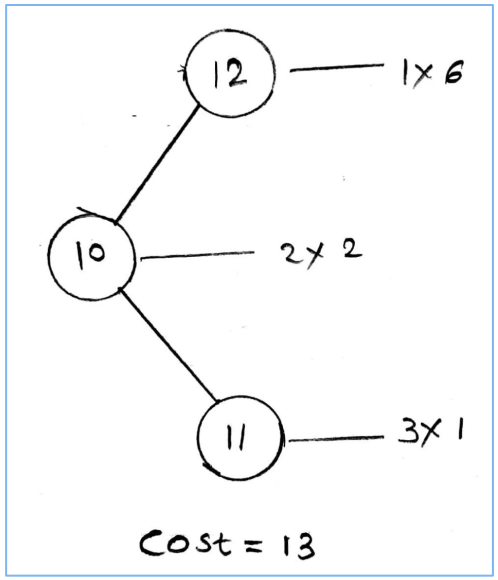
W_i = frequency or probability (also called as weight of the i^{th} node)

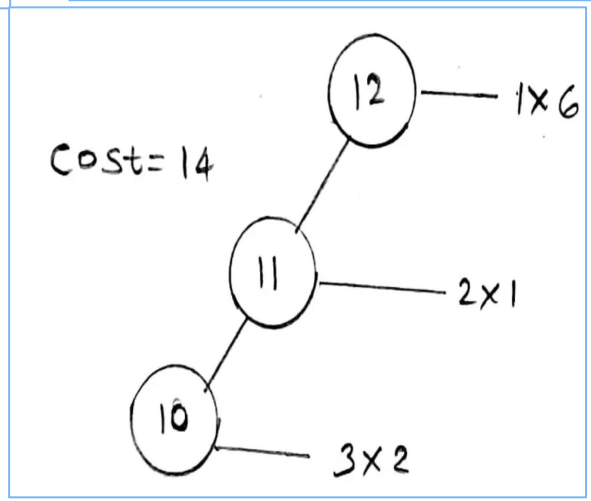
L_i = level of a particular node calculated from the root node treated from level 0

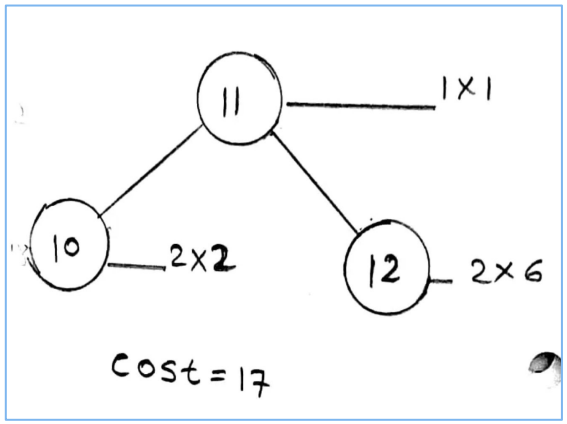
- In BST all nodes have some frequency.
- According to frequency BST has some cost.
- Our objective is find minimum cost of BST.
- Let's take on small example
- Key:- 10,11,12 and Frequency:- 2,1,6





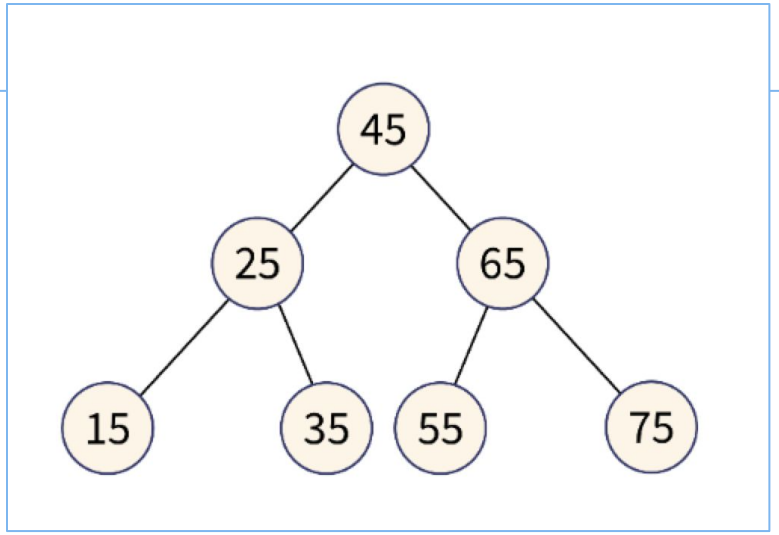






Consider the keys $\{k_1, k_2, \ldots, k_n\}$ such that $k_1 < k_2 < k_3 < \ldots < k_n$. Every successful search for the key k_i has the probability $p(i)$. In addition, every unsuccessful search for the key x has the probability of failure $q(i)$ for $0 \leq i \leq n$, and $k_i < x < k_{i+1}$. We can add a fictitious node as a child for every leaf node.

- For the BSTs , all the keys represent internal nodes.
- all successful searches will always end at an internal node.
- all squares denote external nodes, which are fictitious.
- all unsuccessful searches will end at some external node.
- If there are n keys, there are n + 1 external nodes.
- So all the keys that are not a part of a BST belong to one of (n + 1) equivalence classes E_i for $0 \leq i \leq n$.



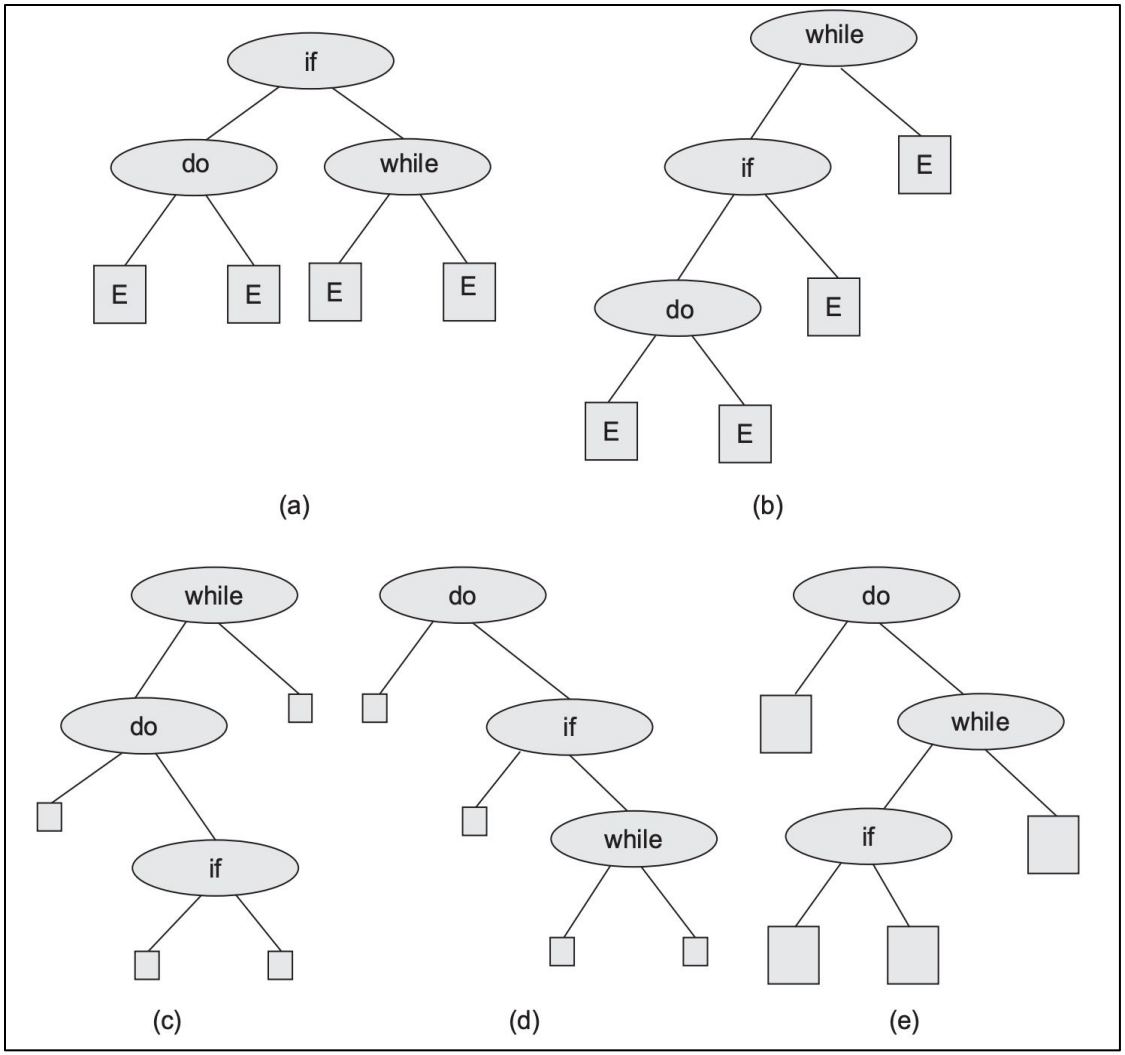
- Hence, the cost of a BST is given as follows:

$$\sum_{1 \le i \le n} p(i) \times \text{level}(k_i) + \sum_{0 \le i \le n} q(i) \times \text{level}(E_i) - 1$$

- Equation defines the cost of a BST in terms of the probabilities of successful and unsuccessful searches and the level of a node.

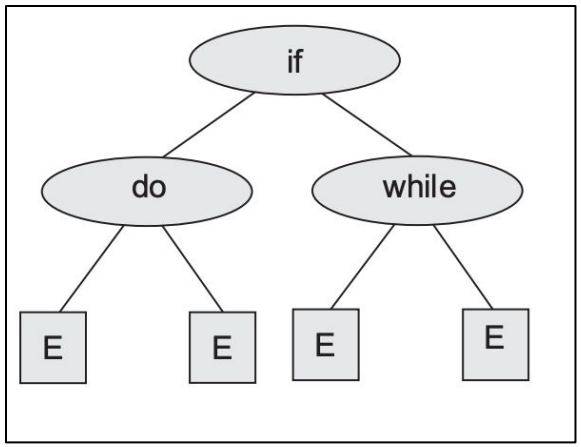
EXAMPLE - Given the keys = {while, do, if} and probabilities p(i) = q(i) = 1/7 for all i. Compute the cost of all possible BSTs and find the OBST.

Solution We get five possible BSTs for the given keys



Let us compute the cost of each BST.

keys = {while, do, if}



$$\text{Cost} = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) + \sum_{0 \le i \le 3} q(i) \times \text{level}(E_i) - 1 = \text{A} + \text{B}$$

$$\text{A} = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) = p_1 \times \text{level}(k_1) + p_2 \times \text{level}(k_2) + p_3 \times \text{level}(k_3)$$

$$= 1/7(2 + 2 + 1)$$

$$= 5/7$$

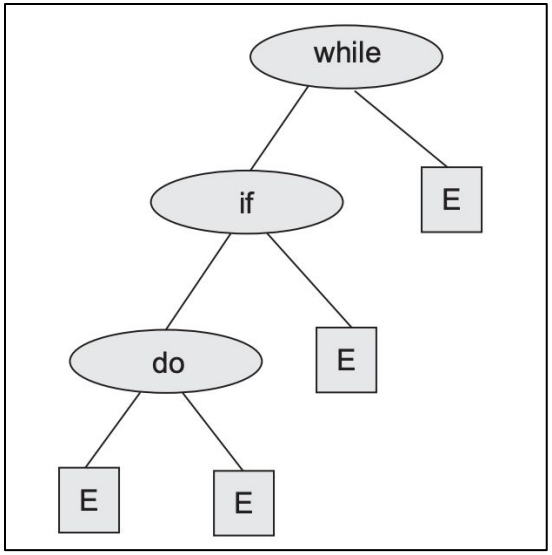
$$\text{B} = \sum_{0 \le i \le 3} q(i) \times (\text{level}(E_i) - 1) = q_0 \times \text{level}(E_0) - 1 + q_1 \times \text{level}(E_1) - 1$$

$$+ q_2 \times \text{level}(E_2) - 1 + q_3 \times \text{level}(E_3) - 1$$

$$= 1/7(2 + 2 + 2 + 2)$$

$$= 8/7$$

Therefore, cost = (5/7) + (8/7) = 13/7



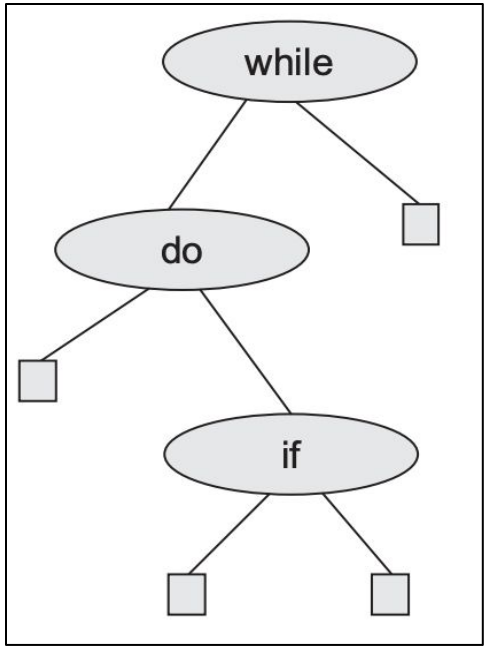
where $A = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) = \frac{1}{7}(1+2+3)$

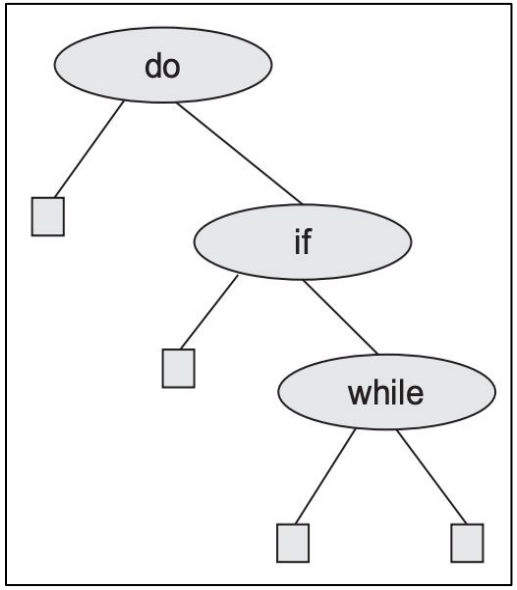
$= 6/7$

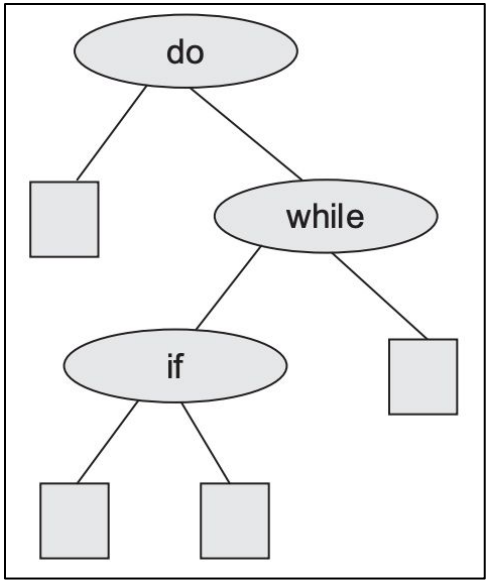
$B = \sum_{0 \le i \le 3} q(i) \times \text{level}(E_i) = \frac{1}{7}(3+3+2+1)$

$= 9/7$

Therefore, cost = (6/7) + (9/7) = 15/7







Similarly cost of each subtree = 15/7

- Practically, we cannot use such an approach to find an OBST as we will need to draw all possible BSTs and then find the cost of all BSTs.
- As the number of keys increases, the number of BSTs also increases.
- Dynamic programming approach can be used to construct an OBST by considering the probabilities of both successful and unsuccessful searches for the given set of keys.
- We construct an OBST step-by-step using the following three formulae:

$$w(i, j) = p(j) + q(j) + w(i, j-1)$$
$$c(i, j) = \min_{(i < a \le j)} \{c(i, a-1) + c(a, j)\} + w(i, j)$$
$$r(i, j) = a$$

where,

$w(i, j)$ is the weight of node (i, j)

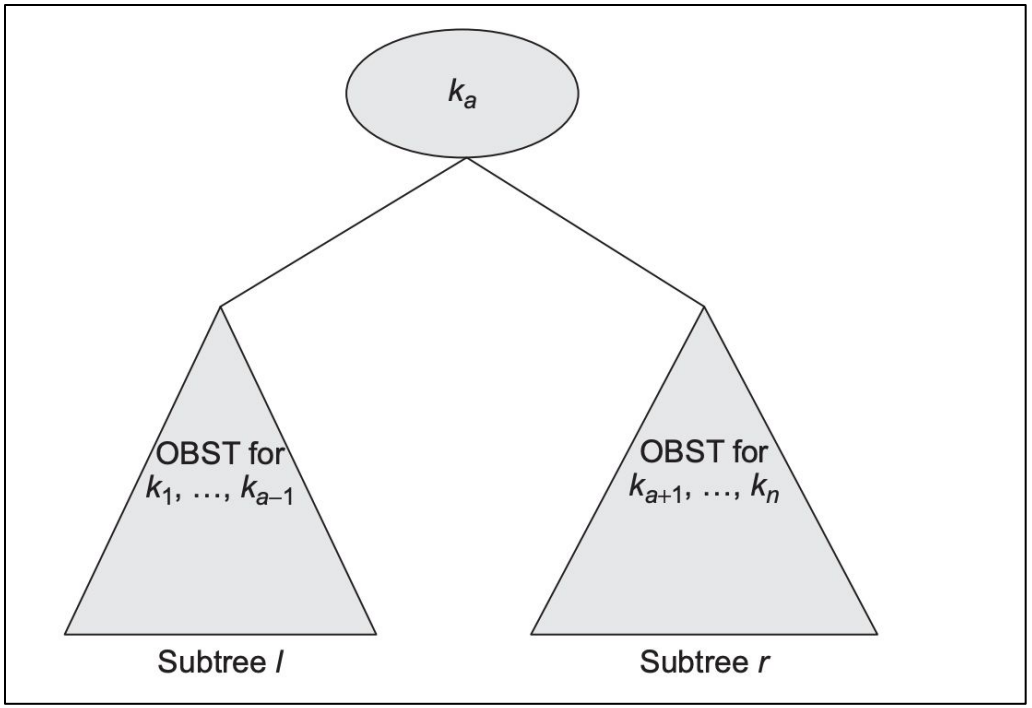
$c(i, j)$ is the cost of node (i, j)

$c(i, a-1)$ is the cost of left subtree

$c(a, j)$ is the cost of right subtree

$r(i, j)$ is the root of the tree

- The dynamic programming approach can be used to construct an OBST stepwise, where the principal of optimality should hold at each step.



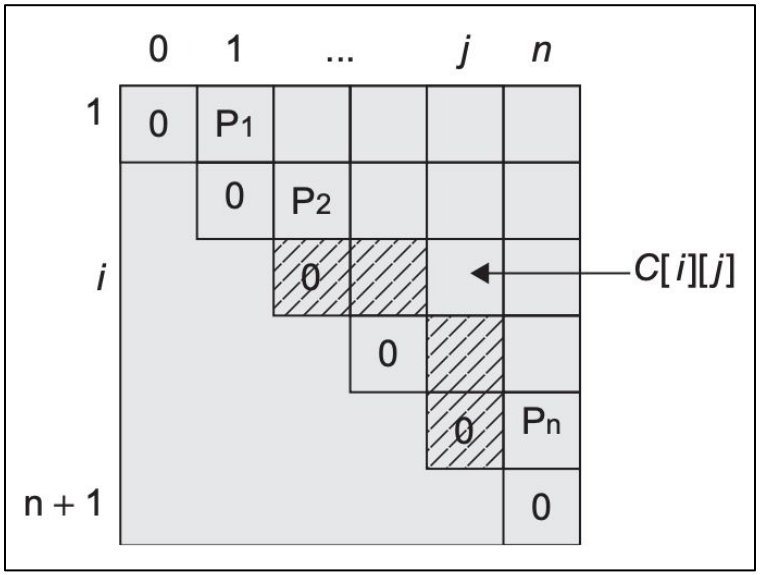
- The steps to find OBST are as follows:

1. We begin by considering all unsuccessful probabilities as initially there are no nodes in the tree. $c(i, i) = 0$, $r(i, i) = 0$, and $w(i, i) = q(i)$ for $0 \le i \le n$, where n is the number of keys.
2. Compute $c(i, j)$ for $j - i = 1$, that is, we are constructing a node of level 1. In addition, compute $w(i, j) = p(j) + q(j) + w(i, j - 1)$, and the root $r(i, j)$ is the value of a which minimizes $c(i, j)$.

$$c(i,j) = \min_{i < a \le j} [c(i,a-1) + c(a,j) + w(i,j)]$$

3. Compute $c(i, j)$ for $j - i = 2$. In addition, compute $w(i, j)$ and $r(i, j)$ as in the previous step.
4. Continue the process till $j - i = n$. Here, w_{on}, c_{on}, and r_{on} denote the weight, cost, and root of OBST, respectively.
5. Finally, we can construct an OBST having the root $r_{on} = a$, which means that the key k_a is the root.

- The initial cost table of the dynamic programming algorithm for constructing an OBST is shown in Fig



- The values needed for computing C[i][j] are shaded in Fig.
- They are the values in row i and to the left of column j, and the values in column j and the rows below row i.

EXAMPLE 10.2 Find an OBST using a dynamic programming for $n = 4$ and keys $(k_1 < k_2 < k_3 < k_4)$ = (do, if, int, while) given that $p(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$.

Solution

Step 1: Initially, $c(i, i) = 0$, $r(i, i) = 0$, and $w(i, i) = q(i)$ for $0 \le i \le 4$.
Hence, $w(0, 0) = 2$, $w(1, 1) = 3$, $w(2, 2) = w(3, 3) = w(4, 4) = 1$
This is shown in Table 10.1.

Table 10.1 OBST computation for Example 10.2 after step 1

	0	1	2	3	4	Initial values ←
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$	
1						
2						
3						
4						

Step 2: $w(i, j) = p(j) + q(j) + w(i, j-1)$

$$c(i,j) = \frac{\min}{i < a \le j} [c(i,a-1) + c(a,j) + w(i,j)]$$

$r = (i, j)$ = value of a which minimizes $c(i, j)$

Let us compute c(i, j) for $j - i = 1$

$w(0, 1) = p(1) + q(1) + w(0, 0)$

$= 3 + 3 + 2 = 8$

$c(0, 1) = w(0, 1) + \min[c(0, 0) + c(1, 1)]$ for $a = 1$

$= 8 + [0 + 0] = 8$

$r(0, 1) = 1$

$w(1, 2) = p(2) + q(2) + w(1, 1) = 3 + 1 + 3 = 7$

$c(1, 2) = w(1, 2) + \min[c(1, 1) + c(2, 2)] = 7 + [0 + 0] = 7$ for $a = 2$

$r(1, 2) = 2$

$w(2, 3) = p(3) + q\ (3) + w(2, 2) = 1 + 1 + 1 = 3$

$c(2, 3) = w(2, 3) + \min[c(2, 2) + c(3, 3)] = 3 + [0 + 0] = 3$ for $a = 3$

$r(2, 3) = 3$

$w(3, 4) = p(4) + q(4) + w(3, 3) = 1 + 1 + 1 = 3$

$c(3, 4) = w(3, 4) + \min[c(3, 3) + c(4, 4)] = 3 + [0 + 0] = 3$

$r(3, 4) = 4$

- This computation is shown in Table

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	Here $j - i = 1$, that is, while calculating c_{ij}, a took only one value, that is, j.
2					
3					
4					

Step 3: Compute $c(i, j)$ for $j - i = 2$

$$w(0, 2) = p(2) + q(2) + w(0, 1)$$
$$= 3 + 1 + 8 = 12$$
$$c(0, 2) = w(0, 2) + \min[c(0, 0) + c(1, 2) \quad \text{for } a = 1,$$
$$c(0, 1) + c(2, 2) \quad \text{for } a = 2]$$
$$= 12 + \min[0 + 7, 8 + 0]$$
$$= 12 + 7 = 19$$
$$r(0, 2) = 1$$
$$w(1, 3) = p(3) + q(3) + w(1, 2) = 1 + 1 + 7 = 9$$
$$c(1, 3) = w(1, 3) + \min[c(1, 1) + c(2, 3) \quad \text{for } a = 2,$$
$$c(1, 2) + c(3, 3) \quad \text{for } a = 3]$$
$$= 9 + \min[0 + 3, 7 + 0]$$
$$= 9 + 3 = 12$$
$$r(1, 3) = 2$$
$$w(2, 4) = p(4) + q(4) + w(2, 3) = 1 + 1 + 3 = 5$$
$$c(2, 4) = w(2, 4) + \min[c(2, 2) + c(3, 4) \text{ for } a = 3 \text{ , } c(2, 3) + c(4, 4)]$$
$$\text{for } a = 4$$
$$= 5 + \min[0 + 3, 3 + 0]$$
$$= 5 + 3 = 8$$
$$r(2, 4) = 3$$

Table shows this computation.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$	Here, $j - i = 2$, that is, while calculating c_{ij}, a took two values.	
3					
4					

Step 4: Compute $c(i, j)$ for $j - i = 3$

$$w(0, 3) = p(3) + q(3) + w(0, 2) = 1 + 1 + 12 = 14$$

$$c(0, 3) = w(0, 3) + \min[c(0, 0) + c(1, 3) \quad \text{for } a = 1, c(0, 1) + c(2, 3) \quad \text{for } a = 2, c(0, 2) + c(3, 4) \quad \text{for } a = 3]$$

$$= 14 + \min[0 + 12, 8 + 3, 19 + 3]$$

$$= 14 + \min[12, 11, 22]$$

$$= 14 + 11 = 25$$

$$r(0, 3) = 2$$

$$w(1, 4) = p(4) + q(4) + w(1, 3) = 1 + 1 + 9 = 11$$

$$c(1, 4) = w(1, 4) + \min[c(1, 1) + c(2, 4) \quad \text{for } a = 2, c(1, 2) + c(3, 4) \quad \text{for } a = 3, c(1, 3) + c(4, 4) \quad \text{for } a = 4]$$

$$= 11 + \min[0 + 8, 7 + 3, 12 + 0]$$

$$= 11 + \min[8, 10, 12]$$

$$= 11 + 8 = 19$$

$$r(1, 4) = 2$$

Table shows this computation.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$	Here $j - i = 3$, that is, while calculating c_{ij}, a took three values.		
4					

Step 5: Compute $c(i, j)$ for $j - i = 4$

$$w(0, 4) = p(4) + q(4) + w(0, 3) = 1 + 1 + 14 = 16$$

$$c(0, 4) = w(0, 4) + \min[c(0, 0) + c(1, 4) \quad \text{for } a = 1,\ c(0, 1) + c(2, 4) \quad \text{for } a = 2,$$
$$c(0, 2) + c(3, 4) \quad \text{for } a = 3,\ c(0, 3) + c(4, 4) \quad \text{for } a = 4]$$
$$= 16 + \min[0 + 19, 8 + 8, 19 + 3, 25 + 0]$$
$$= 16 + \min[19, 16, 22, 25]$$
$$= 16 + 16 = 32$$

$$r(0, 4) = 2$$

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

EXAMPLE 10.2 Find an OBST using a dynamic programming for $n = 4$ and keys $(k_1 < k_2 < k_3 < k_4) = (\text{do, if, int, while})$ given that $p(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$.

