		L	T	P	C
II Year I Semester	**Advanced Data Structures & Algorithm Analysis**	**3**	**0**	**0**	**3**

Course Objectives:

The main objectives of the course is to

- provide knowledge on advance data structures frequently used in Computer Science domain
- Develop skills in algorithm design techniques popularly used
- Understand the use of various data structures in the algorithm design

UNIT – I:

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.
AVL Trees – Creation, Insertion, Deletion operations and Applications
B-Trees – Creation, Insertion, Deletion operations and Applications

UNIT – II:

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications
Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications
Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

UNIT – III:

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths
Dynamic Programming: General Method, All pairs shortest paths, Single Source Shortest Paths – General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem

UNIT – IV:

Backtracking: General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem
Branch and Bound: The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem

UNIT – V:

NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem
NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)
NP Hard Scheduling Problems: Scheduling Identical Processors, Job Shop Scheduling

Textbooks:

1. Fundamentals of Data Structures in C++, Horowitz, Ellis; Sahni, Sartaj; Mehta, Dinesh 2nd Edition Universities Press
2. Computer Algorithms/C++ Ellis Horowitz, SartajSahni, SanguthevarRajasekaran2nd Edition University Press

Reference Books:

1. Data Structures and program design in C, Robert Kruse, Pearson Education Asia
2. An introduction to Data Structures with applications, Trembley & Sorenson, McGraw Hill
3. The Art of Computer Programming, Vol.1: Fundamental Algorithms, Donald E Knuth, Addison-Wesley, 1997.
4. Data Structures using C & C++: Langsam, Augenstein&Tanenbaum, Pearson, 1995
5. Algorithms + Data Structures & Programs:, N.Wirth, PHI
6. Fundamentals of Data Structures in C++: Horowitz Sahni& Mehta, Galgottia Pub.
7. Data structures in Java:, Thomas Standish, Pearson Education Asia

Online Learning Resources:

1. https://www.tutorialspoint.com/advanced_data_structures/index.asp
2. http://peterindia.net/Algorithms.html
3. Abdul Bari,1. Introduction to Algorithms (youtube.com)

UNIT – I:

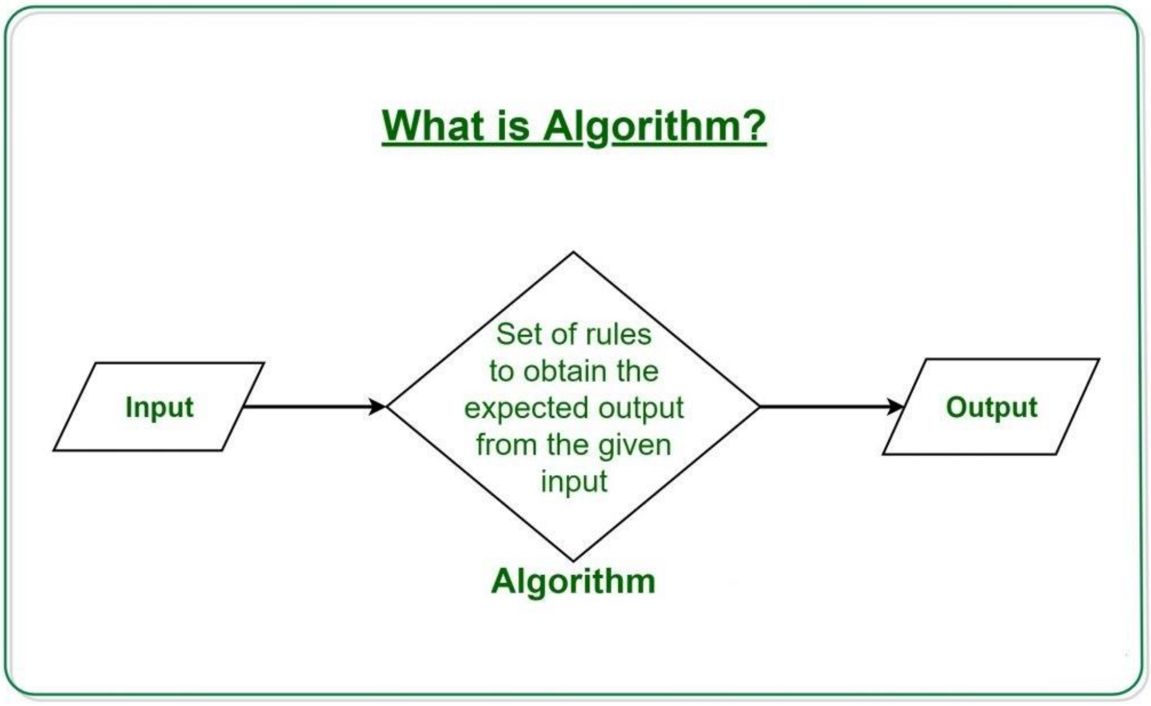
Introduction to Algorithm Analysis

Algorithm is a step-by-step procedure for solving a problem or accomplishing a task. In the context of data structures and algorithms, it is a set of well-defined instructions for performing a specific computational task. Algorithms are fundamental to computer science and play a very important role in designing efficient solutions for various problems.

The word ALGORITHM means "A set of finite rules or instructions to be followed in calculations or other problem solving operations.

A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operation.

Algorithms can be simple and complex based on what we want to achive.

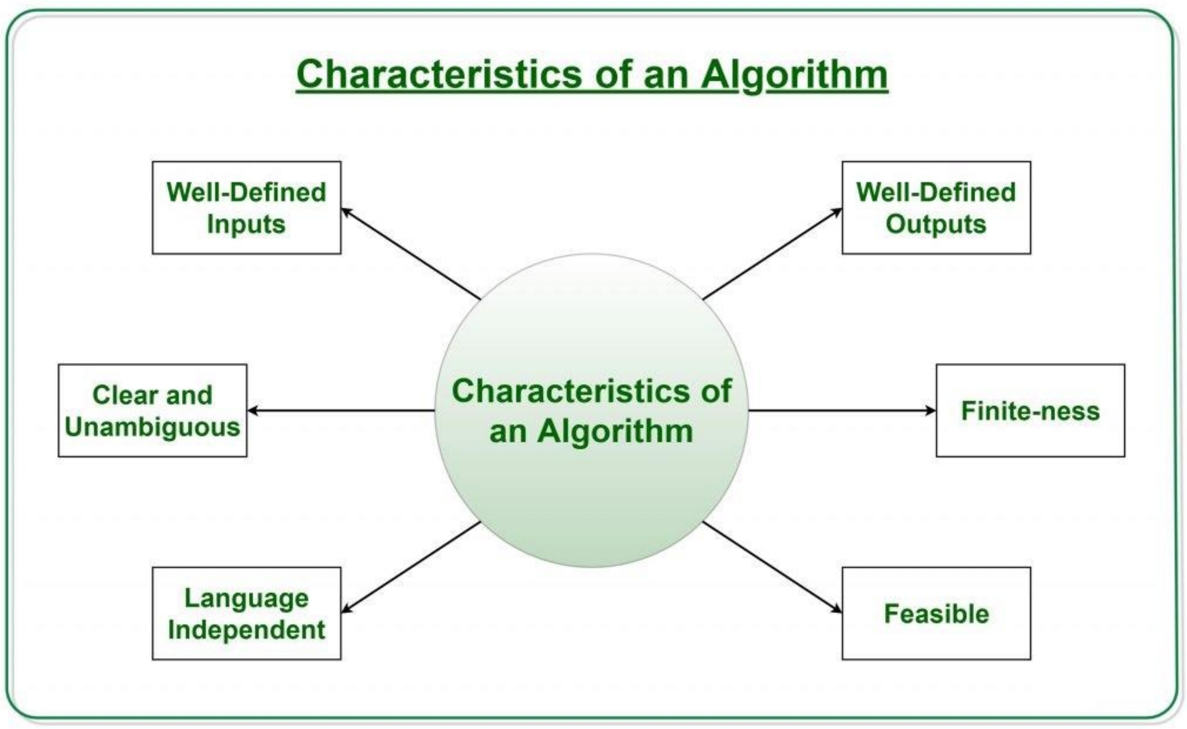

What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Algorithm plays a crucial role in various fields and has many applications Some of the key areas where algorithms are used include:

- COMPUTER SCIENCE
- MATHEMATICS
- OPERATIONS RESEARCH
- ARTIFICIAL INTELLIGENCE
- DATA SCIENCE

<u>What is the need for algorithms?</u>

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
4. They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.



- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
- **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an

algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:
- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Algorithm Analysis: Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?
- To predict the behaviour of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behaviour of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.
  And then we divide it by the total number of inputs.

1. Time Complexity

The **time complexity** of an algorithm is defined as the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a **function of the length of the input** and not the actual execution time of the machine on which the algorithm is running on

How is Time complexity computed?

To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

- If we have statements with basic operations like **comparisons, return statements, assignments, and reading a variable.** We can assume they take **constant time each O(1).**

```
Statement 1: int a=5;            // reading a variable
statement 2; if( a==5) return true;  // return statement
statement 3; int x= 4>5 ? 1:0;     // comparison
statement 4; bool flag=true;      // Assignment
```

This is the result of calculating the overall time complexity.

```
total time = time(statement1) + time(statement2) + ... time (statementN)
```

Overall, T(n)= O(1), which means constant complexity.

N*M number of times.

T(N)= n * m *(t(cout statement))
   = n * m * O(1)
   =O(n*m), **Quadratic Complexity.**

2. Space Complexity :

The amount of memory required by the algorithm to solve a given problem is called the space complexity of the algorithm. Problem-solving using a computer requires memory to hold temporary data or final result while the program is in execution.

How is space complexity computed?

The space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will **require O(n) space**. If we create a two-dimensional array of size **n*n**, this will require **O(n2) space.**

Asymptotic Notations:

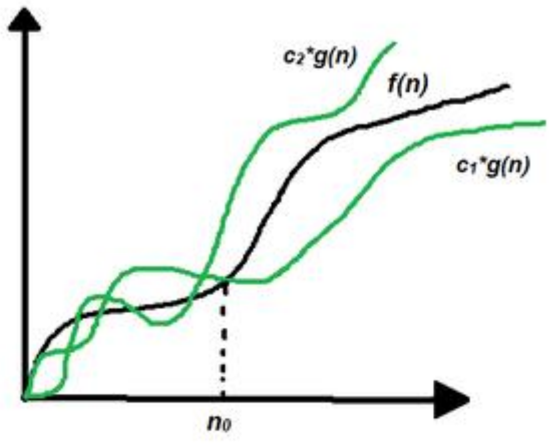
- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- These notations provide a concise way to express the behavior of an algorithm's time or space complexity as the input size approaches infinity.
- Rather than comparing algorithms directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.
- It enables comparisons of algorithms' efficiency by abstracting away machine-specific constants and implementation details, focusing instead on fundamental trends.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

There are mainly three asymptotic notations:

1. **Big-O Notation (O-notation)**
2. **Omega Notation (Ω-notation)**
3. **Theta Notation (Θ-notation)**

1. Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.
.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.
Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation

Mathematical Representation of Theta notation:
Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0}
Note: Θ(g) is a set
The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.
The execution time serves as both a lower and upper bound on the algorithm's time complexity.
It exist as both, most, and least boundaries for a given input value.
A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example**,** Consider the expression **3n3 + 6n2 + 6000 = Θ(n3)**, the dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.
Examples :

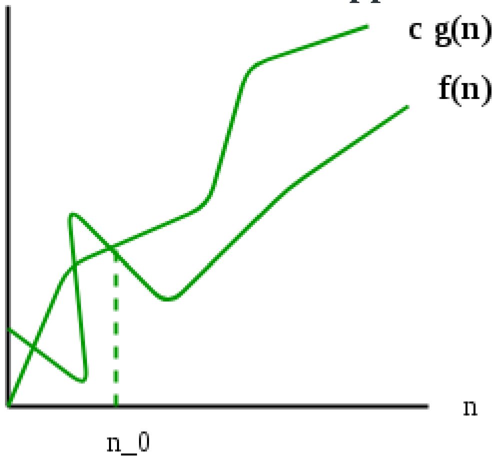
{ 100 , log (2000) , 10^4 } belongs to **Θ(1)**
{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Θ(n)**
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Θ(n2)**
Note: Θ provides exact bounds.

2. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.
.It is the most widely used notation for Asymptotic analysis.
.It specifies the upper bound of a function.
.The maximum time required by an algorithm or the worst-case time complexity.
.It returns the highest possible output value(big-O) for a given input.
.Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \le f(n) \le cg(n)$ for all $n \ge n0$

It returns the highest possible output value (big-O)for a given input.
The execution time serves as an upper bound on the algorithm's time



complexity. f(n) = O(g(n))

Mathematical Representation of Big-O Notation:

O(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n0$ }

For example**,** Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is O(n2).
Note: O(n2) also covers linear time.

If we use Θ notation to represent the time complexity of Insertion sort, we have to use two statements for best and worst cases:

- The worst-case time complexity of Insertion Sort is Θ(n2).
- The best case time complexity of Insertion Sort is Θ(n).

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

Examples :

{ 100 , log (2000) , 10^4 } belongs to **O(1)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **O(n)**
U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **O(n^2)**

Note: Here, **U represents union**, we can write it in these manner because **O provides exact or upper bounds .**

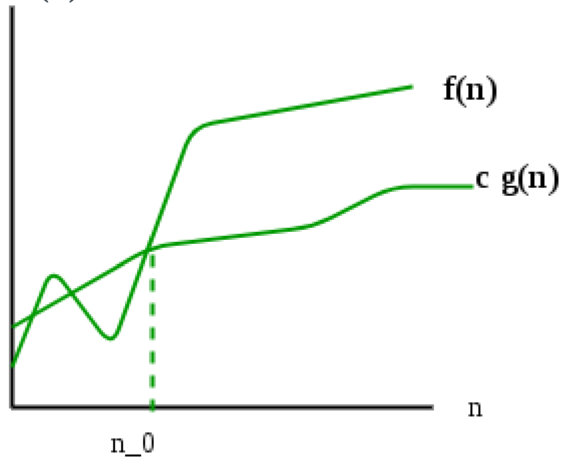
3. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c*g(n) ≤ f(n) for all n ≥ n0



f(n) = Omega(g(n))

Mathematical Representation of Omega notation :

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

Examples :

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Ω(n^2)**

U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Ω(n)**

U { 100 , log (2000) , 10^4 } belongs to **Ω(1)**

Note: Here, **U represents union,** we can write it in these manner because **Ω provides exact or lower bounds.**

AVL TREES

The first type of self-balancing binary search tree to be invented is the AVL tree. The name AVL tree is coined after its inventor's names − Adelson-Velsky and Landis.

In AVL trees, the difference between the heights of left and right subtrees, known as the **Balance Factor**, must be at most one. Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.

BALANCE FACTOR = HEIGHT(LEFT SUBTREE) − HEIGHT(RIGHT SUBTREE)

There are usually four cases of rotation in the balancing algorithm of AVL trees: LL, RR, LR, RL.

<u>LL Rotations</u>

LL rotation is performed when the node is inserted into the right subtree leading to an unbalanced tree. This is a single left rotation to make the tree balanced again −

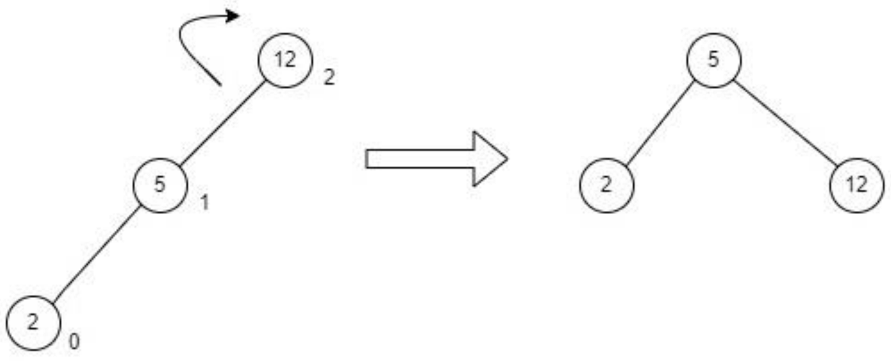


Fig : LL Rotation

The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node.

RR Rotations

RR rotation is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again −

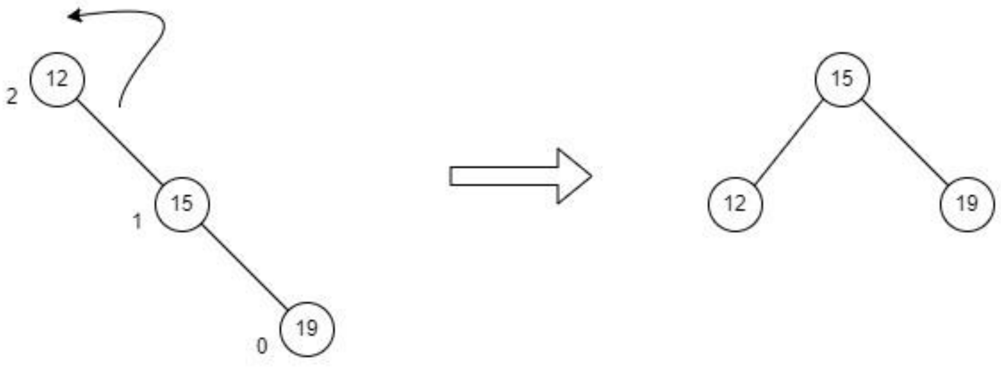


Fig : RR Rotation

The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node.

LR Rotations

LR rotation is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. There are multiple steps to be followed to carry this out.

- Consider an example with "A" as the root node, "B" as the left child of "A" and "C" as the right child of "B".
- Since the unbalance occurs at A, a left rotation is applied on the child nodes of A, i.e. B and C.
- After the rotation, the C node becomes the left child of A and B becomes the left child of C.
- The unbalance still persists, therefore a right rotation is applied at the root node A and the left child C.
- After the final right rotation, C becomes the root node, A becomes the right child and B is the left child.

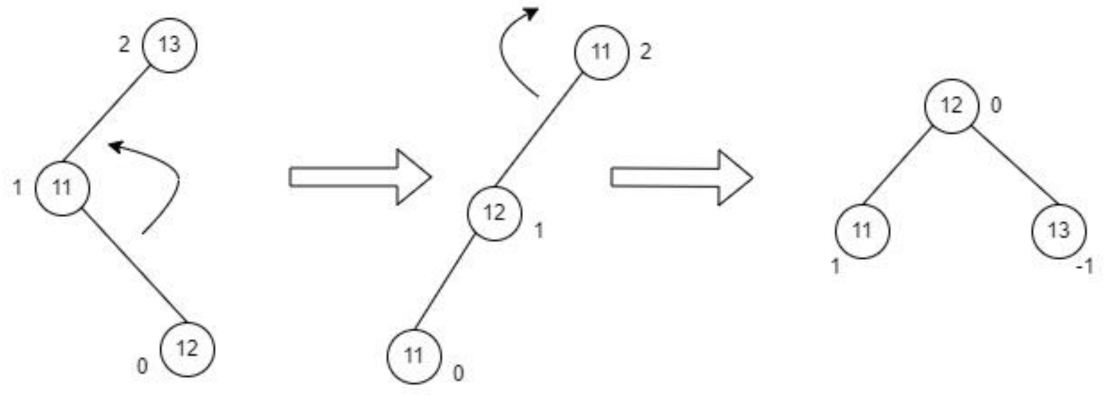


Fig : LR Rotation

RL Rotations

RL rotation is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation. There are multiple steps to be followed to carry this out.

- Consider an example with "A" as the root node, "B" as the right child of "A" and "C" as the left child of "B".
- Since the unbalance occurs at A, a right rotation is applied on the child nodes of A, i.e. B and C.
- After the rotation, the C node becomes the right child of A and B becomes the right child of C.
- The unbalance still persists, therefore a left rotation is applied at the root node A and the right child C.
- After the final left rotation, C becomes the root node, A becomes the left child and B is the right child.

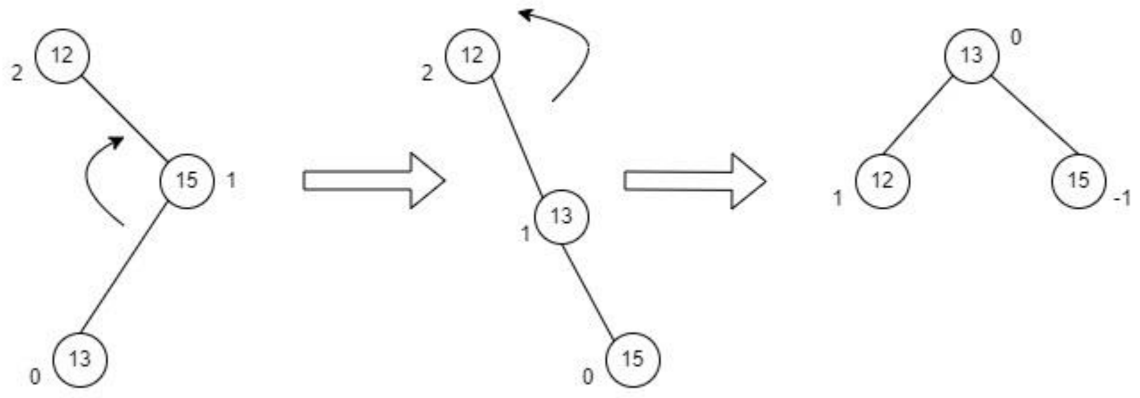


Fig : RL Rotation

Basic Operations of AVL Trees

The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are − **Insertion** and **Deletion**.

Insertion operation

The data is inserted into the AVL Tree by following the Binary Search Tree property of insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements.

However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.

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.

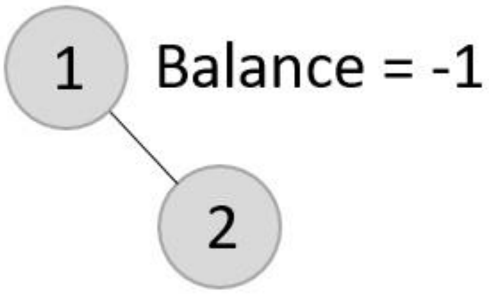
Step 4 − If the tree is not empty, we perform the Binary Search Tree
insertion operation and check the balancing factor of the node
in the tree.

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

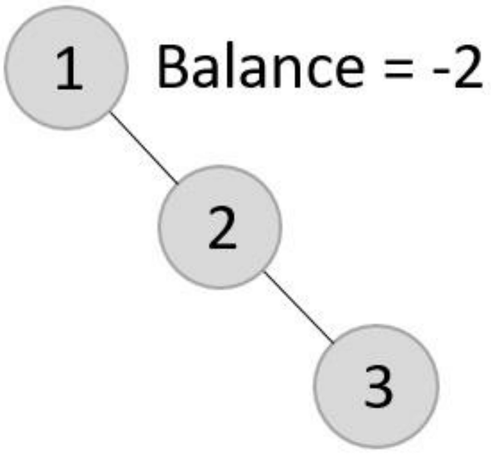
Let us understand the insertion operation by constructing an example AVL tree with 1 to 7 integers.

Starting with the first element 1, we create a node and measure the balance, i.e., 0.

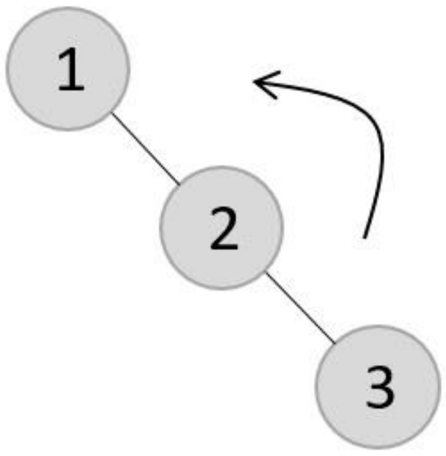
Since both the binary search property and the balance factor are satisfied, we insert another element into the tree.



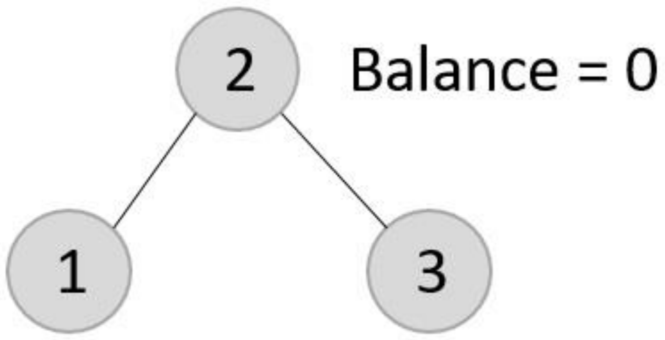
The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree.



