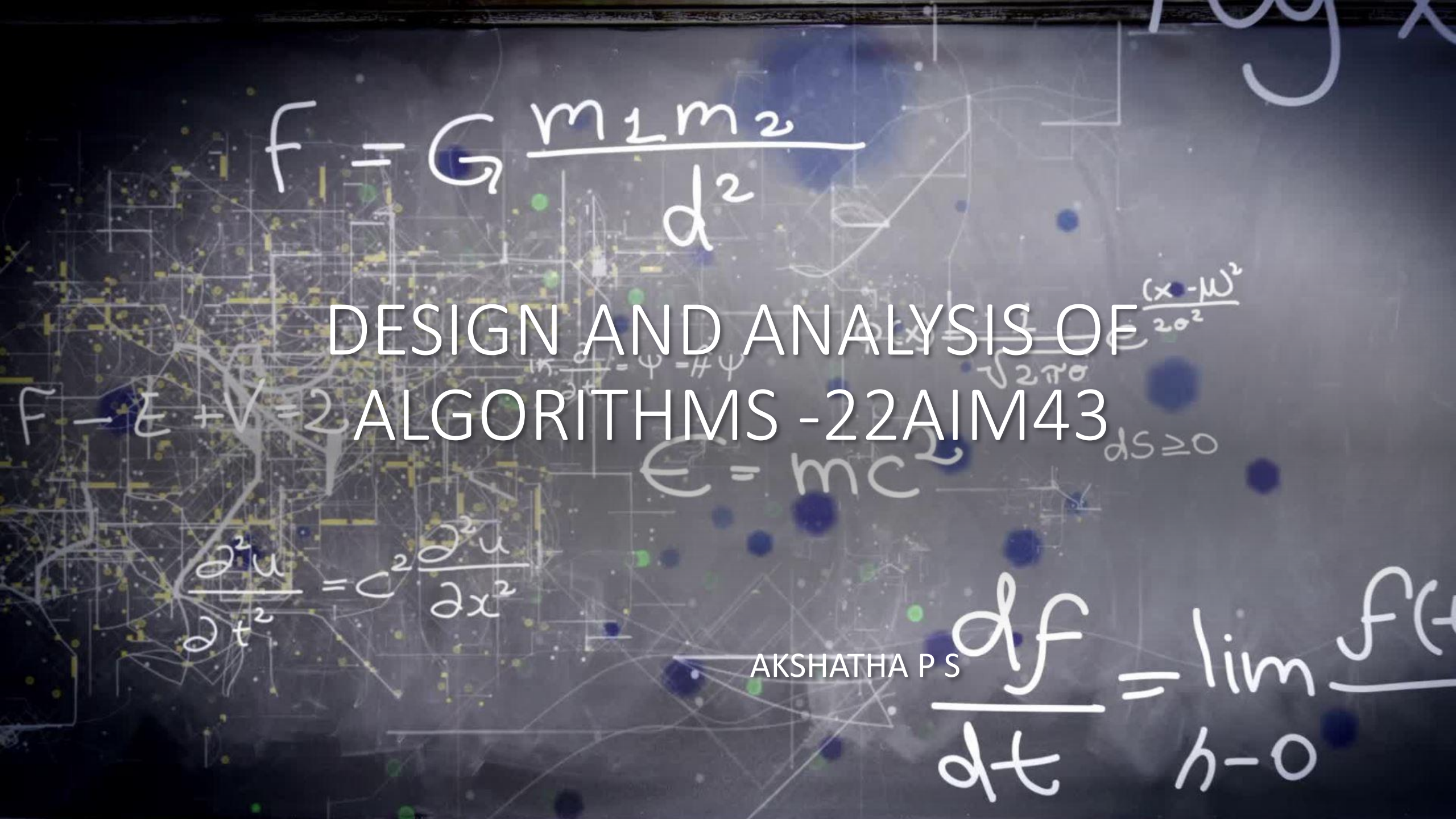


DESIGN AND ANALYSIS OF ALGORITHMS -22AIM43

AKSHATHA P S

MODULE-1	**INTRODUCTION**	**22AIM43.1**	**8 Hours**
Introduction to Algorithms, Role of algorithms in computing, Time and Space Complexity of Algorithms, Asymptotic notations, worst-case, Average-case and Best-case analysis, Analysis Framework- Empirical analysis- Mathematical analysis for Recursive and Non-recursive algorithms.			
Case Study	Illustrate real-world applications of algorithms and growth functions.		
Text Book	Text Book 1:1.1,1.2,1.3		
MODULE-2	**DIVIDE AND CONQUER**	**22AIM43.2**	**8 Hours**
Divide and Conquer Methodology: Binary search, Merge sort, Quick sort, Finding the maximum and minimum, Strassen's matrix, advantages and disadvantages of divide and conquer.			
Case Study	Compare and contrast the time complexity and suitability of the bubble sort, merge sort, and quicksort algorithms. Provide scenarios where one might be preferred over the others.		
Text Book	Text Book 1: 2.1,2.2		
MODULE-3	**GREEDY METHOD AND DYNAMIC PROGRAMMING**	**22AIM43.3**	**8 Hours**
Greeedy method: Introduction, Job scheduling problem, Minimum Spanning tree algorithms – Kruskals & Prims. Optimal Tree Problem: Huffman Trees. **DYNAMIC PROGRAMMING:** Introduction, Knapsack problems, Travelling Salesman problem. Transitive closure - Warshall's and Floyds algorithm.			

MODULE-4	**DECREASE & CONQUER, TRANSFORM & CONQUER**	**22AIM43.4**	**8 Hours**
Decrease &conquer: Introduction – Decrease by constant, decrease by constant factor-Fake Coin Problem-Russian Peasant Multiplication, variable size decrease. **Transform & conquer**: Introduction, Balanced Search trees – AVL trees & 2-3 trees, Red Black Trees			
Text Book	Text Book 1: 5.1,5.2,5.3,5.4,5.5,5.6		
MODULE-5	**BACKTRACKING, BRANCH AND BOUND**	**22AIM43.5,22AIM43.6**	**8 Hours**
Backtracking: Introduction, N Queens problem, subset sum problem, **Branch and Bound:** Introduction, Travelling Salesman problem, Knapsack problem, Assignment problem, NP-Hard and NP-Complete problems: Basic concepts, non-deterministic algorithms.			

Suggested Learning Resources:

Text Books:

1. Anany Levitin,"Introduction to the Design &Analysis of Algorithms",3rd Edition, PEARSON Education, 2012.

ReferenceBook:

1.Thomas H Cormen, Charles E Leiserson, Ronald R Rivest & Clifford Stein, "Introduction to Algorithms ", THIRD Edition, Eastern Economy Edition

Module 4: Decrease & Conquer, Transform & Conquer

Description:

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are

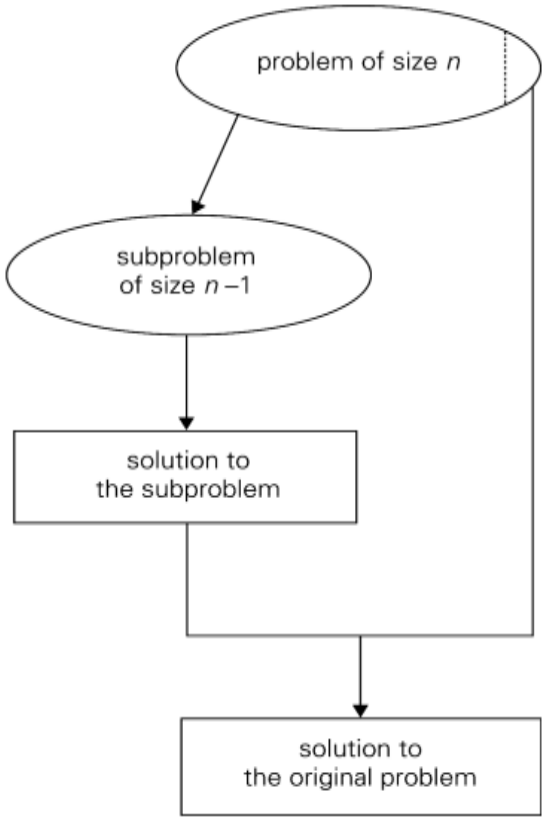
1. Decrease by a constant :(usually by 1):
    a. insertion sort
    b. graph traversal algorithms (DFS and BFS)
    c. topological sorting
    d. algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half)
    a. binary search and bisection method

3. Variable size decrease
    a. Euclid's algorithm

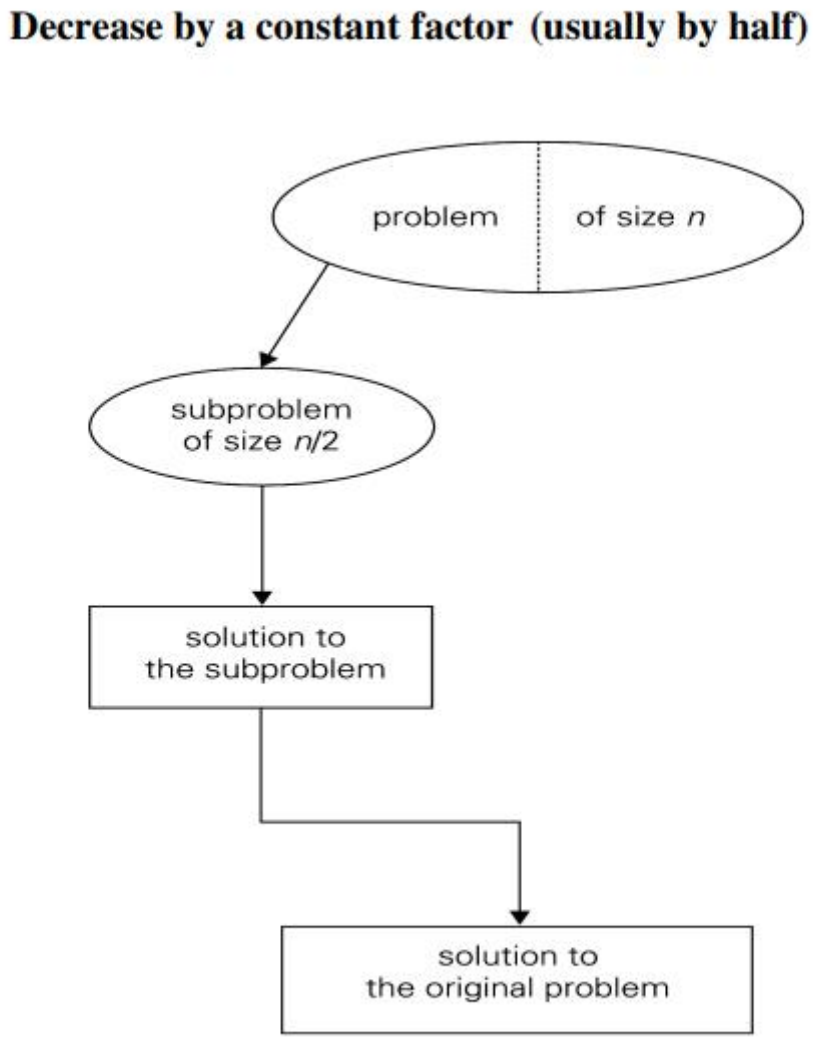
Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant :(usually by 1):



Decrease by a Constant : In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one , although other constant size reductions do happen occasionally. Below are example problems :

- Insertion sort
- Graph search algorithms: DFS, BFS
- Topological sorting
- Algorithms for generating permutations, subsets



Decrease by a Constant factor: This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. A reduction by a factor other than two is especially rare. Decrease by a constant factor algorithms are very efficient especially when the factor is greater than 2 as in the fake-coin problem. Below are example problems :

- Binary search
- Fake-coin problems
- Russian peasant multiplication

Variable-Size-Decrease : In this variation, the size-reduction pattern varies from one iteration of an algorithm to another. As, in problem of finding gcd of two number though the value of the second argument is always smaller on the right-handside than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Advantages of Decrease and Conquer:

1.Simplicity: Decrease-and-conquer is often simpler to implement compared to other techniques like dynamic programming or divide-and-conquer.

2.Efficient Algorithms: The technique often leads to efficient algorithms as the size of the input data is reduced at each step, reducing the time and space complexity of the solution.

3.Problem-Specific: The technique is well-suited for specific problems where it's easier to solve a smaller version of the problem.

Disadvantages of Decrease and Conquer:

1.Problem-Specific: The technique is not applicable to all problems and may not be suitable for more complex problems.

2.Implementation Complexity: The technique can be more complex to implement when compared to other techniques like divide-and-conquer, and may require more careful planning.

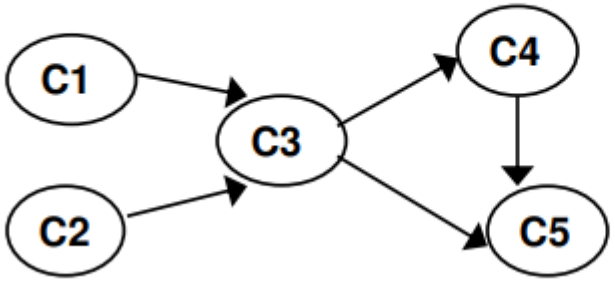
Example on Decrease by a Constant: Topological Sorting Using Source Removal Method

Source removal method:

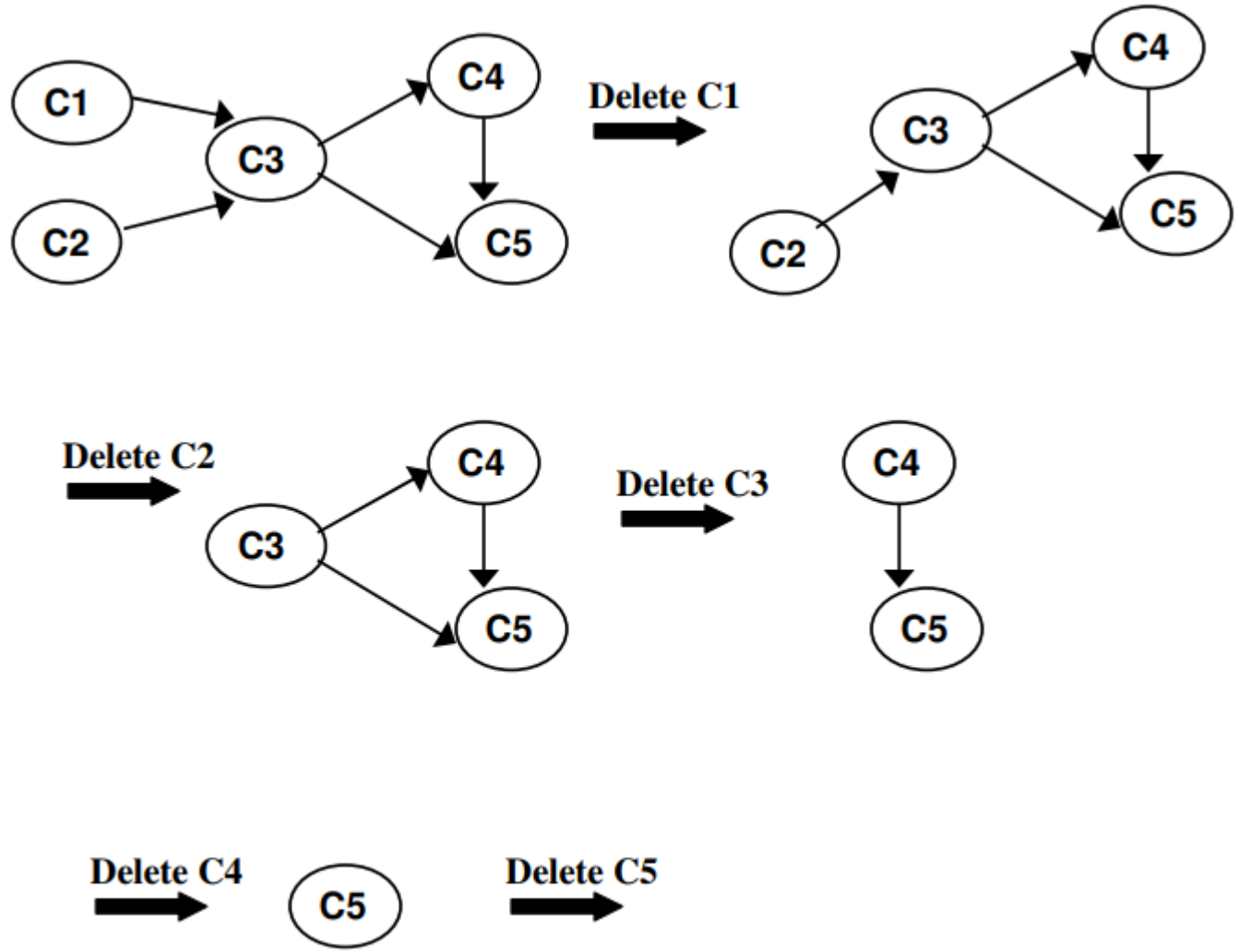
- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges
- Delete it along with all the edges outgoing from it.