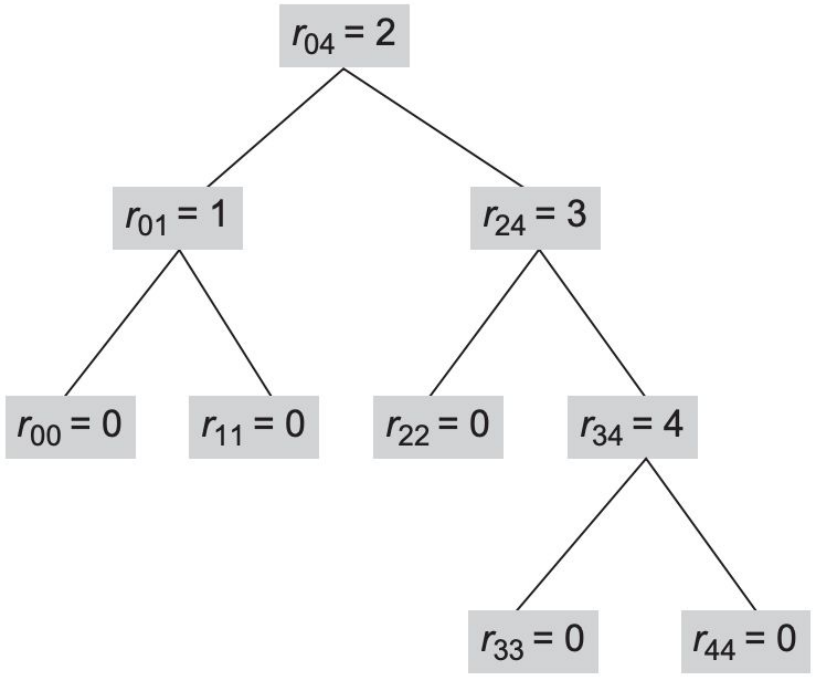
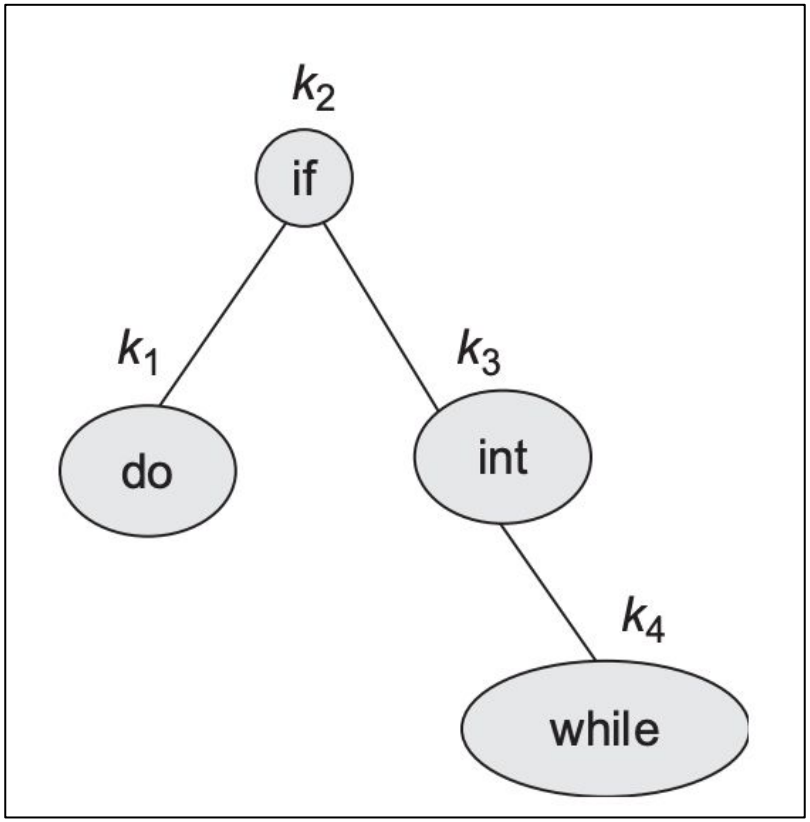
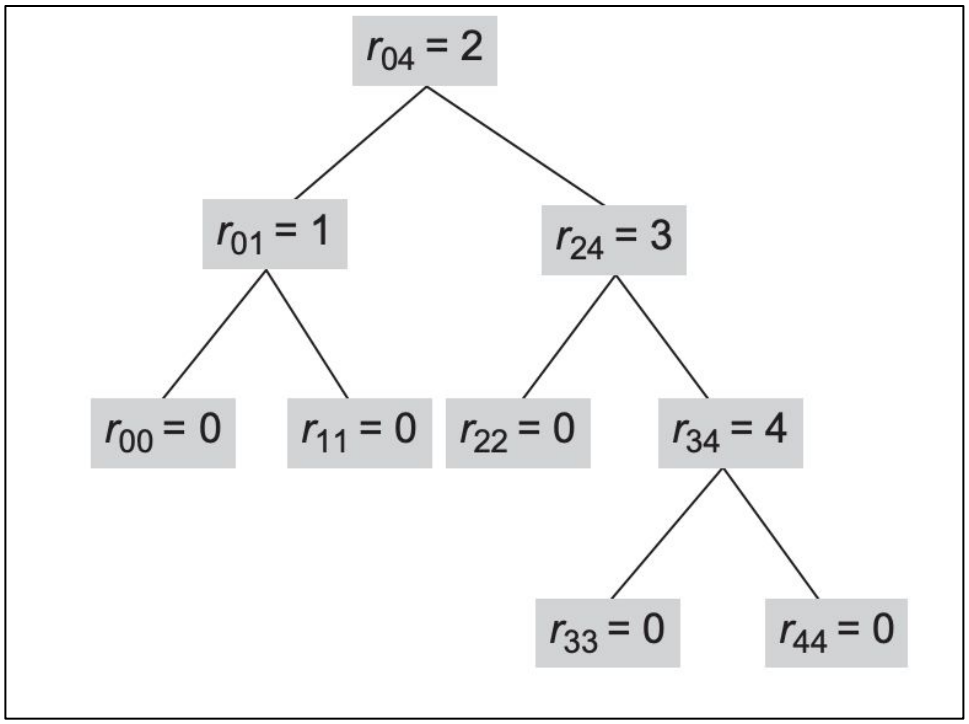


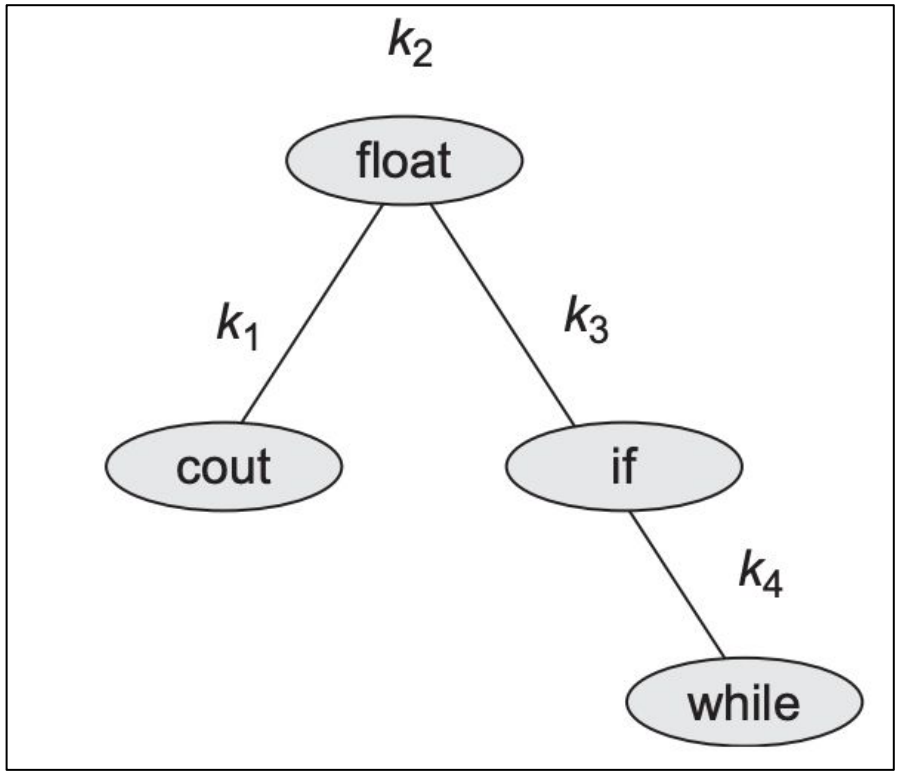
Fig. 10.8 Tree and *r* values



EXAMPLE 10.3 Find an OBST using the dynamic programming approach for $n = 4$, keys = (count, float, if, while). Compute $w(i, j)$, $r(i, j)$, and $c(i, j)$ for $0 \le i \le j \le 4$ given that $p(1) = 1/20$, $p(2) = 1/5$, $p(3) = 1/10$, $p(4) = 1/20$, $q(0) = 1/5$, $7(1) = 1/10$, $q(2) = 1/5$, $q(3) = 1/20$, and $q(4) = 1/20$. Using $r(i, j)$, construct an OBST.

	0	1	2	3	4
0	$w_{00} = 0.2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 0.1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 0.2$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 0.05$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 0.05$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 0.35$ $c_{01} = 0.35$ $r_{01} = 1$	$w_{12} = 0.5$ $c_{12} = 0.5$ $r_{12} = 2$	$w_{23} = 0.35$ $c_{23} = 0.35$ $r_{23} = 3$	$w_{34} = 0.15$ $c_{34} = 0.15$ $r_{34} = 4$	
2	$w_{02} = 0.75$ $c_{02} = 1.1$ $r_{02} = 2$	$w_{13} = 0.65$ $c_{13} = 1$ $r_{13} = 2$	$w_{24} = 0.45$ $c_{24} = 0.6$ $r_{24} = 3$		
3	$w_{03} = 0.9$ $c_{03} = 1.6$ $r_{03} = 2$	$w_{14} = 0.75$ $c_{14} = 1.35$ $r_{14} = 2$			
4	$w_{04} = 1$ $c_{04} = 1.95$ $r_{04} = 2$				


$r_{04} = 2$
$r_{01} = 1$
$r_{24} = 3$
$r_{00} = 0$
$r_{11} = 0$
$r_{22} = 0$
$r_{34} = 4$
$r_{33} = 0$
$r_{44} = 0$


k_2
float
k_1
k_3
cout
if
k_4
while

```
//OBST operations
#include<iostream>
using namespace std;
#define MAX 10
int keys[MAX],p[MAX],q[MAX];

struct node
{
    int key;
    node *left,*right;
};

class obst
{
    int n;
    int R[MAX][MAX],C[MAX][MAX],W[MAX][MAX];
```

```
public:
obst(int m)
    {
        n=m;
        for(int i=0;i<=n;i++)
        {
            for(int  j=0;j<=n;j++)
            {
            R[i][j]=0;
            C[i][j]=0;
            W[i][j]=0;
            }
        }
    }
    node* construct(int,int);
    void display(node* root);
    int cost(int*,int*,int);
};
```

```
void obst::display(node* root)
{
    if(root!=NULL)
    {
        cout<<root->key<<" ";
        display(root->left);
        display(root->right);
    }
}

node* obst::construct(int i,int j)
{
    node* p;

    if(i==j)
        p=NULL;
    else
    {
        p=new node;
        p->key=keys[R[i][j]];
        p->left=construct(i,R[i][j]-1);
        p->right=construct(R[i][j],j);
    }

    return p;
}
```

```
int obst::cost(int* p,int* q,int n)
{
    //Constructing the weight matrix
    for(int i=0;i<=n;i++)
    {
        W[i][i]=q[i];

        for(int j=i+1;j<=n;j++)
        {
            W[i][j]=W[i][j-1]+p[j]+q[j];
        }
    }
//Constructing the cost matrix
    for(int i=0;i<=n;i++)
    {

        C[i][i]=W[i][i];
    }

    for(int i=0;i<=n-1;i++)
    {
        int j=i+1;
        C[i][j]=C[i][i]+C[j][j]+W[i][j];
        R[i][j]=j;
    }
```

```
    for(int h=2;h<=n;h++)
    {
        for(int i=0;i<=n-h;i++)
        {
            int j=i+h;
            int m=R[i][j-1];
            int min=C[i][m-1]+C[m][j];
            for(int k=m+1;k<=R[i+1][j];k++)
            {
                int x=C[i][k-1] + C[k][j];
                if(x<min)
                {
                    m=k;
                    min=x;
                }
            }
            C[i][j]=W[i][j]+min;
            R[i][j]=m;
        }
    }
```

```
    cout<<"\nThe weight matrix: ";
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=n;j++)
        cout<<W[i][j]<<" ";

        cout<<"\n";
    }

    cout<<"\nThe cost matrix: ";
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=n;j++)
        cout<<C[i][j]<<" ";

        cout<<"\n";
    }

    cout<<"The root matrix: ";
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=n;j++)
        cout<<W[i][j]<<" ";

        cout<<"\n";
    }
    return C[0][n];
}
```

```
int main()
{
    int n;
    cout<<"\nEnter the number of keys: ";
    cin>>n;
    obst obj(n);

    for(int i=1;i<=n;i++)
    {
        cout<<"Keys["<<i<<"]:";
        cin>>keys[i];
        cout<<"p["<<i<<"]:";
        cin>>p[i];
    }
 for(int i=0;i<=n;i++)
    {
       cout<<"q["<<i<<"]:";
        cin>>q[i];
    }

    int c=obj.cost(p,q,n);
    cout<<"\nMinimum cost: "<<c;
    node* root=obj.construct(0,n);
    cout<<"\nInorder traversal";
    obj.display(root);
    cout<<"\n";
    return 0;
}
```

Height Balanced Tree- AVL tree

- An AVL tree is a BST where the heights of the left and right subtrees of the root differ by utmost 1 and the left and right subtrees are again AVL trees.
- The formal definition is as follows:

Definition: An empty tree is height-balanced, if T is a non-empty binary tree with T_L and T_R as its left and right subtrees, respectively, with the following properties:

1. T_L and T_R are height-balanced.
2. $-1 \leq |h_L - h_R| \leq 1$, where h_L and h_R are the heights of T_L and T_R, respectively.

- In an AVL tree with n nodes, the searches, insertions, and deletions can all be achieved in time O(log n), even in the worst case.
- To keep the tree height-balanced, we have to find out the balance factor of each node in the tree after every insertion or deletion.

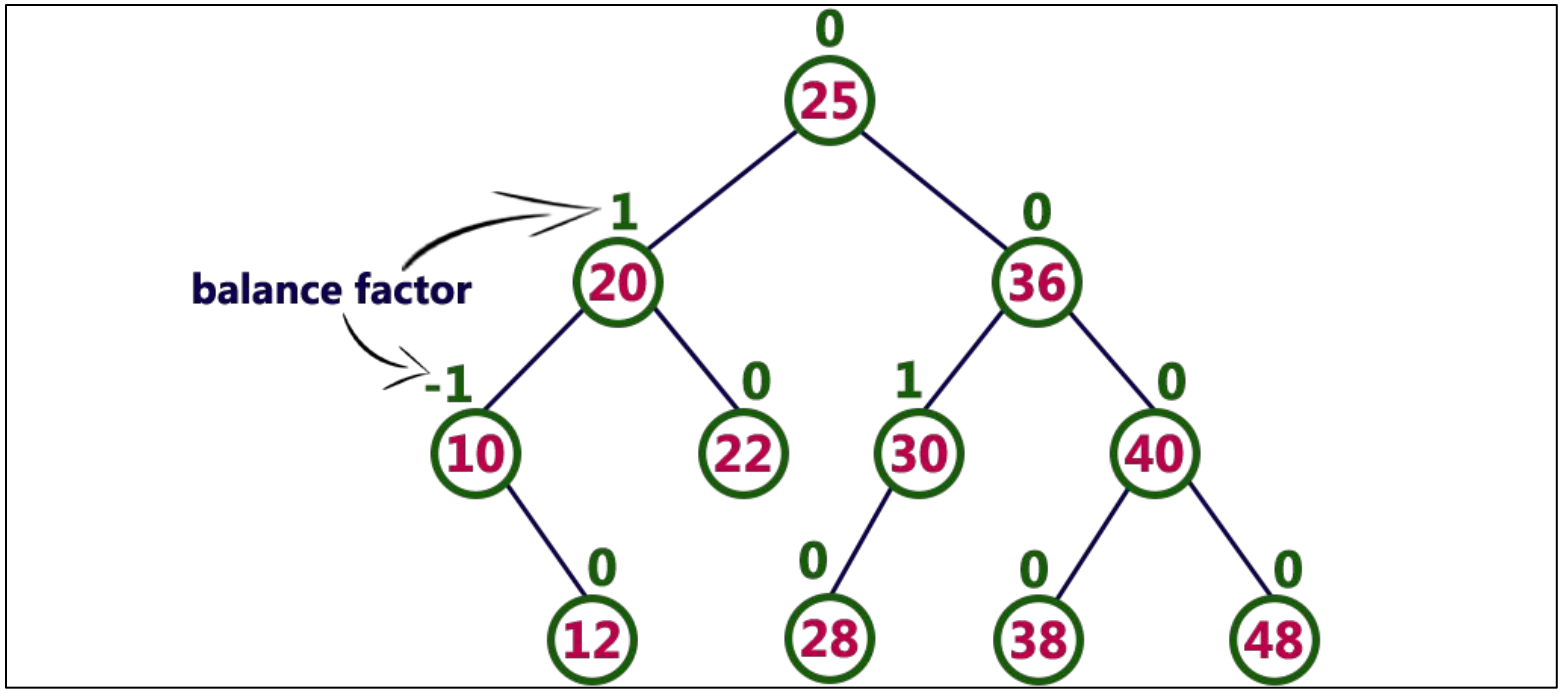
The balance factor of a node T, BF(T), in a binary tree is $h_L - h_R$, where h_L and h_R are the heights of the left and right subtrees of T, respectively. For any node T in an AVL tree, the BF(T) is equal to -1, 0, or 1.

Balance Factor

- In an AVL tree, the balance factor of a node is the height difference between its left and right subtrees. It ensures the tree remains balanced, with a balance factor of 0 indicating equilibrium.
- All nodes have balance factors within the range [-1, 1], indicating a balanced AVL tree. Mathematically, the balance factor (BF) of a node N is given by:

$$b_f = h_l - h_r$$

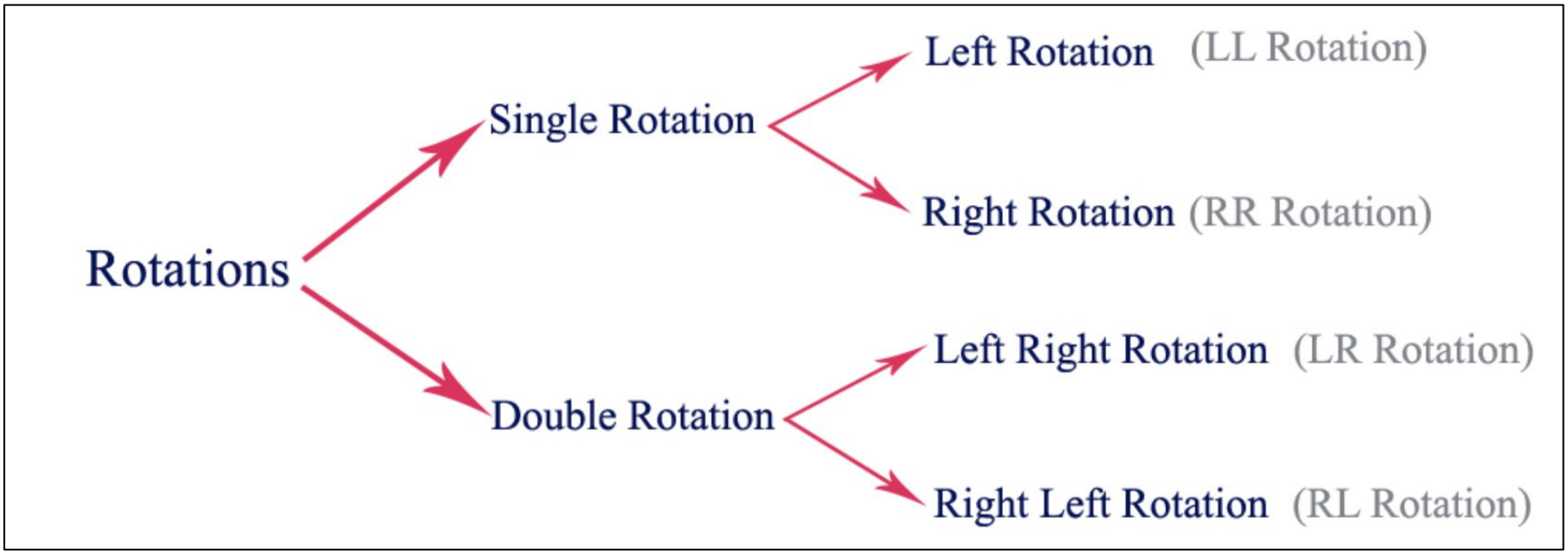
Example of AVL Tree

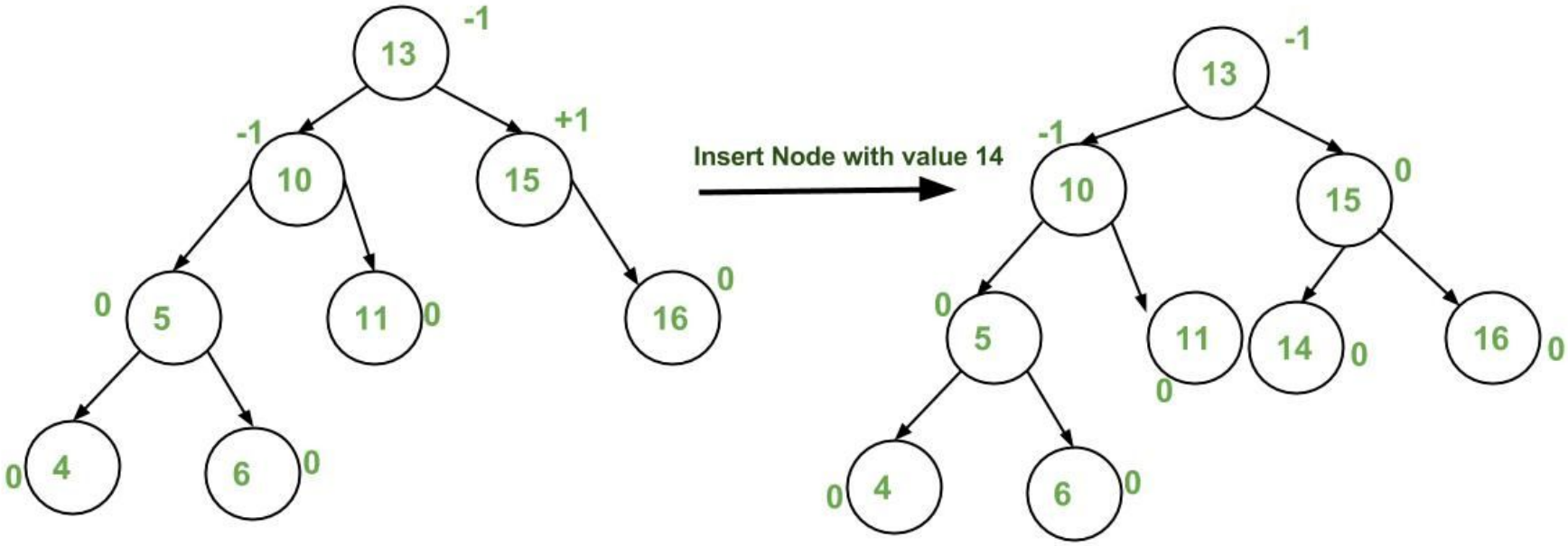


AVL Tree Rotation

- In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.Rotation operations are used to make the tree balanced.
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.

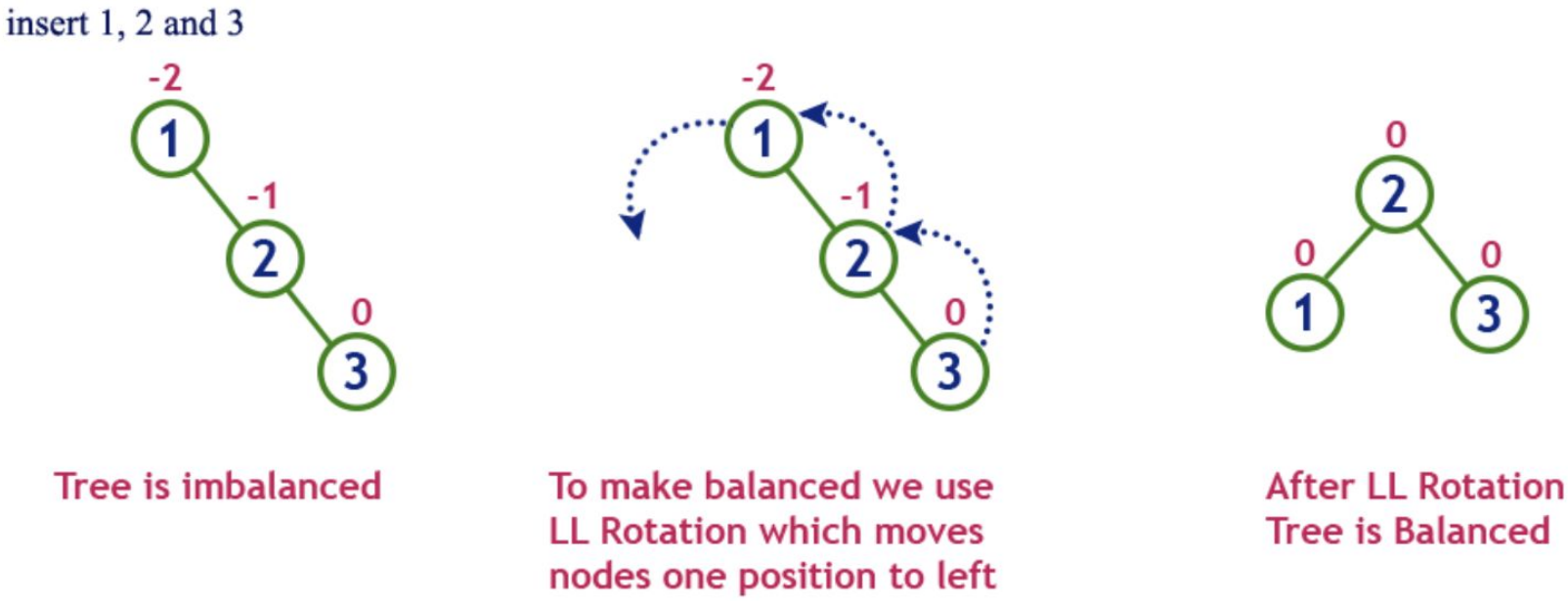
- There are four rotations and they are classified into two types.