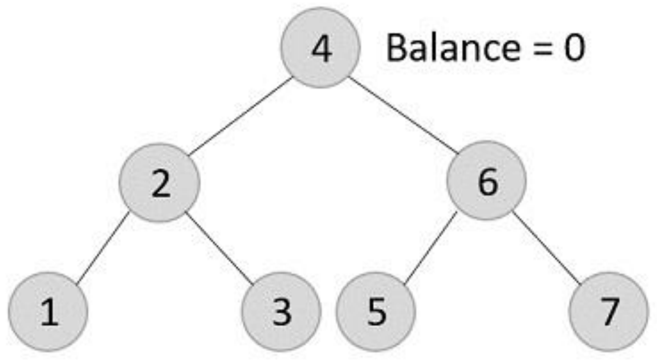
Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes.



The tree is rearranged as −



Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as −



Example

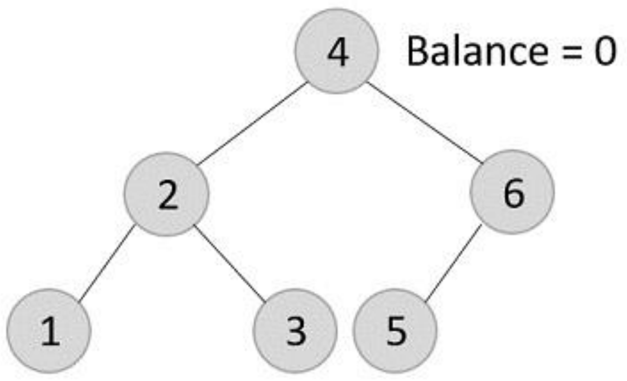
Following are the implementations of this operation in various programming languages −

Deletion operation

Deletion in the AVL Trees take place in three different scenarios −

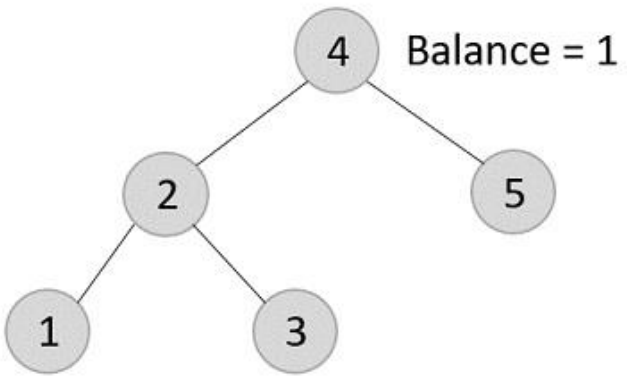
- **Scenario 1 (Deletion of a leaf node)** − If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.
- **Scenario 2 (Deletion of a node with one child)** − If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied.
- **Scenario 3 (Deletion of a node with two child nodes)** − If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor value. Then try to delete the inorder successor node. If the balance factor exceeds 1 after deletion, apply balance algorithms.

Using the same tree given above, let us perform deletion in three scenarios −



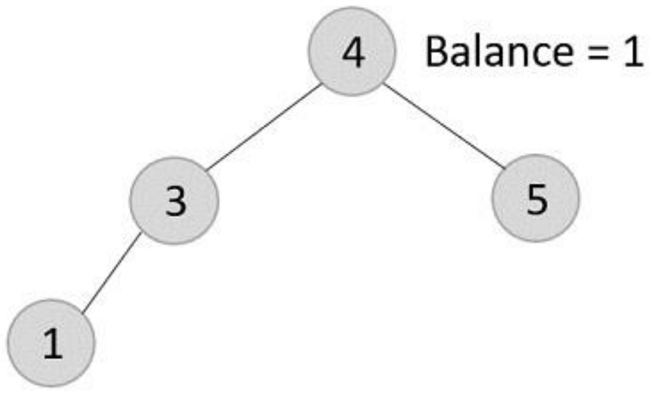
- Deleting element 7 from the tree above −

Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree

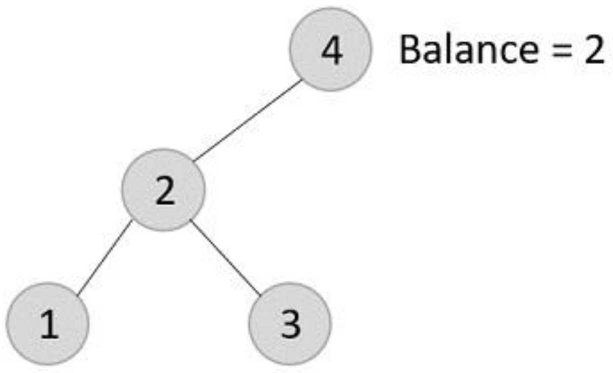


- Deleting element 6 from the output tree achieved −

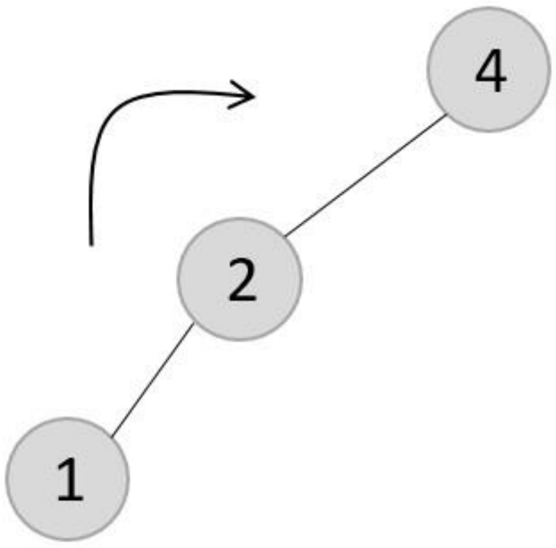
However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.



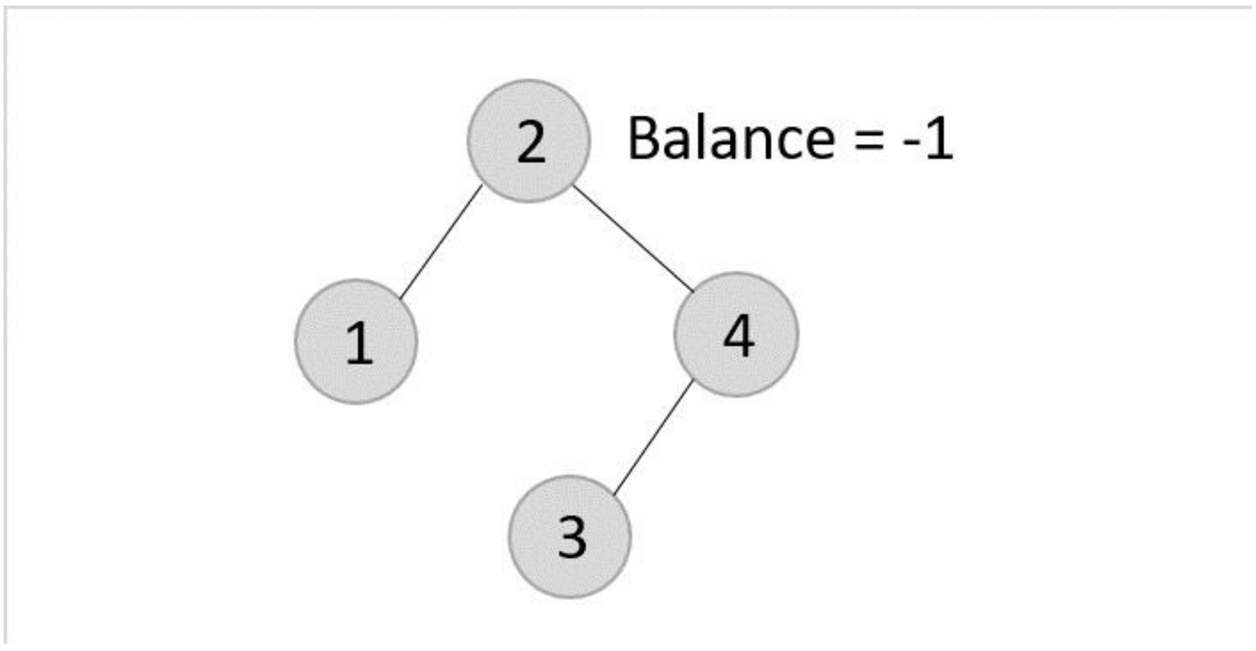
The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete the element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4.



The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here).

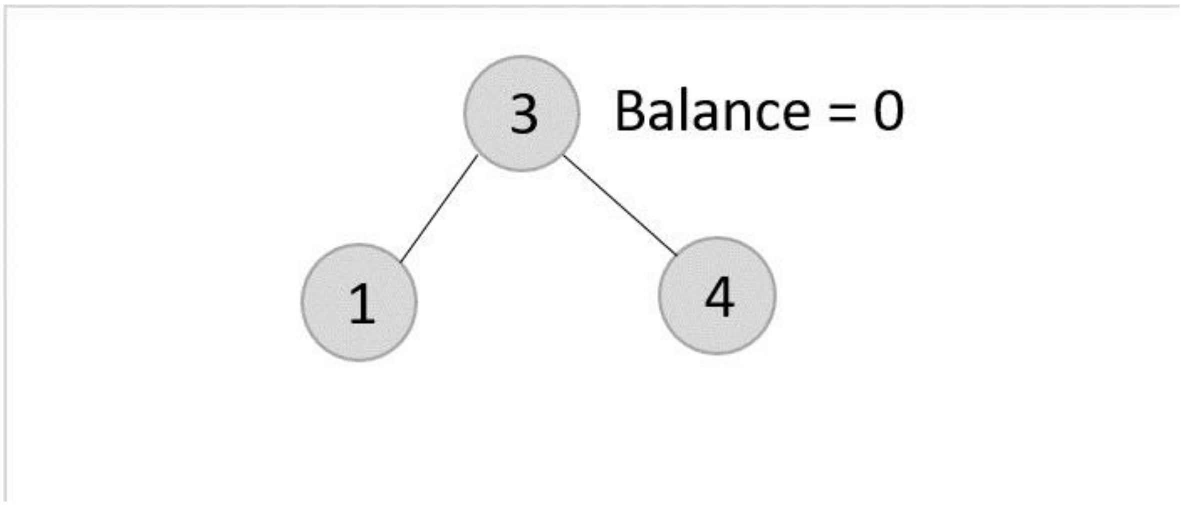


Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.



- Deleting element 2 from the remaining tree −

As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.



The balance of the tree still remains 1, therefore we leave the tree as it is without performing any rotations.