Example:

Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:

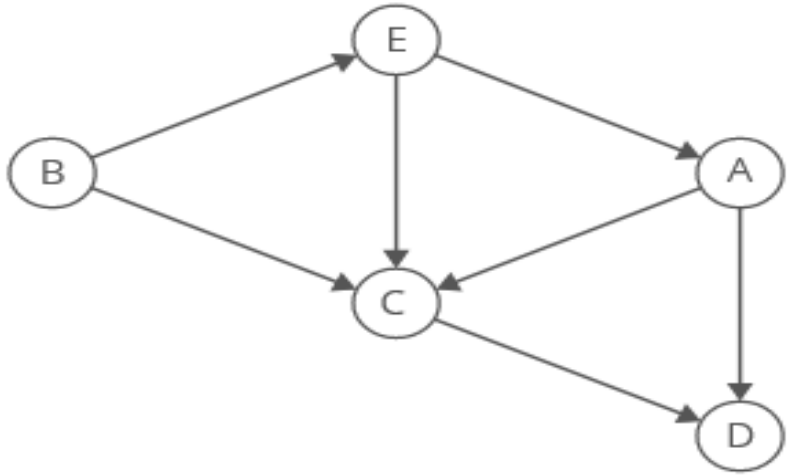


Solution:

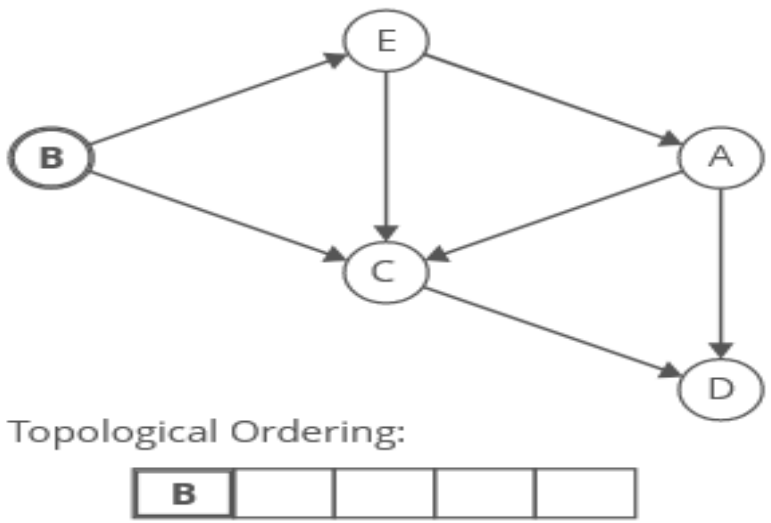


The topological order is C1, C2, C3, C4, C5

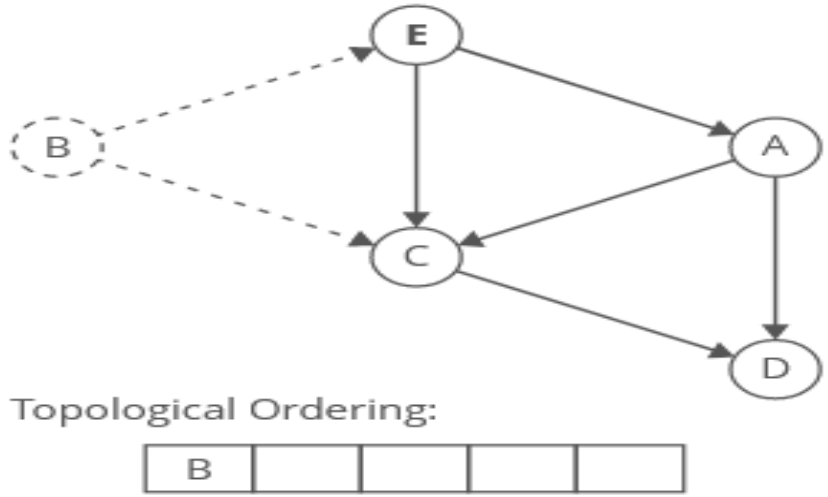
Example 2:



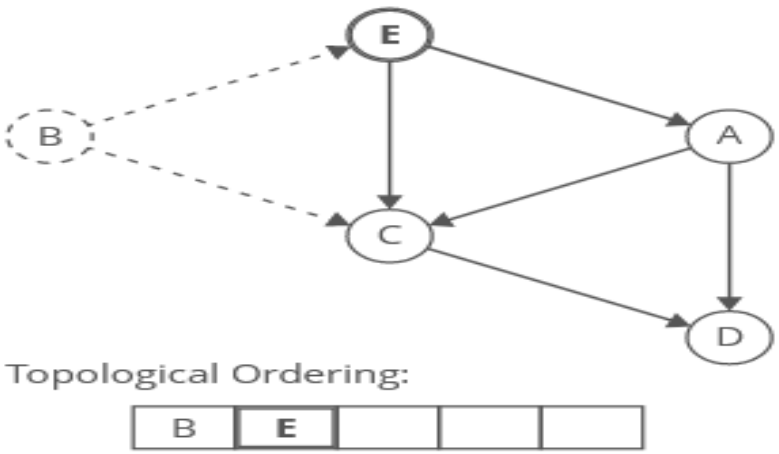
So, we'll find a node with an indegree of zero and add it to the topological ordering.

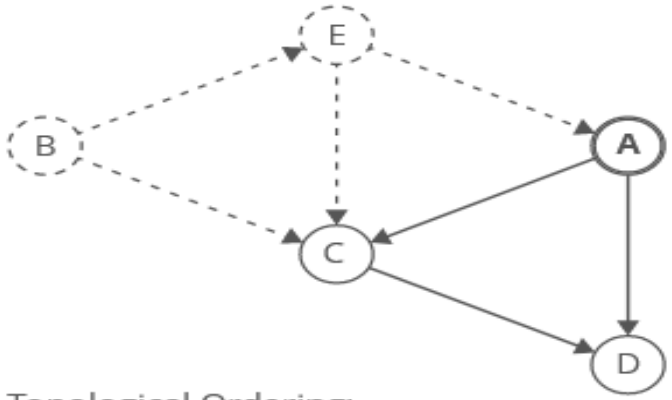


Once a node is added to the topological ordering, we can take the node, and its outgoing edges, out of the graph.



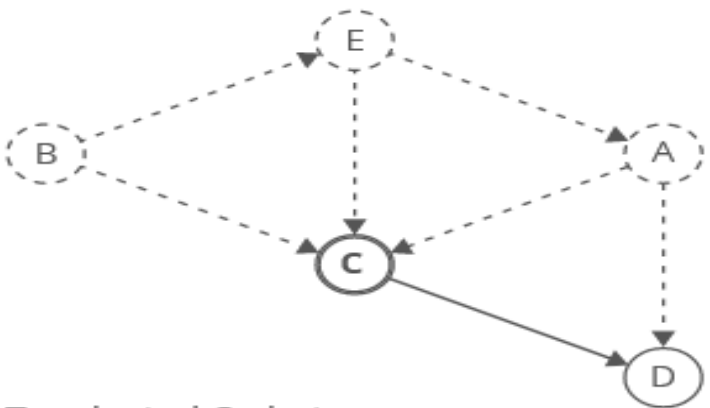
and repeat





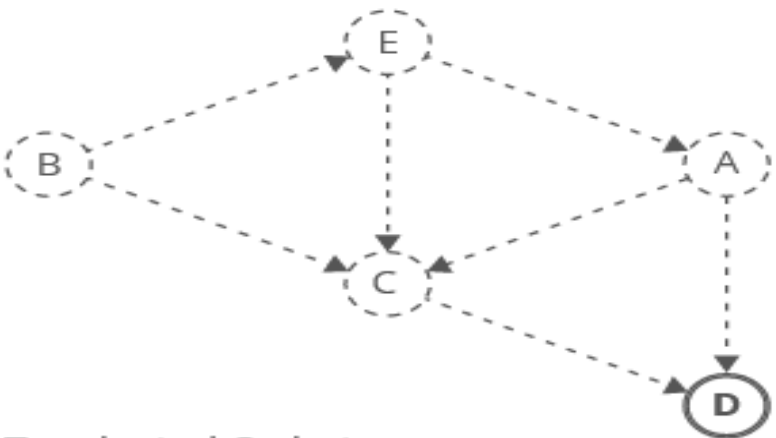
Topological Ordering:

B	E	**A**		



Topological Ordering:

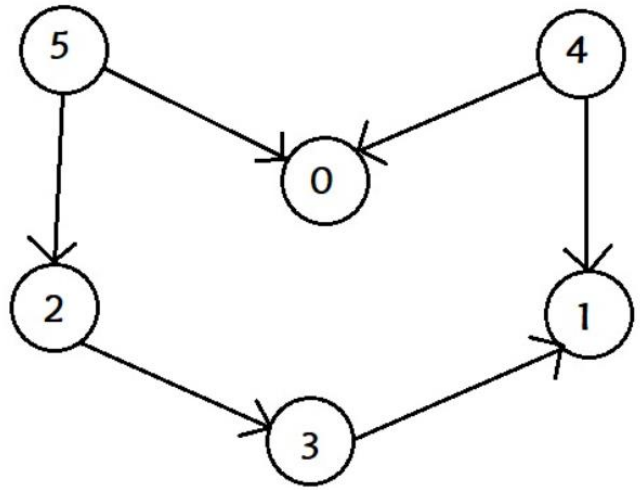
B	E	A	**C**	



Topological Ordering:

B	E	A	C	**D**

Exercise:



Example on Decrease by a Constant Factor: Fake Coin Problem-Russian Peasant Multiplication

Definition

- Best illustrates the decrease-by-a-constant-factor strategy
- Among n identical-looking coins, one is fake.

Definition

- Among n identical-looking coins, one is fake.
- Problem: to design an efficient algorithm for detecting the fake coin
  - Assumption: fake coin is known to be lighter than the genuine one

How to compare?

- With a balance scale, any two sets of coins can be compared

How to compare?

- Based on tipping to the left, to the right, or staying even, the Balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.

How to compare?

- Staying even,

How to compare?

- Tipping to the left,

Fake Coin - Right

How to compare?

- Tipping to the right,

Basic Idea

- If n is odd (n−1) is even
  - Keep one extra coin aside
  - Split (n−1) coins into two piles and put each half on the balance scale
  - If the piles weigh the same, the coin put aside must be fake 
  - If the balance is not even, then we choose the lightest pile of coins to be the one containing the fake.  
- Proceed in the same manner with the lighter pile - one with the fake coin

Basic Idea

- If n is even
  - Split n coins into two piles and put each half on the balance scale

  - If the piles weigh the same, No fake coin
  - If the balance is not even, then we choose the lightest pile of coins to be the one containing the fake.

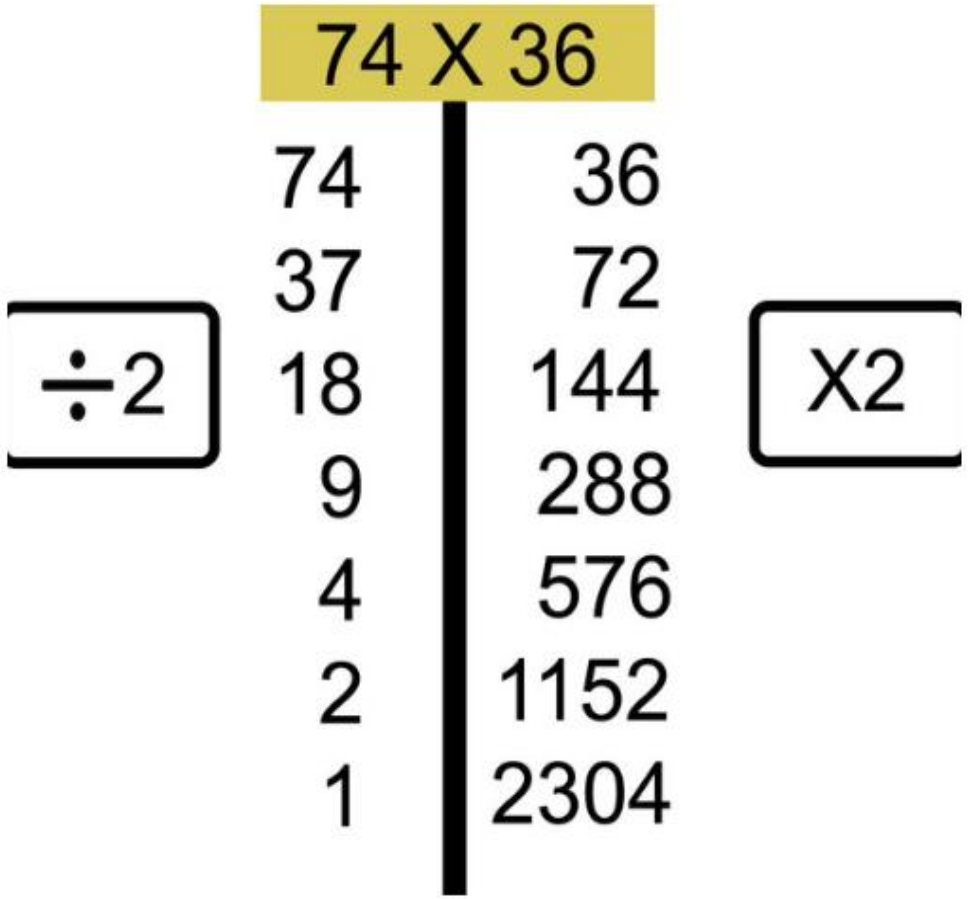
- Proceed in the same manner with the lighter pile - one with the fake coin