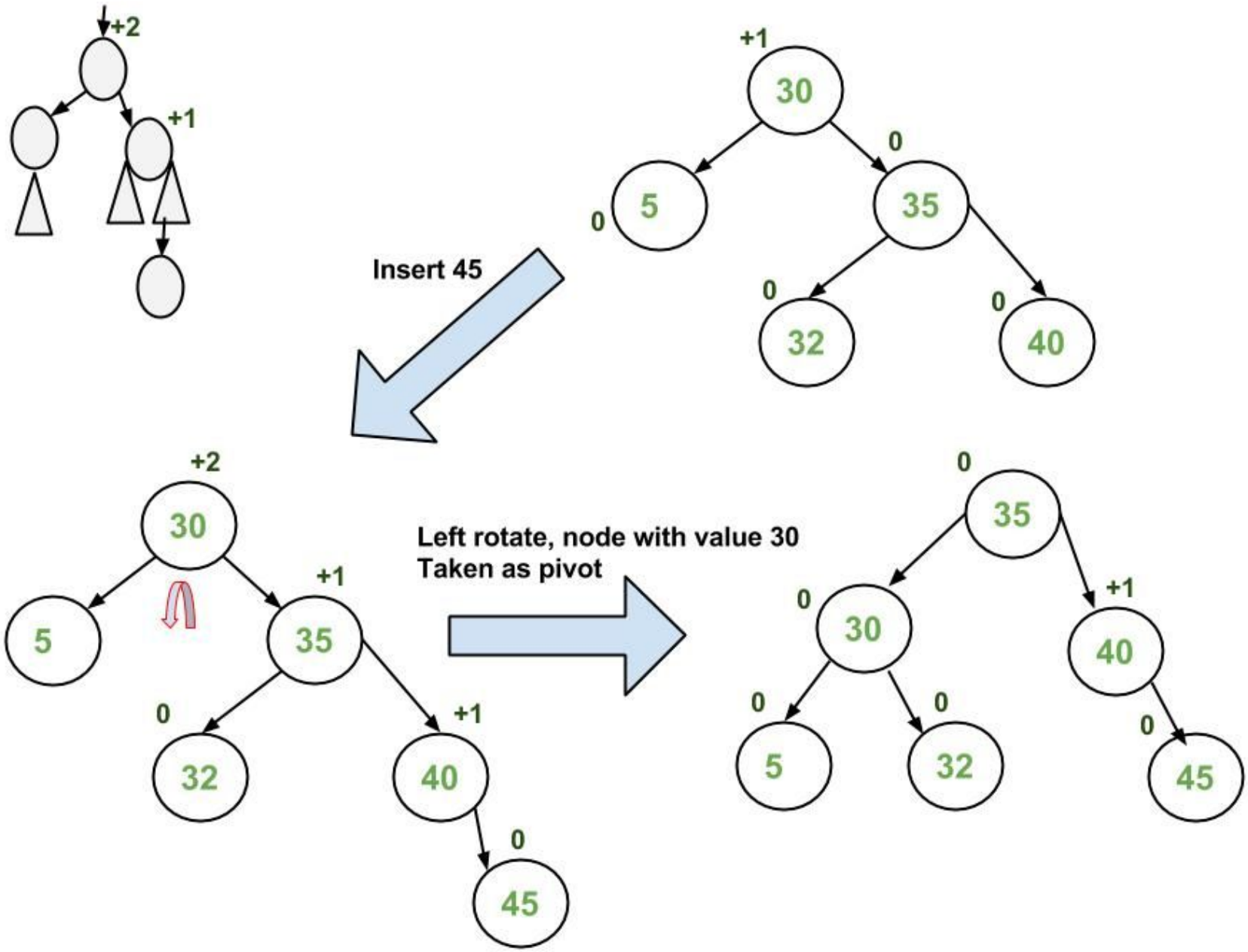



13
-1
-1
10
+1
15
0
5
11
0
16
0
0
4
6
0
Insert Node with value 14
13
-1
-1
10
15
0
0
5
11
14
0
16
0
0
0
4
6
0

Single Left Rotation (LL Rotation) -

- If a node in a tree becomes unbalanced due to the nodes of the right subtree, then we left-rotate the parent node to balance the tree.
- Left rotation is performed on a right-skewed binary search tree.
- In LL Rotation, every node moves one position to left from the current position.
- To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

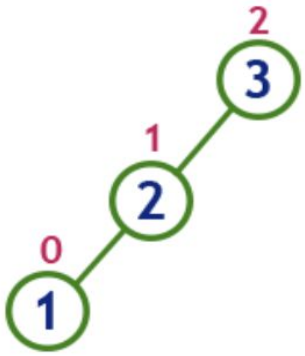



+2
+1
+1
30
0
5
35
0
32
0
40
Insert 45
+2
30
+1
5
35
0
+1
32
40
0
45
Left rotate, node with value 30
Taken as pivot
0
35
0
+1
30
40
0
0
0
5
32
45

Single Right Rotation (RR Rotation)

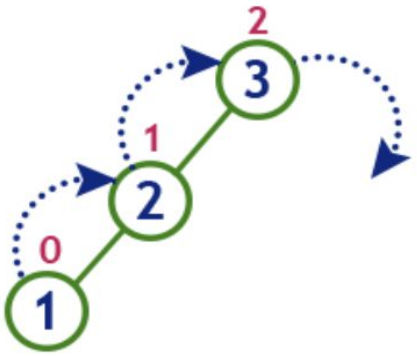
- If a tree is not height-balanced due to the left subtree, we rotate it in the rightward direction to balance it.
- Right rotation is performed from left-skewed binary search trees.
- In RR Rotation, every node moves one position to right from the current position.
- To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

insert 3, 2 and 1

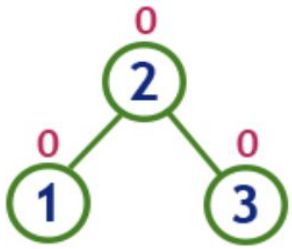


Tree is imbalanced

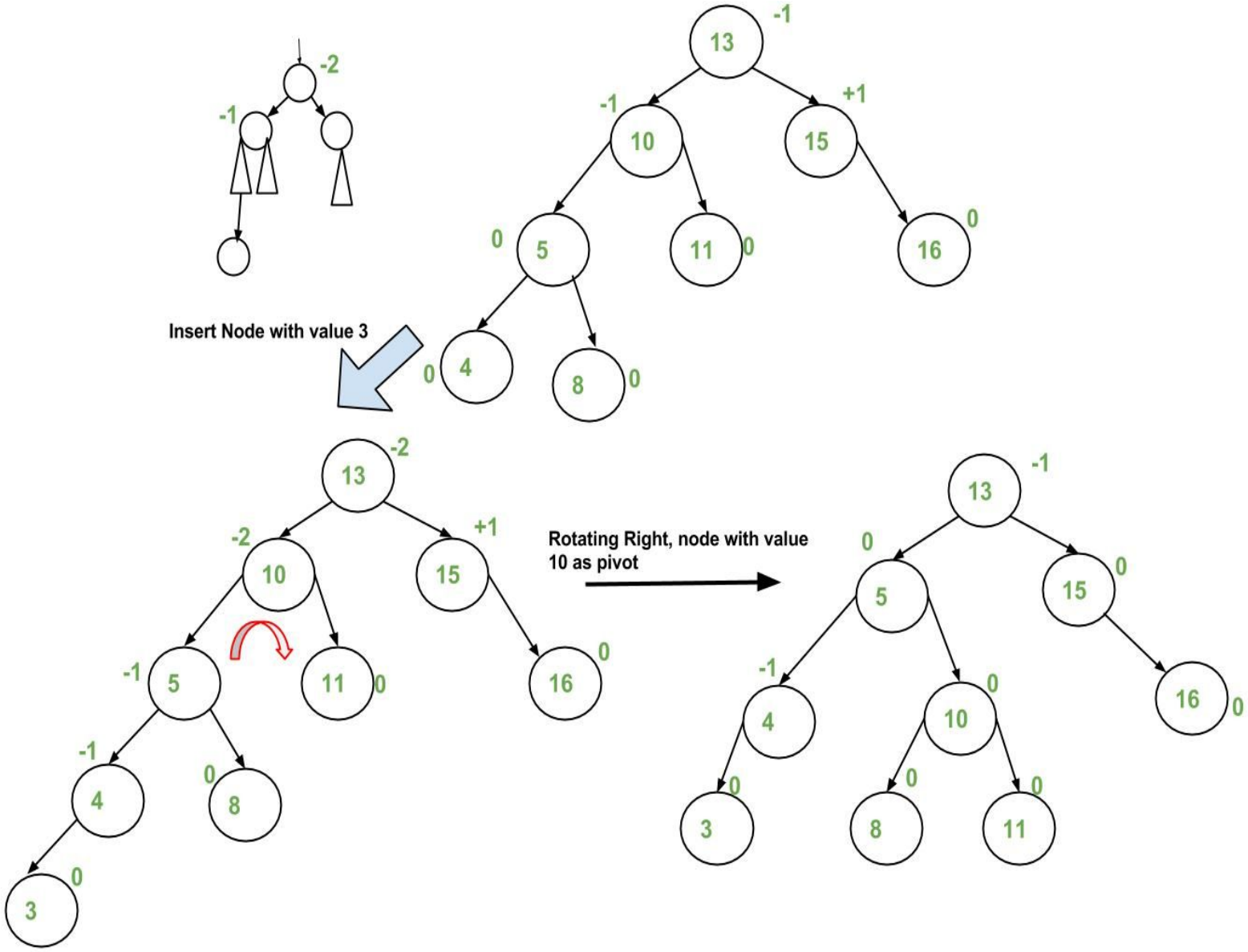
because node 3 has balance factor 2



To make balanced we use RR Rotation which moves nodes one position to right

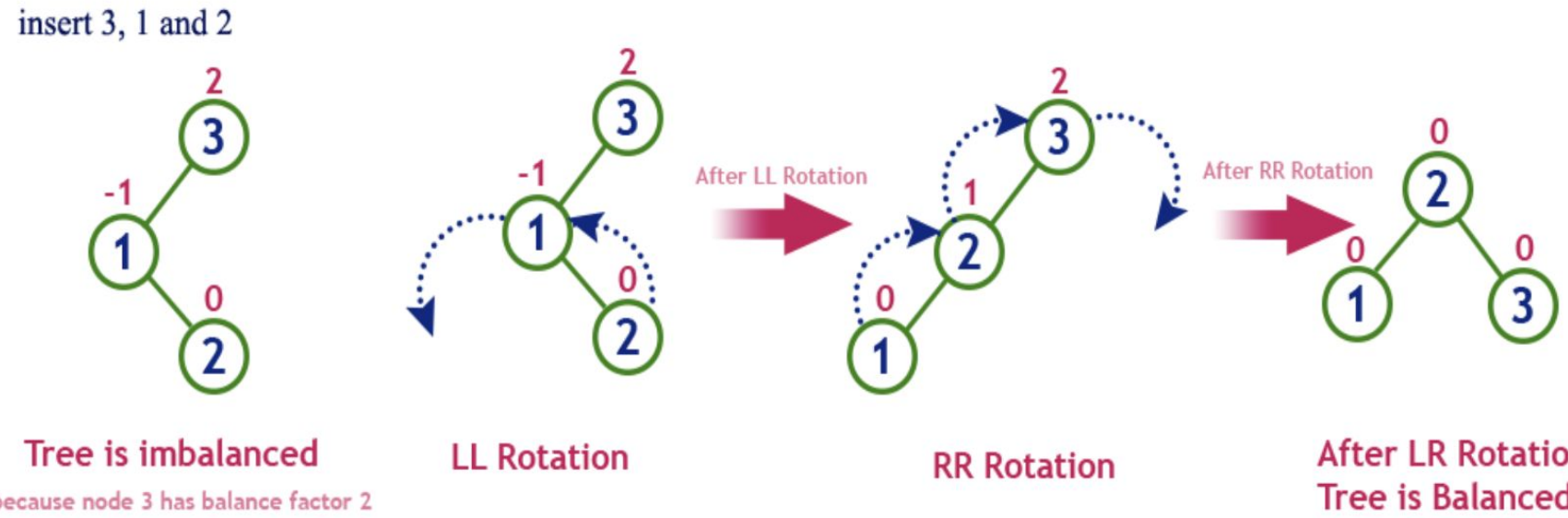


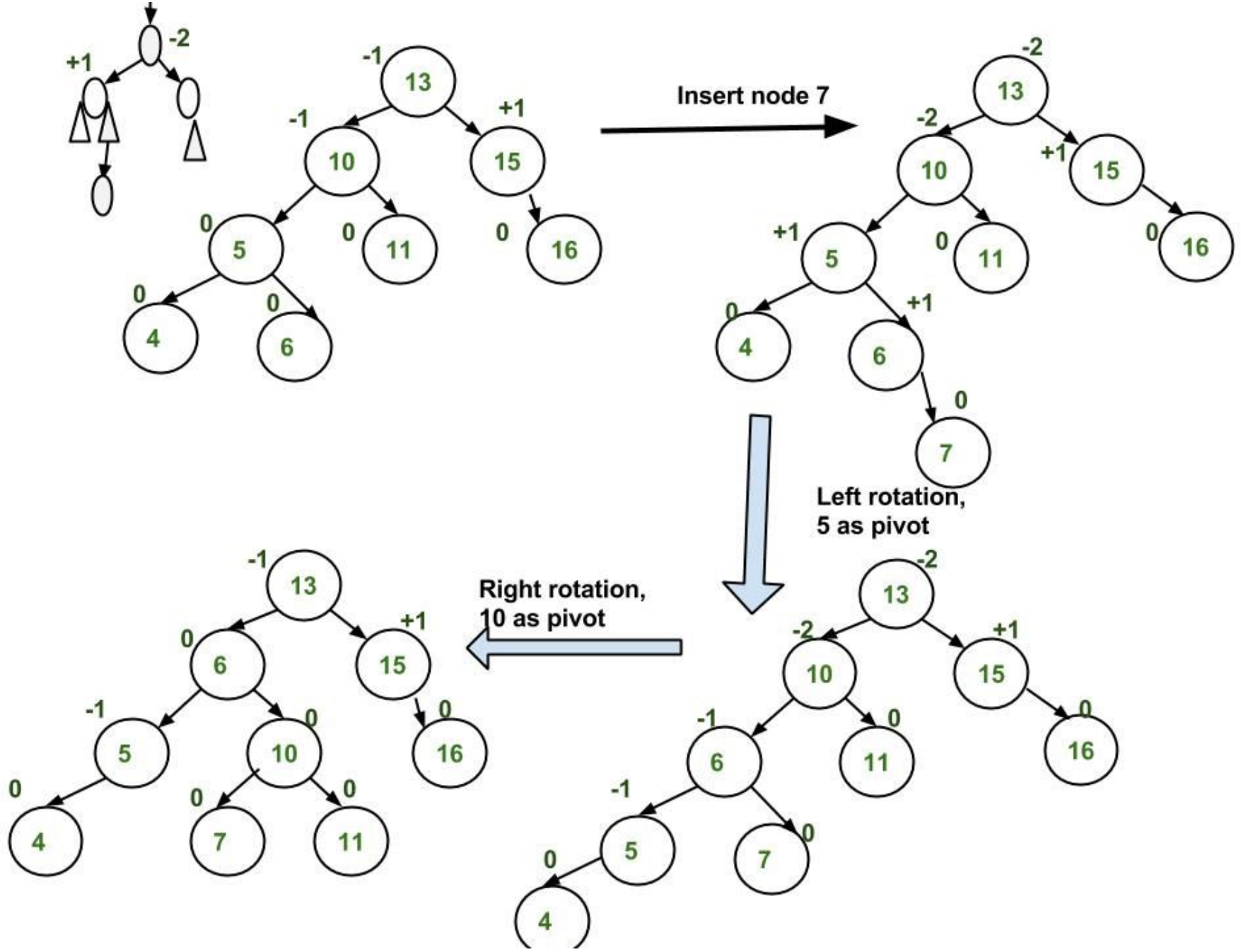
After RR Rotation Tree is Balanced


-2
-1
13
-1
+1
10
15
0
5
11
0
16
0
Insert Node with value 3
0
4
8
0
13
-2
-2
+1
10
15
-1
5
11
0
16
0
-1
0
4
8
3
0
Rotating Right, node with value 10 as pivot
13
-1
0
5
15
0
-1
4
0
10
16
0
0
0
3
8
11

Left Right Rotation (LR Rotation)

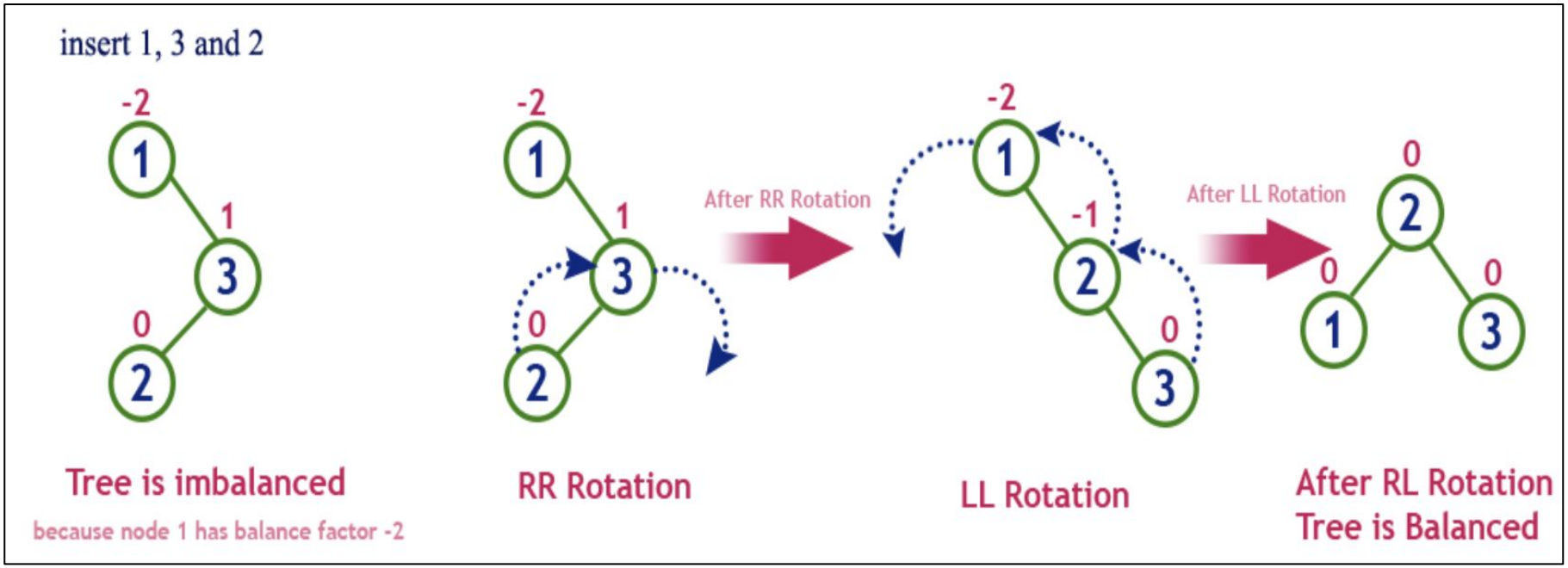
- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AVL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

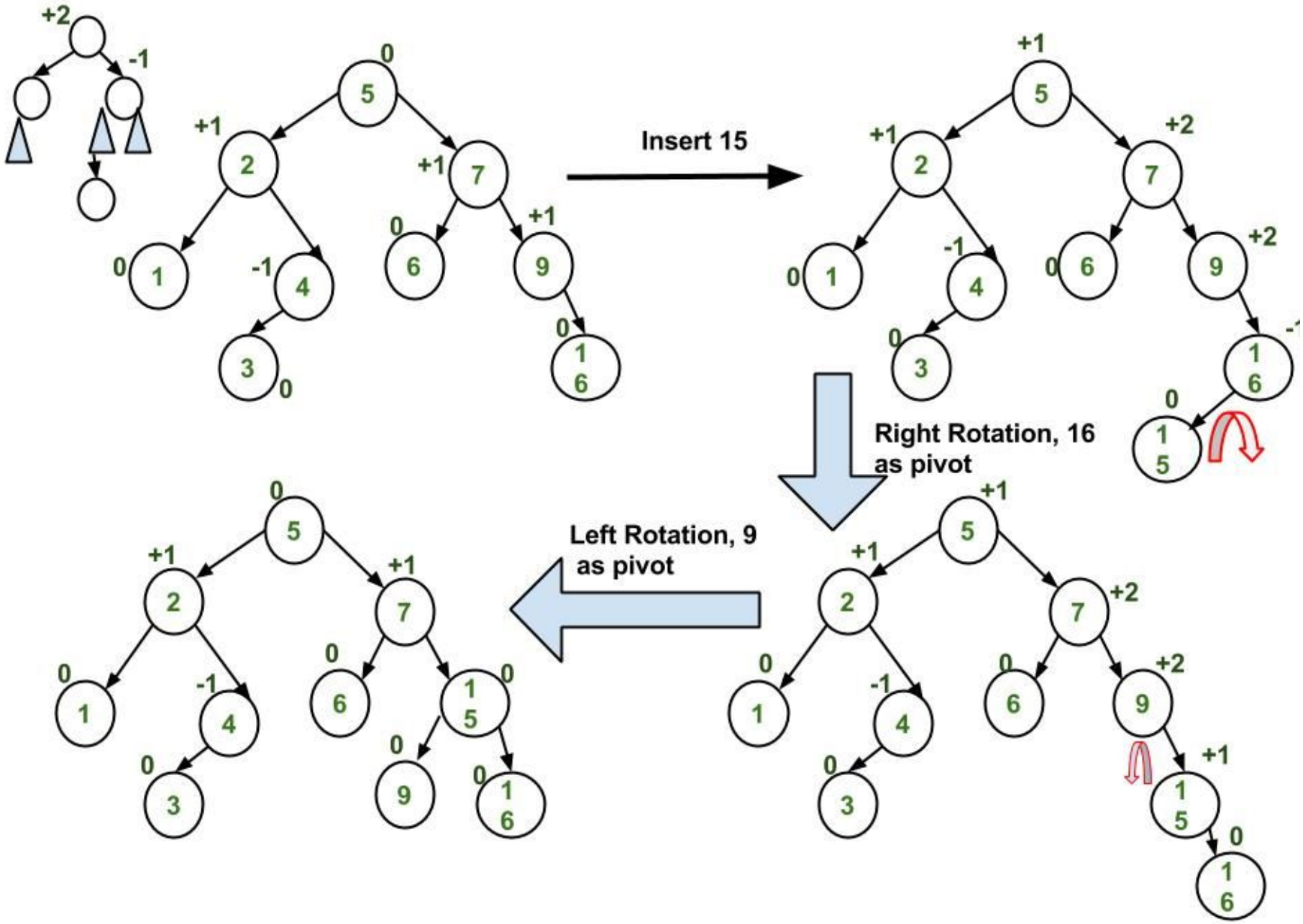



+1
-2
-1
13
-1
10
+1
15
0
5
0
11
0
16
0
4
0
6
Insert node 7
-2
13
-2
10
+1
15
+1
5
0
11
0
16
0
4
+1
6
0
7
Left rotation,
5 as pivot
-2
13
-2
10
+1
15
-1
6
0
11
0
16
-1
5
0
7
0
4
Right rotation,
10 as pivot
-1
13
0
6
+1
15
-1
5
0
10
0
16
0
4
0
7
0
11

Right Left Rotation (RL Rotation)

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




Insert 15
Right Rotation, 16
as pivot
Left Rotation, 9
as pivot

Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree -

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

Step 1 - Read the search element from the user.

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

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

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

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

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

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

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

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

Insertion Operation in AVL Tree -

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

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

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

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

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

Algorithm

- The following steps are involved in performing the insertion operation of an AVL Tree −

  Step 1 − Create a node

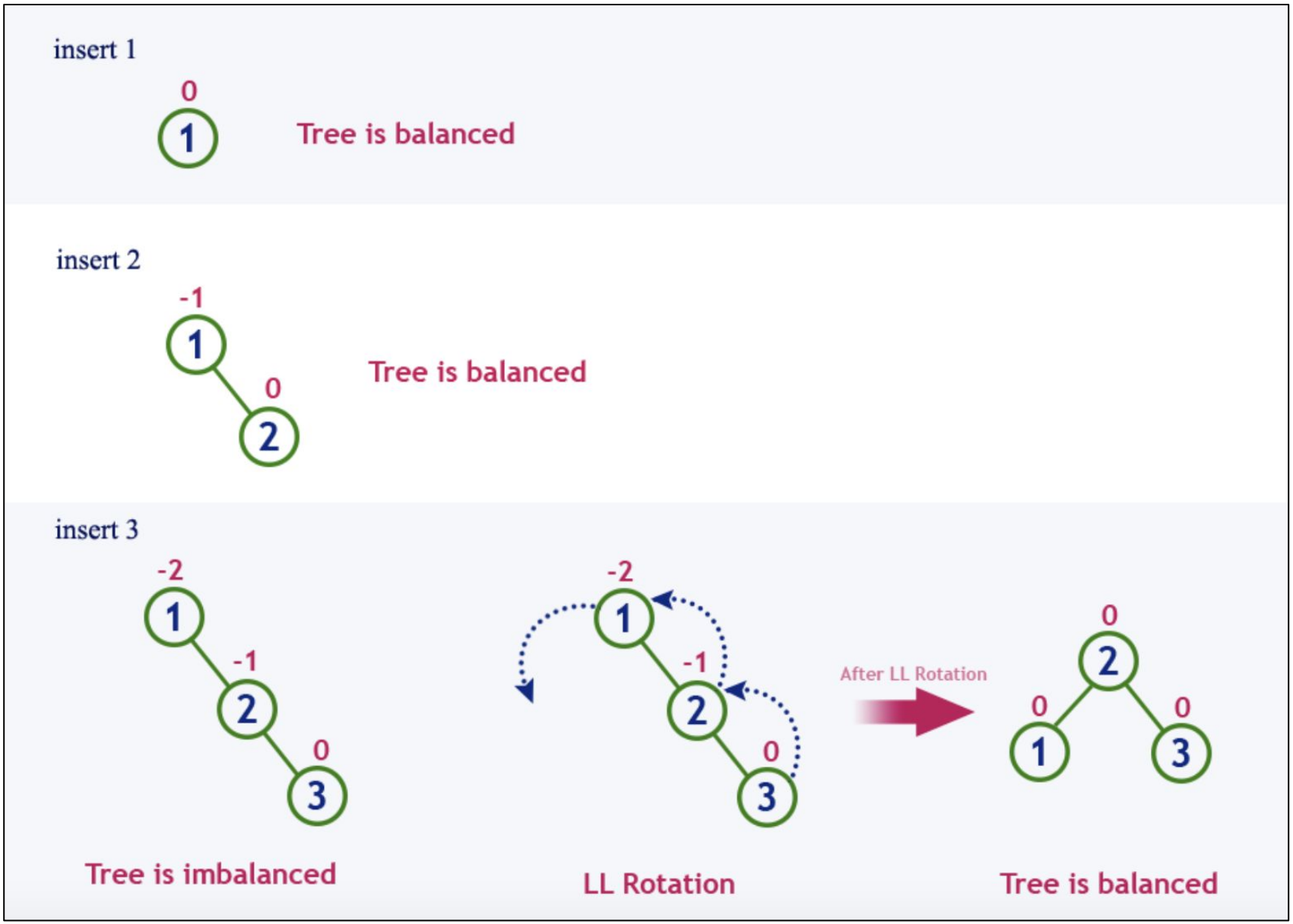
  Step 2 − Check if the tree is empty

  Step 3 − If the tree is empty, the new node created will become the root node of the AVL Tree.

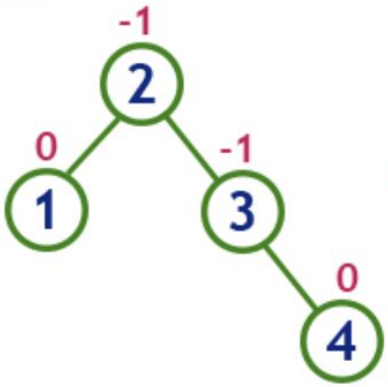
  Step 4 − If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.