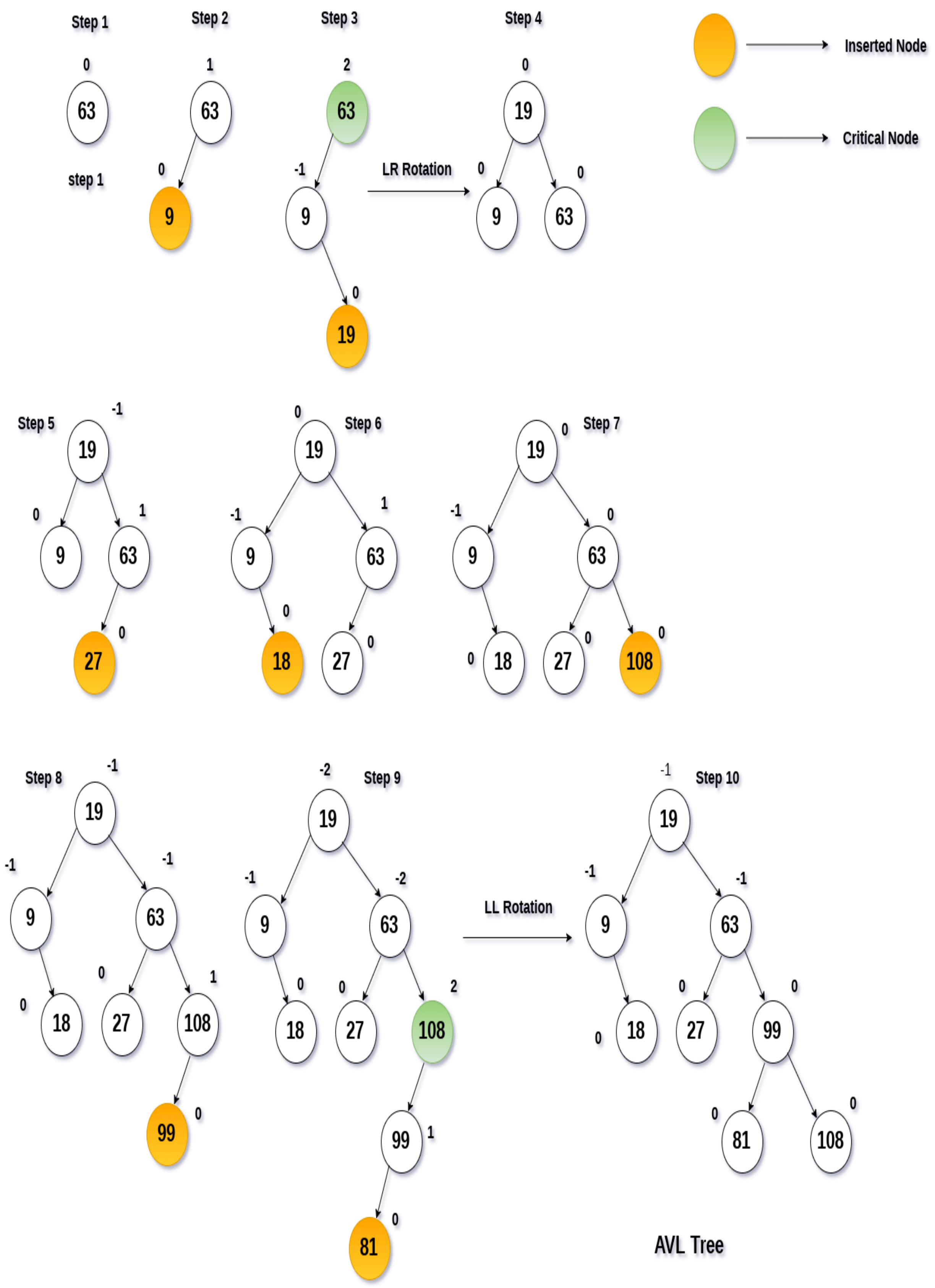
Example

Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of binary search tree.


Step 1
step 1
Step 2
Step 3
LR Rotation
Step 4
Inserted Node
Critical Node
Step 5
Step 6
Step 7
Step 8
Step 9
LL Rotation
Step 10
AVL Tree

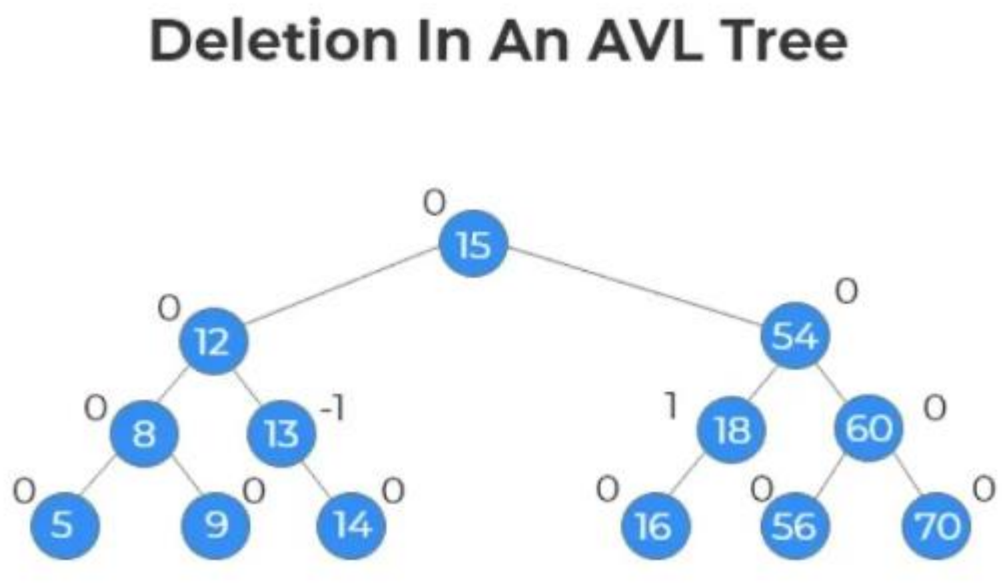
Deletion in an AVL Tree

- Deletion in an AVL tree is similar to that in a BST.
- Deletion of a node tends to disturb the balance factor. Thus to balance the tree, we again use the Rotation mechanism.
- Deletion in AVL tree consists of two steps:
  - **Removal of the node:** The given node is removed from the tree structure. The node to be removed can either be a leaf or an internal node.
  - **Re-balancing of the tree:** The elimination of a node from the tree can cause disturbance to the balance factor of certain nodes. Thus it is important to re- balance theb_fact of the nodes; since the balance factor is the primary aspect that ensures the tree is an AVL Tree.

Note: There are certain points that must be kept in mind during a deletion process.

- If the node to be deleted is a leaf node, it is simply removed from the tree.
- If the node to be deleted has one child node, the child node is replaced with the node to be deleted simply.
- If the node to be deleted has two child nodes then,
  - Either replace the node with it's **inorder predecessor** , i.e, **the largest element of the left sub tree.**
  - Or replace the node with it's **inorder successor** , i.e, **the smallest element of the right sub tree.**

Let us consider the same example as above with some additional elements as shown. We can see in the image the balance factor of each node in the tree.

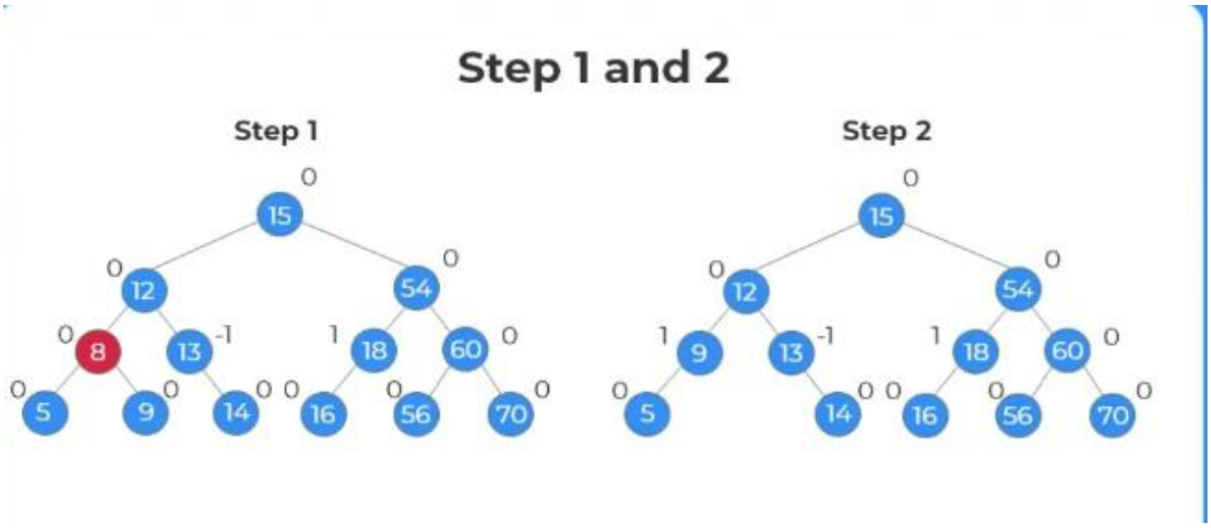


Step 1:

- The node to be deleted from the tree is **8**.
- If we observe it is the parent node of the node **5** and **9.**
- Since the node 8 has two children it can be replaced by either of it’s child nodes.

Step 2:

- The node **8** is deleted from the tree.
- As the node is deleted we replace it with either of it’s children nodes.
- Here we **replaced** the node with the **inorder successor** , i.e, **9.**
- Again we check the balance factor for each node.

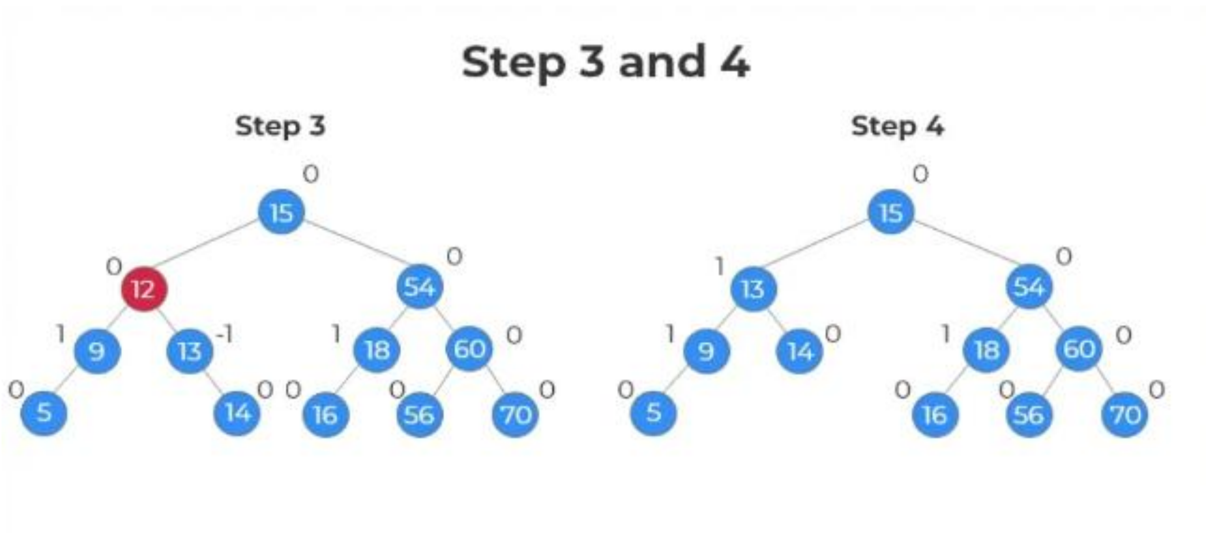


Step 3:

- Now The next element to be deleted is **12**.
- If we observe, we can see that the node 12 has a left subtree and a right subtree.
- We again can replace the node by either it's inorder successor or inorder predecessor.
- In this case we have replaced it by the inorder successor.

Step 4:

- The node **12** is deleted from the tree.
- Since we have replaced the node with the inorder successor, the tree structure looks like shown in the image.
- After removal and replacing check for the balance factor of each node of the tree.

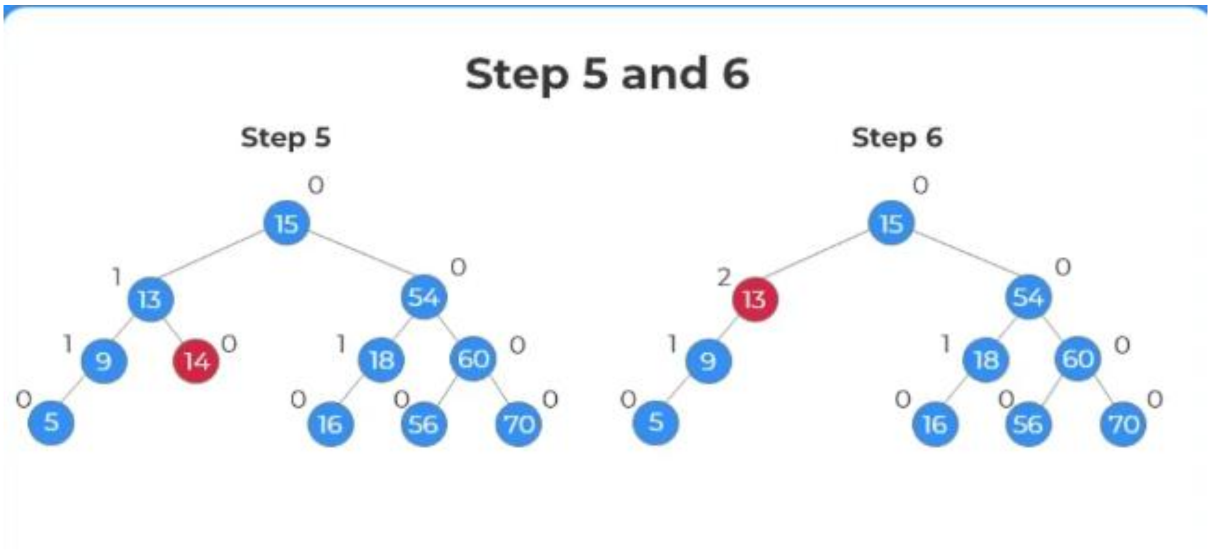


Step 5:

- The next node to be eliminated is **14**.
- It can be seen clearly in the image that 14 is a leaf node.
- Thus it can be eliminated easily from the tree.