Example on Decrease by a Constant Factor: Russian Peasant Multiplication

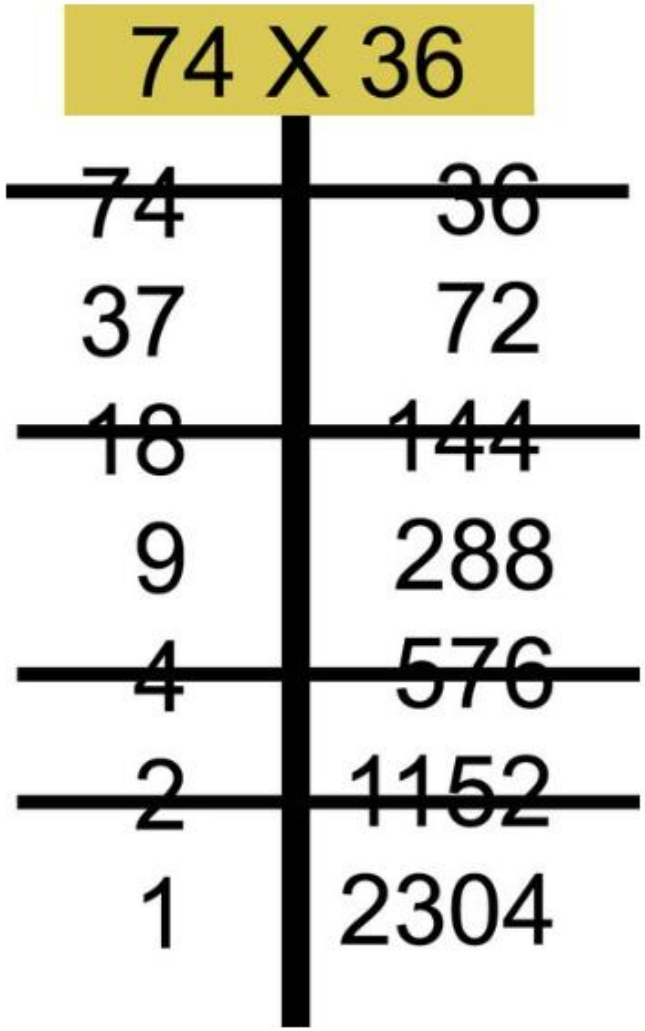
Russian peasant multiplication is an interesting way to multiply numbers that uses a process of halving and doubling. Like standard multiplication and division, Russian peasant multiplication is an algorithm; however, it allows you to multiply any two whole numbers using only multiplication and division by 2. Although Russian peasant multiplication is not as quick as the multiplication method that is now standard, it's still fun to try.

How it works

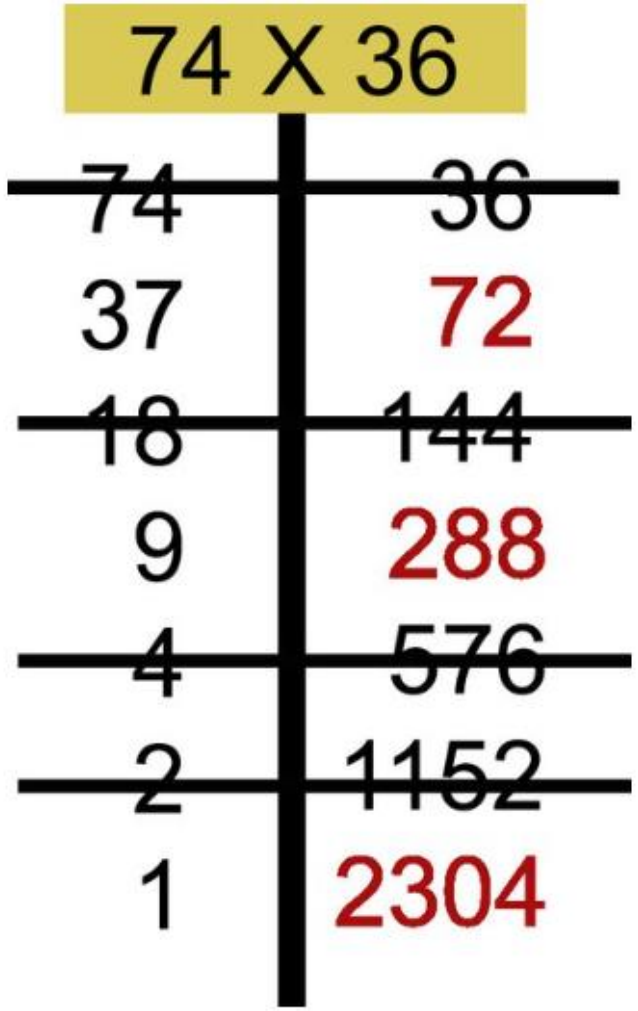
Write the problem on the top and make a column under each of the numbers of the problem. Every time you half the number on the left and double the number on the right. when you need to half an odd number take out 1 so it is even and half it. We will talk about what remains in a minute.



Next, cross out the rows in which the number on the left is even. Extend your line to cross out the corresponding number on the right as well.



Find the sum of the remaining numbers in the right column. (the numbers in the right column that you did not strike through.) The sum of these numbers is equal to the product you would get from multiplying the original numbers using the standard method.



72
288
2304 +
2664

Exercises:

1. 146 X 37
2. 33 X 22
3. 64 X 61

TRANSFORM AND CONQUER

Transform and conquer is a design paradigm where a given problem is transformed to another domain.

Use Cases:

1. Data Compression:

Data is transformed into a more compressed form to take up less space. In transform coding, the source signal is transformed into a new representation, usually using a linear transformation such as a discrete cosine transform (DCT). In contrast, in conquer coding, the source data is first partitioned into smaller blocks, then entropy-coded.

2. Machine Learning:

Transform and conquer is used in machine learning to transform data into a form that is easier to work with. This is done by feeding the computer data, which is then used to train the computer to recognize patterns.

3. Image Processing:

Data is transformed into a form that is more conducive to image analysis. Image Processing usually works by transforming an image into another form, and then conquering it to extract the desired information. It works by first converting an image into digital form, and then performing operations on the digital representation of the image to achieve the desired result.

Advantages:

- **Efficiency:** The transform and conquer algorithm is very efficient in terms of both time and space complexity.
- **Generality:** The transform and conquer algorithm can be applied to a wide variety of problems, including those that are NP-hard.
- **Flexibility:** The transform and conquer algorithm is highly flexible and can be adapted to solve different types of problems.
- **Scalability:** The transform and conquer algorithm is very scalable and can be applied to problems of any size.

Limitations:

- **Complexity in identifying the problem:** It is often very much complex to identify a problem in which we can apply this technique or which variation of the technique will be more fruitful.
- **Time-consuming manual process:** The process of manually coding and analyzing data can be time-consuming, especially when working with large data sets.

AVL TREES

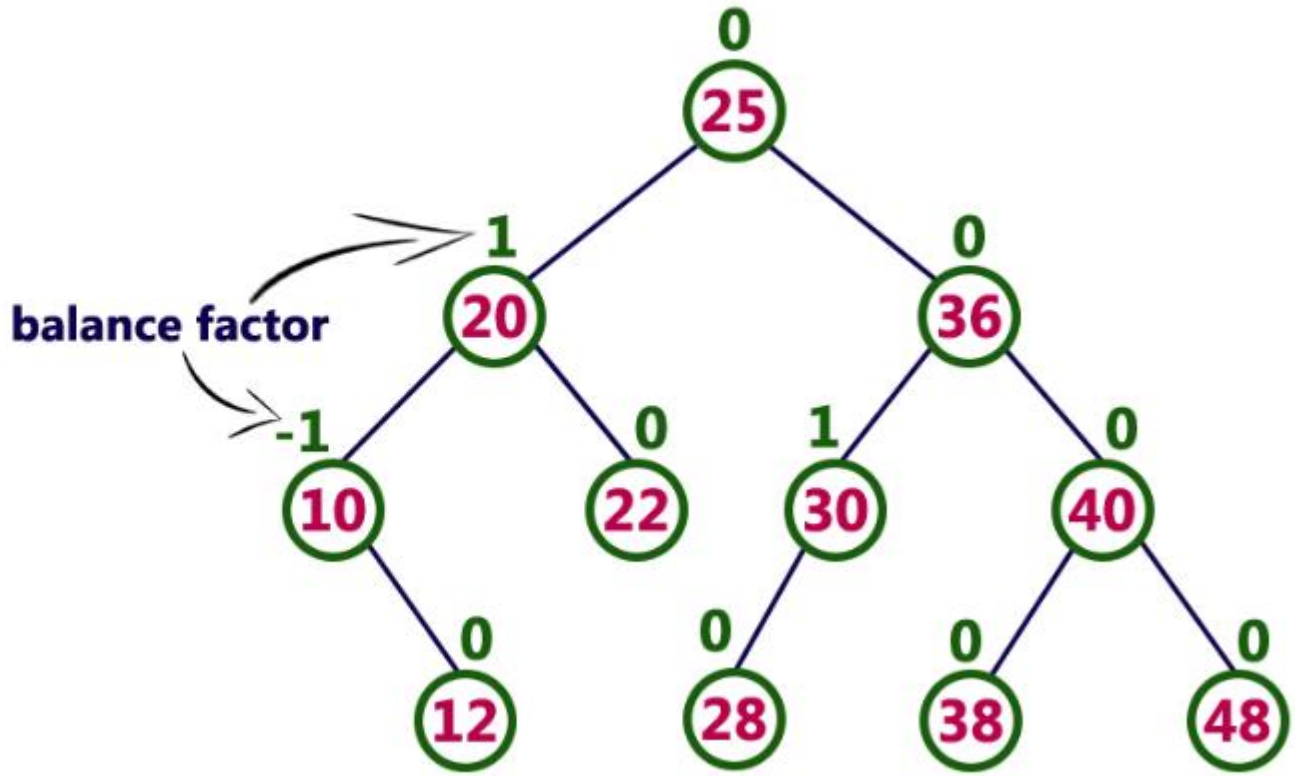
AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**.

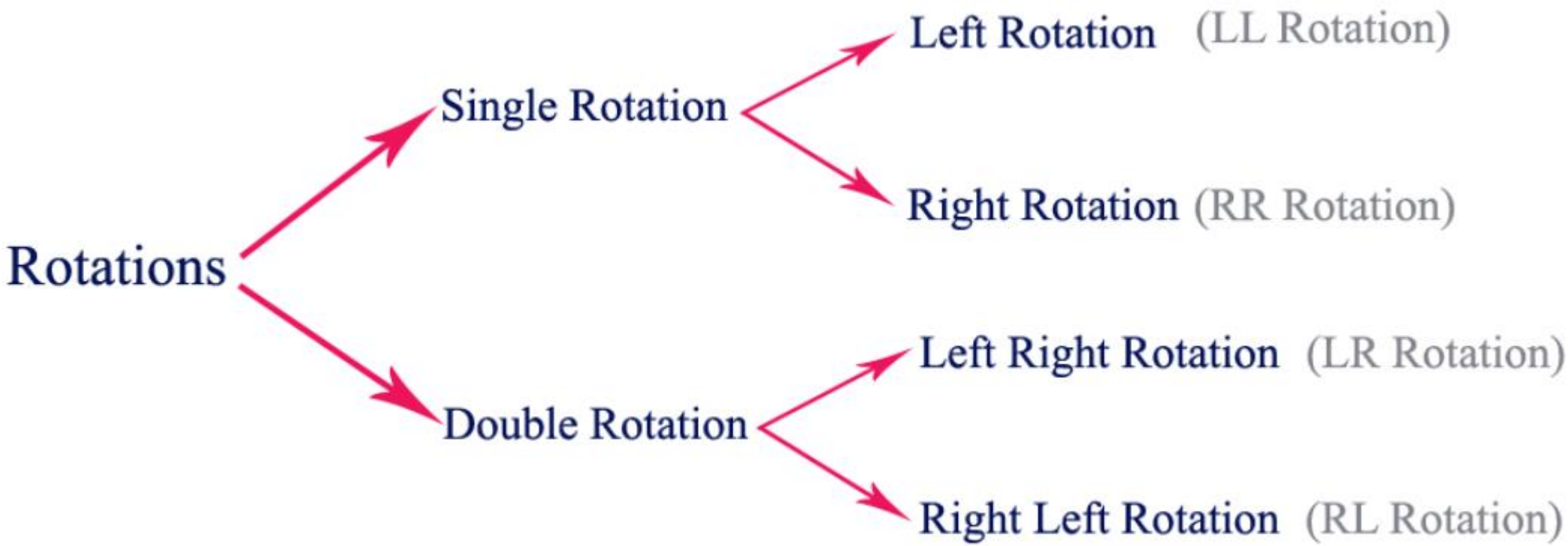
Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree



AVL Tree Rotations

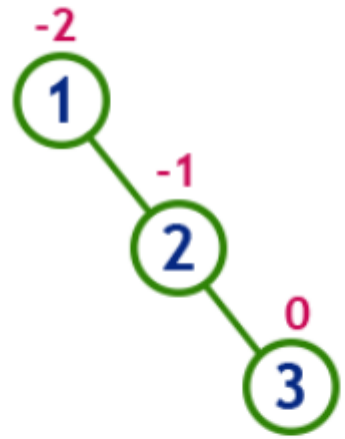
Rotation is the process of moving nodes either to left or to right to make the tree balanced.



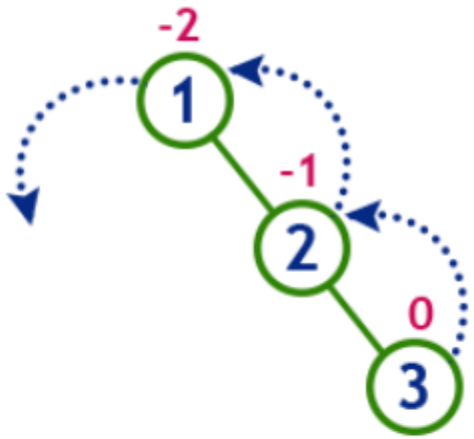
Single Left Rotation (LL Rotation)

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...

