NOTES

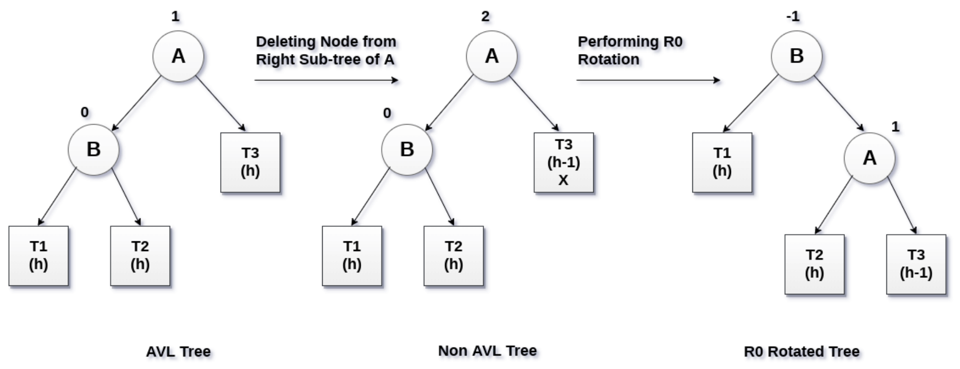
Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness.

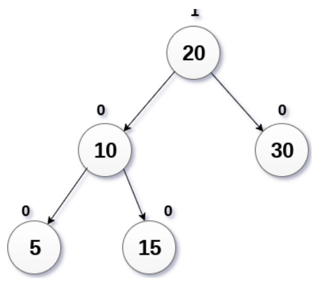
For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

R0 rotation (Node B has balance factor 0)

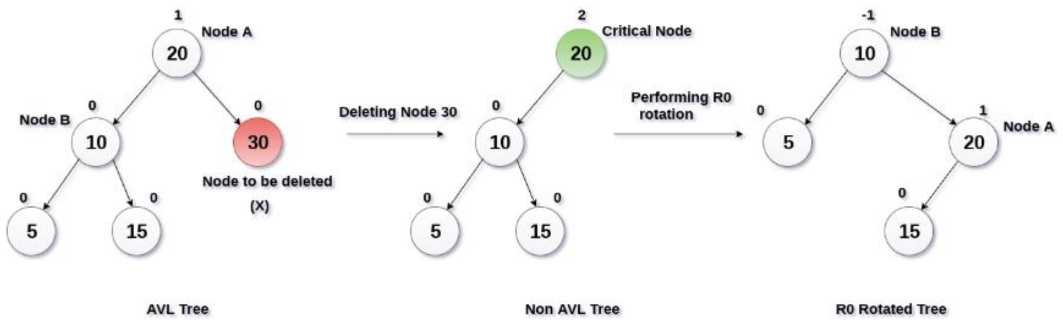
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation. The node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 become the left and right sub-tree of the node A. The process involved in R0 rotation is shown in the following image.



An Example

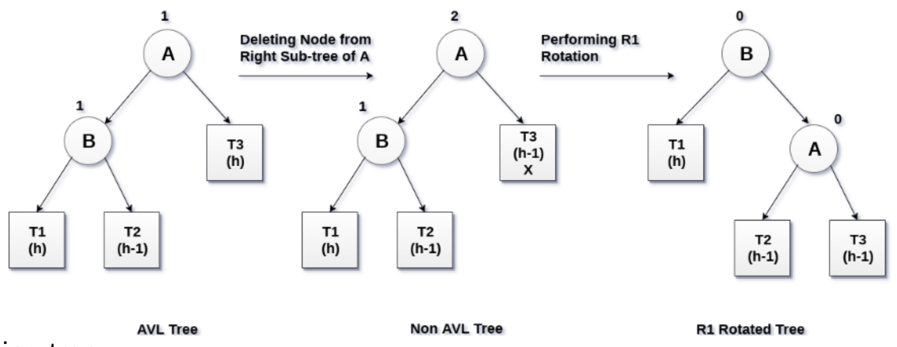


Delete the node 30 from the AVL tree

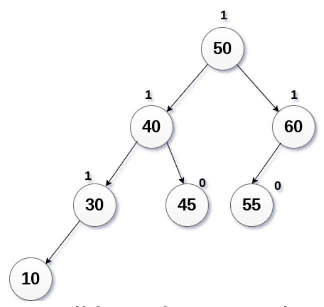


R1 Rotation (Node B has balance factor 1)