Step 6:

- As the node **14** is deleted, we check the balance factor of all the nodes.
- We can see the balance factor of the node **13** is **2**.
- This violates the terms of the AVL tree thus we need to balance it using the rotation mechanism.

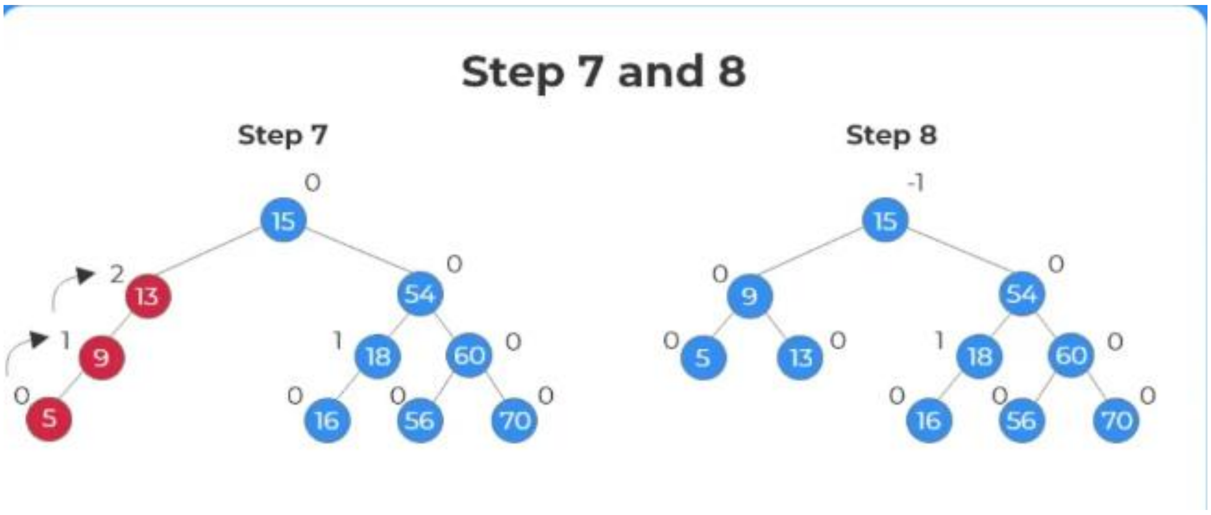


Step 7:

- In order to balance the tree, we identify the rotation mechanism to be applied.
- Here we need to use **LL Rotation.**
- The nodes involved in the rotation is shown as follows.

Step 8:

- The nodes are rotated and the tree satisfies the conditions of an AVL tree.
- The final structure of the tree is shown as follows.
- We can see all the nodes have their balance factor as **‘0’ , ‘1’** and **‘-1’.**



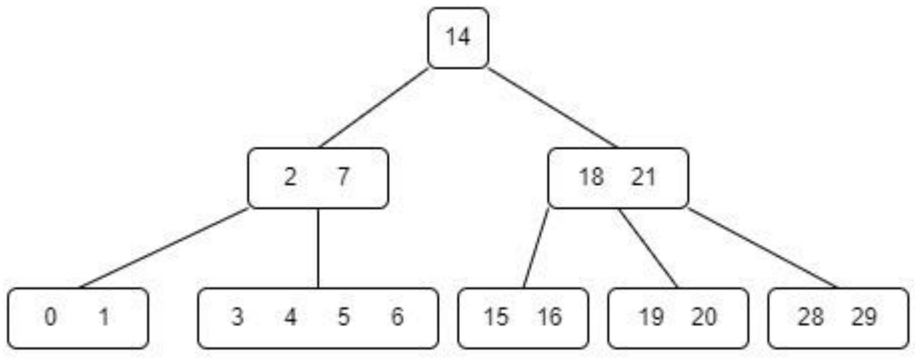
B TREES

B trees are extended binary search trees that are specialized in m-way searching, since the order of B trees is 'm'. Order of a tree is defined as the maximum number of children a node can accommodate. Therefore, the height of a b tree is relatively smaller than the height of AVL tree and RB tree.

They are general form of a Binary Search Tree as it holds more than one key and two children.

The various properties of B trees include −

- Every node in a B Tree will hold a maximum of m children and (m-1) keys, since the order of the tree is m.
- Every node in a B tree, except root and leaf, can hold at least m/2 children
- The root node must have no less than two children.
- All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
- A B tree always maintains sorted data.



B trees are also widely used in disk access, minimizing the disk access time since the height of a b tree is low.

Note − A disk access is the memory access to the computer disk where the information is stored and disk access time is the time taken by the system to access the disk memory.

Insertion Example

Let us understand the insertion operation with the illustrations below.

UNIT - 2

Heap Data Structure

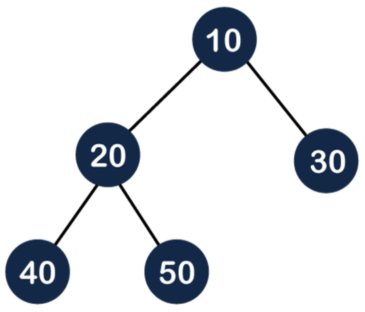
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

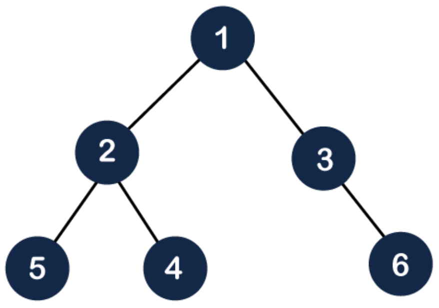
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?
There are two types of the heap:

- Min Heap
- Max heap

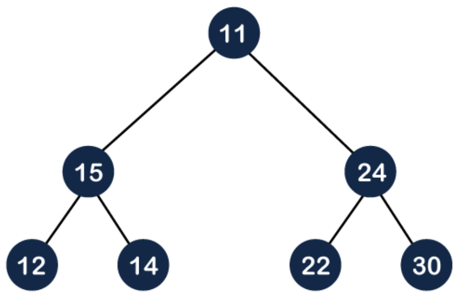
Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Let's understand the min-heap through an example.



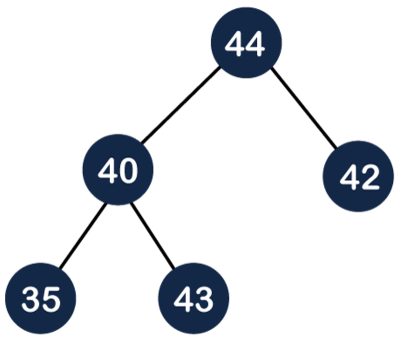
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

```
// algorithm to insert an element in the max heap.
insertHeap(A, n, value)
{
    n=n+1; // n is incremented to insert the new element
    A[n]=value; // assign new value at the nth position
    i = n; // assign the value of n to i
    // loop will be executed until i becomes 1.
    while(i>1)
    {
        parent= floor value of i/2; // Calculating the floor value of i/2
        // Condition to check whether the value of parent is less than the given node or not
        if(A[parent]<A[i])
        {
            swap(A[parent], A[i]);
            i = parent;
        }
        else
        {
            return;
        }
    }
}
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

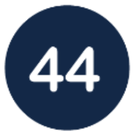
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

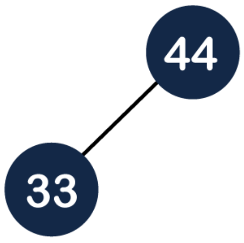
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.

- Secondly, the value of the parent node should be greater than the either of its child.

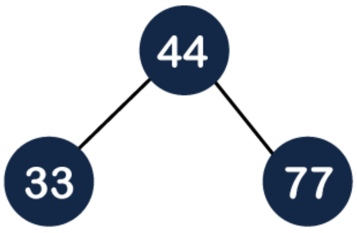
Step 1: First we add the 44 element in the tree as shown below:



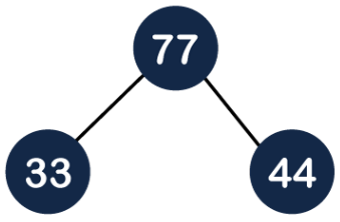
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



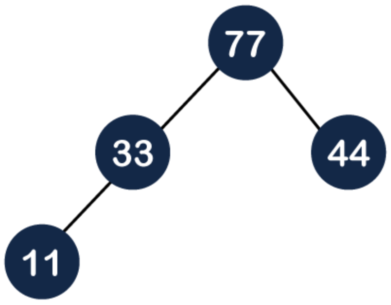
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



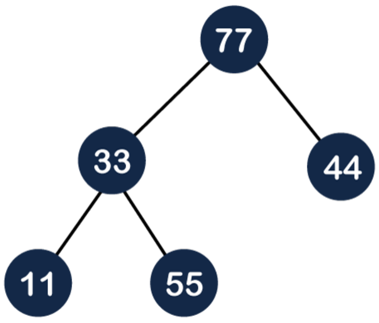
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



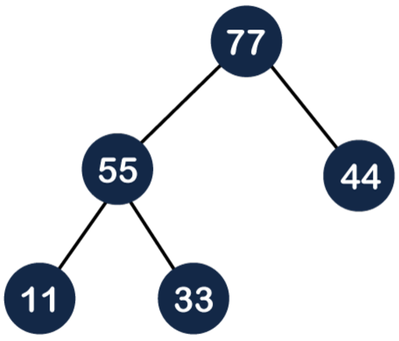
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



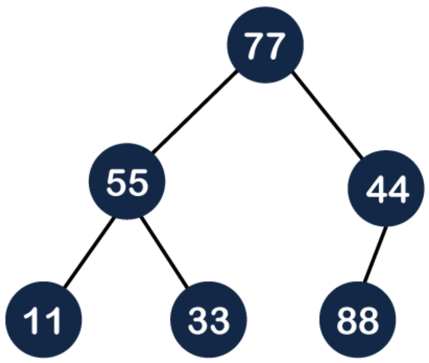
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to heapify the tree

```
1. MaxHeapify(A, n, i)
2. {
```

```
int largest =i;
int l= 2i;
int r= 2i+1;
while(l<=n && A[l]>A[largest])
{
largest=l;
}
while(r<=n && A[r]>A[largest])
{
largest=r;
}
if(largest!=i)
{
swap(A[largest], A[i]);
heapify(A, n, largest);
}}
```

Deletion in Heap:

Given a Binary Heap and an element present in the given Heap. The task is to delete an element from this Heap.

The standard deletion operation on Heap is to delete the element present at the root node of the Heap. That is if it is a Max Heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.

Process of Deletion:

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

Illustration:

```
Suppose the Heap is a Max-Heap as:
     10
    /    \
   5      3
  / \
 2   4
```
The element to be deleted is root, i.e. 10.

Process:

The last element is 4.

Step 1: Replace the last element with root, and delete it.
```
      4
    /    \
   5      3
  /
 2
```
Step 2: Heapify root.

Final Heap:
```
      5
    /    \
   4      3
  /
 2
```

Delete operation

When it comes to deleting an element from the max heap, there are two cases:

Case-01: Deletion of the leaf nodes

This case is straightforward to handle. We delete/disconnect the leaf node from the max heap in this case. After deleting the leaf node, we don’t need to change any value in the max heap.

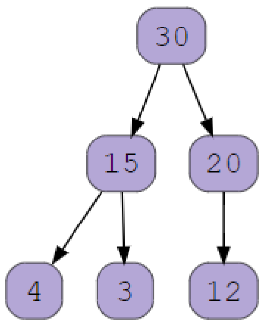
Case-02: Deletion of some other node

This case is strenuous as deleting a node other than the leaf node disturbs the heap properties. This case takes two steps for deletion:

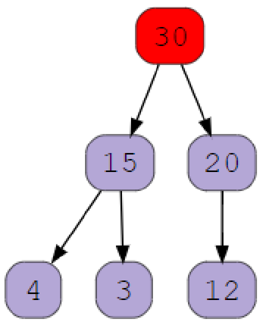
1. Delete the selected element from the given heap tree and replace the value of the last node with deleted node.
2. Check the max heap properties for the entire heap and keep calling the heapify() function until we get the max heap.

Let's visualize it from an example. In this example, we delete the root node:

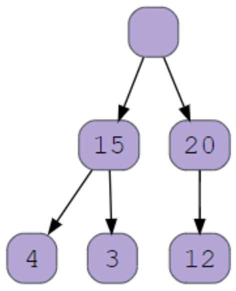
Consider this max heap



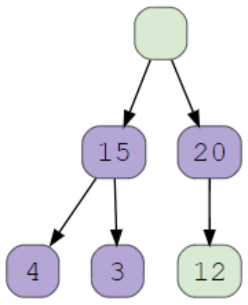
Let's delete root node

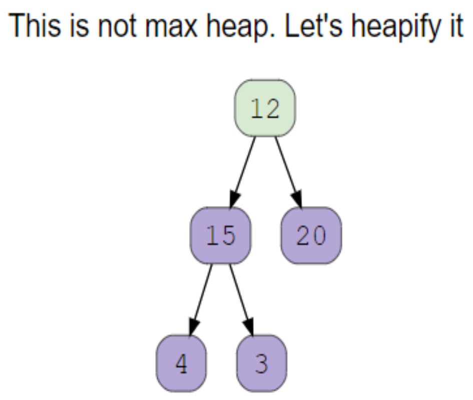


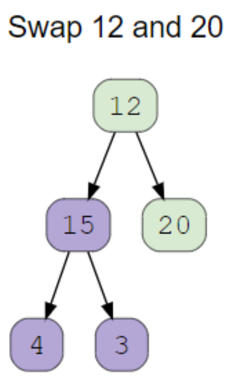
Root node has been deleted

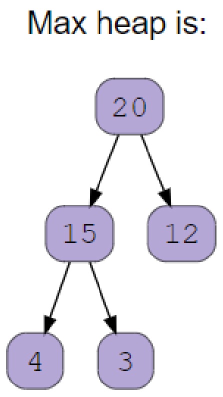


Replace last node with deleted node









Applications of Heap Data Structure

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used.

1. Priority Queues: Heaps are commonly used to implement priority queues, where elements with higher priority are extracted first. This is useful in many applications such as scheduling tasks, handling interruptions, and processing events.

2. Sorting Algorithms: Heapsort, a comparison-based sorting algorithm, is implemented using the Heap data structure. It has a time complexity of O(n log n), making it efficient for large datasets.
3. Graph algorithms: Heaps are used in graph algorithms such as Prim's Algorithm, Dijkstra's algorithm., and the A* search algorithm.
4. Lossless File Compression: Heaps are used in data compression algorithms such as Huffman coding, which uses a priority queue implemented as a min-heap to build a Huffman tree.
5. Medical Applications: In medical applications, heaps are used to store and manage patient information based on priority, such as vital signs, treatments, and test results.
6. Load balancing: Heaps are used in load balancing algorithms to distribute tasks or requests to servers, by processing elements with the lowest load first.
7. Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of this post for details.
8. Resource allocation: Heaps can be used to efficiently allocate resources in a system, such as memory blocks or CPU time, by assigning a priority to each resource and processing requests in order of priority.

Basic Graph Terminology:

1. Graph

A Graph **G** is a non-empty set of vertices (or nodes) **V** and a set of edges **E**, where each edge connects a pair of vertices. Formally, a graph can be represented as **G= (V, E)**. Graphs can be classified based on various properties, such as directedness of edges and connectivity.

2. Vertex (Node)

A Vertex, often referred to as a **Node**, is a fundamental unit of a graph. It represents an **entity** within the graph. In applications like social networks, vertices can represent individuals, while in road networks, they can represent intersections or locations.

3. Edge

An Edge is a **connection between two vertices** in a graph. It can be either directed or undirected. In a directed graph, edges have a specific direction, indicating a one-way connection between vertices. In contrast, undirected graphs have edges that do not have a direction and represent bidirectional connections.

4. Degree of a Vertex

The Degree of a Vertex in a graph is the **number of edges incident** to that vertex. In a directed graph, the degree is further categorized into the in-degree (number of incoming edges) and out-degree (number of outgoing edges) of the vertex.

5. Path

A Path in a graph is a **sequence of vertices** where each adjacent pair is connected by an edge. Paths can be of varying lengths and may or may not visit the same vertex more than once. The shortest path between two vertices is of particular interest in algorithms such as Dijkstra's algorithm for finding the shortest path in weighted graphs.

6. Cycle

A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same). Cycles are essential in understanding the connectivity and structure of a graph and play a significant role in cycle detection algorithms.

Advanced Graph Terminology:

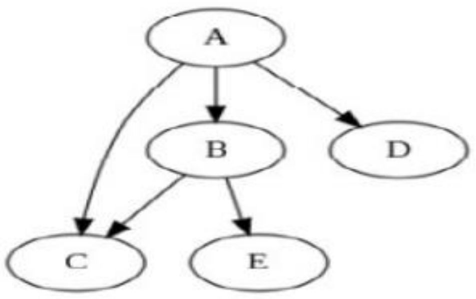
1. Directed Graph (Digraph):

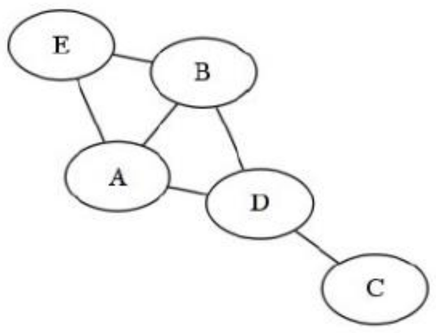