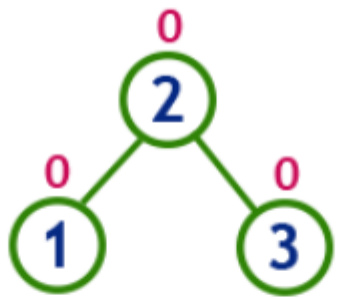
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

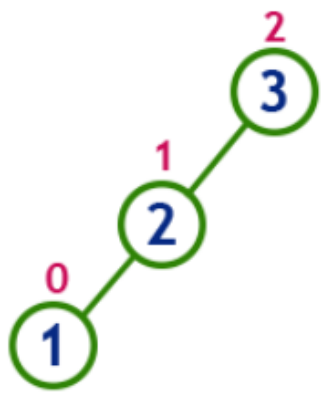


After LL Rotation Tree is Balanced

Single Right Rotation (RR Rotation)

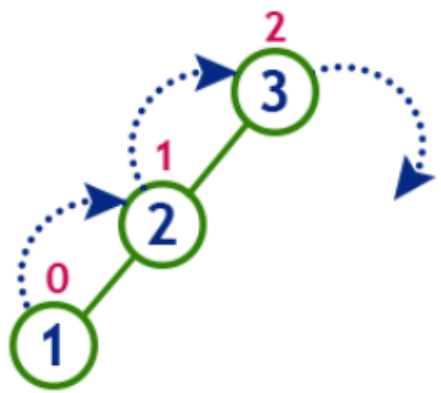
In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

insert 3, 2 and 1

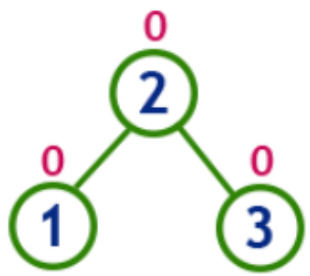


Tree is imbalanced

because node 3 has balance factor 2



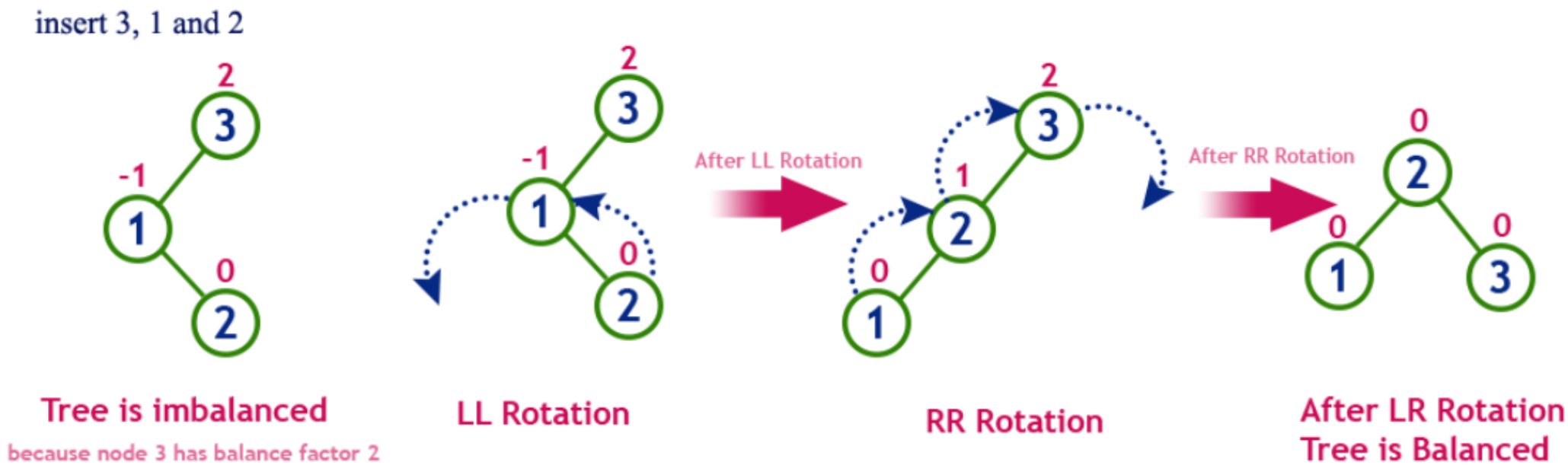
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

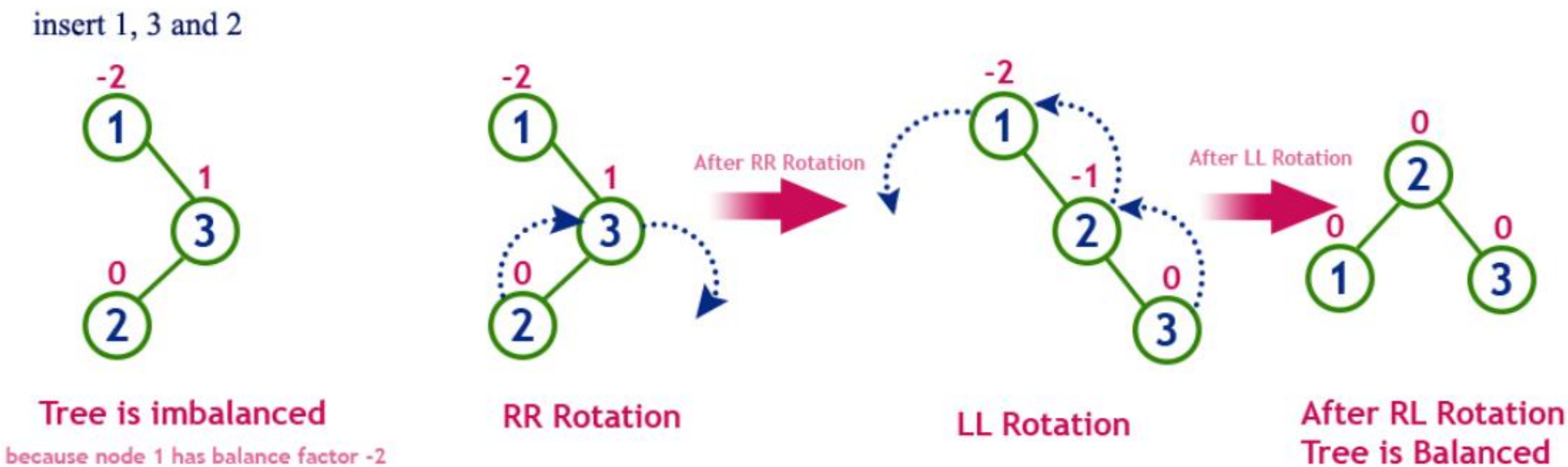
Left Right Rotation (LR Rotation)

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...



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...