  Step 5 − Suppose the balancing factor exceeds ±1, we apply suitable rotations on the said node and resume the insertion from Step 4.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

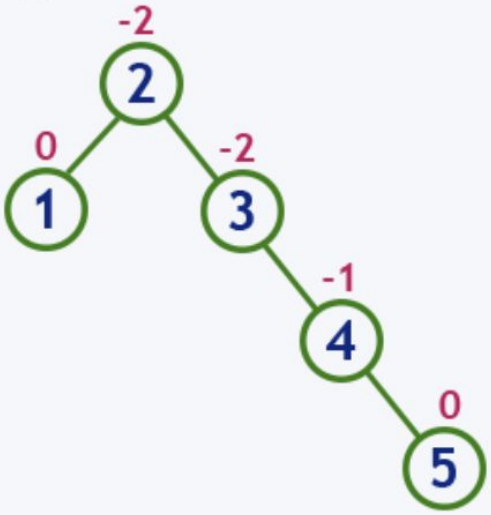


insert 4

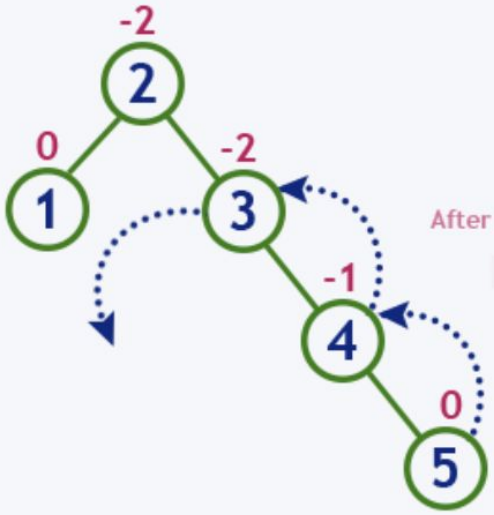


Tree is balanced

insert 5

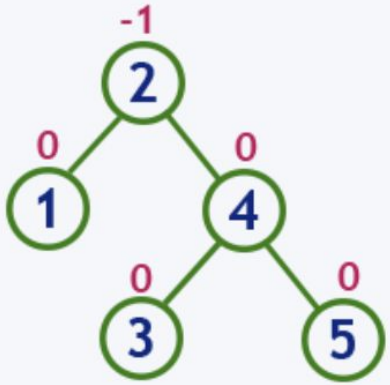


Tree is imbalanced



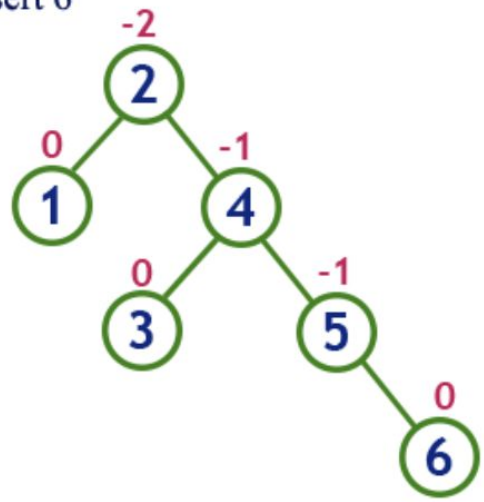
LL Rotation at 3

After LL Rotation at 3

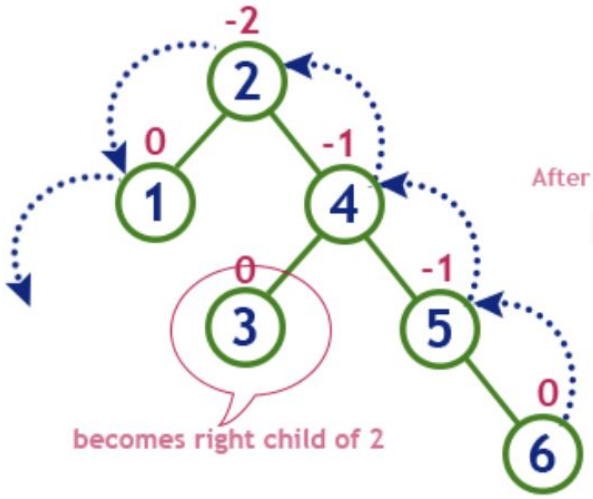


Tree is balanced

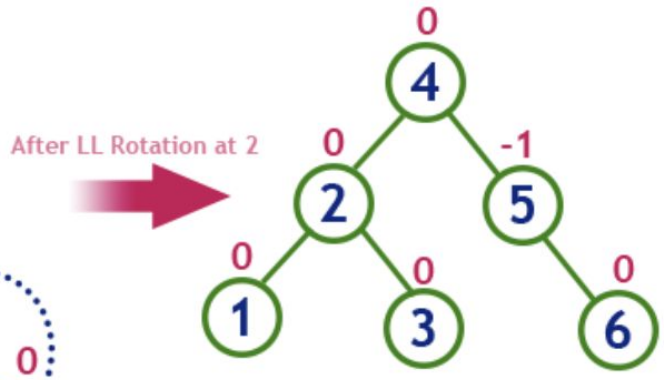
insert 6



Tree is imbalanced

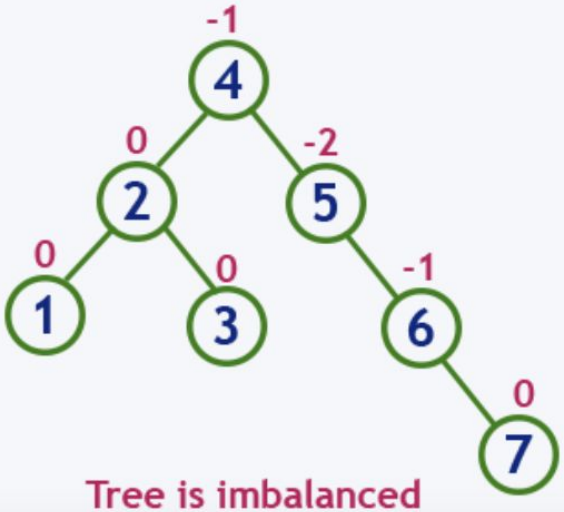


LL Rotation at 2

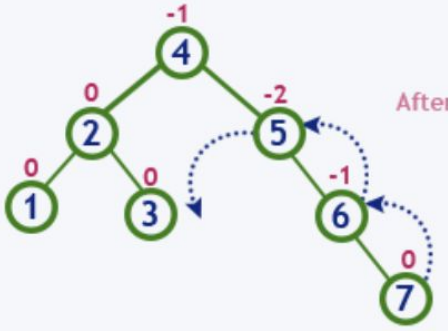


Tree is balanced

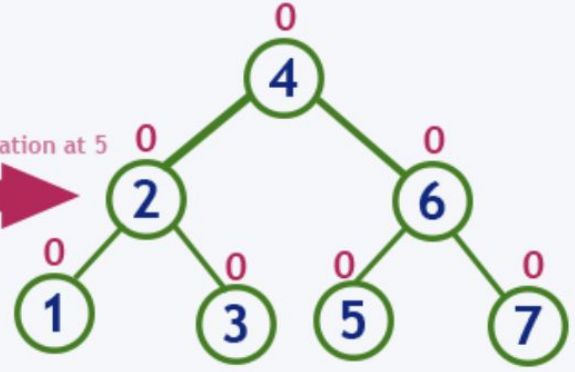
insert 7



Tree is imbalanced

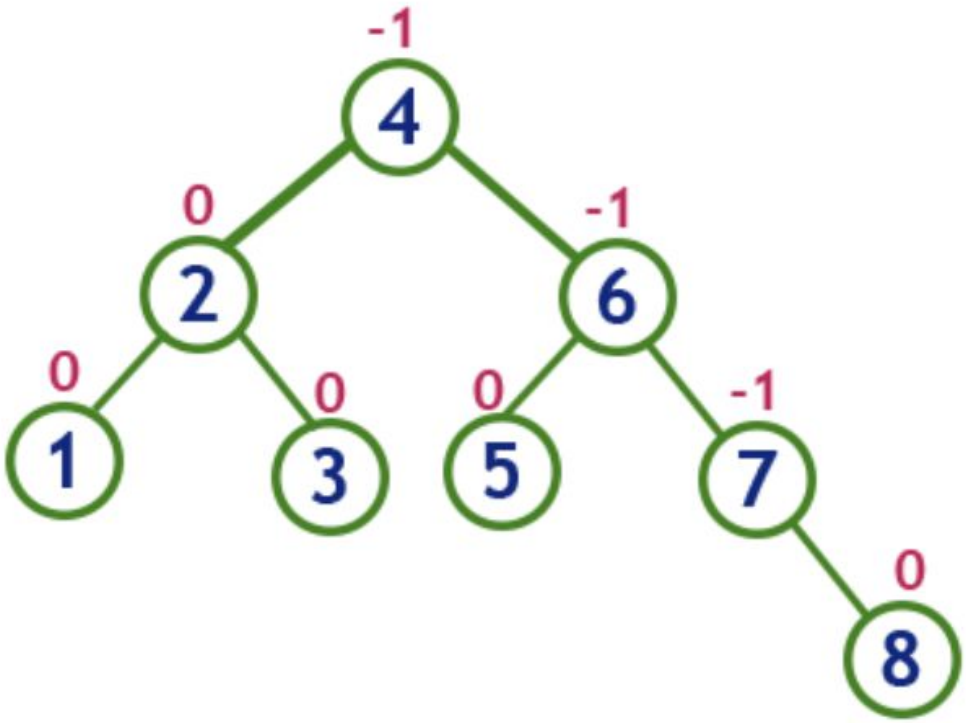


LL Rotation at 5



Tree is balanced

insert 8

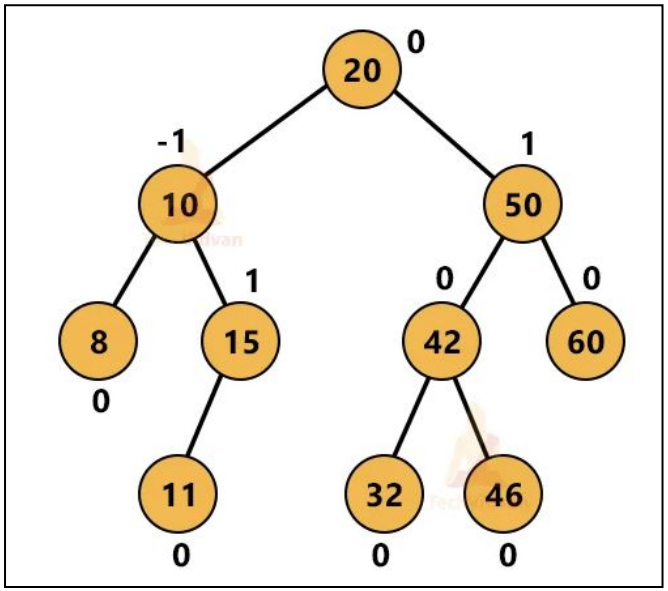


Tree is balanced

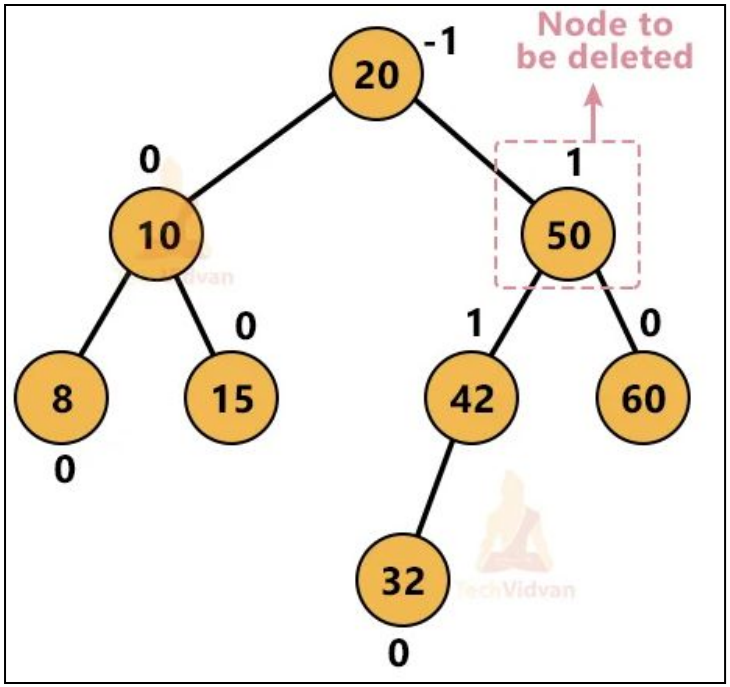
Deletion Operation in AVL Tree

- The deletion operation in AVL Tree is similar to deletion operation in BST.
- But after every deletion operation, we need to check with the Balance Factor condition.
- If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.
- We know that deletion of any node in any binary search tree is handled in three different cases:
    1. The node is a leaf node
    2. The node is an internal node having one child
    3. The node is an internal node having two child nodes

Consider the following tree:



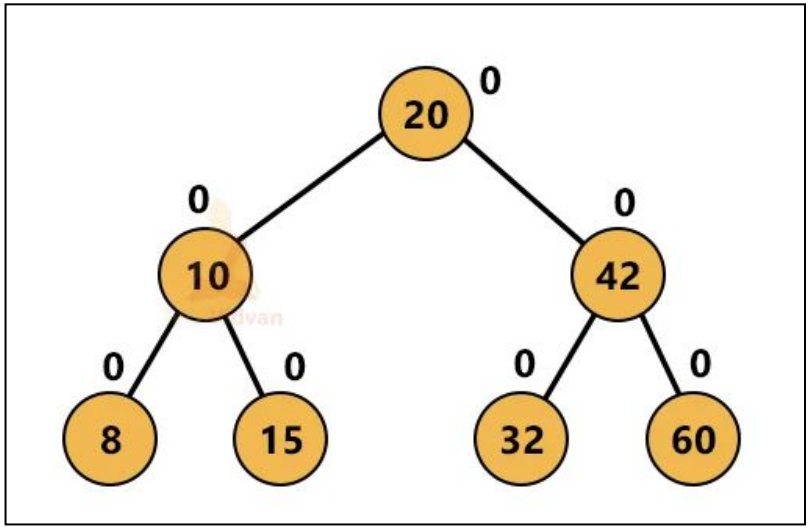
Suppose we wish to delete 50.

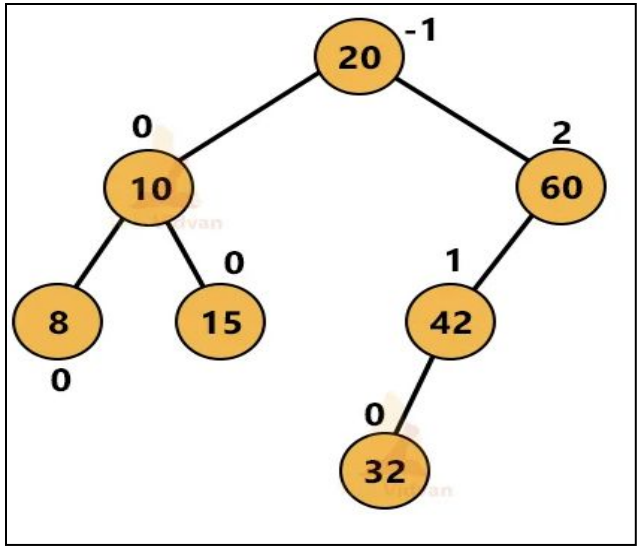


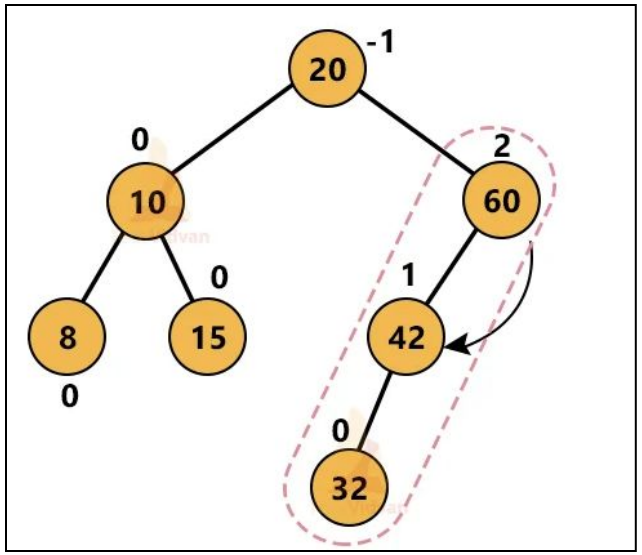
On deleting 50, 60 will be replaced by 50 and 50 will be deleted as shown:

Now, the tree has become unbalanced due to the nodes: 60, 42 and 32.

This is a left-skewed tree so will perform right rotation to balance it. Thus, after performing the right rotation, the tree will become as follows:







```
//C++ program to Implement AVL Tree

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#define pow2(n) (1 << (n))

using namespace std;

//Node Declaration

struct avl_node

{

    int data;

    struct avl_node *left;

    struct avl_node *right;

}*root;
```

```
// Class Declaration

class avlTree

{

    public:

        int height(avl_node *);

        int diff(avl_node *);

        avl_node *rr_rotation(avl_node *);

        avl_node *ll_rotation(avl_node *);

        avl_node *lr_rotation(avl_node *);

        avl_node *rl_rotation(avl_node *);

        avl_node* balance(avl_node *);

        avl_node* insert(avl_node *, int );

        avl_node *deleteAVL(avl_node *, int );

        void display(avl_node *, int);

        void inorder(avl_node *);

        avlTree()

        {

            root = NULL;

        }

};
```

```
//Main Contains Menu
int main()
{
    int choice, item,dval;
    avlTree avl;
    while (1)
    {
        cout<<"\n---------------------"<<endl;
        cout<<"AVL Tree Implementation"<<endl;
        cout<<"\n---------------------"<<endl;
        cout<<"1.Insert Element into the tree"<<endl;
        cout<<"2.Display Balanced AVL Tree"<<endl;
        cout<<"3.Delete"<<endl;
        cout<<"4.Exit"<<endl;
        cout<<"Enter your Choice: ";
        cin>>choice;
```

```
switch(choice)
        {
        case 1:
            cout<<"Enter value to be inserted: ";
            cin>>item;
            root = avl.insert(root, item);
            break;
        case 2:
            if (root == NULL)
            {
                cout<<"Tree is Empty"<<endl;
                continue;
            }
            cout<<"Balanced AVL Tree:"<<endl;
            avl.display(root, 1);
            break;
        case 3:
            cout<<"delete:"<<endl;
            cout<<"Enter value to be deleted: ";
            cin>>dval;
            root = avl.deleteAVL(root,dval);
            cout<<endl;
            break;
```

```
        case 4:
                exit(1);
                break;
        default:
            cout<<"Wrong Choice"<<endl;
           }
    }
    return 0;
}
```

```
// Height of AVL Tree
int avlTree::height(avl_node *temp)
{
    int h = 0;
    if (temp != NULL)
    {
        int l_height = height (temp->left);
        int r_height = height (temp->right);
        int max_height = max (l_height, r_height);
        h = max_height + 1;
    }
    return h;
}
```

```
//Height Difference

 int avlTree::diff(avl_node *temp)

{

    int l_height = height (temp->left);

    int r_height = height (temp->right);

    int b_factor= l_height - r_height;

    return b_factor;

}
```

```
// Right- Right Rotation
avl_node *avlTree::rr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
 }
// Left- Left Rotation
avl_node *avlTree::ll_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
}
```

```
// Left - Right Rotation
avl_node *avlTree::lr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = ll_rotation (temp);
    return rr_rotation (parent);
}

// Right- Left Rotation
avl_node *avlTree::rl_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = rr_rotation (temp);
    return ll_rotation (parent);
}
```

```
//Balancing AVL Tree
avl_node *avlTree::balance(avl_node *temp)
{
    int bal_factor = diff(temp);
    if (bal_factor > 1)
    {
        if (diff (temp->left) > 0)
            temp = rr_rotation (temp);
        else
            temp = lr_rotation (temp);
    }
    else if (bal_factor < -1)
    {
        if (diff (temp->right) > 0)
            temp = rl_rotation (temp);
        else
            temp = ll_rotation (temp);
    }
    return temp;
}
```

```
//Insert Element into the tree
avl_node *avlTree::insert(avl_node *root, int value)
{
    if (root == NULL)
    {
        root = new avl_node;
        root->data = value;
        root->left = NULL;
        root->right = NULL;
        return root;
    }
    else if (value < root->data)
    {
        root->left = insert(root->left, value);
        root = balance (root);
    }
    else if (value >= root->data)
    {
        root->right = insert(root->right, value);
        root = balance (root);
    }
    return root;
}
```

```
//Display AVL Tree
void avlTree::display(avl_node *ptr, int level)
{
    int i;
    if (ptr!=NULL)
    {
        display(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
        cout<<"Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout<<"        ";
        cout<<ptr->data;
        display(ptr->left, level + 1);
    }
}
//Inorder Traversal of AVL Tree
void avlTree::inorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    inorder (tree->left);
    cout<<tree->data<<"  ";
    inorder (tree->right);
}
```

```
avl_node *avlTree::deleteAVL(avl_node *root, int dval)
{
    avl_node *temp;
    if (root != NULL)
    {
        if (dval < root->data)
        {
            root->left = deleteAVL(root->left, dval);
            if (diff(root) == -2)
            {
                if (diff(root->right) <= 0)
                {
                    cout << "\n RR rotation \n";
                    root = rr_rotation(root);
                }
                else
                {
                    cout << "\n RL rotation \n";
                    root = rl_rotation(root);
                }
            }
        }
```

```
else if (dval > root->data)
        {
            root->right = deleteAVL(root->right, dval);
            if (diff(root) == 2)
            {
                if (diff(root->left) >= 0)
                {
                    cout << "\n LL rotation \n";
                    root = ll_rotation(root);
                }
                else
                {
                    cout << "\n LR rotation \n";
                    root = lr_rotation(root);
                }
            }
        }
```

```
        else
        {
            if (root->right == NULL)
                return (root->left);
            else
            {
                // find leftmost of right
                temp = root->right;
                while (temp->left != NULL)
                    temp = temp->left;
                root->data = temp->data;
                root->right = deleteAVL(root->right, temp->data);
                if (diff(root) == 2)
                {
                    if (diff(root->left) >= 0)
                        root = ll_rotation(root);
                    else
                        root = lr_rotation(root);
                }
            }
        }
    }
    else
        return NULL;
  return root;
}
```

Examples on AVL Tree

1. Construct an AVL tree for the following data: 30, 31, 32, 23, 22, 28, 24, 29, 26, 27, 34, 36
2. Construct an AVL tree for the following data: STA, add, lda , mov , jmp, trim, xchg, mVi, div, nop, in, jnz
3. Construction of AVL tree (a) Key = STA (b) Key = ADD (c) Key = LDA (d) Key = MOV (e) Key = JMP (f) Key = TRIM
4. Construct an AVL tree for the set of keys = {50, 55, 60, 15, 10, 40, 20, 45, 30, 70, 80}

Applications of AVL Tree:

- It is used to index huge records in a database and also to efficiently search in that.
- For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
- Database applications, where insertions and deletions are less common but frequent data lookups are necessary Software that needs optimized search.
- It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

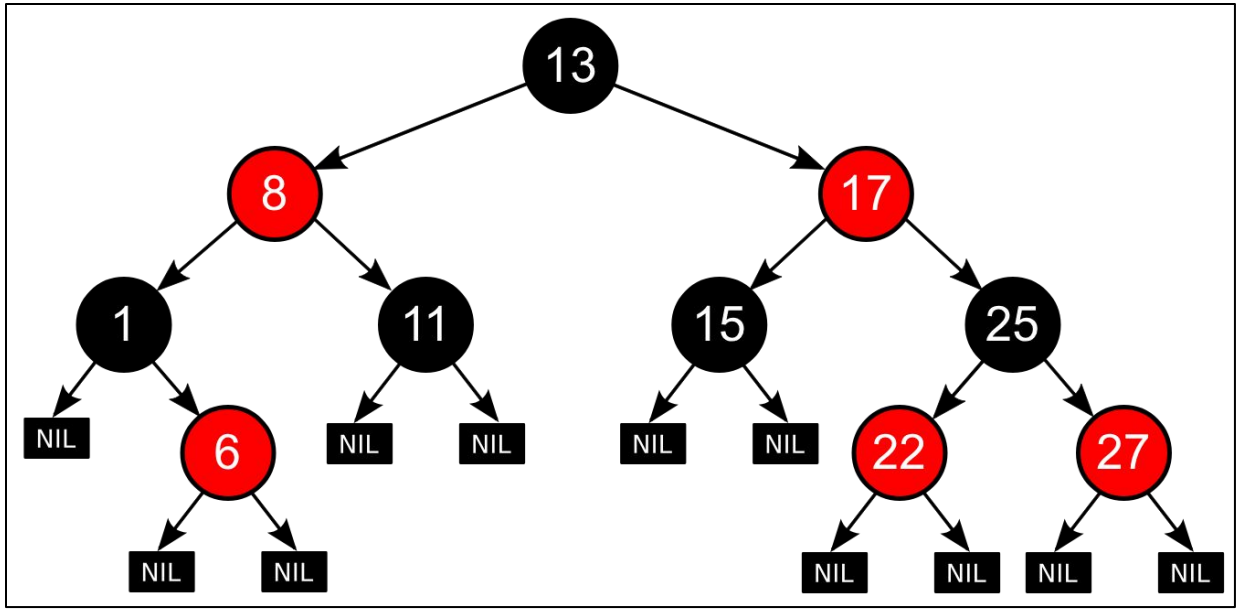
1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

Red-Black Tree

- Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.
- A red–black tree is a BST with one extra bit of storage per node: its colour, which can either be red or black.
- The tree is balanced by constraining the way nodes can be coloured on any path from the root to a leaf; red-black tree ensures that no such path is more than twice as long as any other.



Red-Black Tree

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree.

Every Red Black Tree has the following properties.

Properties of Red Black Tree

1. Red - Black Tree must be a Binary Search Tree.
2. The ROOT node must be colored BLACK.
3. The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).There are no two adjacent red nodes (A red node cannot have a red parent or red child)
4. Every simple path from root to descendant leaf node contains same number of black nodes.
5. Every new node must be inserted with RED color.
6. All the external nodes (leaf nodes) must be colored BLACK.

Insertion into RED BLACK Tree

While inserting a new node, the new node is always inserted as RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations:-

1. **Recolor**
2. **Rotation**

To understand the insertion operation, let us understand the keys required to define the following nodes:

1. Let **u** be the newly inserted node.
2. **p** is the parent node of **u**.
3. **g** is the grandparent node of **u**.
4. **Un** is the uncle node of **u**.

Insertion into RED BLACK Tree

Case -1

- The imbalancing is concerned with the color of grandparent child i.e., Uncle Node.
- If **uncle node is red** then four cases arises, and by doing **recoloring** imbalancing can be removed.
- **Left Right imbalance:**
  - In this, Red-Black Tree violates its property in such a manner that parent and inserted child will be in red color at a position of left and right with respect to grandparent.
  - Therefore, it is termed as LEFT RIGHT imbalance.

Insertion into RED BLACK Tree

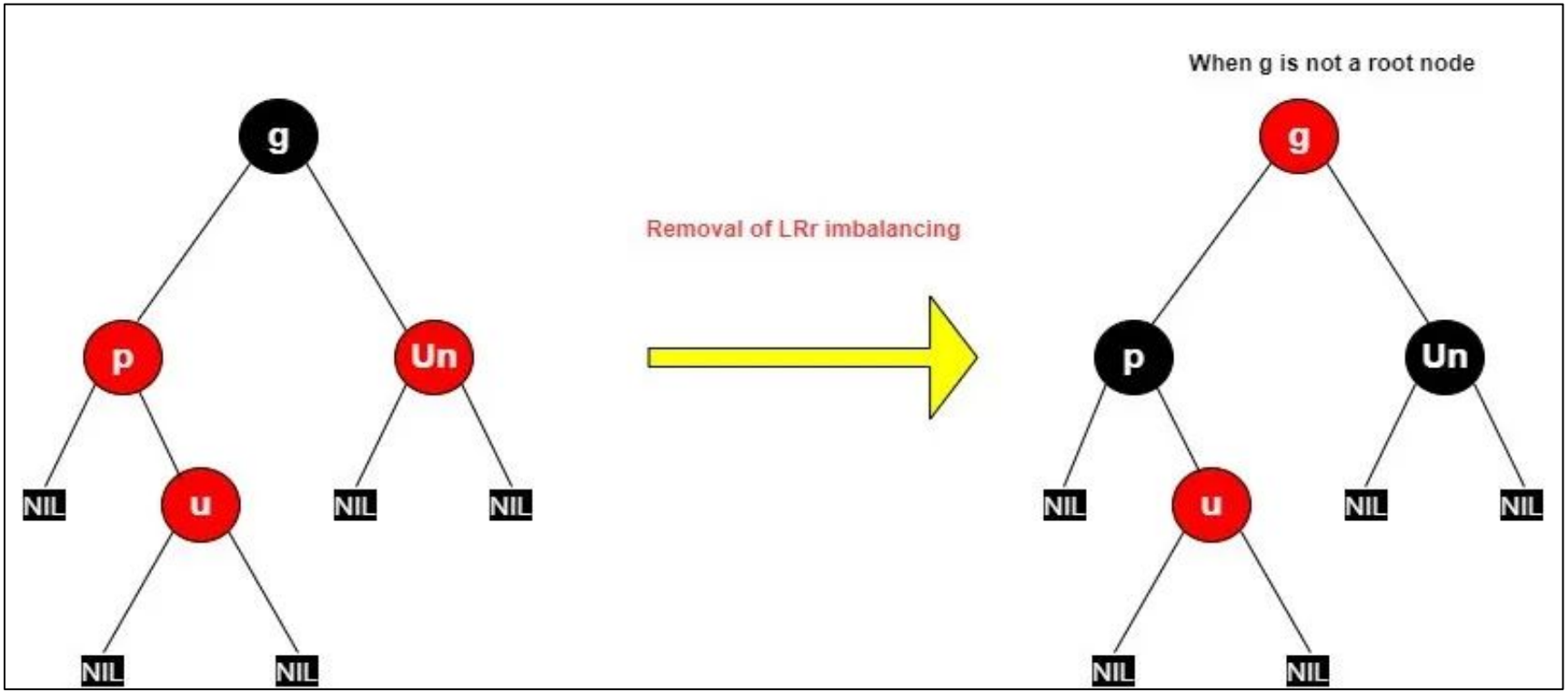
Removal of LRr imbalance can be done by:

i. Changing the color of node **p** from red to black.

ii. Changing the color of node **Un** from red to black.

iii. Changing the color of node **g** from black to red, if **g** is not a root node.