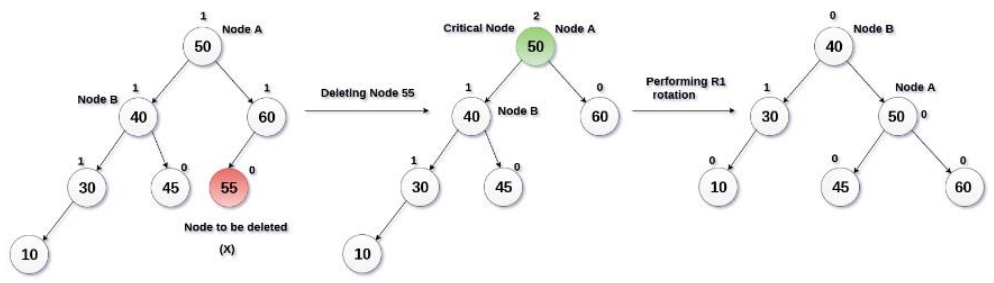
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.



Consider the following tree-

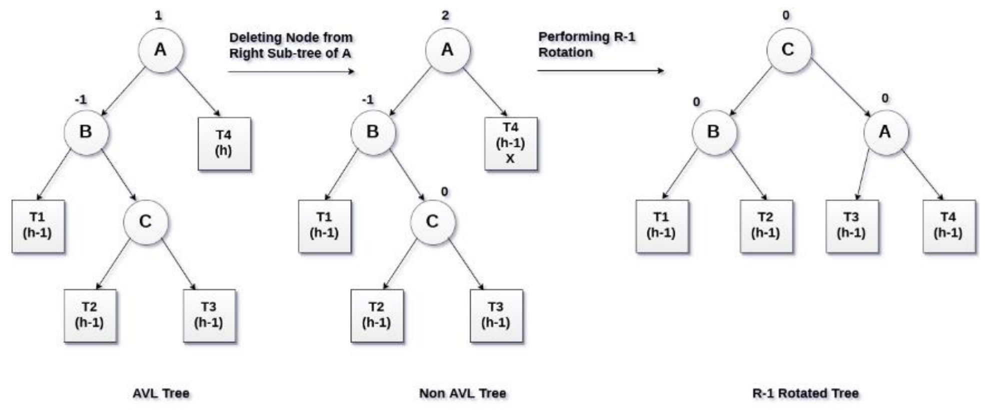


Delete Node 55 from that tree: This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).

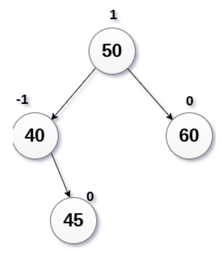


R-1 Rotation (Node B has balance factor -1)

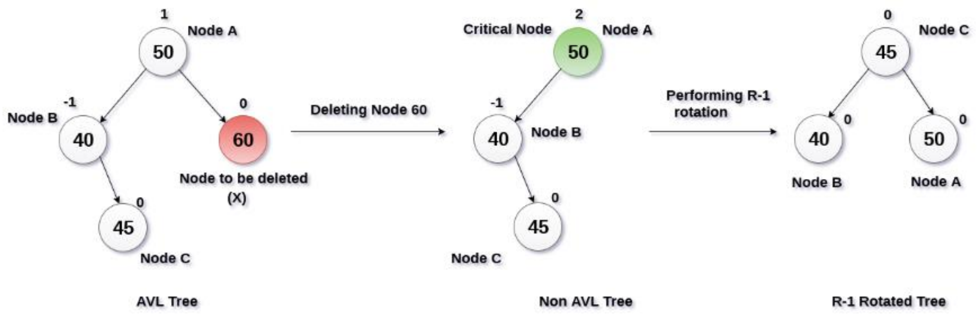
R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively. The sub-trees T1, T2 becomes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



Consider the AVL tree given here



Delete Node 60: in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.