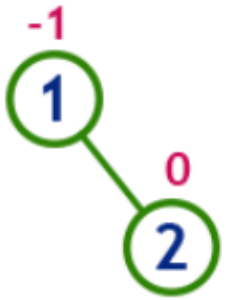


Example: Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1

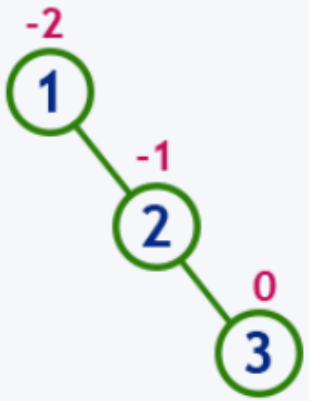
Tree is balanced

insert 2

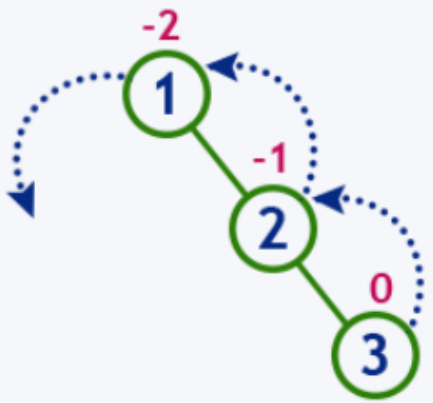


Tree is balanced

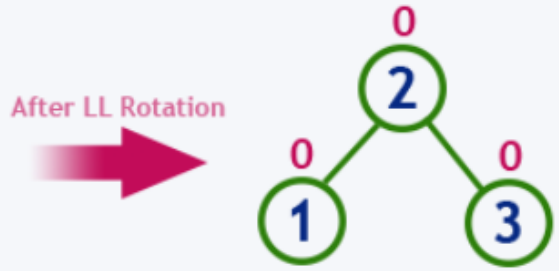
insert 3



Tree is imbalanced

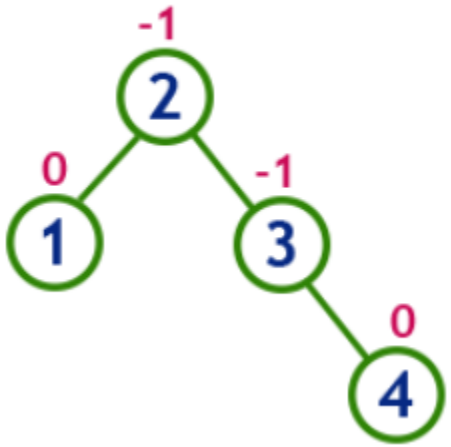


LL Rotation



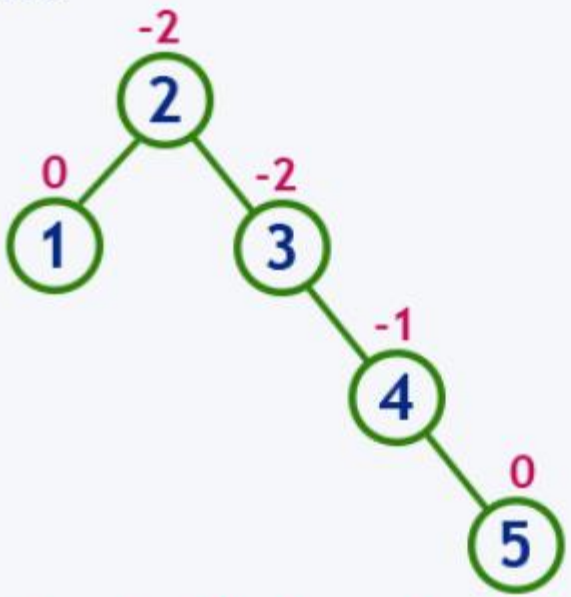
Tree is balanced

insert 4

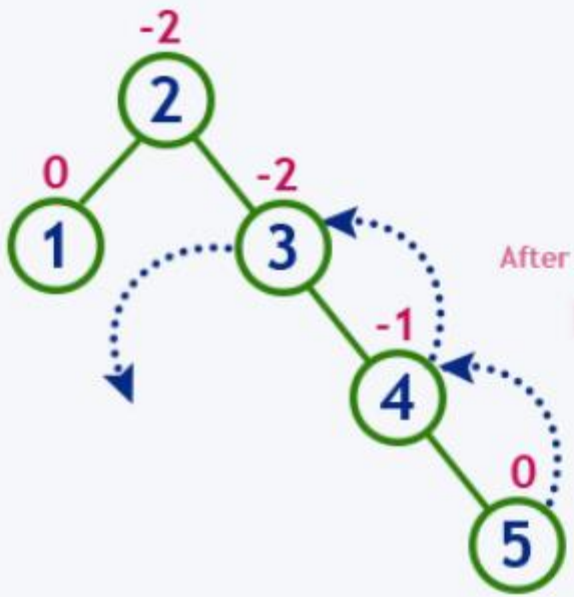


Tree is balanced

insert 5

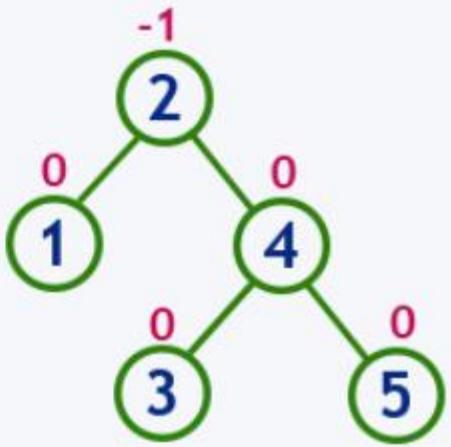


Tree is imbalanced



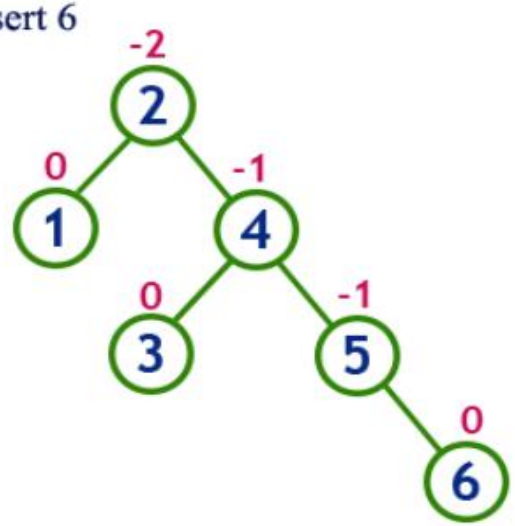
LL Rotation at 3

After LL Rotation at 3

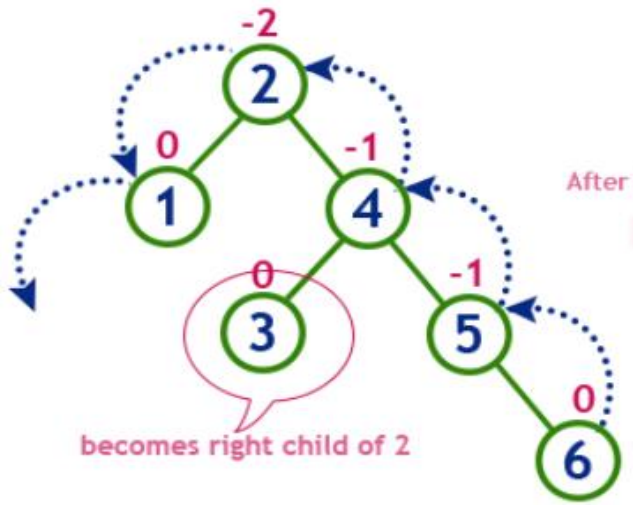


Tree is balanced

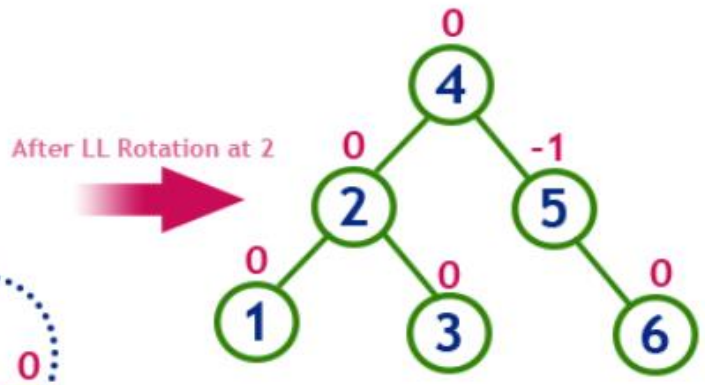
insert 6



Tree is imbalanced

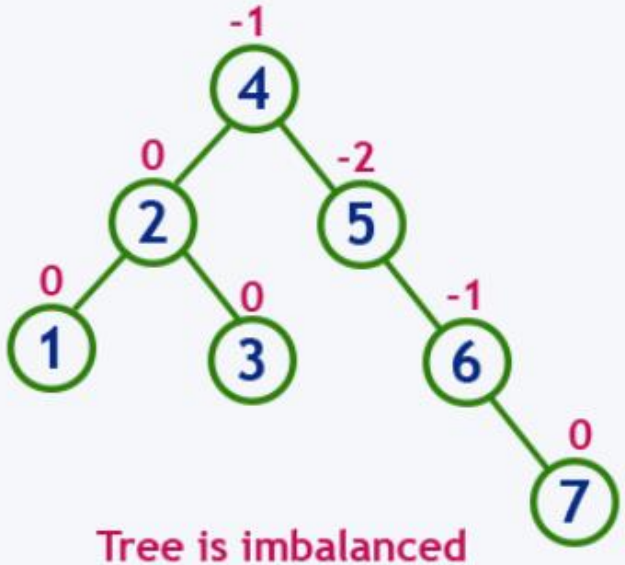


LL Rotation at 2

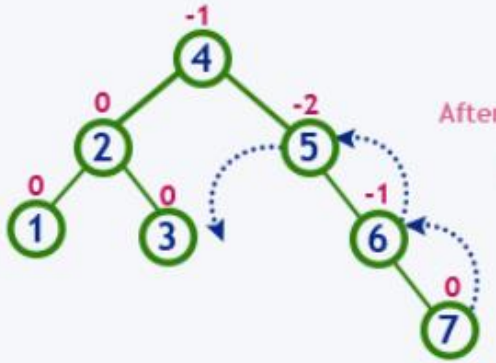


Tree is balanced

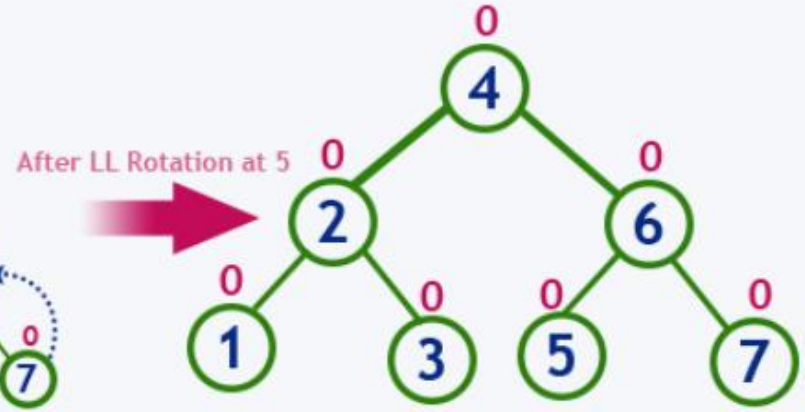
insert 7



Tree is imbalanced

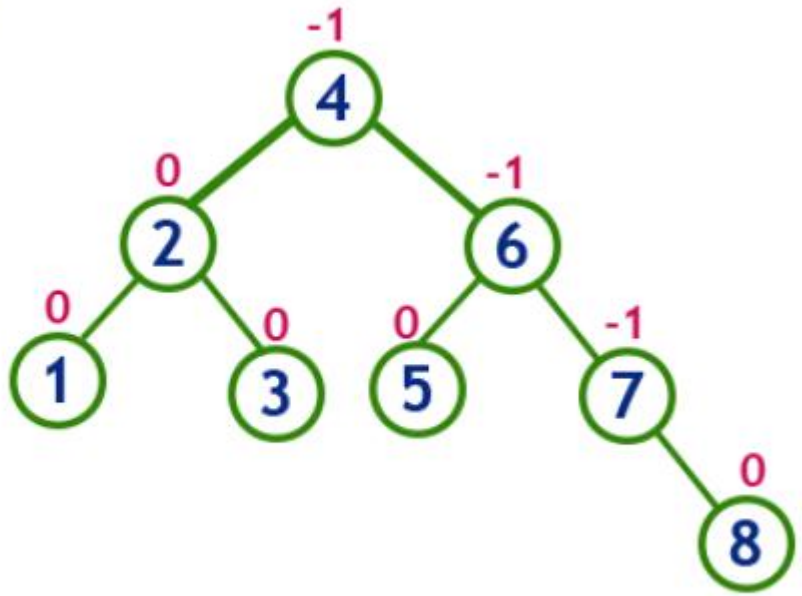


LL Rotation at 5



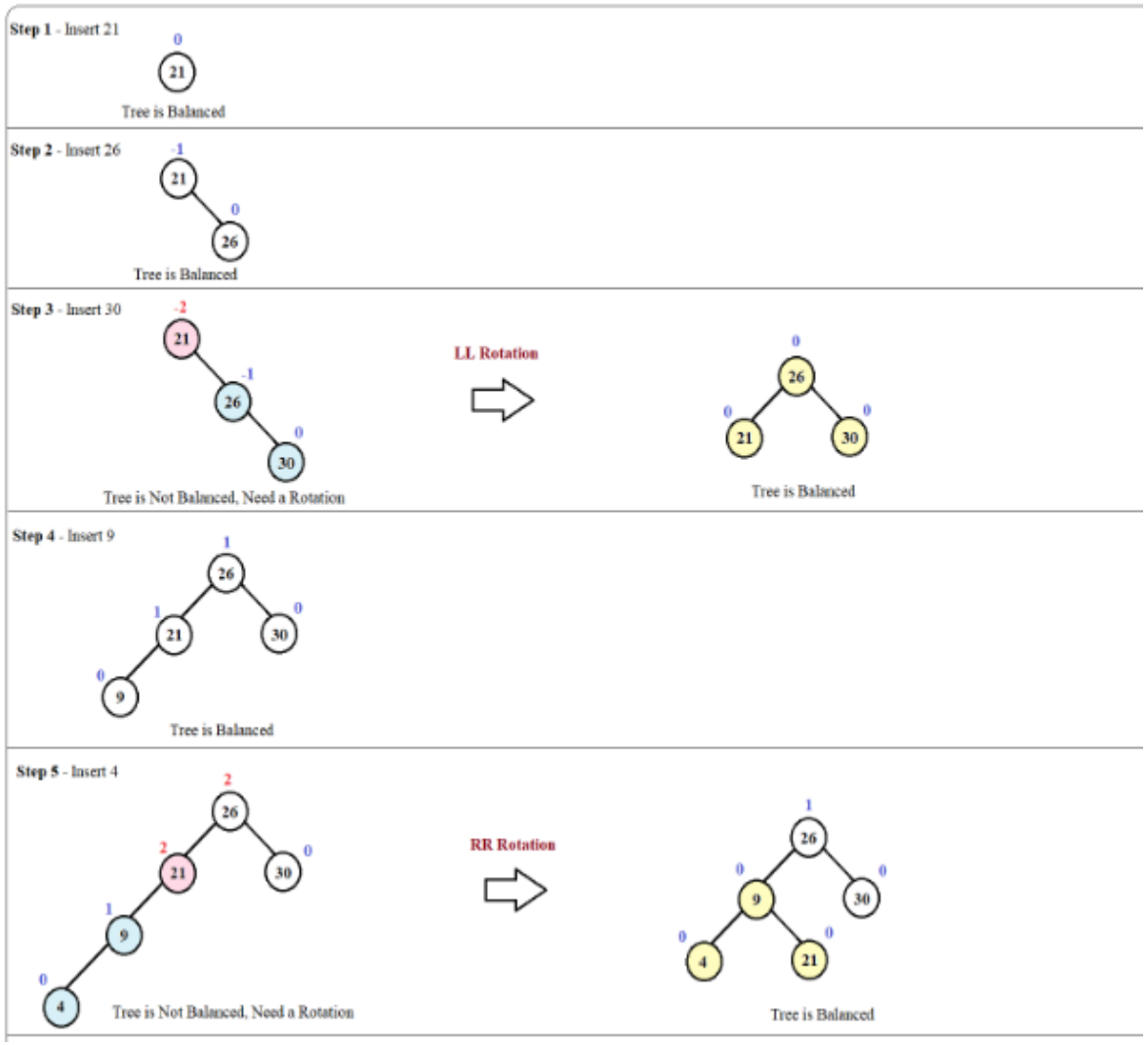
Tree is balanced

insert 8

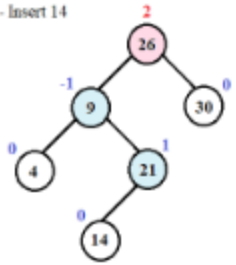


Tree is balanced

Construct AVL tree for the following data 21,26,30,9,4,14,28,18,15,10,2,3,7

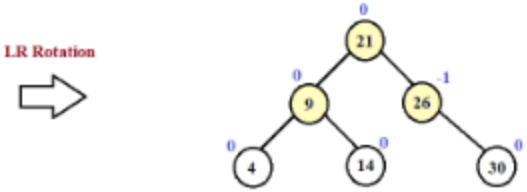


Step 6 - Insert 14



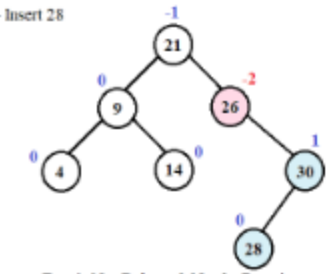
Tree is Not Balanced, Need a Rotation

LR Rotation



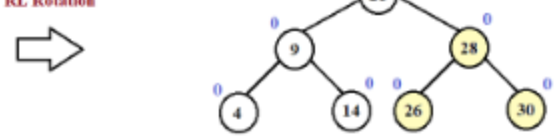
Tree is Balanced

Step 7 - Insert 28



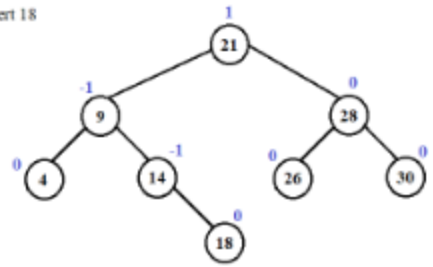
Tree is Not Balanced, Need a Rotation

RL Rotation



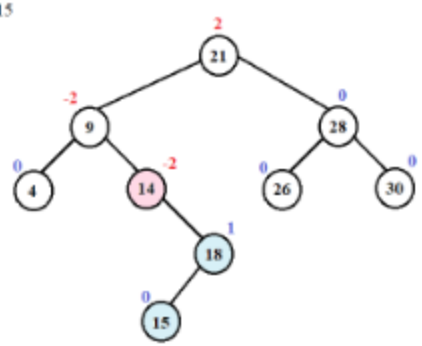
Tree is Balanced

Step 8 - Insert 18



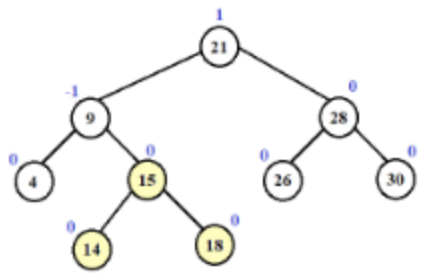
Tree is Balanced

Step 9 - Insert 15



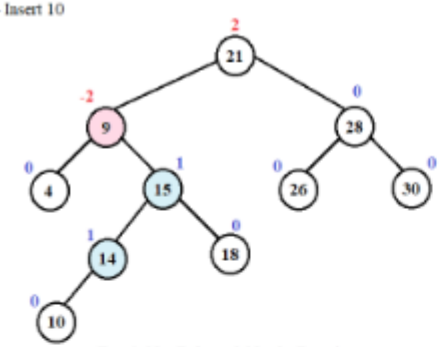
Tree is Not Balanced, Need a Rotation

RL Rotation



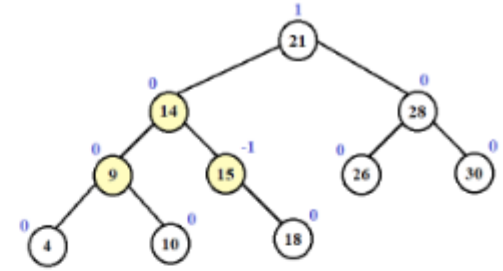
Tree is Balanced

Step 10 - Insert 10



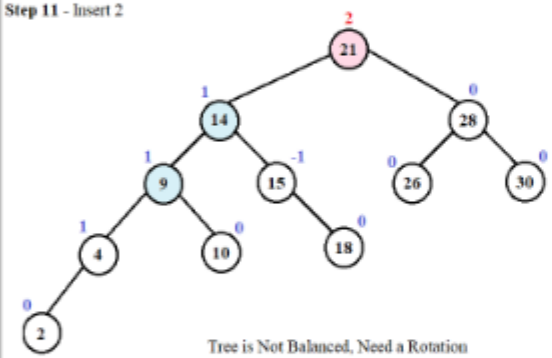
Tree is Not Balanced, Need a Rotation

RL Rotation



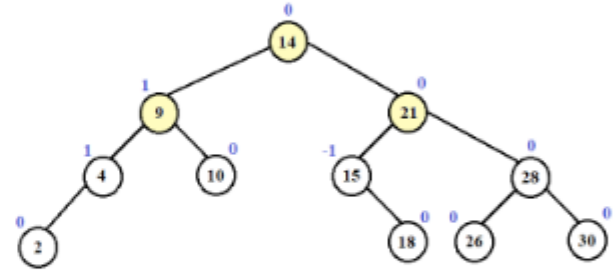
Tree is Balanced

Step 11 - Insert 2



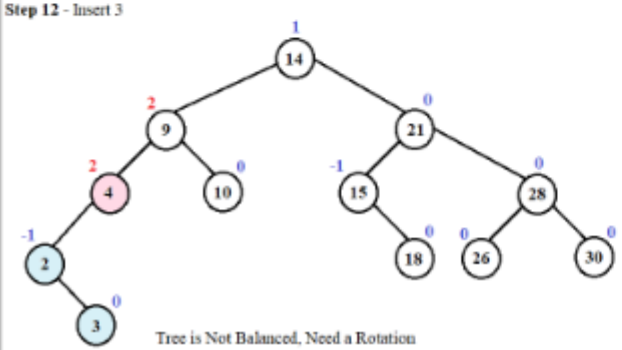
Tree is Not Balanced, Need a Rotation

RR Rotation



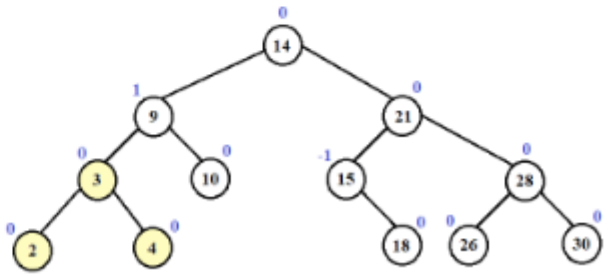
Tree is Balanced

Step 12 - Insert 3



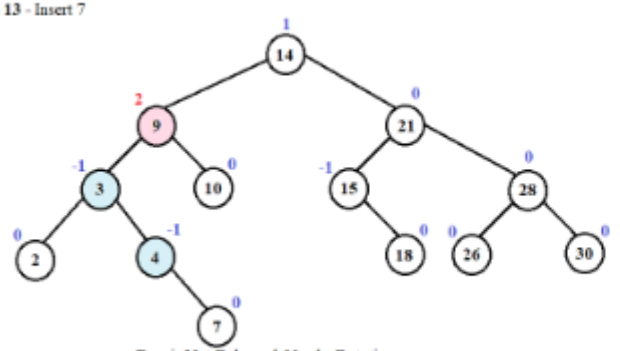
Tree is Not Balanced, Need a Rotation

LR Rotation



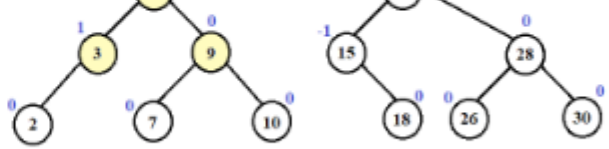
Tree is Balanced

Step 13 - Insert 7



Tree is Not Balanced, Need a Rotation

LR Rotation

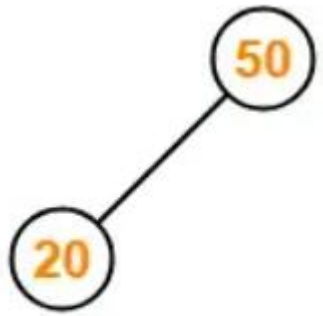


Tree is Balanced

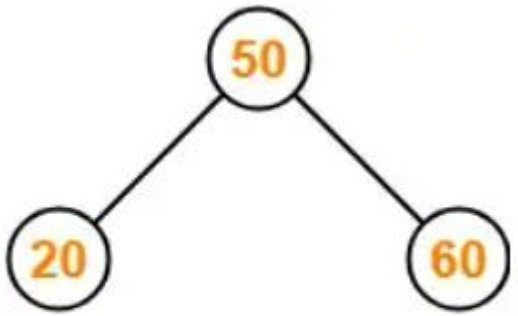
Construct AVL Tree for the following sequence of numbers-