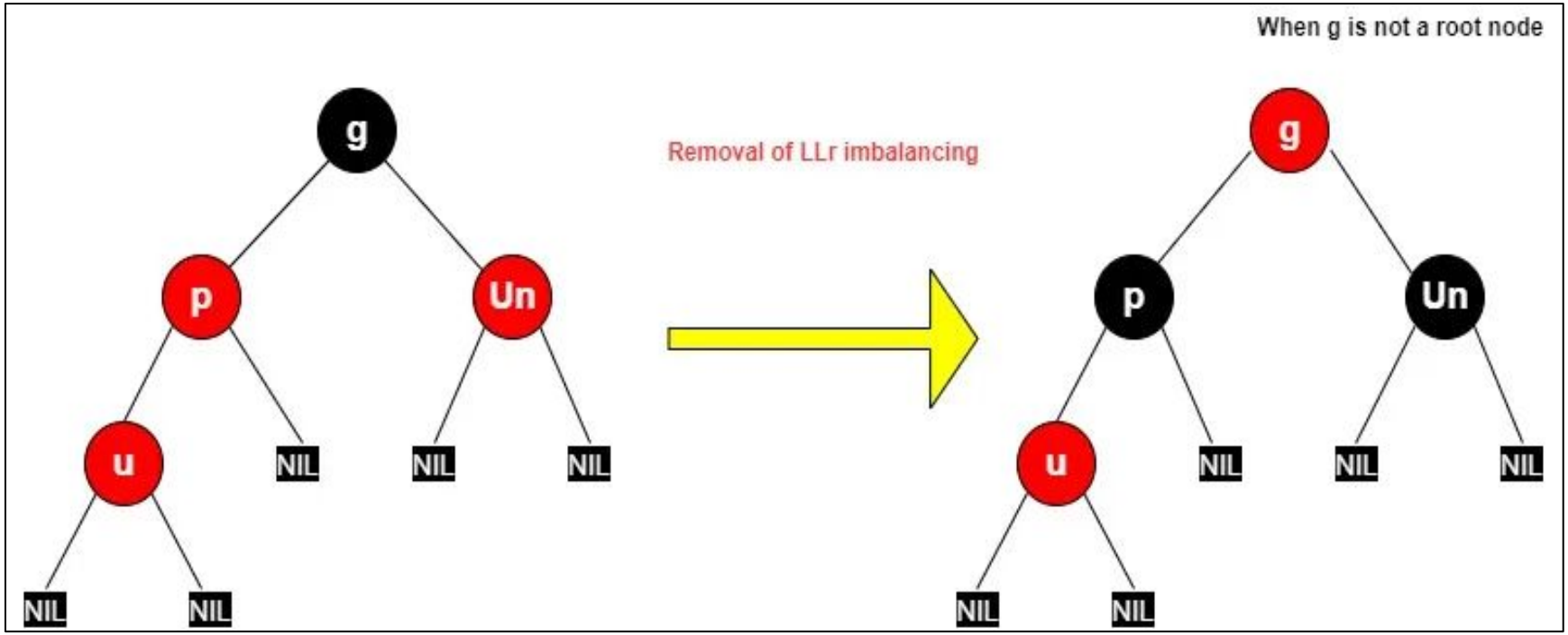
Note: If given g is root node then there will be no changes in color of g.



Insertion into RED BLACK Tree

2. Left Left imbalance:

- In this, Red-Black Tree violates its property in such a manner that parent and inserted child will be in red color at a position of left and left with respect to grandparent.
- Therefore, it is termed as LEFT LEFT imbalance.



Insertion into RED BLACK Tree

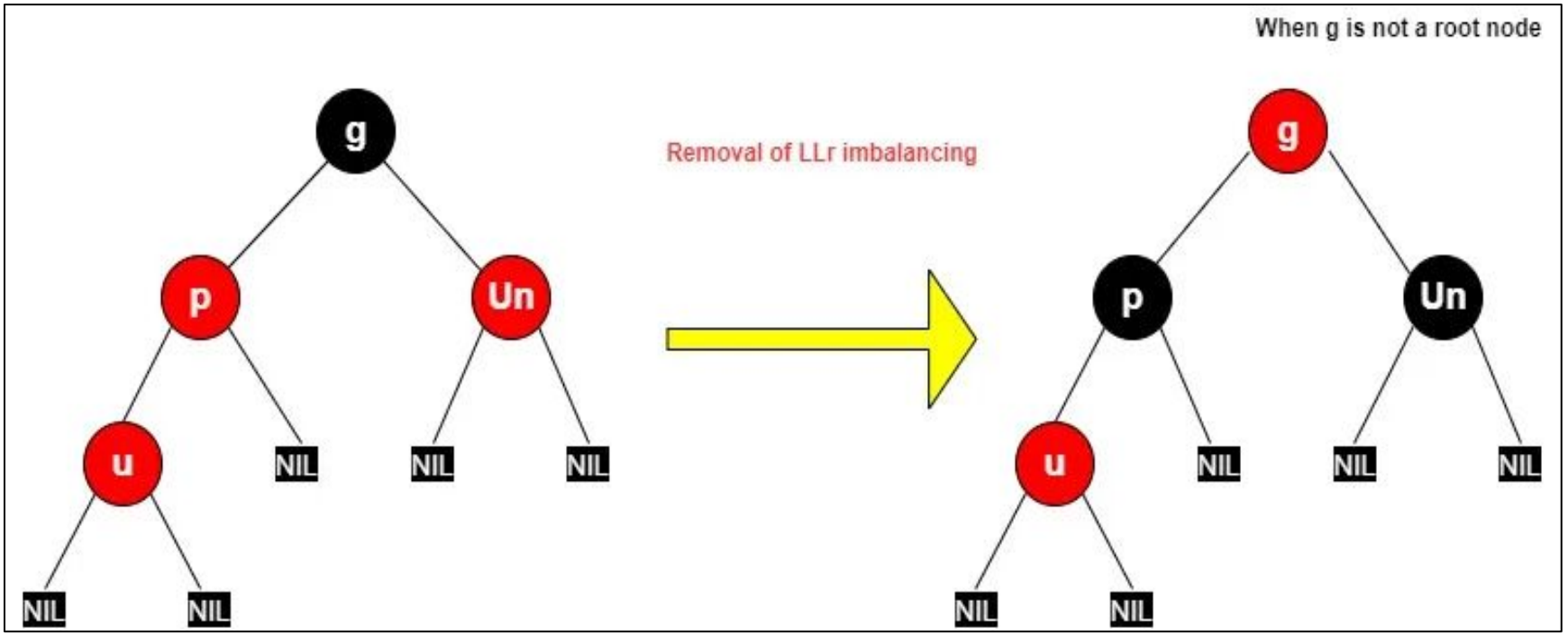
Removal of LLr imbalance can be done by:

i. Changing the color of node **p** from red to black.

ii. Changing the color of node **Un** from red to black.

iii. Changing the color of node **g** from black to red, if **g** is not a root node.

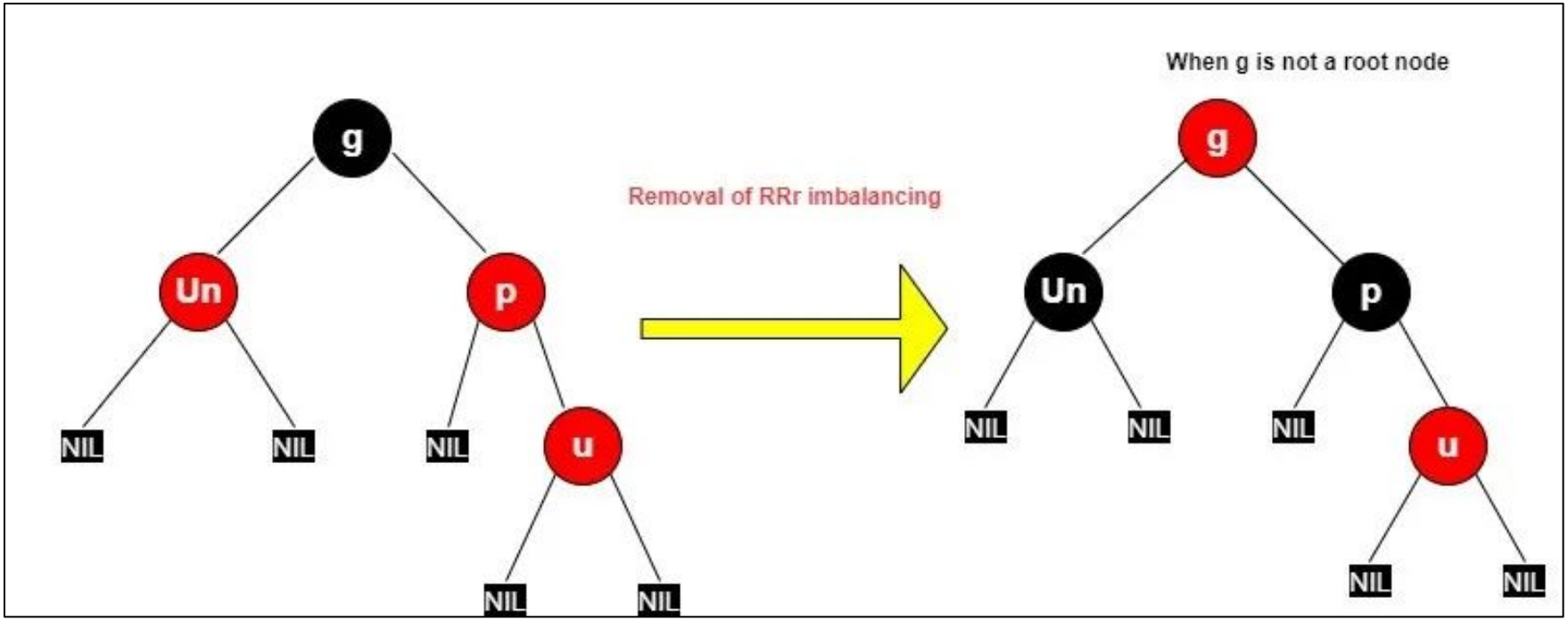
Note: If given **g** is root node then there will be no changes in color of **g**.



Insertion into RED BLACK Tree

3. Right Right imbalance:

- In this, Red-Black Tree violates its property in such a manner that parent and inserted child will be in red color at a position of right and right with respect to grandparent.
- Therefore, it is termed as RIGHT RIGHT imbalance.



Insertion into RED BLACK Tree

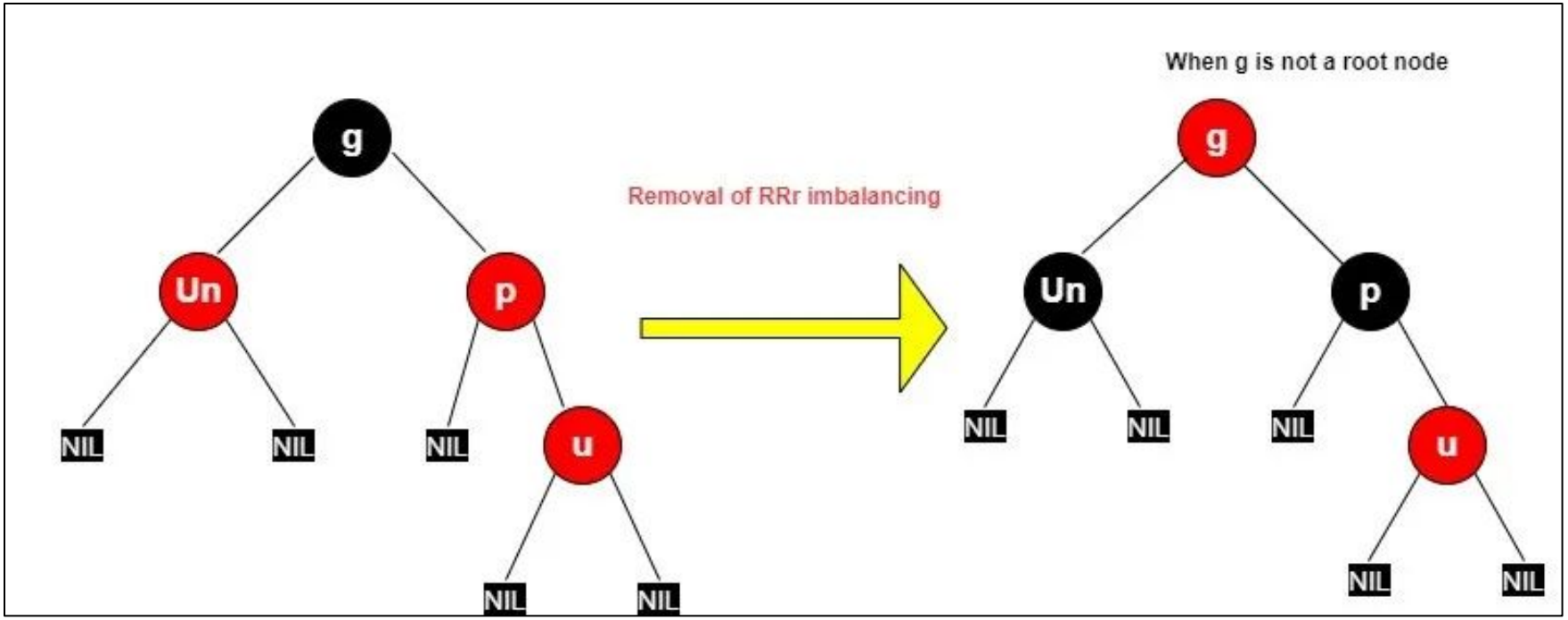
Removal of RRr imbalance can be done by:

i. Changing the color of node **p** from red to black.

ii. Changing the color of node **Un** from red to black.

iii. Changing the color of node **g** from black to red, if **g** is not a root node.

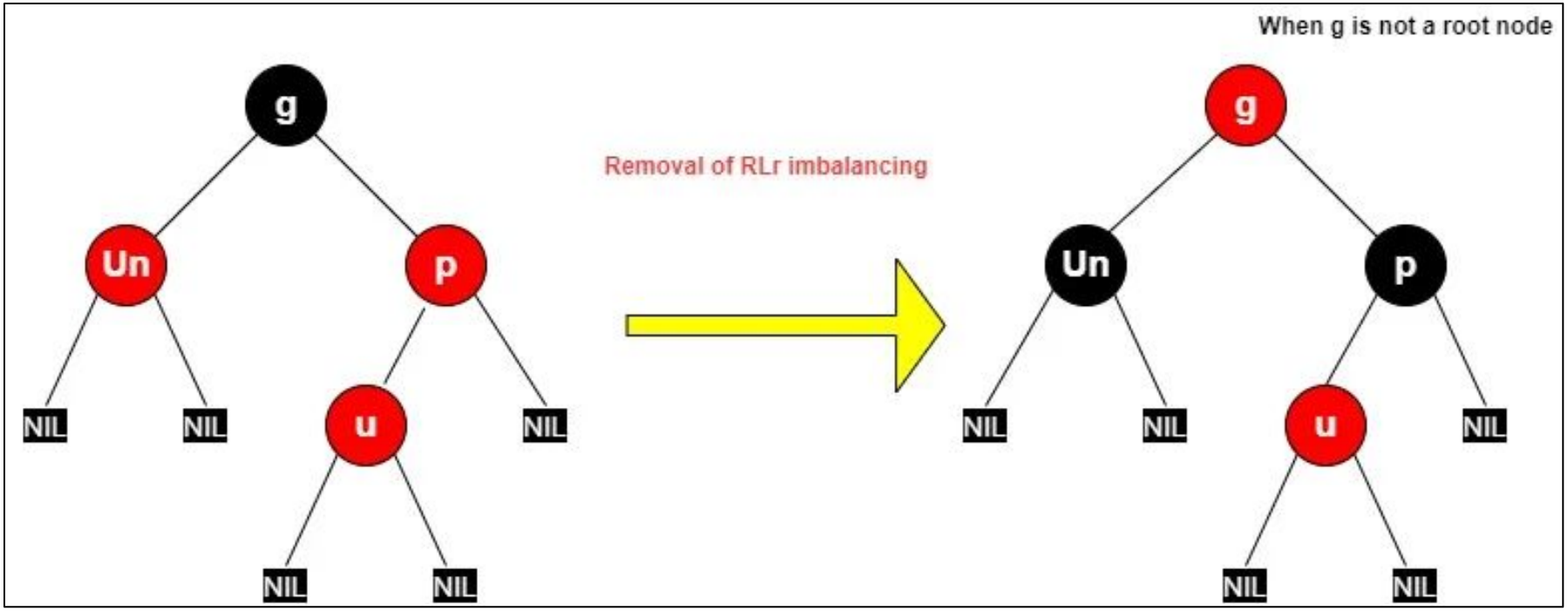
Note: If given **g** is root node then there will be no changes in color of **g**.



Insertion into RED BLACK Tree

4. Right Left imbalance:

- In this, Red-Black Tree violates its property in such a manner that parent and inserted child will be in red color at a position of right and left with respect to grandparent.
- Therefore, it is termed as RIGHT LEFT imbalance.



Insertion into RED BLACK Tree

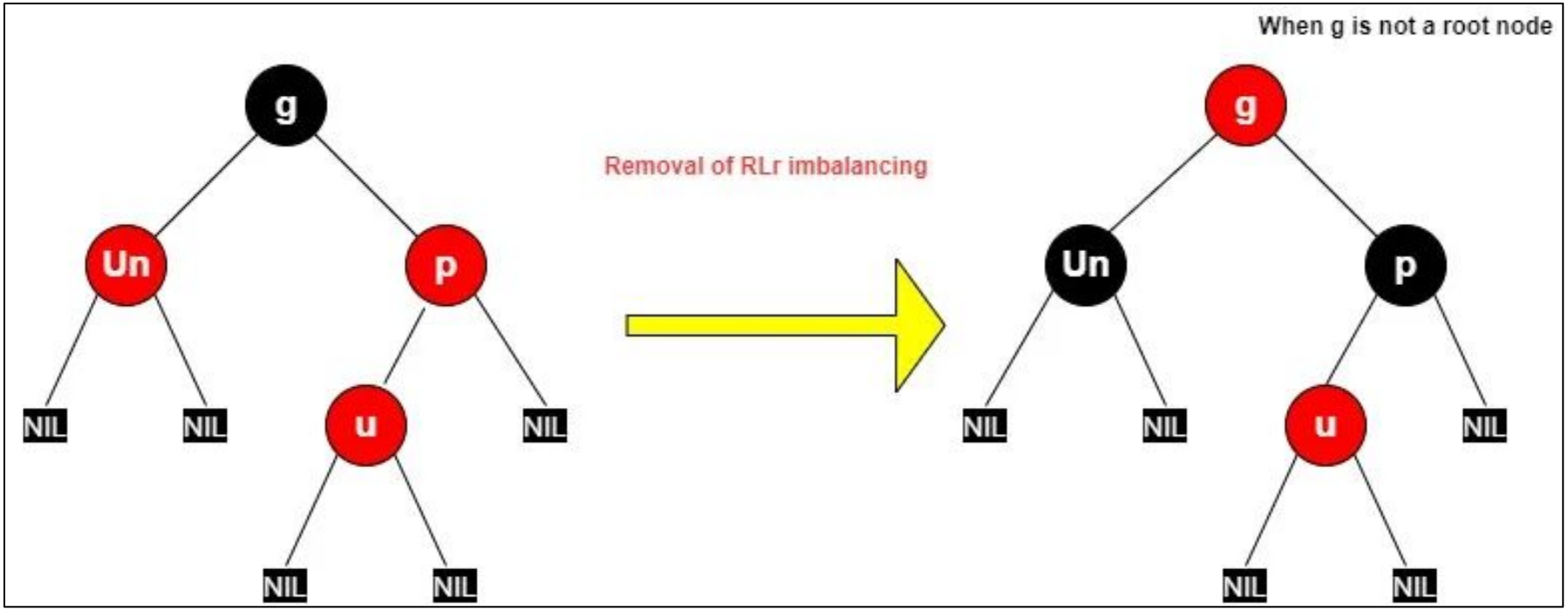
Removal of RRr imbalance can be done by:

i. Changing the color of node **p** from red to black.

ii. Changing the color of node **Un** from red to black.

iii. Changing the color of node **g** from black to red, if **g** is not a root node.

Note: If given **g** is root node then there will be no changes in color of **g**.



Insertion into RED BLACK Tree

Case - 2

- The imbalancing can also be occurred when the child of grandparent i.e., uncle node is black.
- Then also four cases will arise and in these cases imbalancing can be removed by using **rotation** technique.

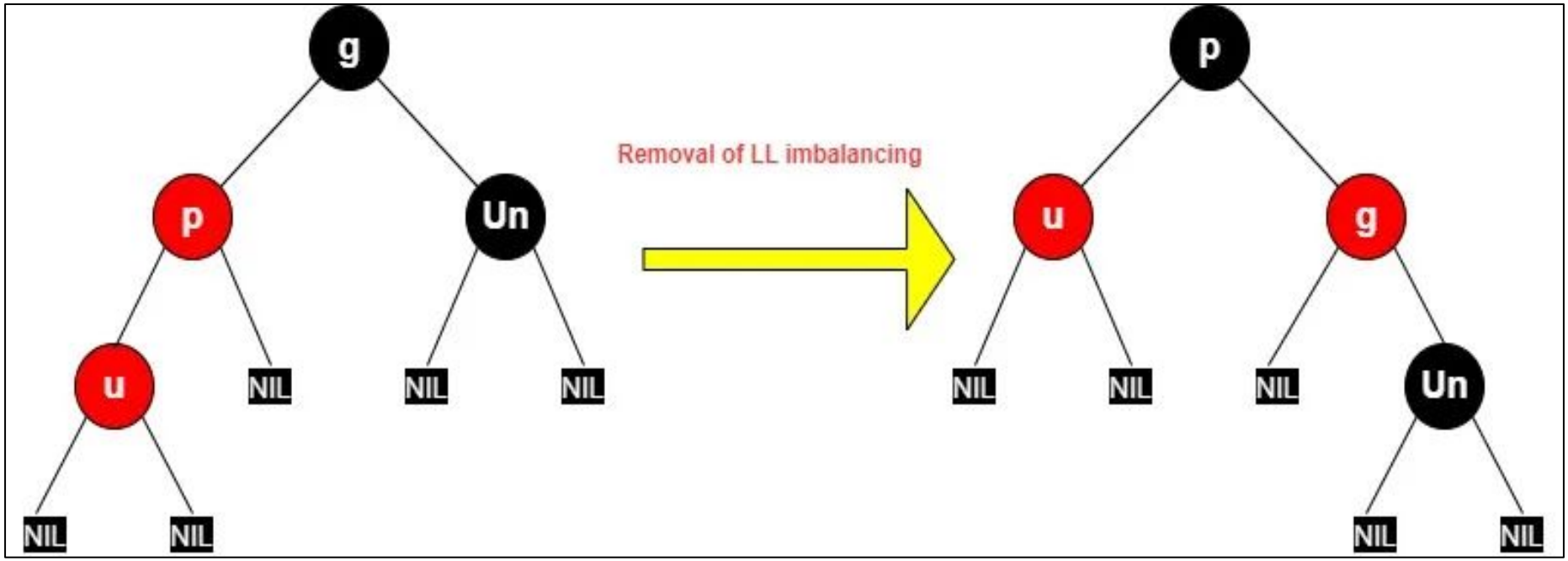
    1. **LR imbalancing**
    2. **LL imbalancing**
    3. **RR imbalancing**
    4. **RL imbalancing**

Insertion into RED BLACK Tree

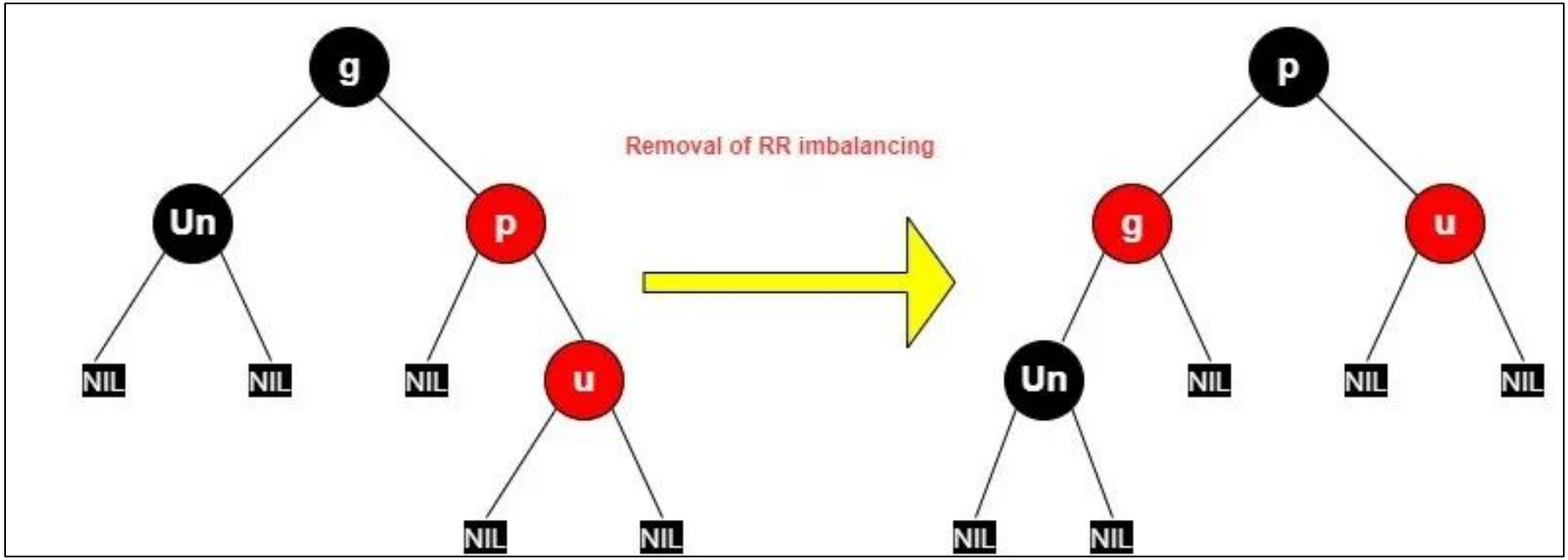
LL and RR imbalancing can be removed by following two steps —

i. Apply single rotation of **p** about **g**.

ii. Recolor **p** to black and **g** to red.