A Directed Graph consists of nodes (vertices) connected by directed edges (arcs). Each edge has a specific direction, meaning it goes from one node to another. Directed Graph is a network where information flows in a specific order. Examples include social media follower relationships, web page links, and transportation routes with one-way streets.

2. Undirected Graph:

In an Undirected Graph, edges have no direction. They simply connect nodes without any inherent order. For example, a social network where friendships exist between people, or a map of cities connected by roads (where traffic can flow in both directions).

a. *Directed/Undirected:* In a directed graph the direction of the edges must be considered





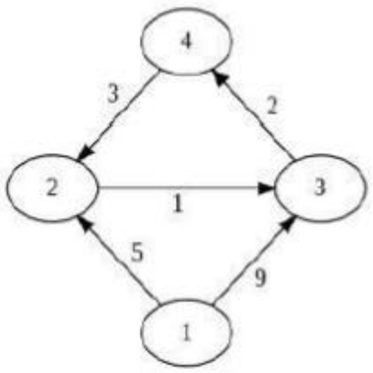
3. Weighted Graph:

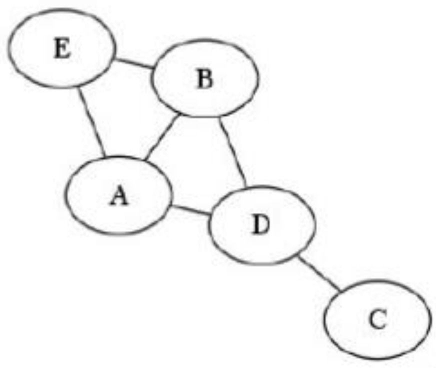
Weighted graphs assign numerical values (weights) to edges. These weights represent some property associated with the connection between nodes. For example, road networks with varying distances between cities, or airline routes with different flight durations, are examples of weighted graphs.

4. Unweighted Graph:

An unweighted graph has no edge weights. It focuses solely on connectivity between nodes. For example: a simple social network where friendships exist without any additional information, or a family tree connecting relatives.

b. *Weighted/ Unweighted:* A weighted graph has values on its edge.





5. Connected Graph:

A graph is connected if there is a path between any pair of nodes. In other words, you can reach any node from any other node. Even a single-node graph is considered connected. For larger graphs, there's always a way to move from one node to another.

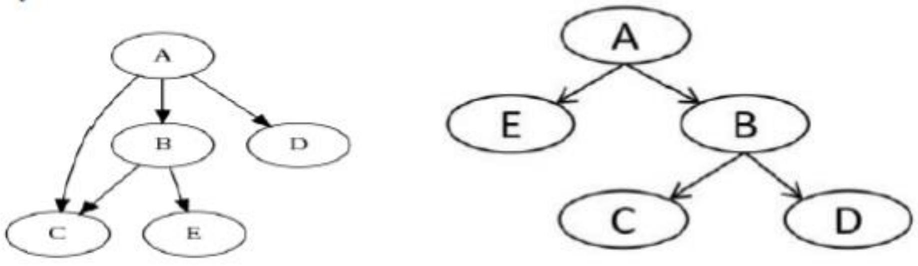
6. Acyclic Graph:

An acyclic graph contains no cycles (closed loops). In other words, you cannot start at a node and follow edges to return to the same node. Examples include family trees (without marriages between relatives) or dependency graphs in software development.

7. Cyclic Graph:

A cyclic graph has at least one cycle. You can traverse edges and eventually return to the same node. For example: circular road system or a sequence of events that repeats indefinitely.

c. *Cyclic/Acyclic*: A **cycle** is a path that begins and ends at same vertex and A graph with no cycles is **acyclic**.



8. Connected Graph

A Graph is connected if there is a **path between every pair of vertices** in the graph. In a directed graph, the concept of strong connectivity refers to the existence of a directed path between every pair of vertices.

9. Disconnected Graph:

A disconnected graph has isolated components that are not connected to each other. These components are separate subgraphs.

10. Tree

A Tree is a **connected graph with no cycles**. It is a fundamental data structure in computer science, commonly used in algorithms like binary search trees and heap data structures. Trees have properties such as a single root node, parent-child relationships between nodes, and a unique path between any pair of nodes.

Applications of Graph:

- **Transportation Systems**: Google Maps employs graphs to map roads, where intersections are vertices and roads are edges. It calculates shortest paths for efficient navigation.
- **Social Networks**: Platforms like Facebook model users as vertices and friendships as edges, using graph theory for friend suggestions.
- **World Wide Web**: Web pages are vertices, and links between them are directed edges, inspiring Google's Page Ranking Algorithm.
- **Resource Allocation and Deadlock Prevention**: Operating systems use resource allocation graphs to prevent deadlocks by detecting cycles.
- **Mapping Systems and GPS Navigation**: Graphs help in locating places and optimizing routes in mapping systems and GPS navigation.

- **Graph Algorithms and Measures**: Graphs are analyzed for structural properties and measurable quantities, including dynamic properties in networks.

Representations of Graph

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

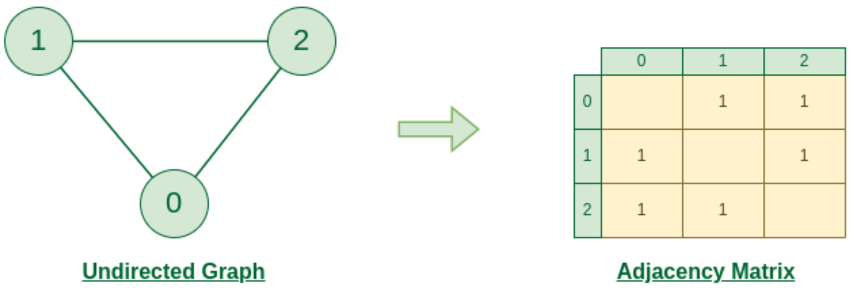
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.
- If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.

Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat**[**destination])** because we can go either way.



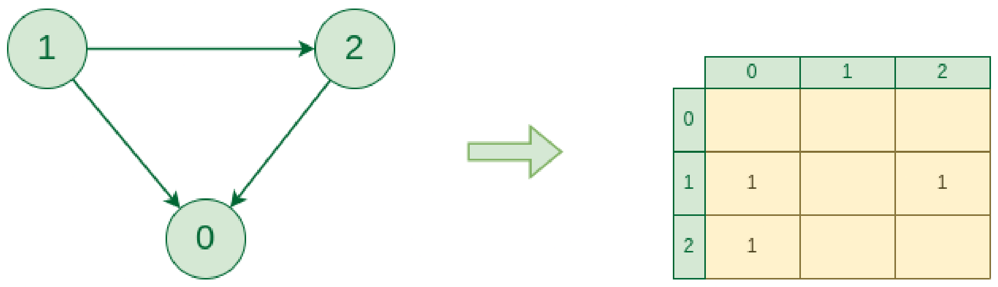
	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix

Undirected Graph to Adjacency Matrix

Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



	0	1	2
0			
1	1		1
2	1		

Directed Graph

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Directed Graph to Adjacency Matrix

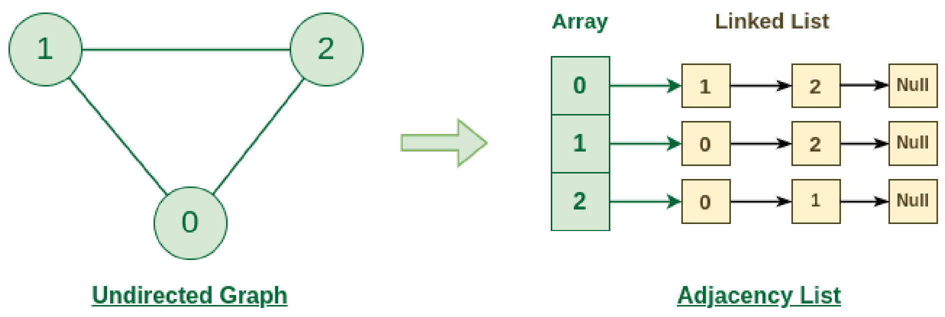
Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.
Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.

Representation of Undirected Graph as Adjacency list:

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

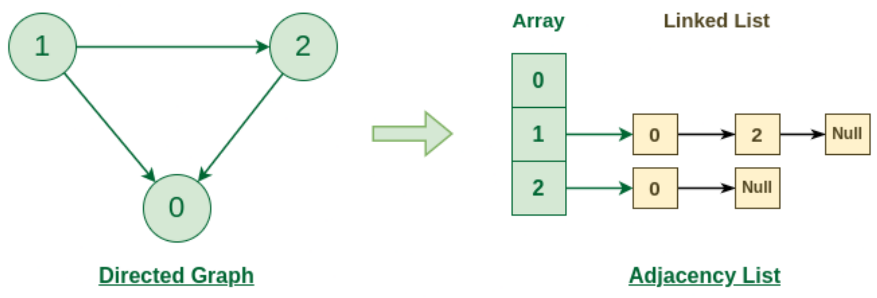


Graph Representation of Undirected graph to Adjacency List

Undirected Graph to Adjacency list

Representation of Directed Graph as Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

Directed Graph to Adjacency list

Basic search traversal :