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

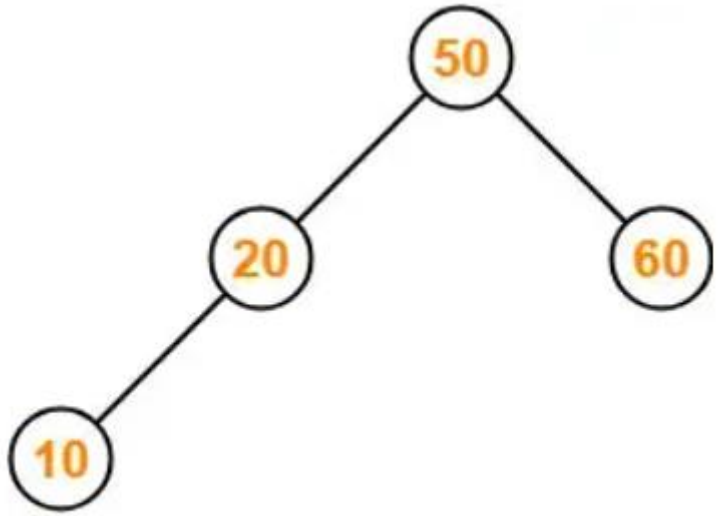
Tree is Balanced



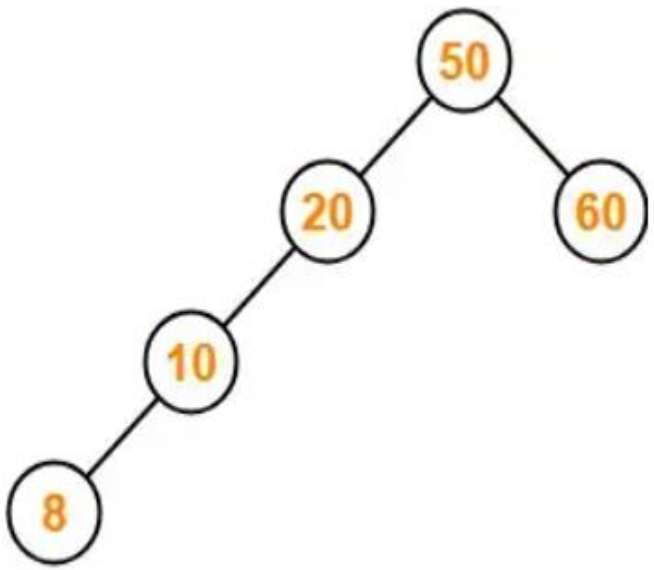
Tree is Balanced



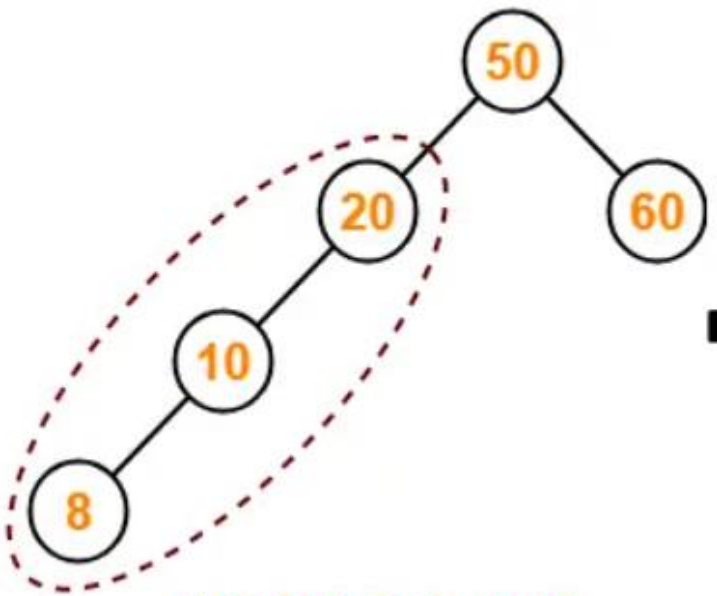
Tree is Balanced



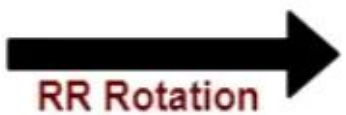
Tree is Balanced

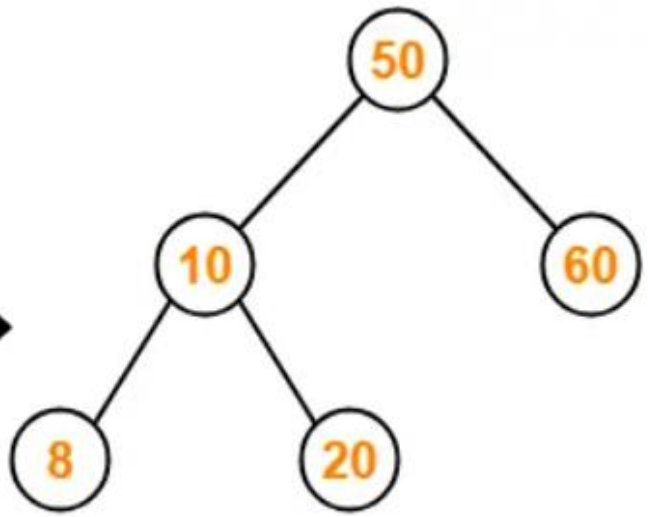


Tree is Imbalanced

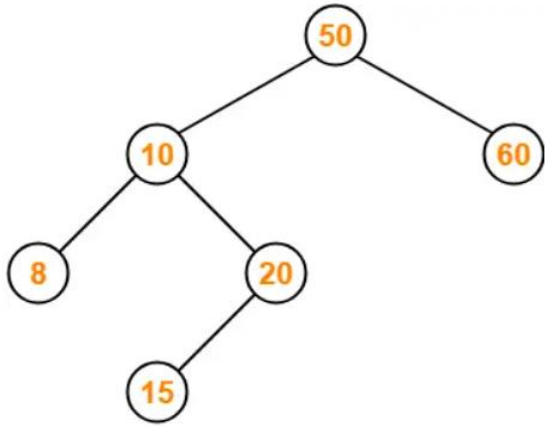


Tree is Imbalanced

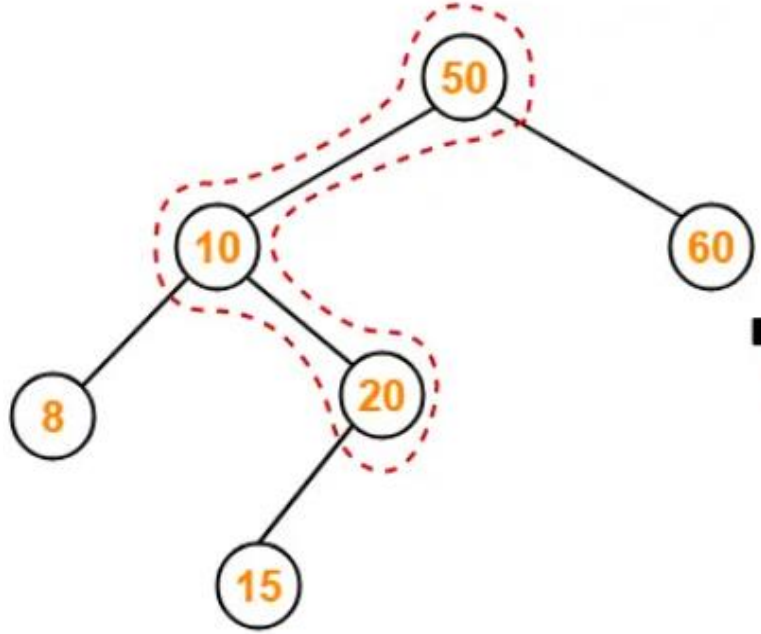




Tree is Balanced

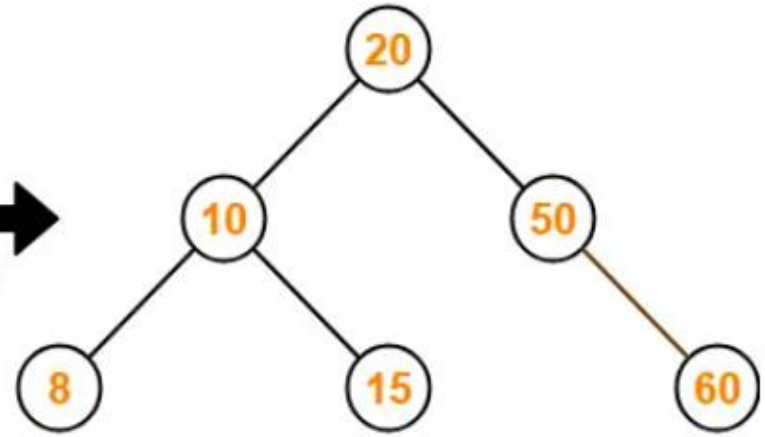


Tree is Imbalanced

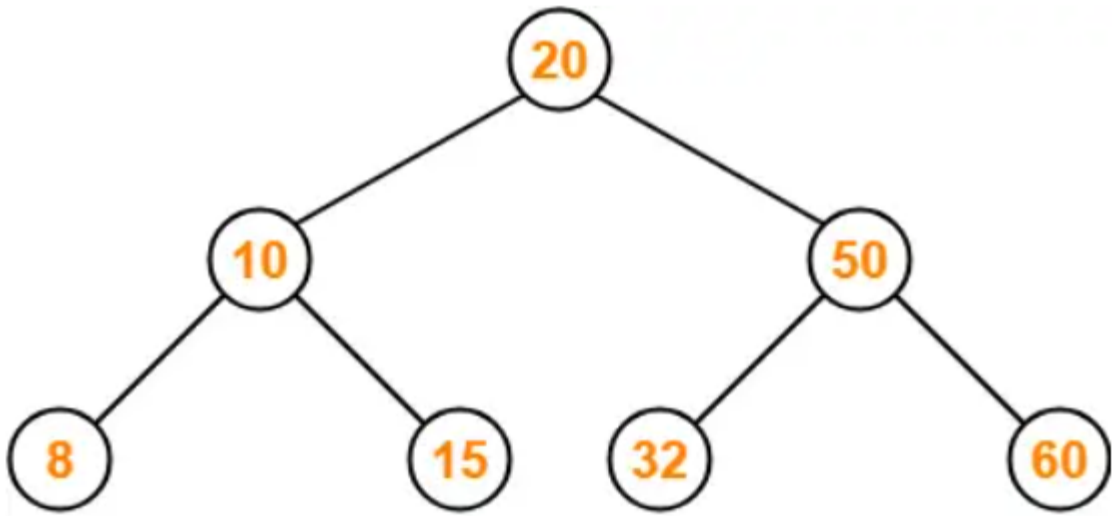


Tree is Imbalanced

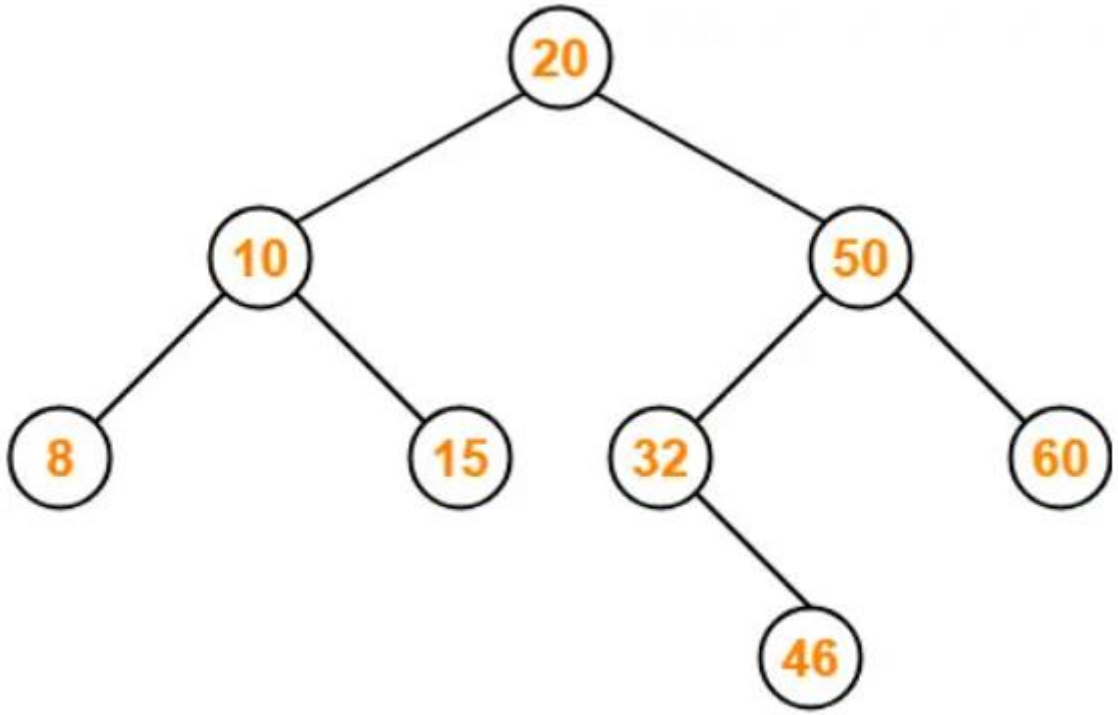
LR Rotation



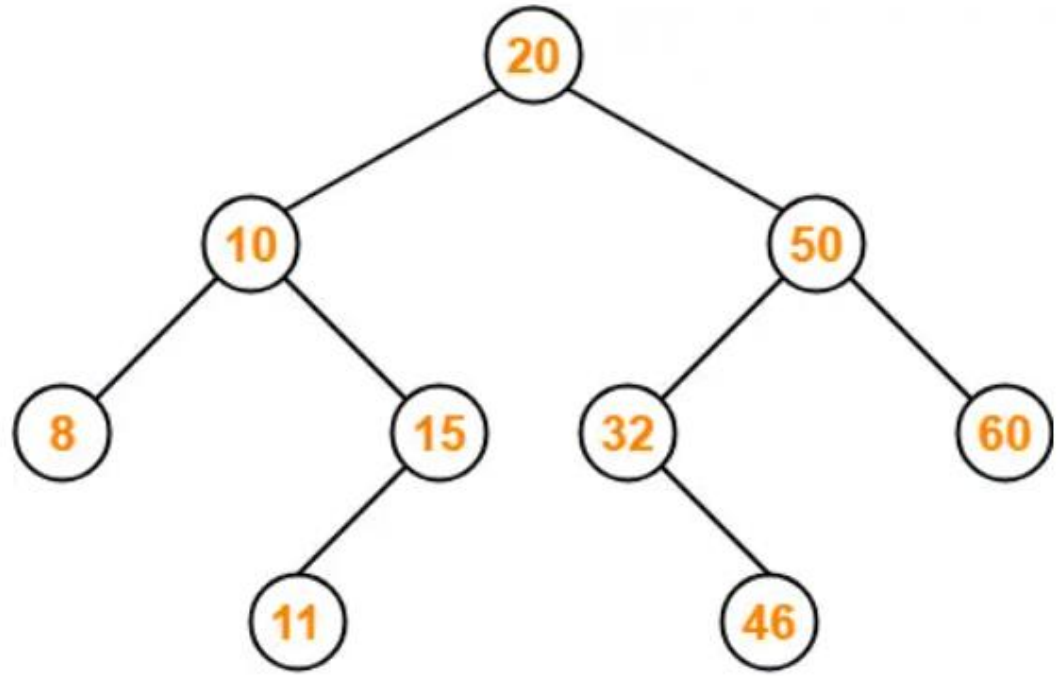
Tree is Balanced



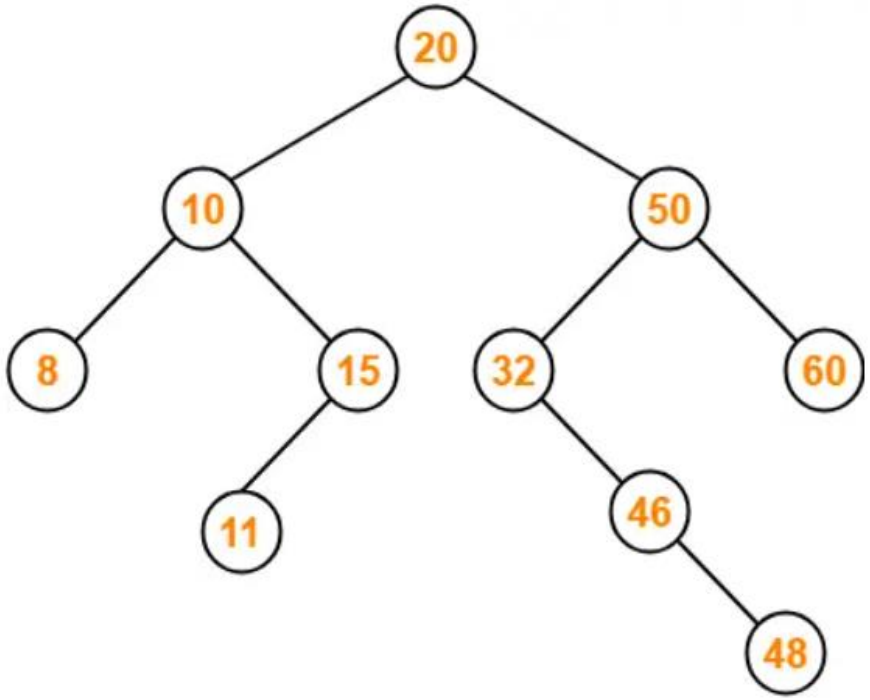
Tree is Balanced



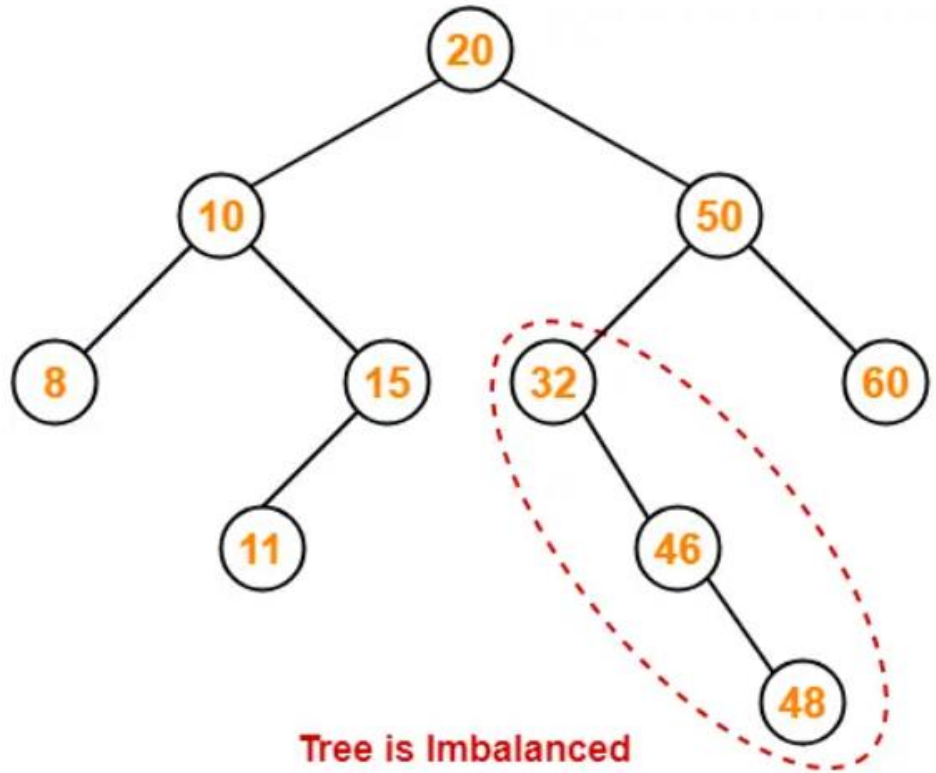
Tree is Balanced