Insertion into RED BLACK Tree

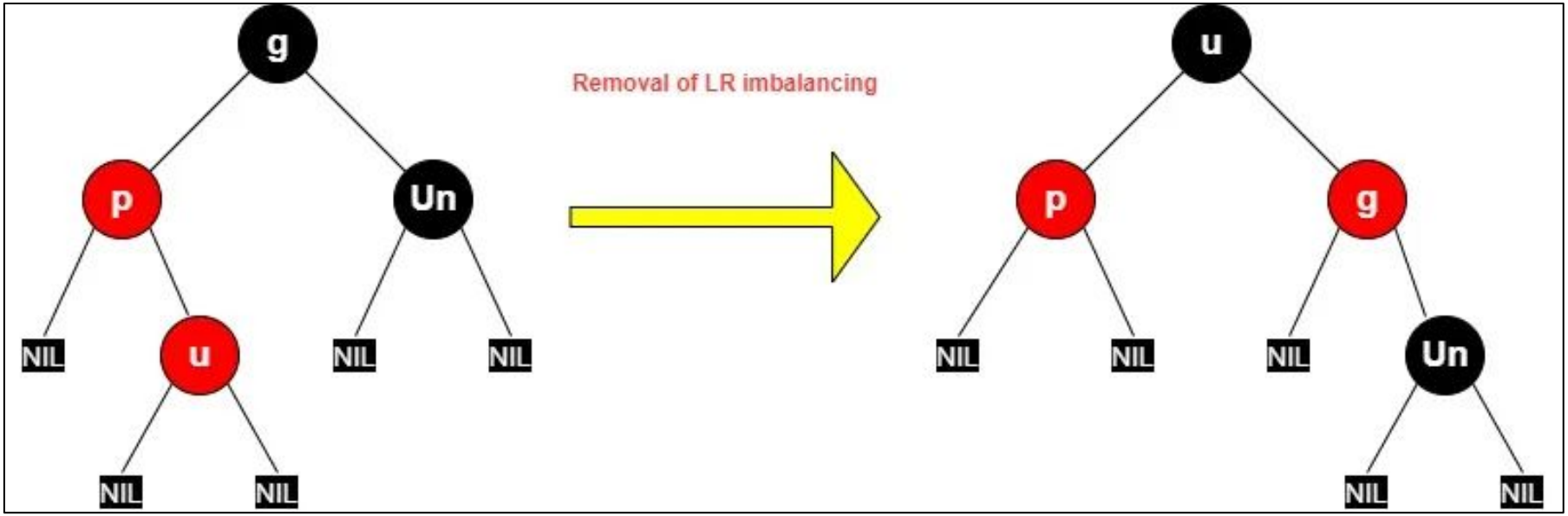


Insertion into RED BLACK Tree

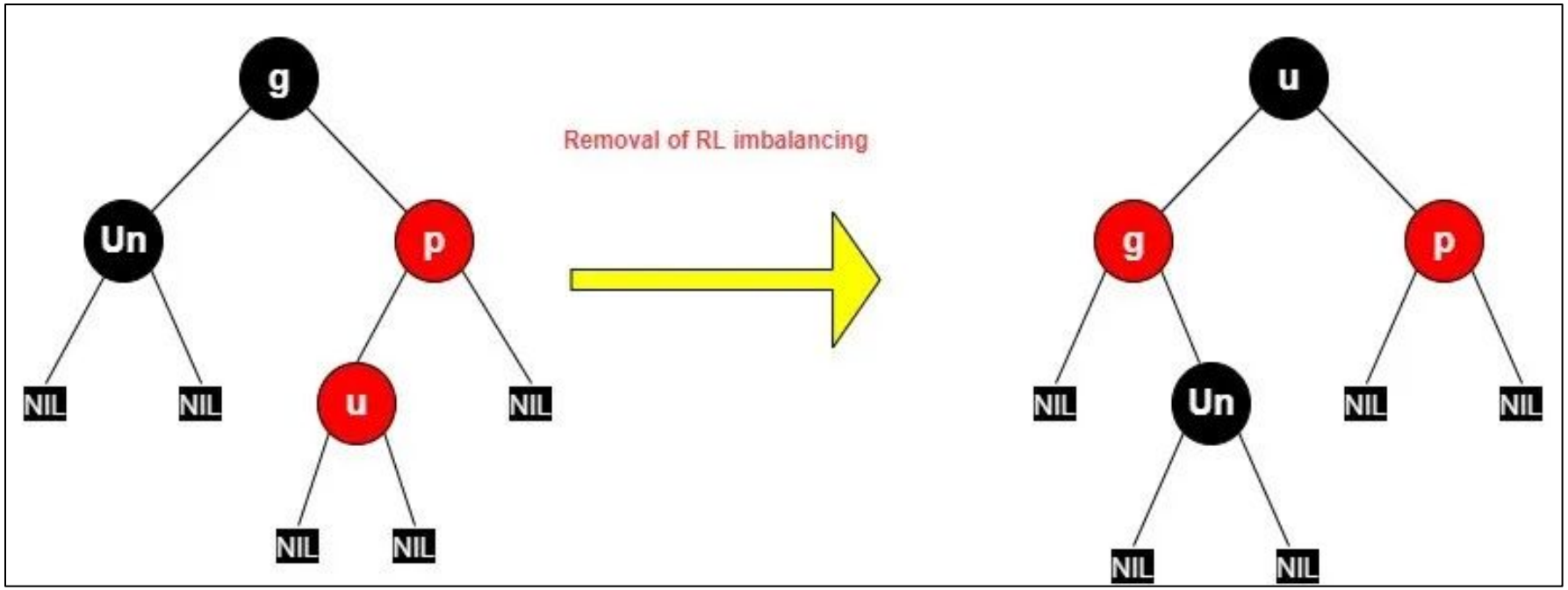
LR and RL imbalancing can be removed by following two steps —

i. Apply double rotation of **u** about **p** followed by **u** about **g**.

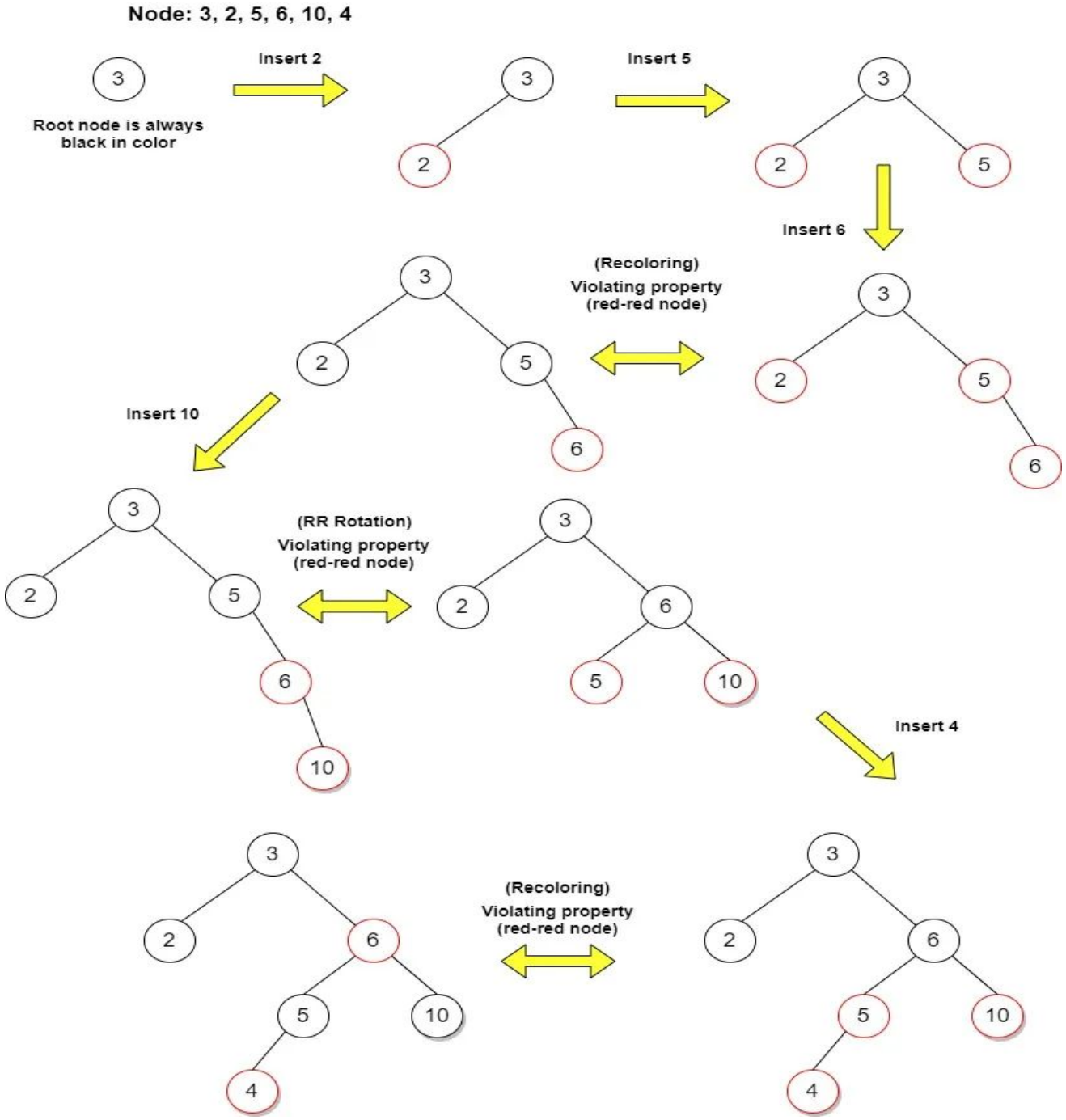
ii. For LR/RL imbalancing, recolor **u** to black and recolor **p** and **g** to red.



Insertion into RED BLACK Tree



Example

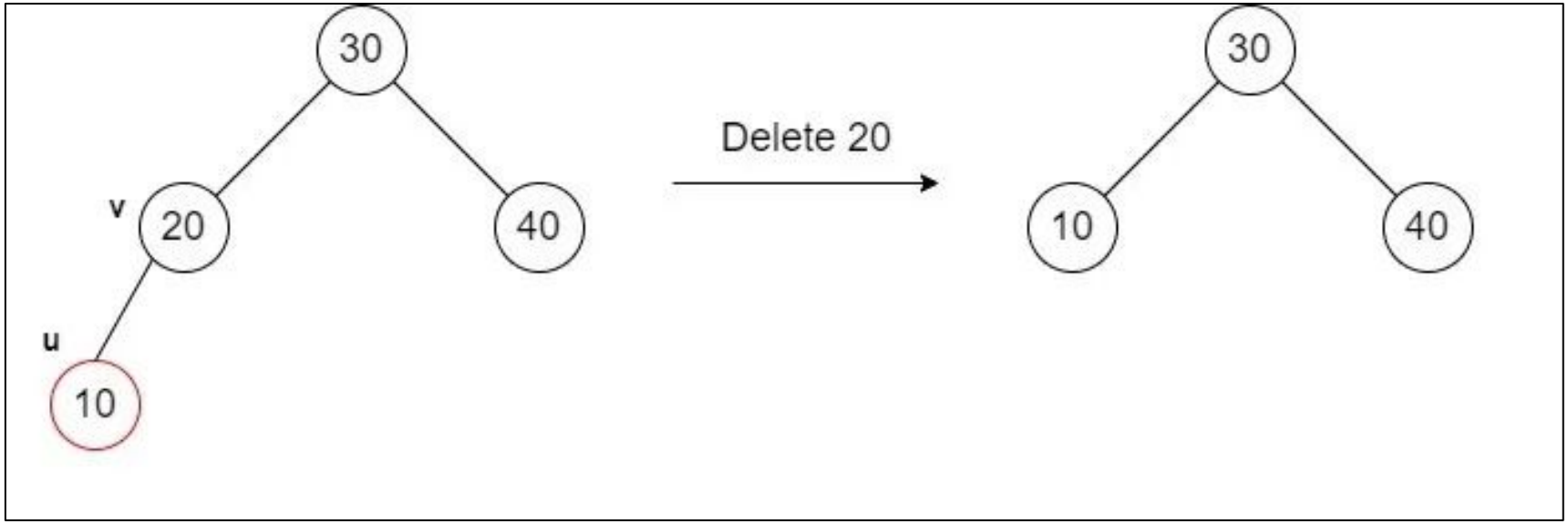

Node: 3, 2, 5, 6, 10, 4
3
Root node is always black in color
Insert 2
3
2
Insert 5
3
2
5
Insert 6
3
2
5
6
(Recoloring)
Violating property
(red-red node)
3
2
5
6
Insert 10
3
2
5
6
10
(RR Rotation)
Violating property
(red-red node)
3
2
6
5
10
Insert 4
3
2
6
5
10
4
(Recoloring)
Violating property
(red-red node)
3
2
6
5
10
4

Deletion Operation on RED BLACK Tree

- Deletion in a red-black tree is a bit more complicated than insertion.
- When a node is to be deleted, it can either have no children, one child or two children.
- Here are the steps involved in deleting a node in a red-black tree:
  1. If the **node to be deleted has no children**, simply remove it and update the parent node.
  2. If the **node to be deleted has only one child**, replace the node with its child.
  3. If the **node to be deleted has two children**, then replace the node with its in-order successor, which is the leftmost node in the right subtree. Then delete the in-order successor node as if it has at most one child.
  4. After the node is deleted, the red-black properties might be violated. To restore these properties, some color changes and rotations are performed on the nodes in the tree. The changes are similar to those performed during insertion, but with different conditions.
  5. The deletion operation in a red-black tree takes O(log n) time on average, making it a good choice for searching and deleting elements in large data sets.

Cases -

- Let **v** be the node to be deleted and **u** be the child that replaces **v** (Note that **u** is NULL when **v** is a leaf and color of NULL is considered as black)
  - **Simple Case:** If either **u** or **v** is red, we mark the replaced child as black (No change in black height). Both **u** and **v** cannot be red as **v** is parent of **u** and two consecutive reds are not allowed in red-black tree.

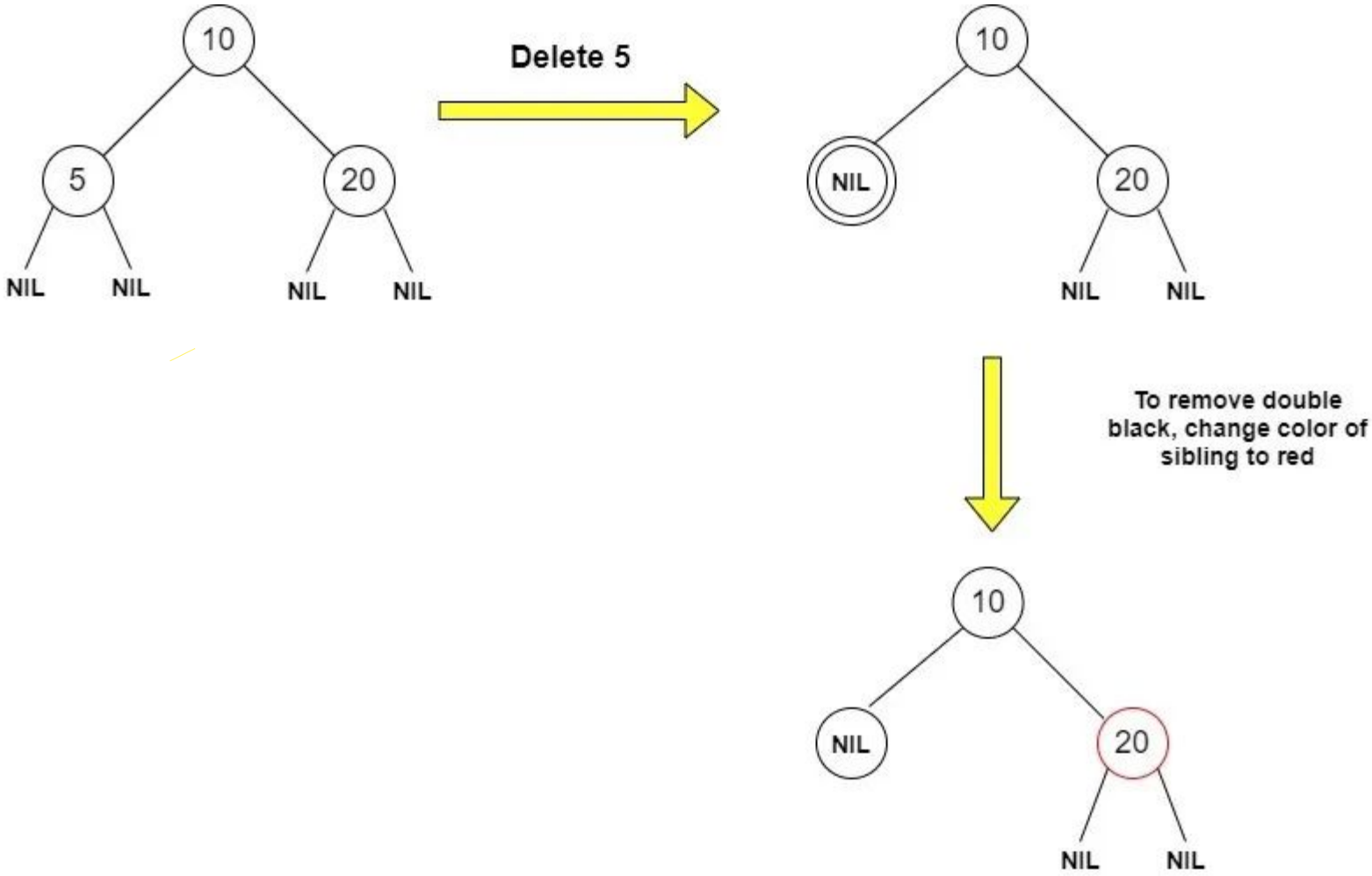


2. If both u and v are black -

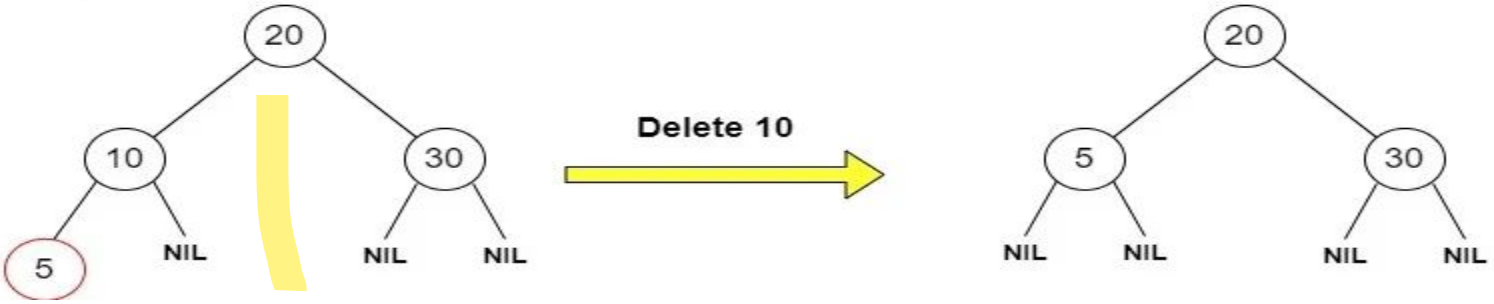
1. Color **u** as double black. Now our task reduces to convert this double black to single black. Note that If **v** is a leaf, then **u** is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.
2. Do following while the current node **u** is double black and it is not root. Let sibling of node be **s**. If sibling **s** is black and at least one of sibling's children is red (let it be **r**), perform rotations(s). (already discussed under Insertion)
    1. **Left Left Case:** s is left child of its parent and r is left child of s or both children of s are red.
    2. **Left Right Case:** s is left child of its parent and r is right child
    3. **Right Right Case:** s is right child of its parent and r is right child of s or both children of s are red.
    4. **Right Left Case:** s is right child of its parent and r is left child of s.
3. If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

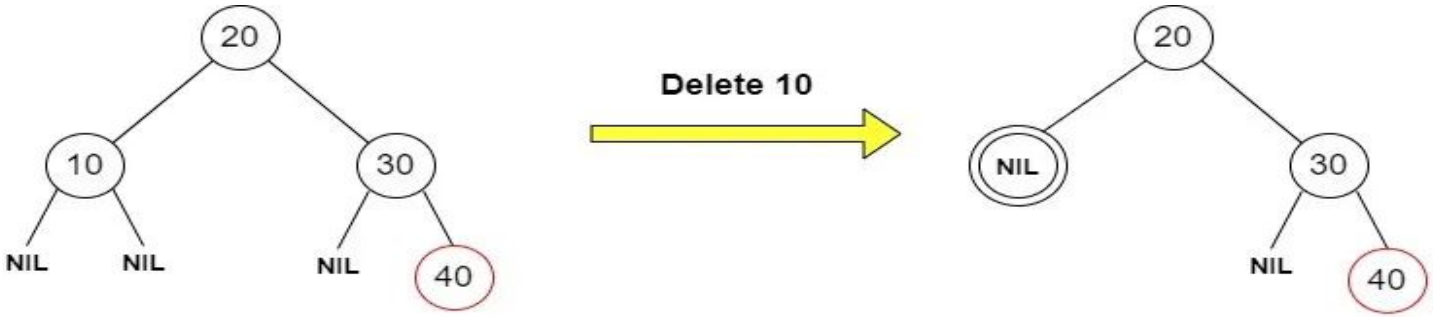
In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

Example

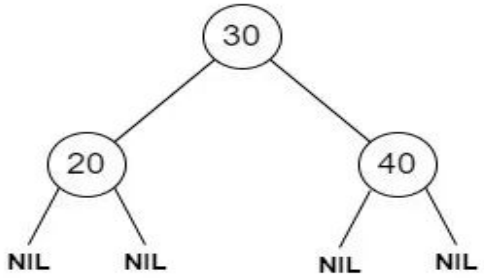

10
5
20
NIL
NIL
NIL
NIL
Delete 5
10
NIL
20
NIL
NIL
To remove double black, change color of sibling to red
10
NIL
20
NIL
NIL