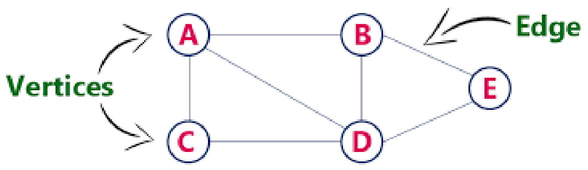


Graph

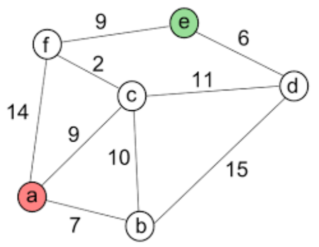
A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

Graph is a data structure that consists of following two components:

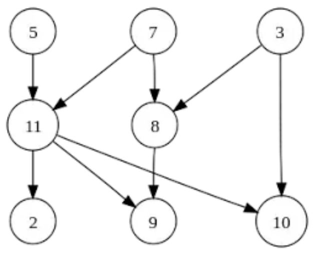
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph (di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.



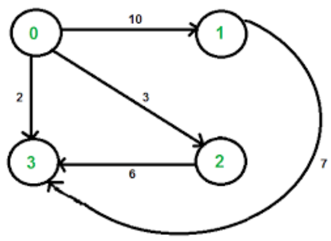
Undirected Graph



Weighted Graph



Directed Graph



Weighed-Directed Graph

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.

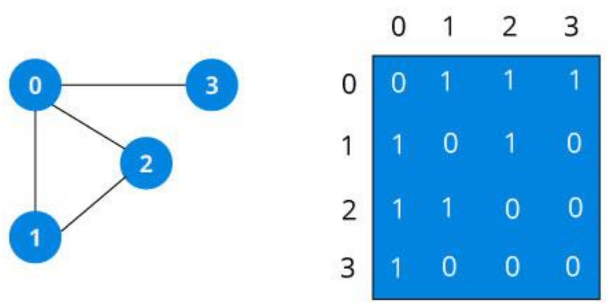
Following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be ar[][], a slot ar[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If ar[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Take a look at diagram here-



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

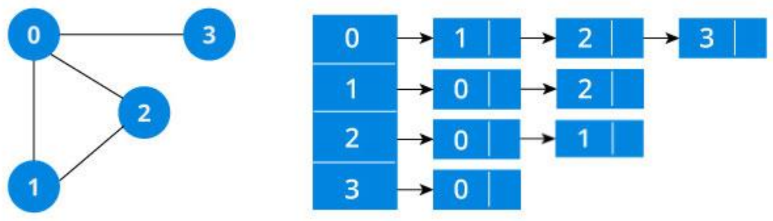
- **Advantage:** Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).
- **Disadvantage:** Consumes more space O(V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

```
class Graph:
    #creating constructor
    def __init__(self, size):
        #add size to object
        self.size= size
        #create an empty list
        self.adjMatrix = []
        #create sub list for each node
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])



    #method to add edges to the matrix
    def add_edgs(self, v1, v2):
        if v1 == v2:
            print("Loop it is")
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    #method to remove the edge also
    def remove_edge(self, v1, v2):
        #check if edge is there
        if self.adjMatrix[v1][v2] == 0:
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0

    def print_graph(self):
        for row in self.adjMatrix:
            for val in row:
                print(val,end = "  ")
            print()

g = Graph(4)
g.add_edgs(0,1)
g.add_edgs(0,2)
g.add_edgs(0,3)
g.add_edgs(1,0)
g.add_edgs(1,2)
g.add_edgs(2,0)
g.add_edgs(2,1)
g.add_edgs(3,0)
```

```
g.print_graph()
```

Adjacency List

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the i^{th} vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.



Advantage:

- Saves space O(|V|+|E|) . In the worst case, consumes $O(V^2)$ space.
- Adding a vertex is easier.

Disadvantage:

- Queries like whether there is an edge from vertex u to vertex v are not efficient.