Tree is Balanced

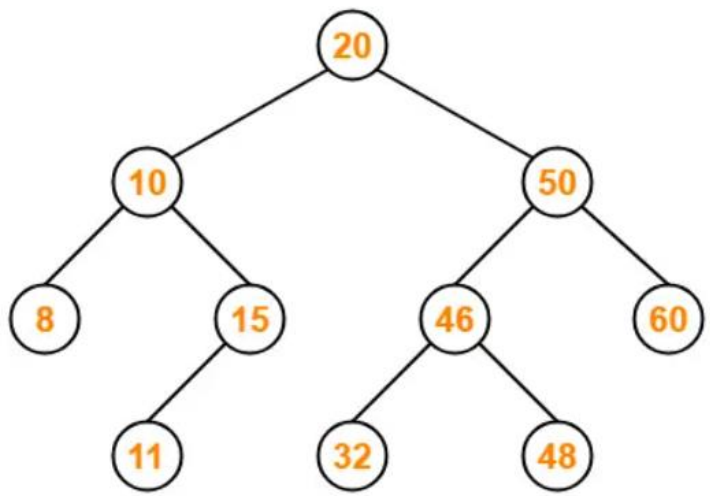


Tree is imbalanced



Tree is Imbalanced

LL Rotation

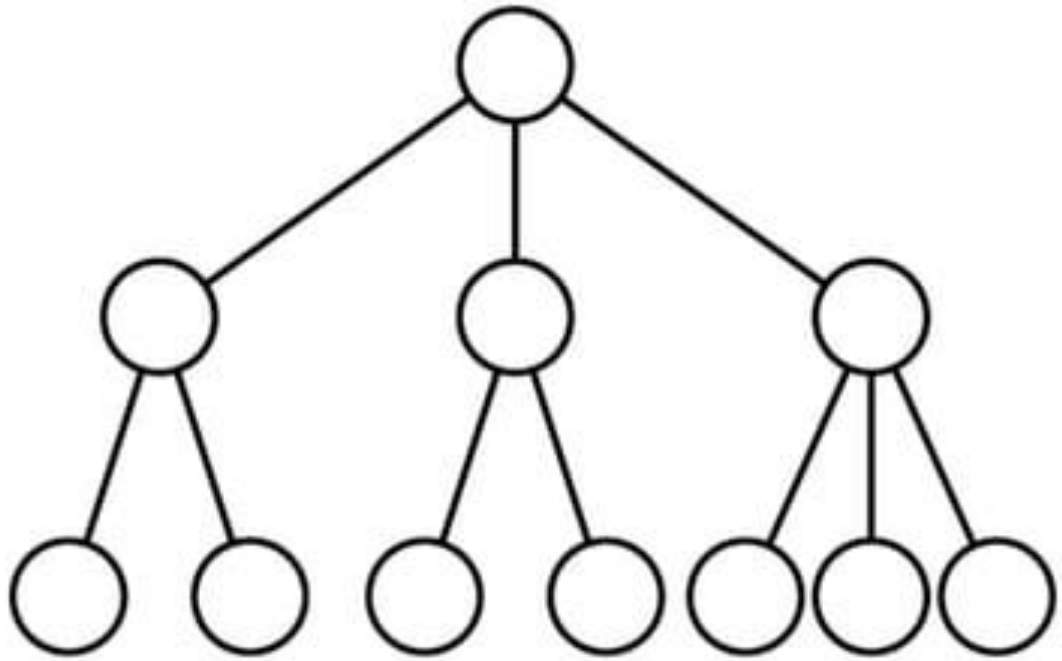


Tree is Balanced

2-3 Trees

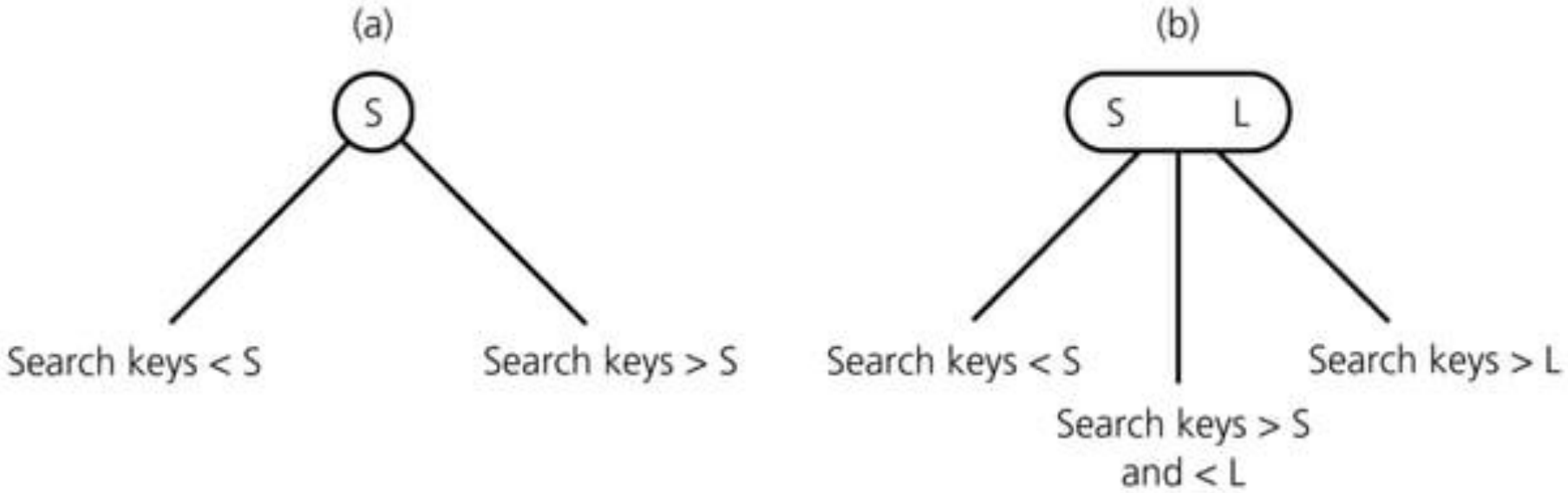
Features

- *each internal node has either 2 or 3 children*
- *all leaves are at the same level*



2-3 Trees with Ordered Nodes

2-node *3-node*



- *leaf node can be either a 2-node or a 3-node*

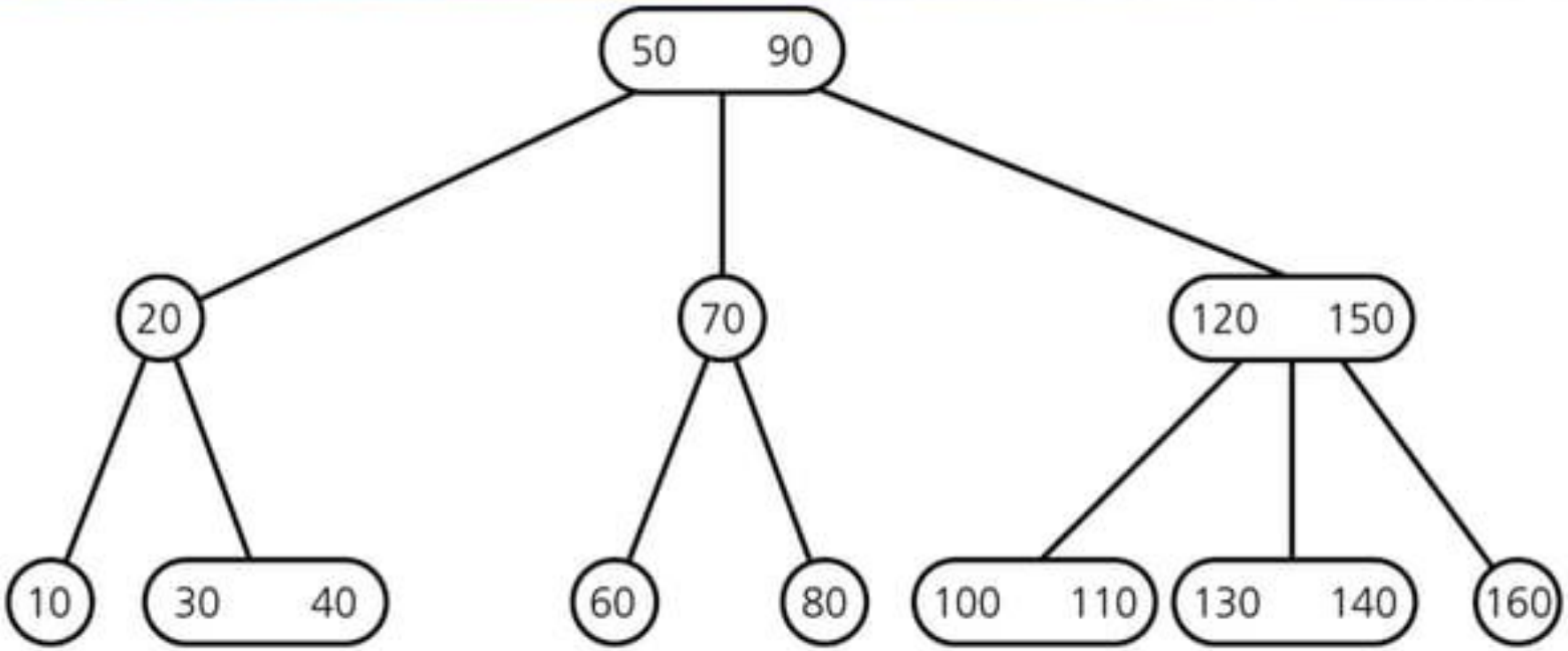
Why 2-3 tree

- Faster searching?
  - Actually, no. 2-3 tree is about as fast as an "equally balanced" binary tree, because you sometimes have to make 2 comparisons to get past a 3-node
- Easier to keep balanced?
  - Yes, definitely.
  - Insertion can split 3-nodes into 2-nodes, or promote 2-nodes to 3-nodes to keep tree approximately balanced!

Why is this better?