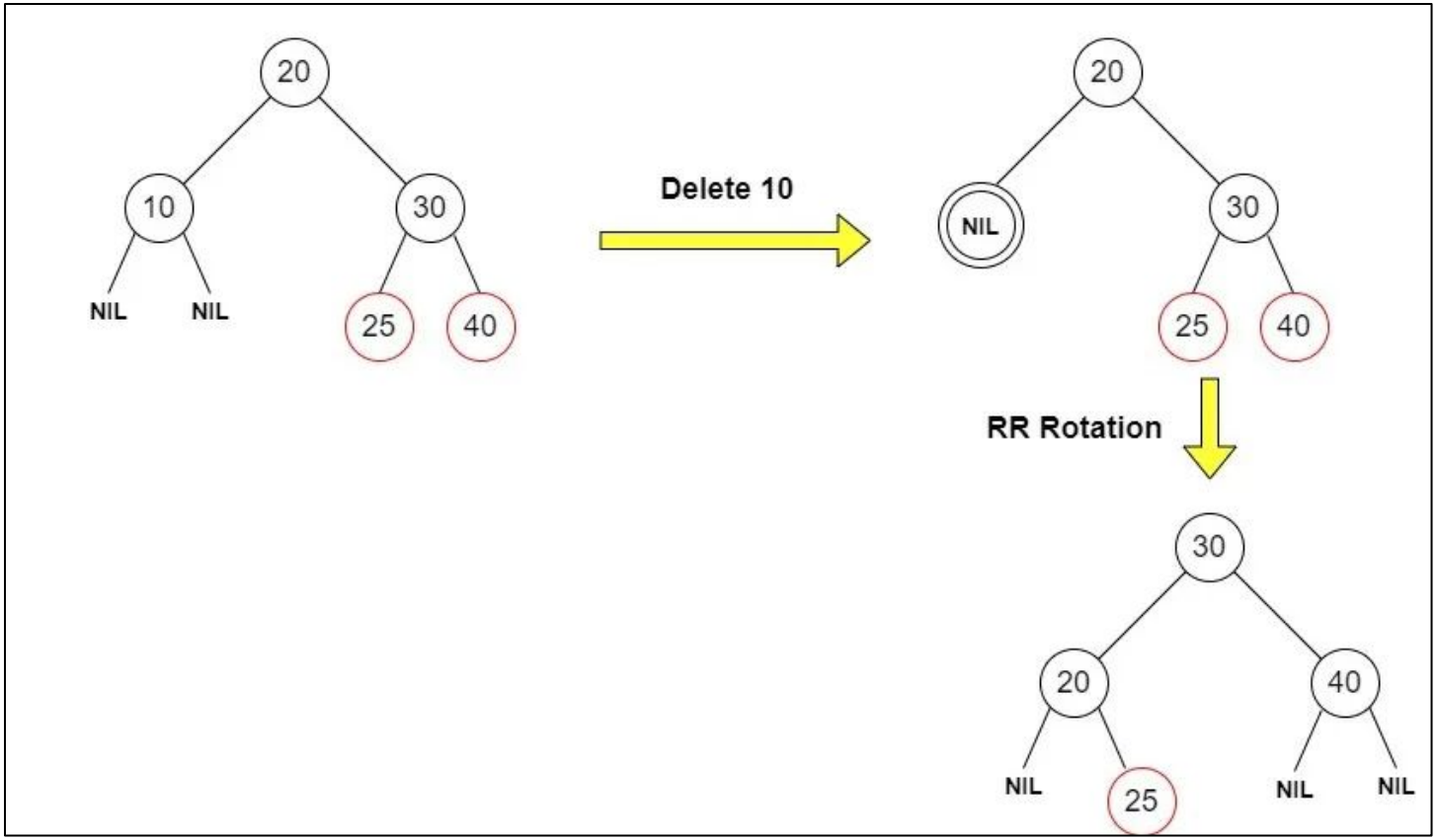
Simple Case

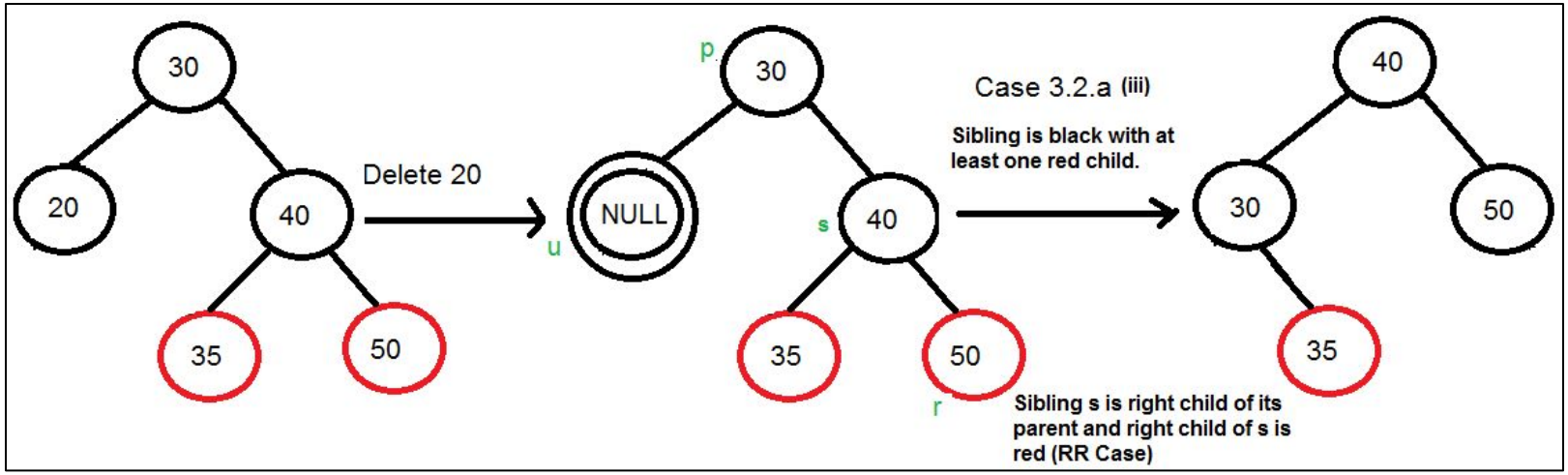


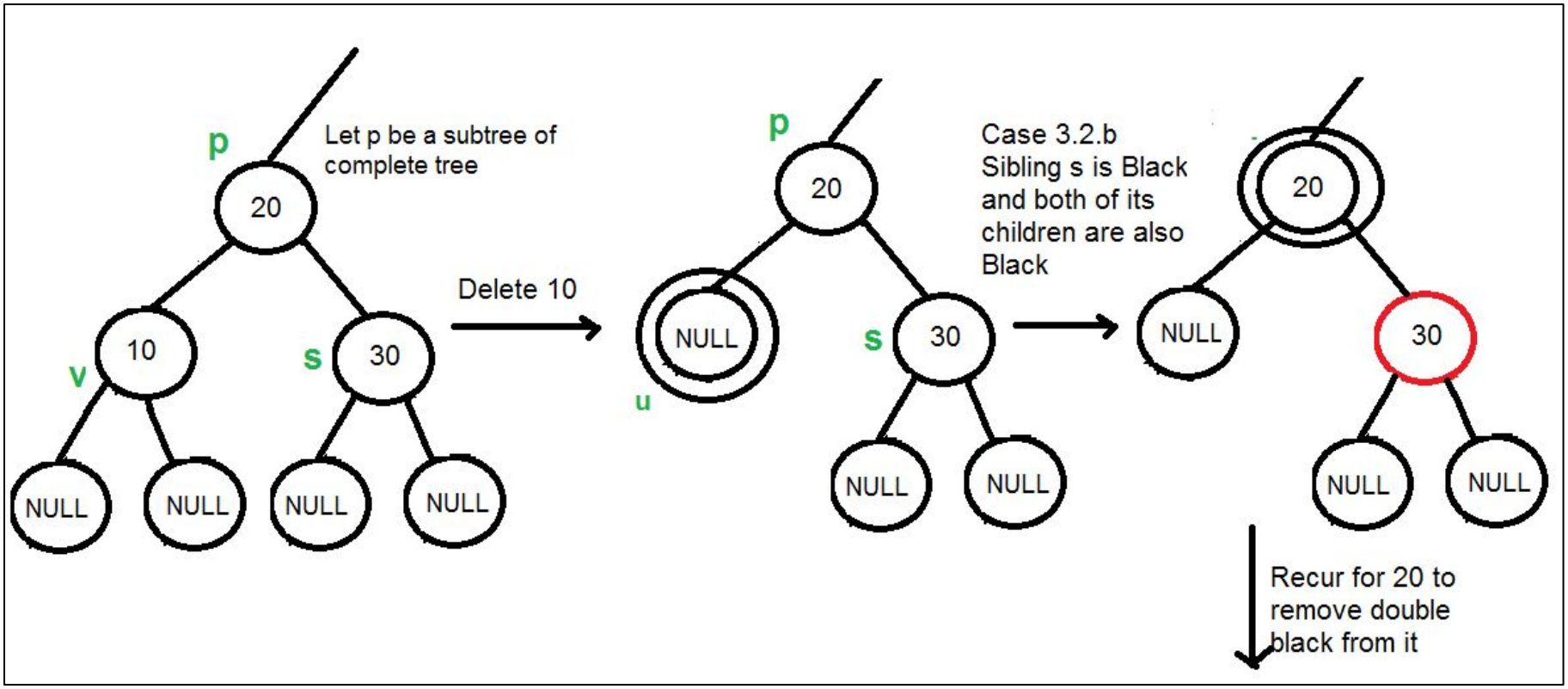


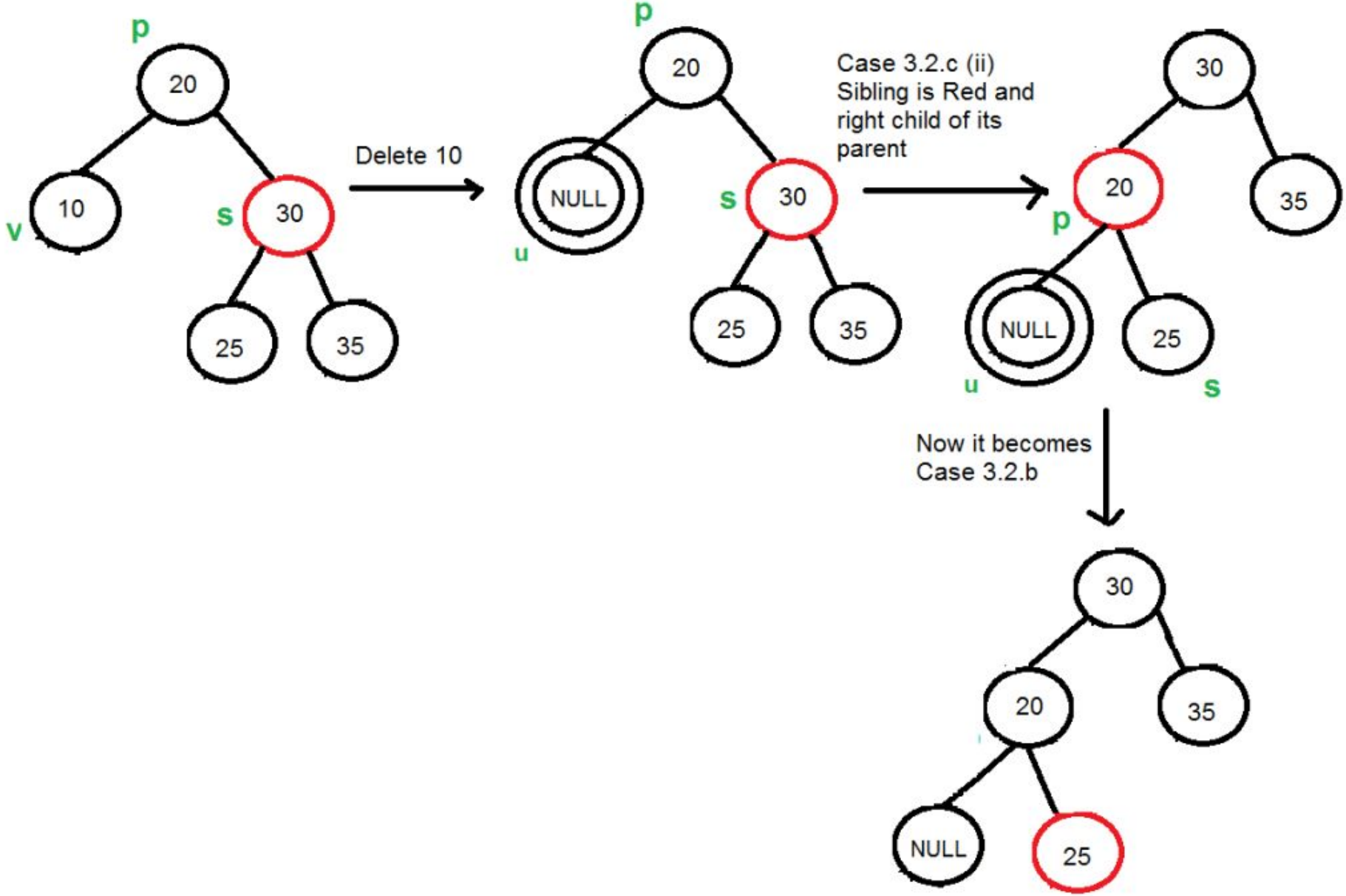
RR Rotation




20
10
30
NIL
NIL
25
40
Delete 10
20
NIL
30
25
40
RR Rotation
30
20
40
NIL
25
NIL
NIL


30
20
40
35
50
Delete 20
u
p
30
NULL
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)


p
20
Let p be a subtree of complete tree
v
10
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
u
NULL
s
30
NULL
NULL
Case 3.2.b
Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

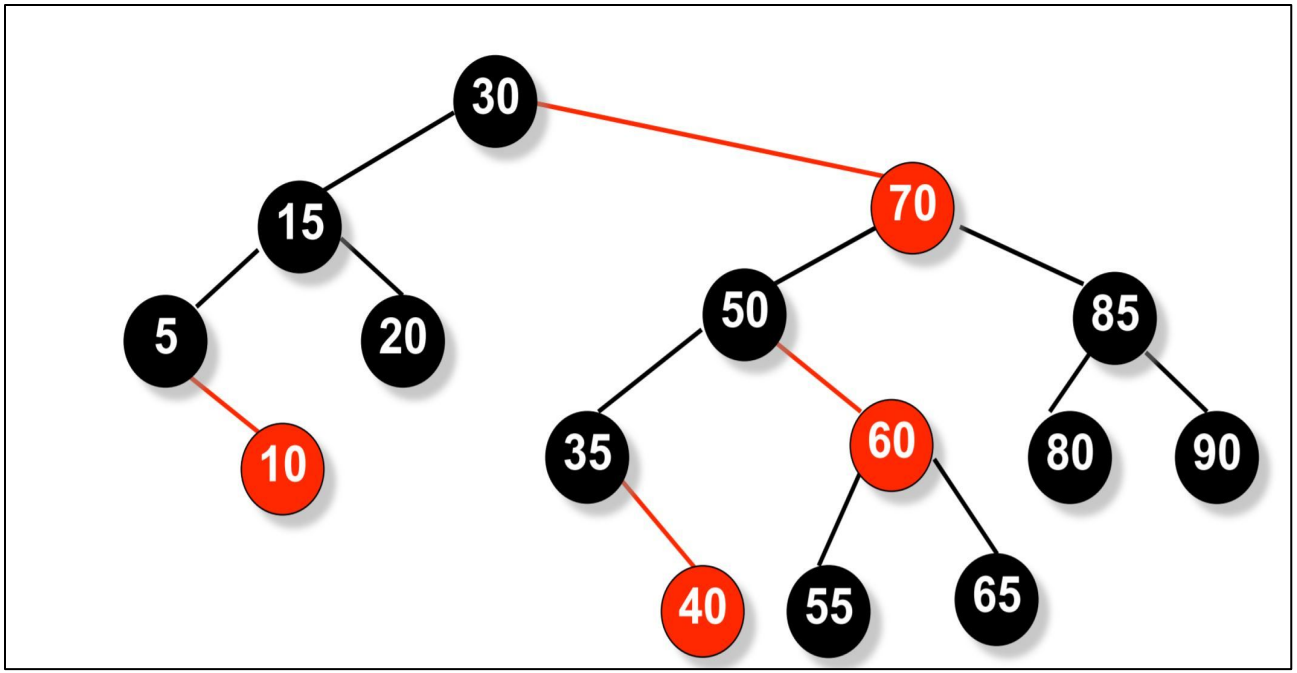

p
20
v
10
s
30
25
35
Delete 10
p
20
u
NULL
s
30
25
35
Case 3.2.c (ii)
Sibling is Red and
right child of its
parent
30
p
20
35
u
NULL
25
s
Now it becomes
Case 3.2.b
30
20
35
NULL
25

AA tree

- A BST of n nodes is said to be balanced if the height is O(logn).
- A balanced tree supports efficient operations since most operations only have to traverse or on two root-to-leaf paths.
- There are many implementations of balanced BSTs, including AVL trees, red–black trees, and AA trees.
- An AA tree is another alternative to AVL trees.
- An AA tree is a balanced BST with the following properties:
  1. Every node is coloured either red or black.
  2. The root is black.
  3. If a node is red, both of its children are black.
  4. Every path from a node to a null reference has the same number of black nodes.
  5. **Left children may not be red.**

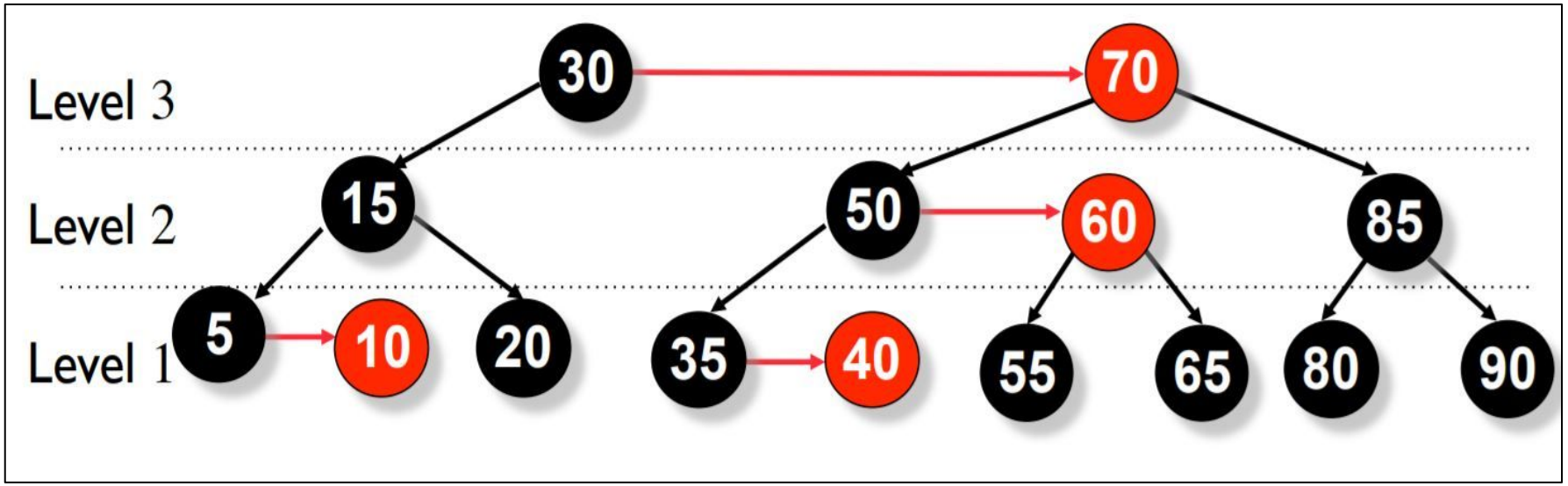
AA tree

Unlike in red-black trees, red nodes on an AA tree can only be added as a right sub-child i.e. no red node can be a left sub-child. The tree below is an AA tree.



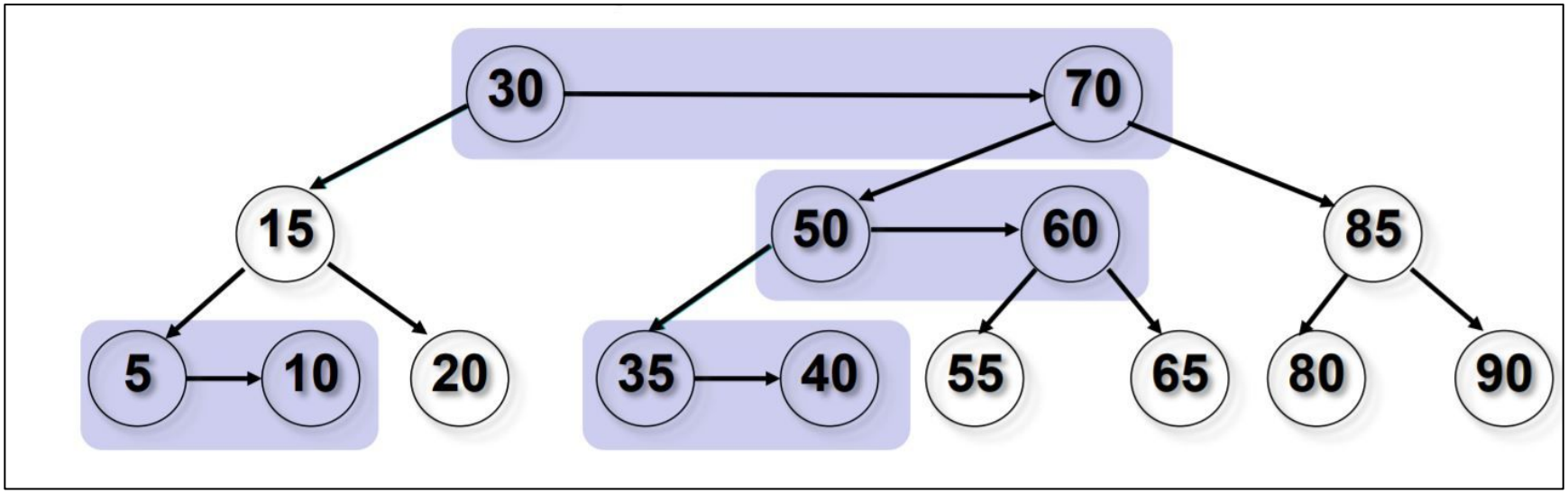
Representing Balance Information in AA Tree

1. The level of every **leaf node** is one.
2. If a node is **red**, its level is the level of its parent.
3. If a node is **black**, its level is one less than the level of its parent.
4. The level of every **left child** is exactly one less than that of its parent.
5. The level of every **right child** is equal to or one less than that of its parent.
6. The level of every **right grand child** is strictly less than that of its grandparent.
7. Every node of level greater than one, has two children.



Horizontal Link

- A link where the child's level is equal to that of its parent is called a horizontal link.
- Horizontal links are always right links.
- Red nodes are simply nodes that are located at the same level as their parents.
- For the AA tree shown above, the image below should help you understand which nodes are red and which are black and which are the horizontal links.

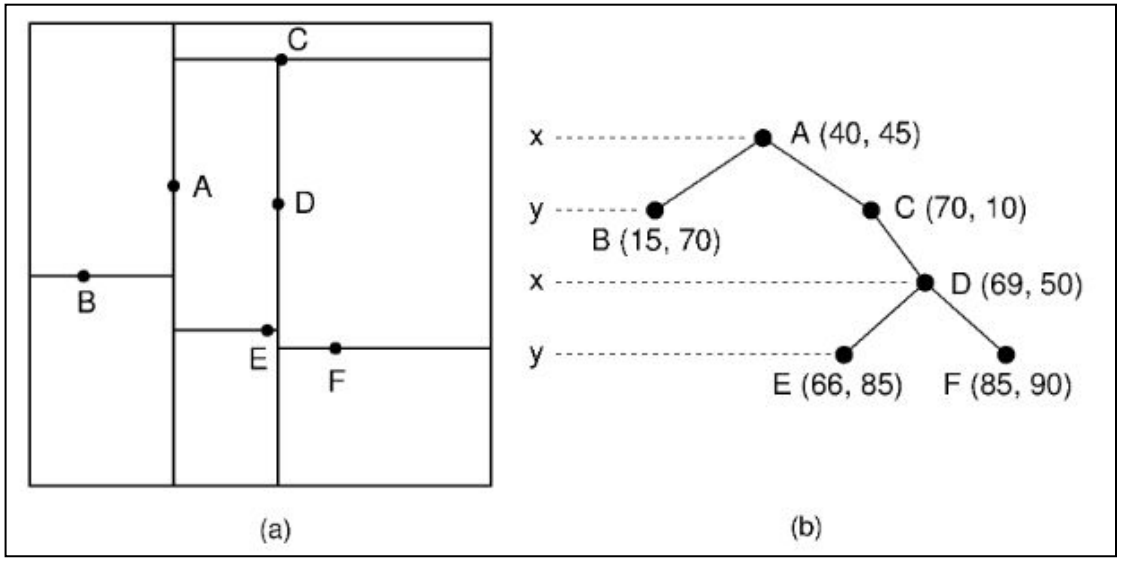


Advantages of AA Trees -

- AA trees are more advantageous as they simplify the algorithms. The following list explains the advantages:
- They eliminate half the reconstructing cases.
- They simplify deletion by removing an annoying case.
    - If an internal node has only one child, that child must be a red child.
    - We can always replace a node with the smallest child in the right subtree; it will either be a leaf node or have a red child.
- An AA tree, which is a balanced BST, supports efficient operations, since most operations only have to traverse one or two root-to-leaf paths.

K-dimensional tree

- A K-Dimensional Tree (also known as K-D Tree) is a space-partitioning data structure for organizing points in a K-Dimensional space.
- This data structure acts similar to a binary search tree with each node representing data in the multi dimensional space.
- The kd tree is a modification to the BST that allows for efficient processing of multi-dimensional search keys.
- The purpose of the tree was to store spatial data with the goal of accomplishing:
    - Nearest neighbor search.
    - Range queries.
    - Fast look-up.

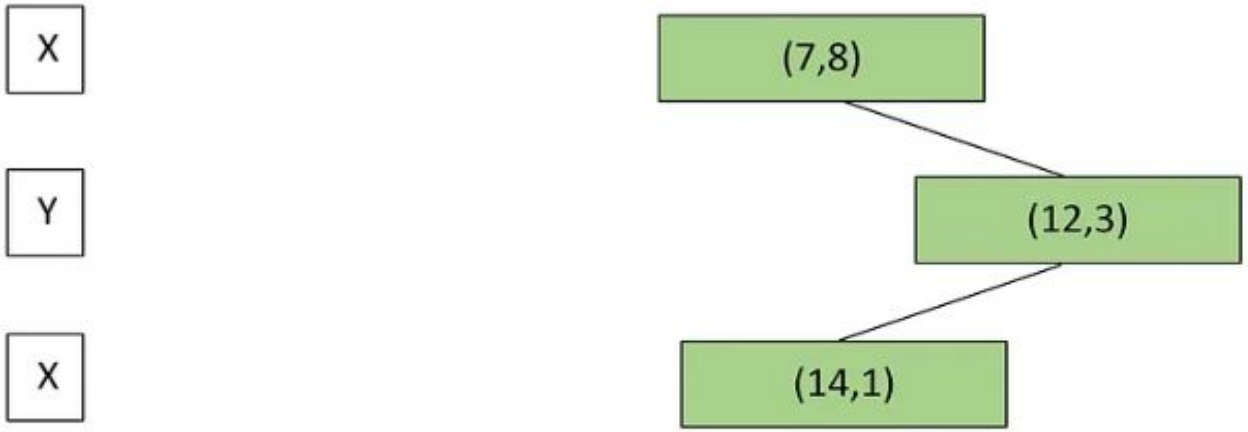


How it works

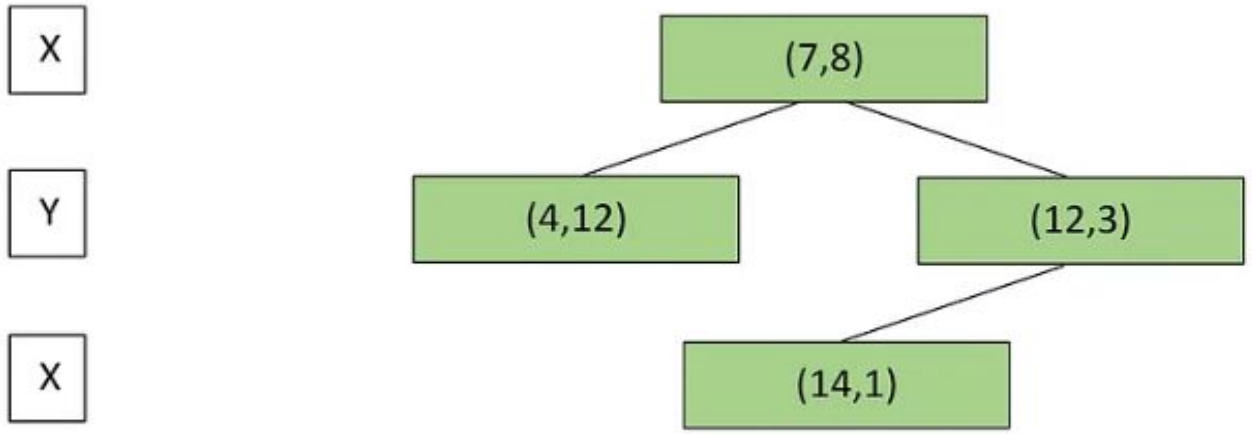
- A simple example to showcase the insertion into a K-Dimensional Tree, we will use a k = 2.
- The points we will be adding are: (7,8), (12,3), (14,1), (4,12), (9,1), (2,7), and (10,19).

The first element inserted is (7,8). It will serve as the base node in the following K-D Tree example.

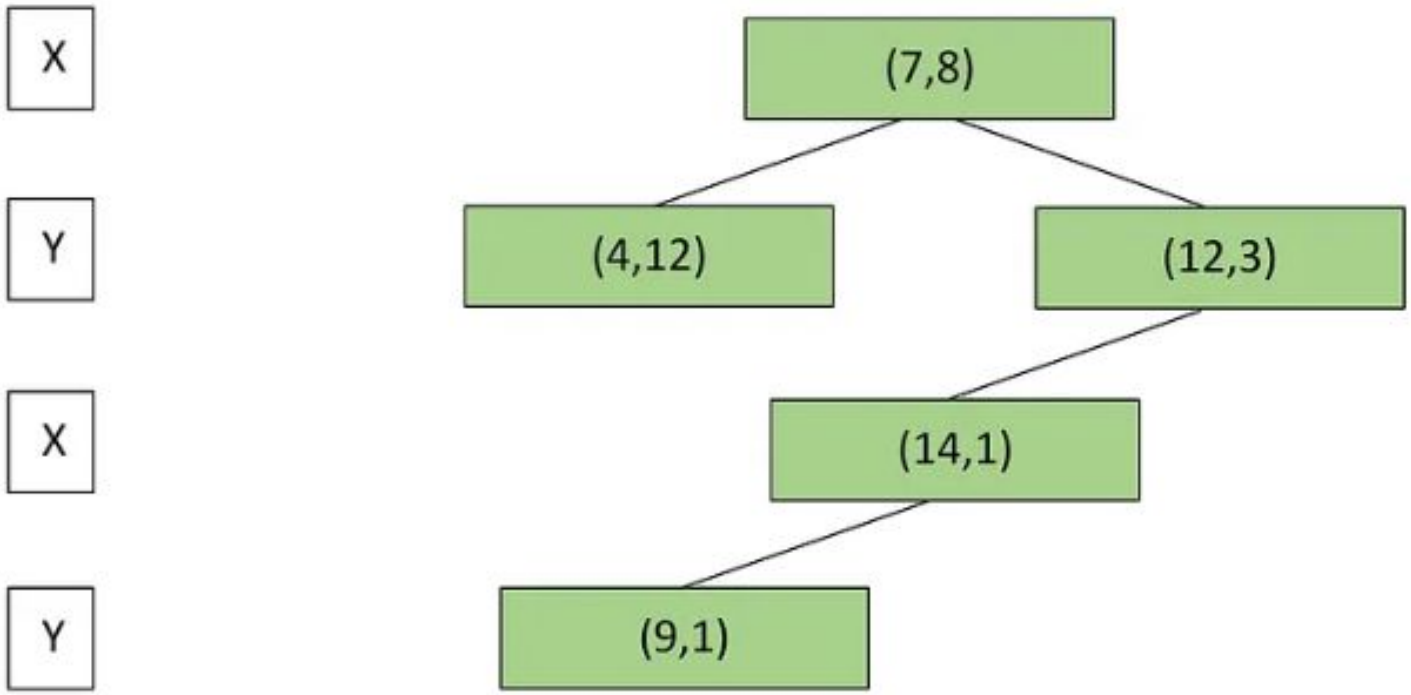
The second element inserted is (12,3). It is placed to the right leaf node of (7,8) because the X value, 12, is greater than the X value of base, 7.



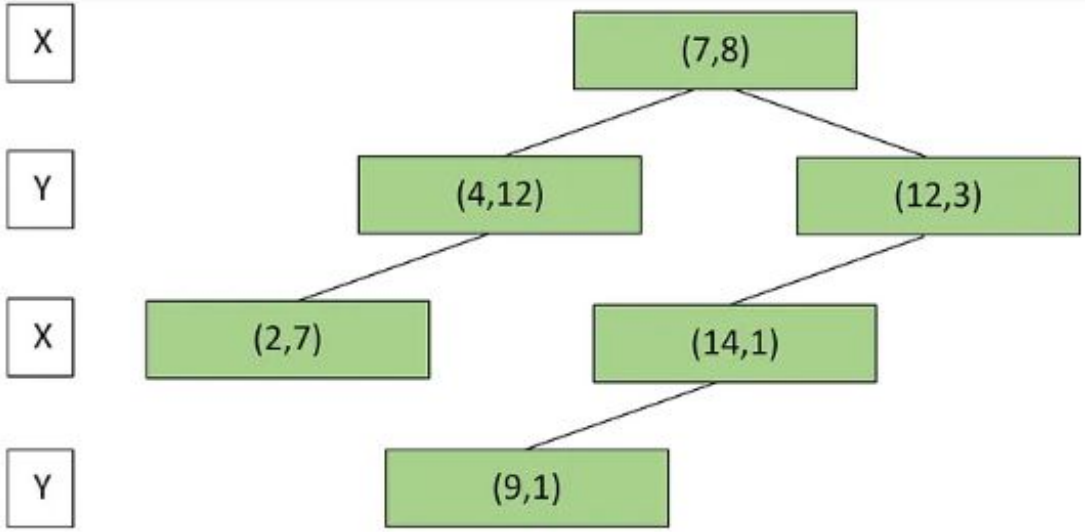
The next element inserted into the K-D Tree is (14,1). At the first level, we compare the X values of the set and since 14 is greater than 7 it moves to the right of the tree. Next is the comparison between (12,3) and (14,1). Each level the comparison operator changes, which means that we will be comparing the Y value of each set. Since the Y of the inserted set is 1 which is less than 3, it is inserted to the left leaf node of (12,3).