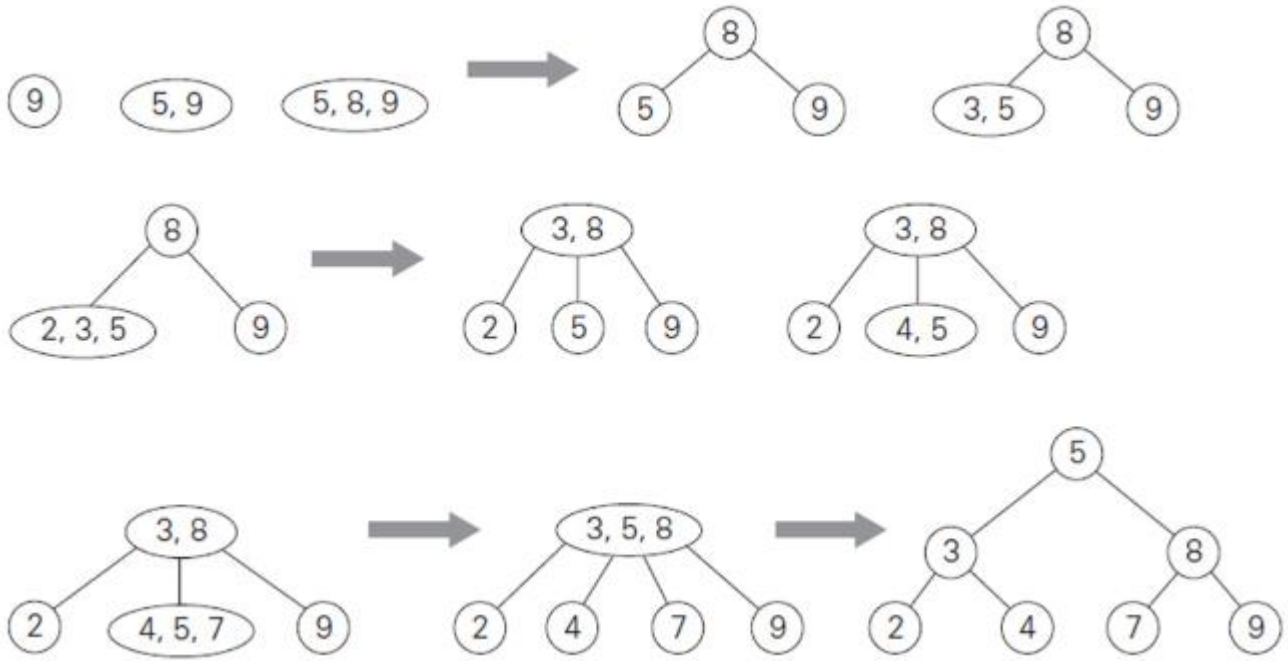
- Intuitively, you unbalance a binary tree when you add height to one path significantly more than other possible paths.
- With the 2-3 insert algorithm, you can only add height to the tree when you create a new root, and this adds one unit of height to all paths simultaneously.
- Hence, the average path length of the tree stays close to log N.

Example of 2-3 Tree



INSERT 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100

INSERT **9, 5, 8, 3, 2, 4, 7**



INSERT 50, 30, 10, 70, 60

Red-black tree

The red-Black tree is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc. In a binary search tree, the searching, insertion and deletion take **O(log2n)** time in the average case, **O(1)** in the best case and **O(n)** in the worst case.

The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees O(log2n) time for all operations like insertion, deletion, and searching.

Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.

As the name suggests that the node is either colored in **Red** or **Black** color. Sometimes no rotation is required, and only recoloring is needed to balance the tree.

Properties:

•It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
•This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
•In the Red-Black tree, the root node is always black in color.
•In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
•If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
•Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Insertion in Red Black tree

The following are some rules used to create the Red-Black tree:

1.If the tree is empty, then we create a new node as a root node with the color black.

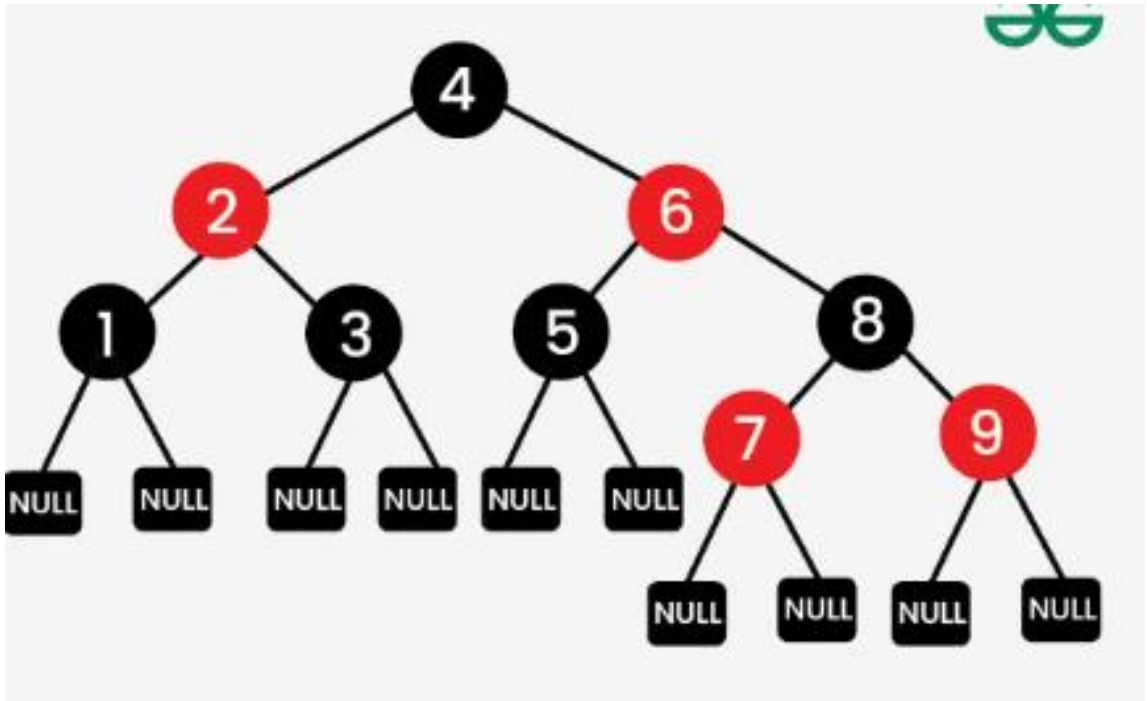
2.If the tree is not empty, then we create a new node as a leaf node with a color red.

3.If the parent of a new node is black, then exit.

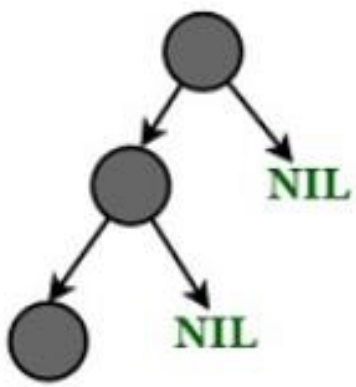
4.If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

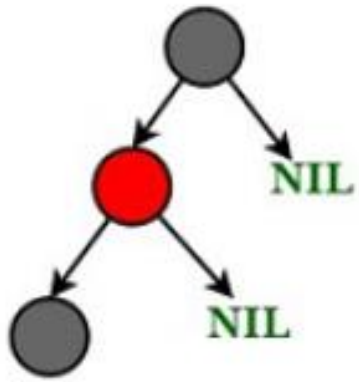
4a) If the color is Black, then we perform rotations and recoloring.

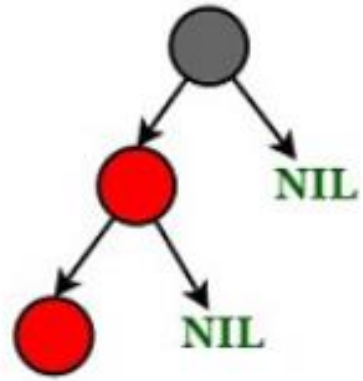
4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.


4
2
6
1
3
5
8
7
9
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

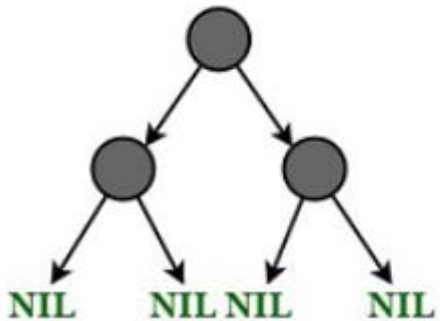
Following are NOT possible 3-noded Red-Black Trees

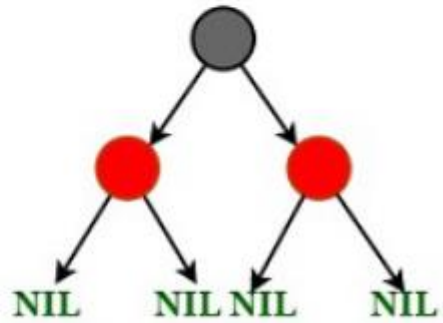


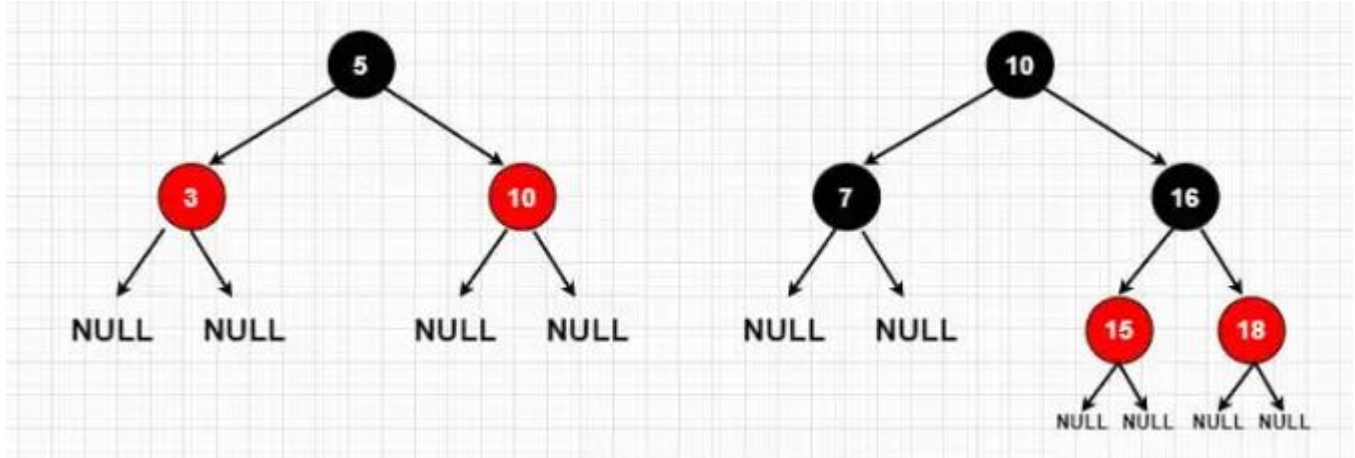


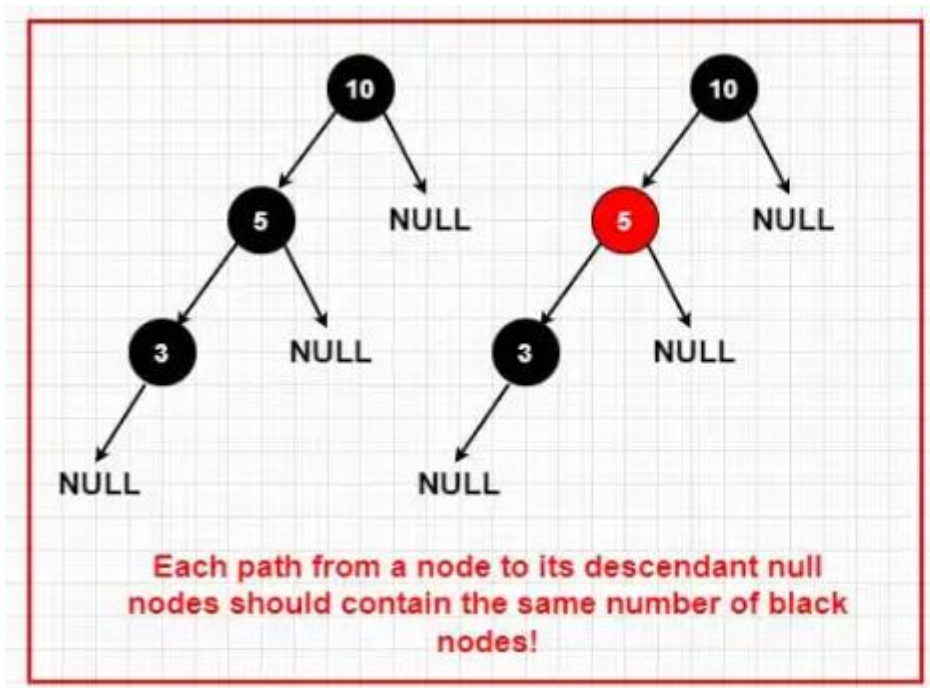


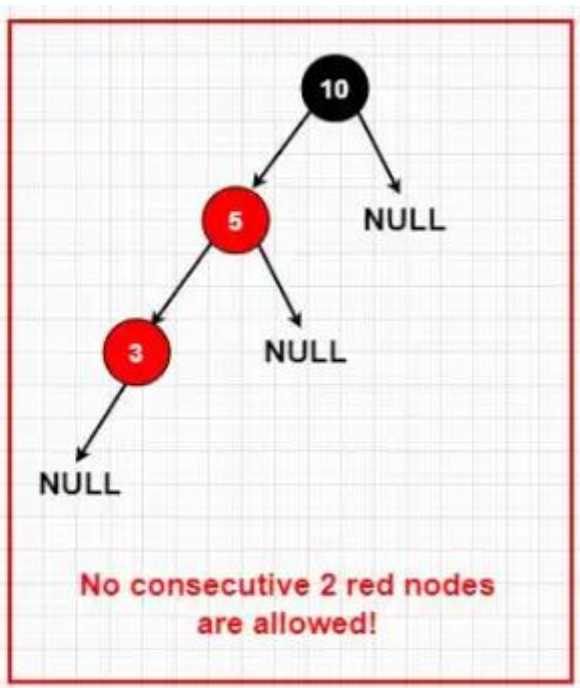
Following are possible Red-Black Trees with 3 nodes






5
3
10
NULL
NULL
NULL
NULL
10
7
16
NULL
NULL
15
18
NULL
NULL
NULL
NULL


10
5
NULL
3
NULL
NULL
10
5
NULL
3
NULL
NULL
Each path from a node to its descendant null nodes should contain the same number of black nodes!


10
5
NULL
3
NULL
NULL
No consecutive 2 red nodes are allowed!

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60

1) If tree is empty, create newnode as root node with colour Black.
2) If tree is not empty, create newnode as leaf node with colour Red.
3) If parent of newnode is Black then exit.
4) If parent of newnode is Red, then check the colour of parent's sibling of newnode:
    a) If colour is Black or NULL then do suitable rotation among node, parent and grandparent and then recolour
    b) If colour is Red then recolour the parent and parent's sibling and also check if grandparent of newnode is not root node then recolour if and recheck from the grandparent.

rules:
-> root color = Black
-> no two adjacent Red nodes
-> count of Black nodes in each path is equal

Point to remember: If rotation is LL or RR, then GP and P node will be recolored, if rotation is RL or LR then GP and Child node is recolored. GP means grand Parent, P means Parent.