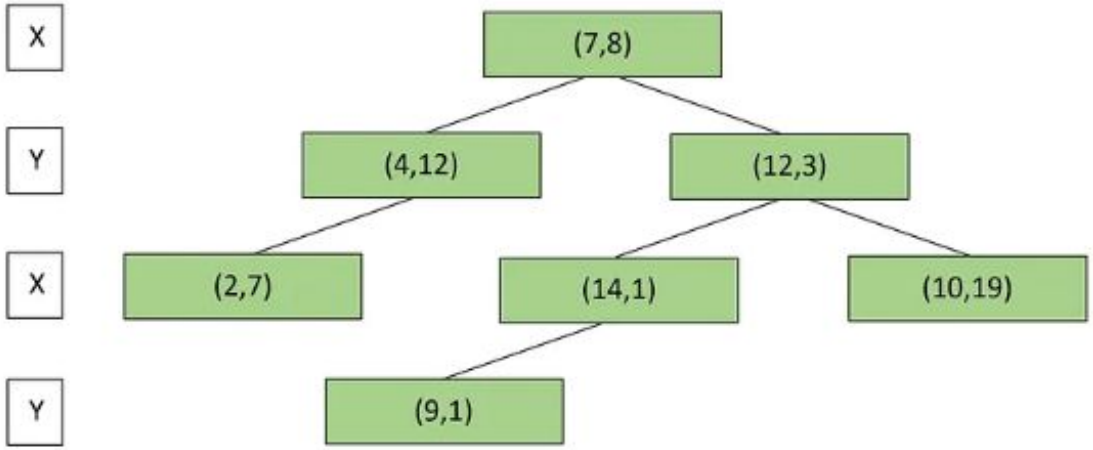
The next set inserted is (4,12). Because the X, 4, is less than the base nodes X, which is 7, the set is inserted to the left leaf node.



The next set is (9,1). Since the X is greater than the base, it moves to the right leaf. At level 2 we compare the Y values, which is less than the current leaf, so we move to the left leaf of this node. On level 3 we restart and compare by the X value again, but if we had multiple dimensions it would moved to the next dimension. Since 9 is less than 14, we insert it to the left of this leaf node.



The next set inserted is (2,7). Since the X is less than 7 it compares to the left leaf node. As the Y value is less than 12, the second levels Y value, this set is inserted to the left of the (4,12) leaf node.

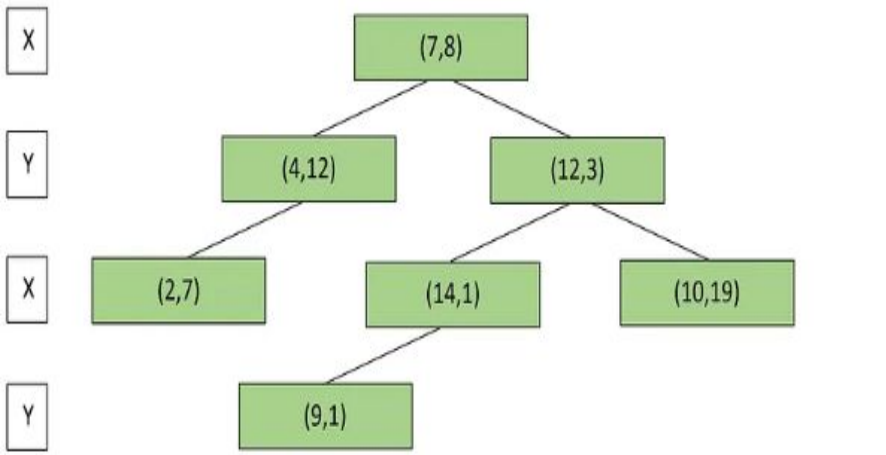


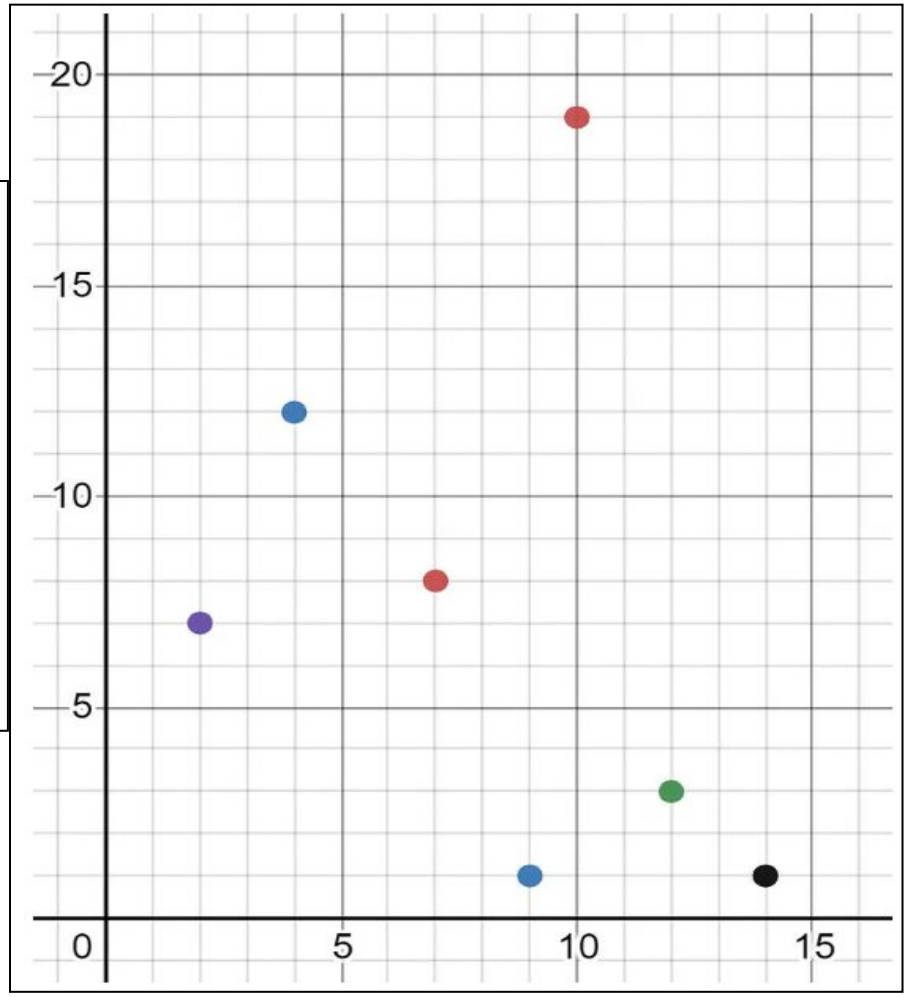
The next set inserted is (10,19). 10 is greater than 7 so it moves to the right leaf node. Additionally, 19 is greater than the second levels Y, so it it inserted to the right leaf node.

- We can see that the depth is equivalent to 3, which is also Log2(n) of the current K-D Tree.

- On a plotted 2 dimensional graph the following K-D Tree will be the equivalent to:

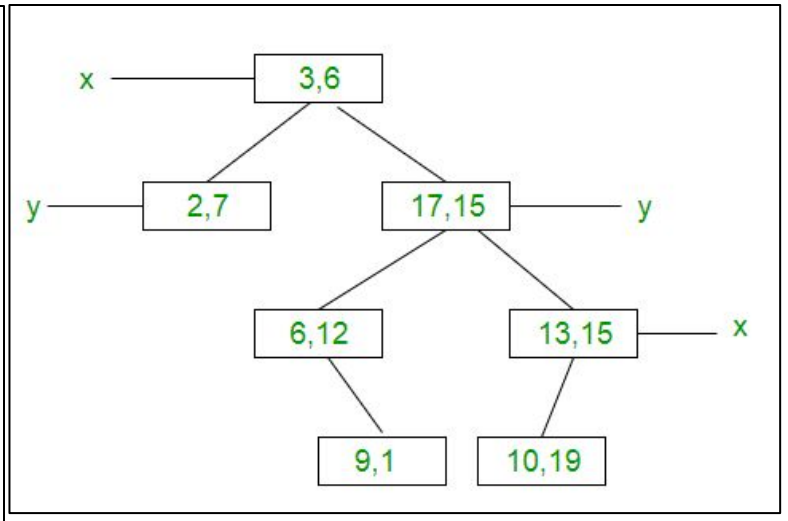
- The points we will be adding are: (7,8), (12,3), (14,1), (4,12), (9,1), (2,7), and (10,19).





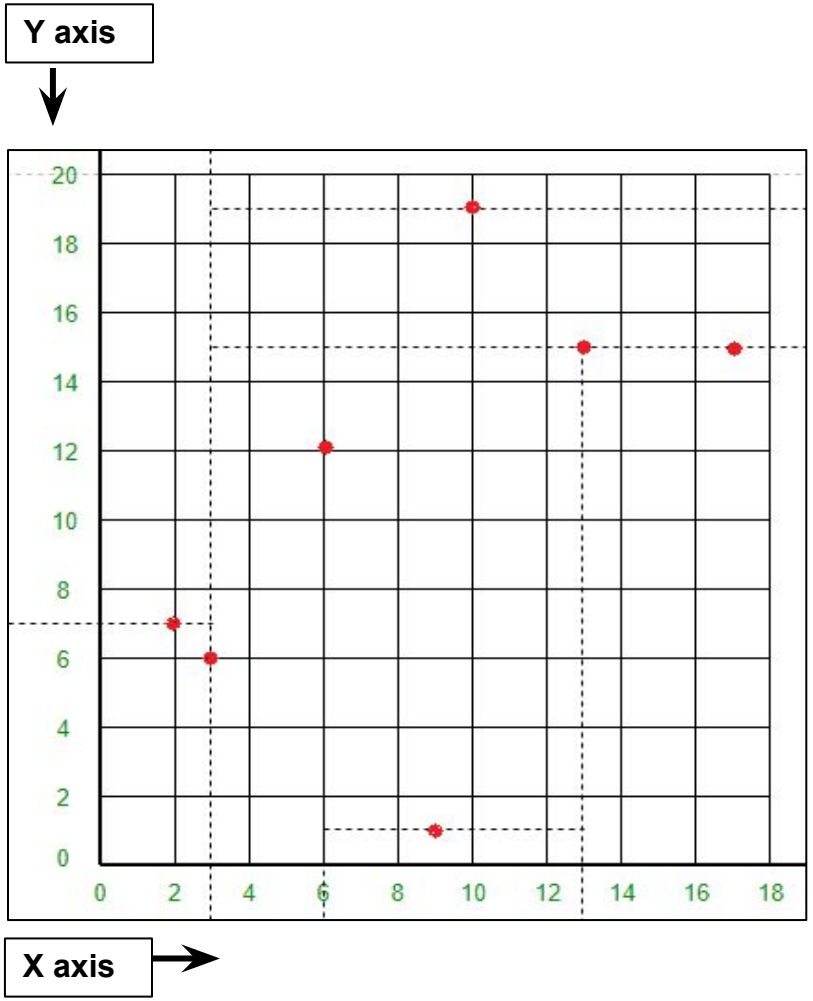
- Consider following points in a 2-D plane: (3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

1. **Insert (3, 6):** Since tree is empty, make it the root node.
2. **Insert (17, 15):** Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the right subtree or in the left subtree. This point will be Y-aligned.
3. **Insert (13, 15):** X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. **Insert (6, 12):** X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 < 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. **Insert (9, 1):**Similarly, this point will lie in the right of (6, 12).
6. **Insert (2, 7):**Similarly, this point will lie in the left of (3, 6).
7. **Insert (10, 19):** Similarly, this point will lie in the left of (13, 15).



How is space partitioned?

1. Point (3, 6) will divide the space into two parts: Draw line X = 3.
2. Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally. Draw line Y = 7 to the left of line X = 3.
3. Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally. Draw line Y = 15 to the right of line X = 3.
4. Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts. Draw line X = 6 to the right of line X = 3 and below line Y = 15.
5. Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts. Draw line X = 13 to the right of line X = 6 and below line Y = 15.
6. Point (9, 1) will divide the space between lines X = 3, X = 6 and Y = 15 into two parts. Draw line Y = 1 between lines X = 3 and X = 13.
7. Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts. Draw line Y = 19 to the right of line X = 3 and above line Y = 15.



Advantages of K Dimension Trees

- **Efficient search:** K-D trees are effective in searching for points in k-dimensional space, such as in nearest neighbor search or range search.
- **Dimensionality reduction:** K-D trees can be used to reduce the dimensionality of the problem, allowing for faster search times and reducing the memory requirements of the data structure.
- **Versatility:** K-D trees can be used for a wide range of applications, such as in data mining, computer graphics, and scientific computing.
- **Balance:** K-d trees are self-balancing, which ensures that the tree remains efficient even when data is inserted or removed.
- **Incremental construction:** K-D trees can be incrementally constructed, which means that data can be added or removed from the structure without having to rebuild the entire tree.
- **Easy to implement:** K-D trees are relatively easy to implement and can be implemented in a variety of programming languages.

SPLAY TREES

- Splay trees are a form of BSTs.
- A splay tree maintains a balance without any explicit balance condition such as color.
- Splay tree is a self-adjusting binary search tree data structure, which means that the tree structure is adjusted dynamically based on the accessed or inserted elements.
- Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.
- In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.
- Splaying is a process of restructuring the tree by making the most recently accessed or inserted element the new root and gradually moving the remaining nodes closer to the root.
- Splay trees are highly efficient in practice due to their self-adjusting nature, which reduces the overall access time for frequently accessed elements.
- However, the main disadvantage of splay trees is that they do not guarantee a balanced tree structure, which may lead to performance degradation in worst-case scenarios.

Operations in a splay tree:

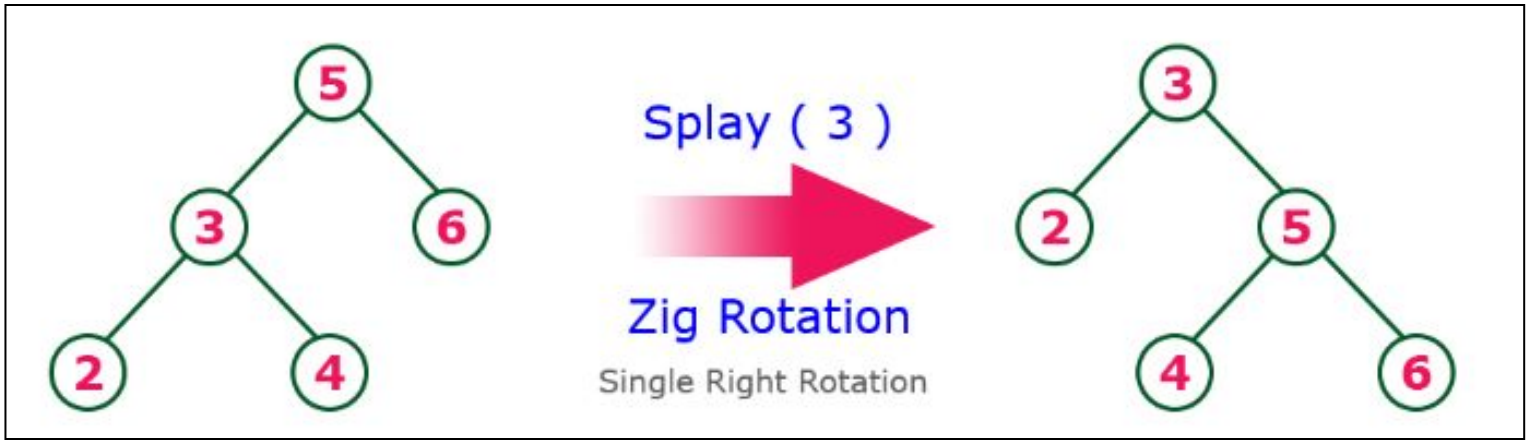
Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

1. **Insertion:** To insert a new element into the tree, start by performing a regular binary search tree insertion. Then, apply rotations to bring the newly inserted element to the root of the tree.
2. **Deletion:** To delete an element from the tree, first locate it using a binary search tree search. Then, if the element has no children, simply remove it. If it has one child, promote that child to its position in the tree. If it has two children, find the successor of the element (the smallest element in its right subtree), swap its key with the element to be deleted, and delete the successor instead.
3. **Search:** To search for an element in the tree, start by performing a binary search tree search. If the element is found, apply rotations to bring it to the root of the tree. If it is not found, apply rotations to the last node visited in the search, which becomes the new root.
4. **Rotation:** The rotations used in a splay tree are either a Zig or a Zig-Zig rotation. A Zig rotation is used to bring a node to the root, while a Zig-Zig rotation is used to balance the tree after multiple

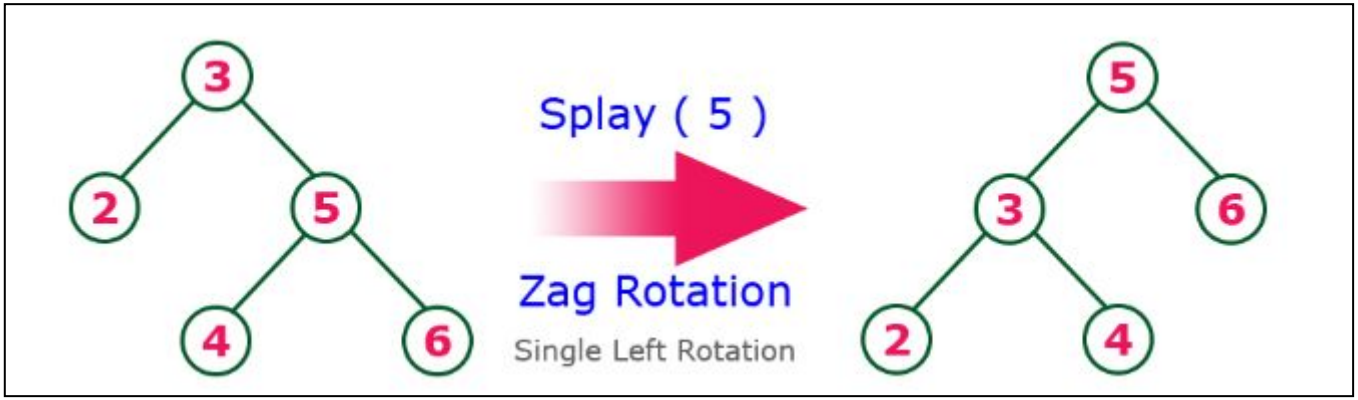
Rotations in Splay Tree

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

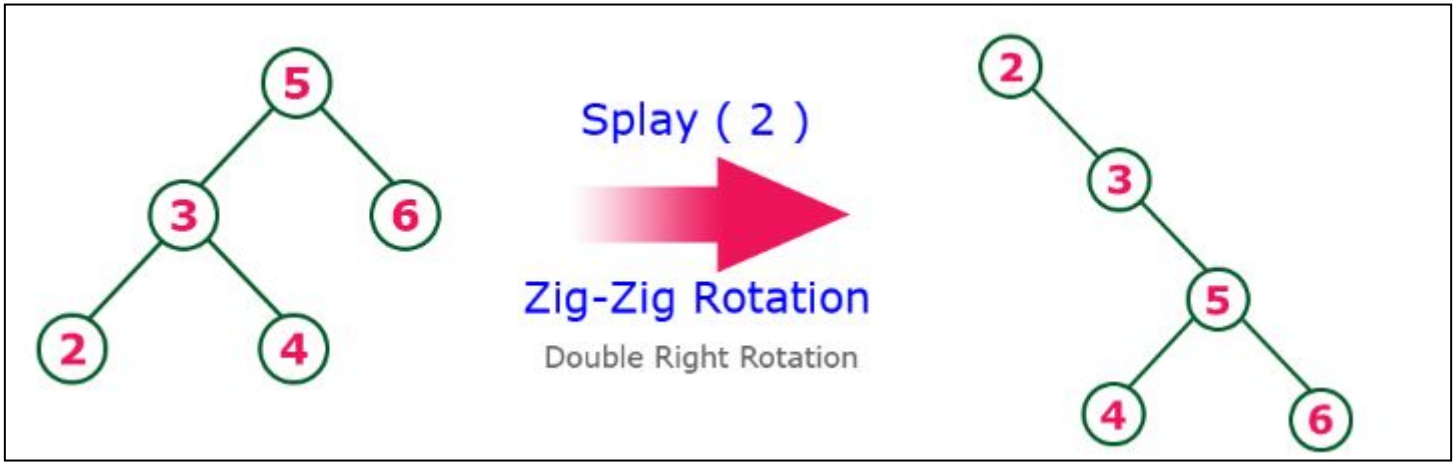
Zig Rotation - The Zig Rotation in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



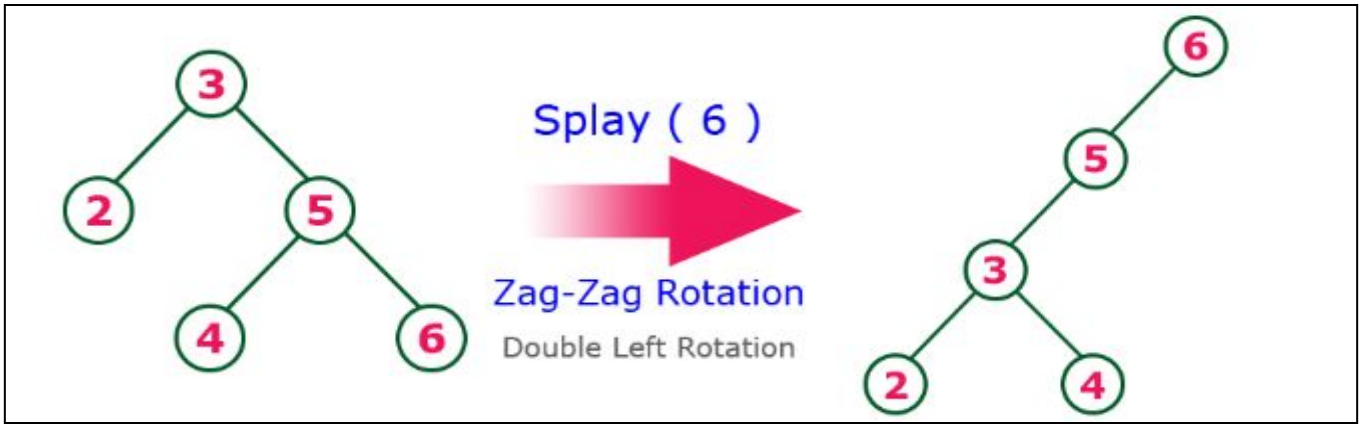
Zag Rotation - The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



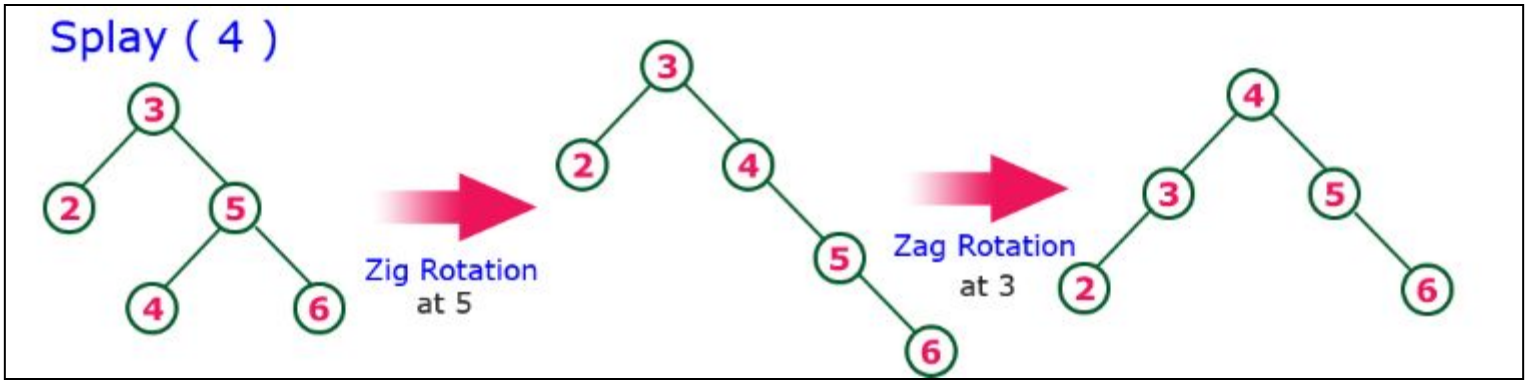
Zig-Zig Rotation -The Zig-Zig Rotation in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



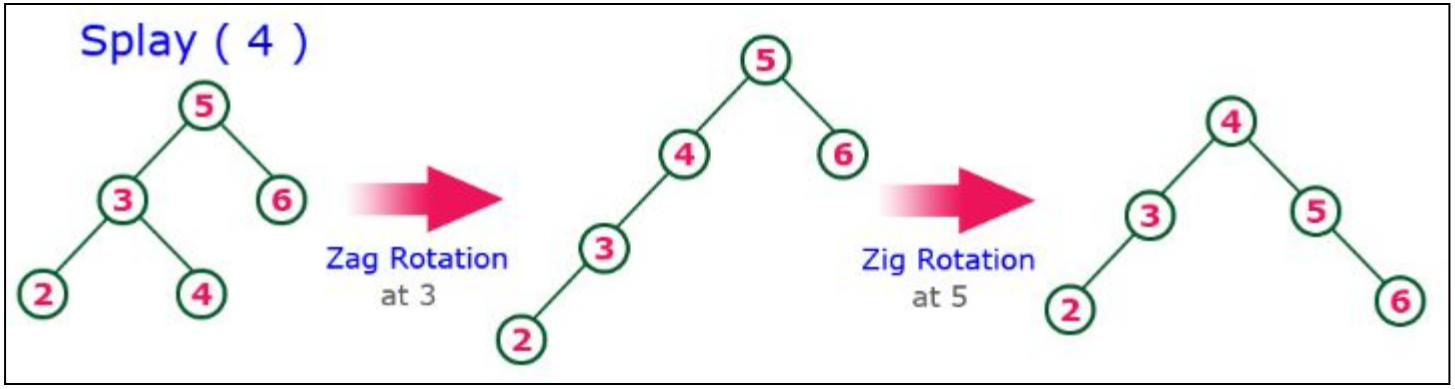
Zag-Zag Rotation - The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



Zig-Zag Rotation - The Zig-Zag Rotation in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation -The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



- **Insertion Operation in Splay Tree**
  - The insertion operation in Splay tree is performed using following steps...
    - **Step 1 -** Check whether tree is Empty.
    - **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
    - **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
    - **Step 4 -** After insertion, **Splay** the **newNode**
- **Deletion Operation in Splay Tree**
  - The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Advantages of Splay Trees:

- Splay trees have amortized time complexity of O(log n) for many operations, making them faster than many other balanced tree data structures in some cases.
- Splay trees are self-adjusting, meaning that they automatically balance themselves as items are inserted and removed. This can help to avoid the performance degradation that can occur when a tree becomes unbalanced.

Disadvantages of Splay Trees:

- Splay trees can have worst-case time complexity of O(n) for some operations, making them less predictable than other balanced tree data structures like AVL trees or red-black trees.
- Splay trees may not be suitable for certain applications where predictable performance is required.

Applications of the splay tree:

- **Caching**: Splay trees can be used to implement cache memory management, where the most frequently accessed items are moved to the top of the tree for quicker access.
- **Database Indexing**: Splay trees can be used to index databases for faster searching and retrieval of data.
- **File Systems**: Splay trees can be used to store file system metadata, such as the allocation table, directory structure, and file attributes.
- **Data Compression:** Splay trees can be used to compress data by identifying and encoding repeating patterns.
- **Text Processing**: Splay trees can be used in text processing applications, such as spell-checkers, where words are stored in a splay tree for quick searching and retrieval.
- **Graph Algorithms:** Splay trees can be used to implement graph algorithms, such as finding the shortest path in a weighted graph.
- **Online Gaming:** Splay trees can be used in online gaming to store and manage high scores, leaderboards, and player statistics.