"The process of traversing all the nodes or vertices on a graph is called graph traversal". We have two traversal techniques on graphs

DFS

BFS

Depth First Search

The DFS explore each possible path to its conclusion before another path is tried. In other words go as a far as you can (if u don't have a node to visit), otherwise, go back and try another way. Simply it can be called as "backtracking".

Breadth First Search

It is one of the simplest algorithms for searching or visiting each vertex in a graph. In this method each node on the same level is checked before the search proceeds to the next level. BFS makes use of a queue to store visited vertices, expanding the path from the earliest visited vertices

Breadth First Search or BFS for a Graph

Breadth First Search (BFS) **is a fundamental** graph traversal algorithm. **It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.**

Initialization: Enqueue the given source vertex into a queue and mark it as visited.

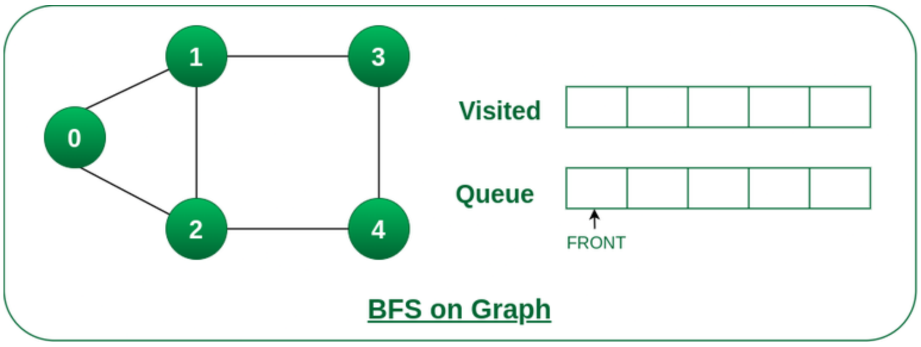
1. **Exploration:** While the queue is not empty:
   - Dequeue a node from the queue and visit it (e.g., print its value).
   - For each unvisited neighbor of the dequeued node:
     - Enqueue the neighbor into the queue.
     - Mark the neighbor as visited.
2. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

How Does the BFS Algorithm Work?

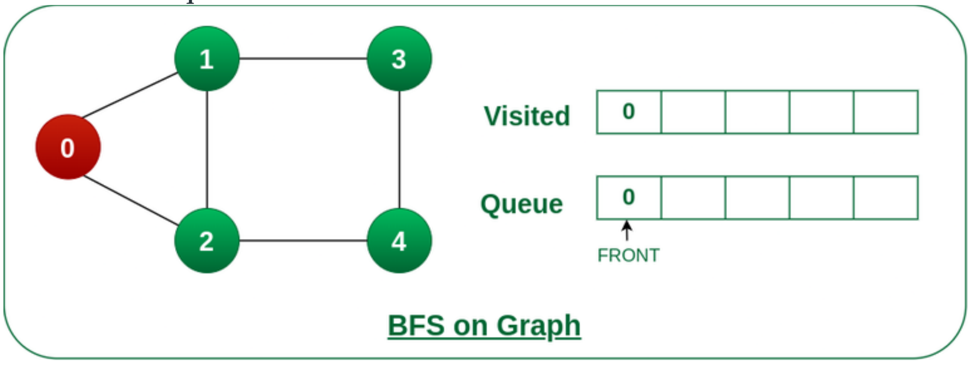
Let us understand the working of the algorithm with the help of the following example where the **source vertex is 0**.

Step1: Initially queue and visited arrays are empty.



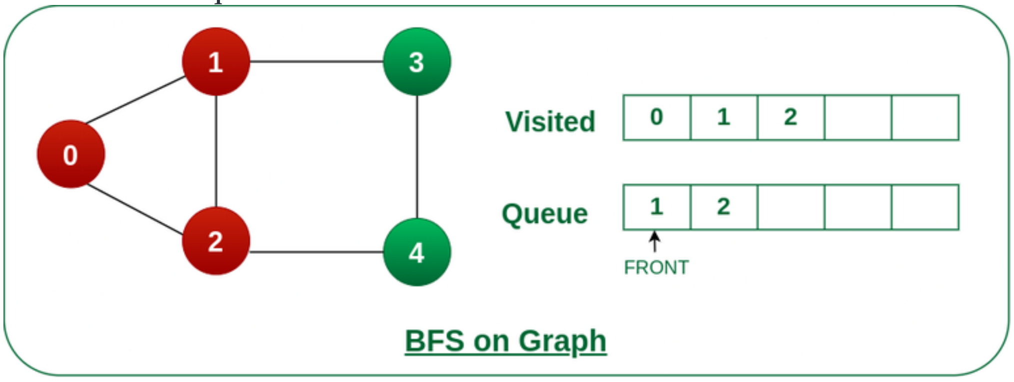
Queue and visited arrays are empty initially.

Step2: Push 0 into queue and mark it visited.



Push node 0 into queue and mark it visited.

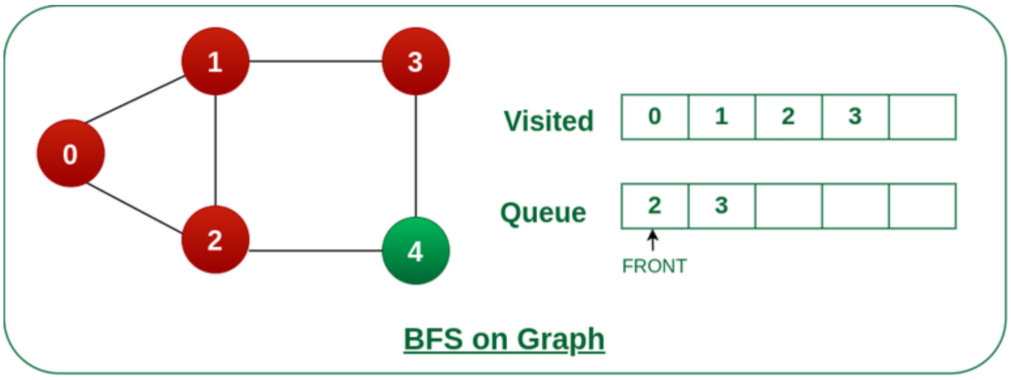
Step 3: Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 0 from the front of queue and visited the unvisited neighbours

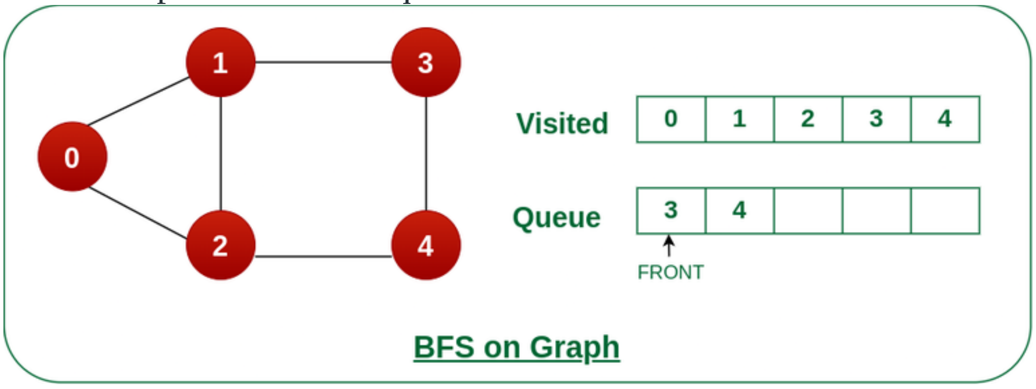
and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



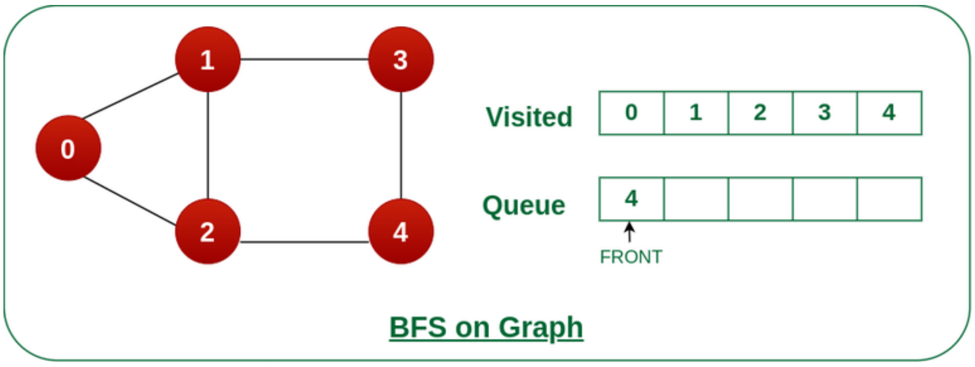
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



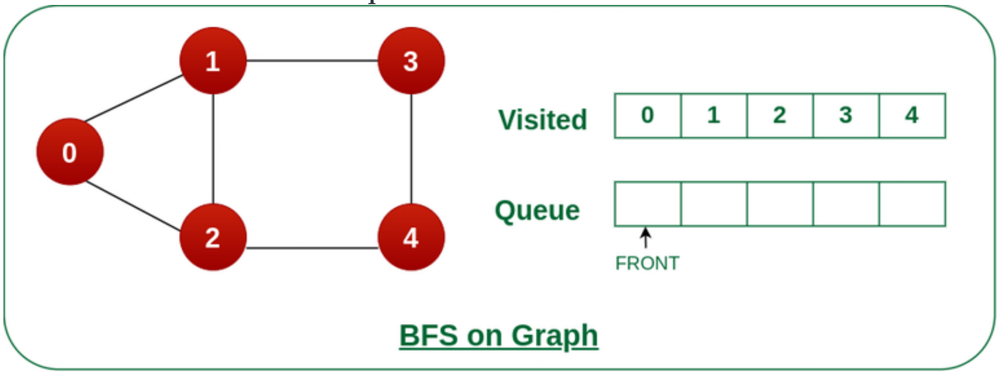
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

Depth First Search or DFS for a Graph

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.
Example:
Input: V = 5, E = 5, edges = {{1, 2}, {1, 0}, {0, 2}, {2, 3}, {2, 4}}, source = 1

Output: 1 2 0 3 4
Explanation: DFS Steps:

- Start at 1: Mark as visited. Output: 1
- Move to 2: Mark as visited. Output: 2
- Move to 0: Mark as visited. Output: 0 (backtrack to 2)
- Move to 3: Mark as visited. Output: 3 (backtrack to 2)
- Move to 4: Mark as visited. Output: 4 (backtrack to 1)

Input: V = 5, E = 4, edges = {{0, 2}, {0, 3}, {0, 1}, {2, 4}}, source = 0

Output: 0 2 4 3 1
Explanation: DFS Steps:

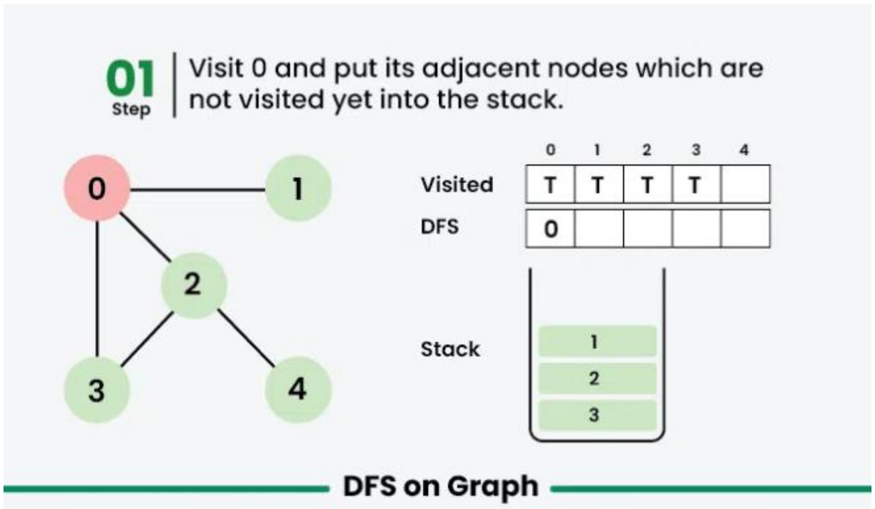
- Start at 0: Mark as visited. Output: 0
- Move to 2: Mark as visited. Output: 2
- Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 0)
- Move to 3: Mark as visited. Output: 3 (backtrack to 0)
- Move to 1: Mark as visited. Output: 1

Recommended Problem

DFS of Graph

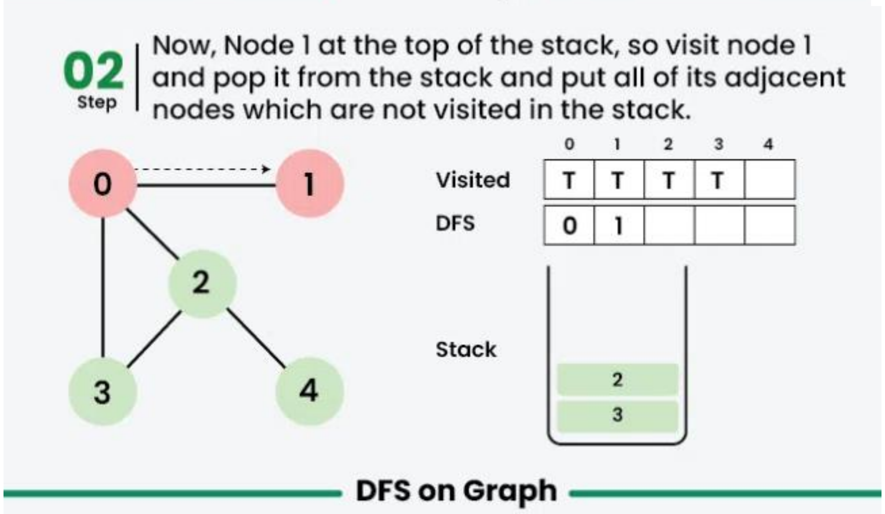
DFS from a Given Source of Undirected Graph:

The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to Preorder Tree Traversal where we visit the root, then recur for its children. In a graph, there maybe loops. So we use an extra visited array to make sure that we do not process a vertex again.
Let us understand the working of **Depth First Search** with the help of the following illustration: for the **source as 0**.



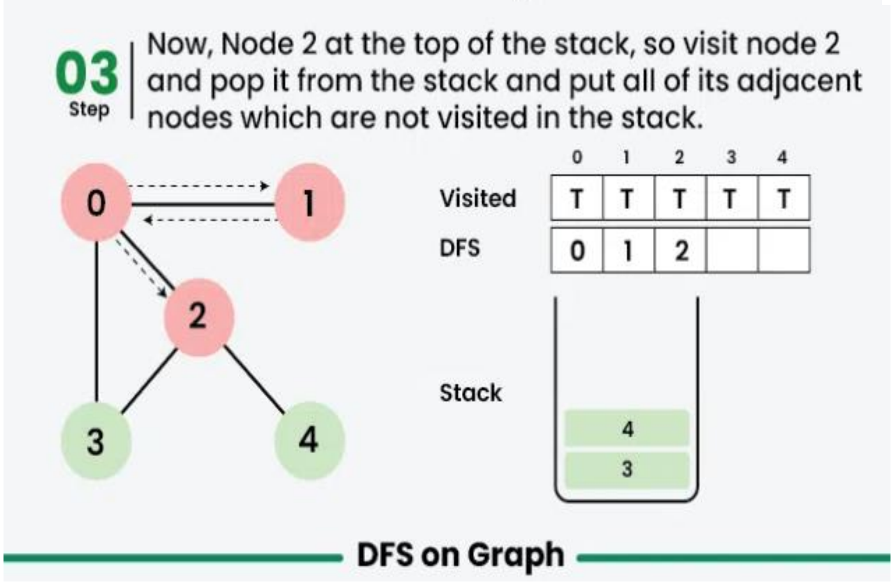
01 Step | Visit 0 and put its adjacent nodes which are not visited yet into the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0				



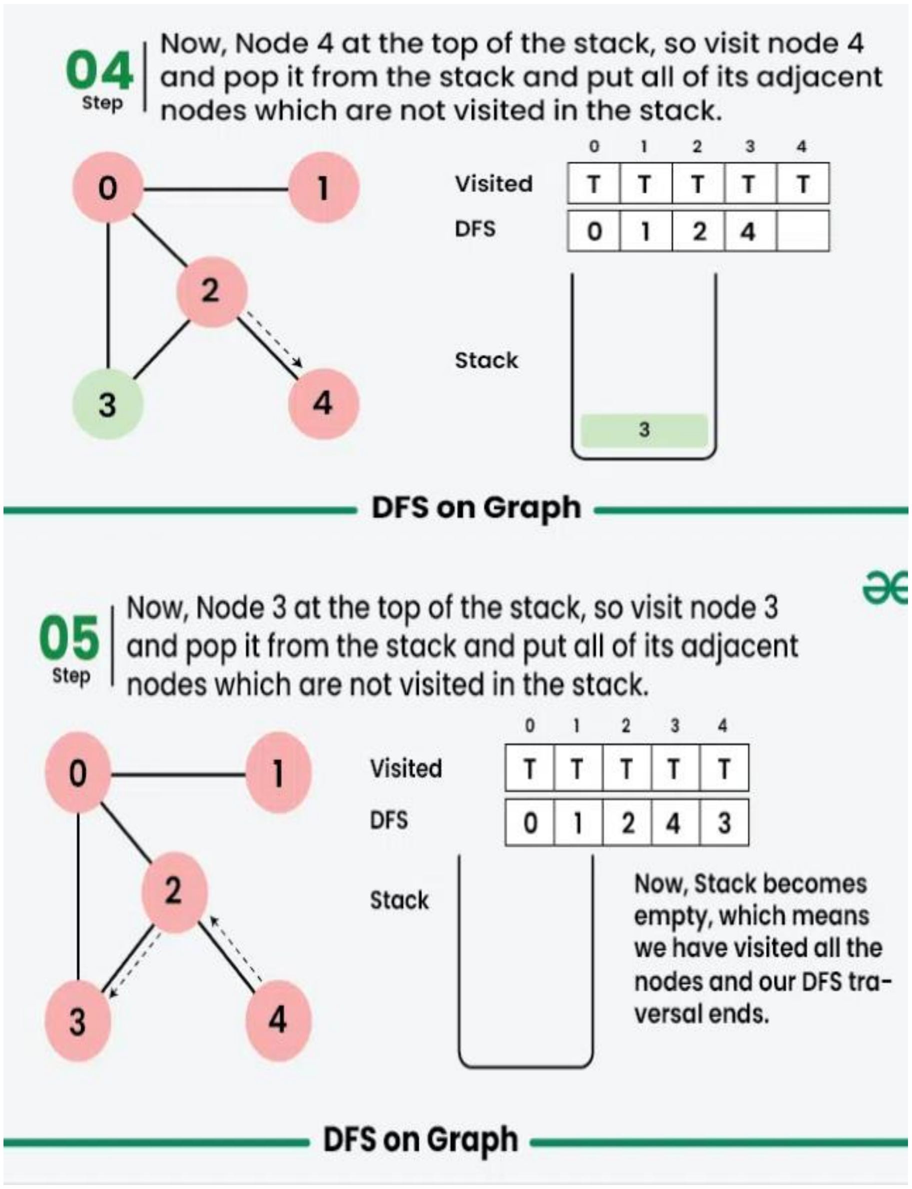
02 Step | Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0	1			



03 Step | Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2		


04 Step

Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	

05 Step

Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

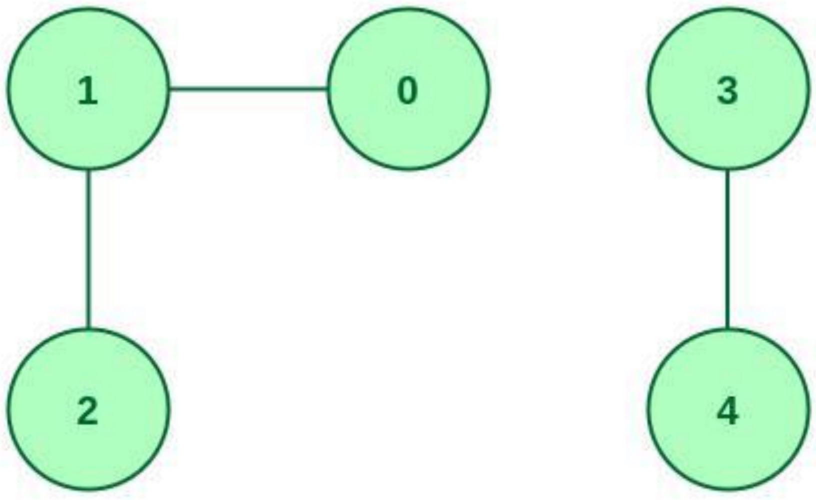
	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	3

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Connected component

Connected component in an undirected graph refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group.

For example in the graph shown below, {0, 1, 2} form a connected component and {3, 4} form another connected component.



Example of connected components

Characteristics of Connected Component:

- A connected component is a set of vertices in a graph that are connected to each other.
- A graph can have multiple connected components.
- Inside a component, each vertex is reachable from every other vertex in that component.

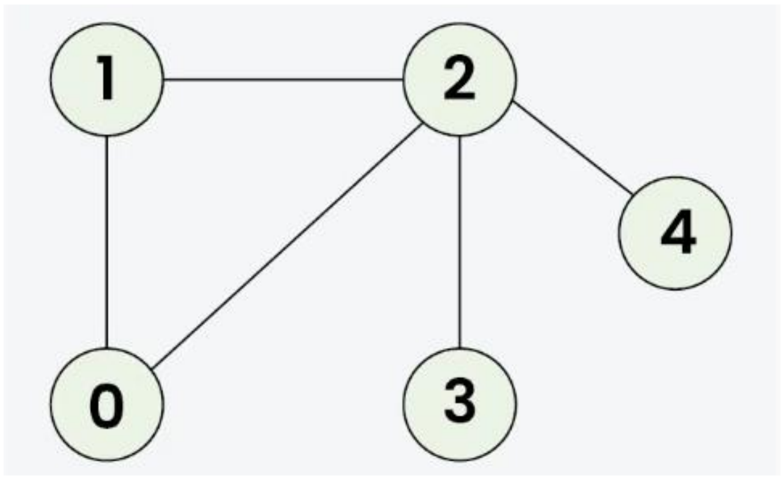
How to identify Connected Component:

There are several algorithms to identify Connected Components in a graph. The most popular ones are:

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Union-Find Algorithm (also known as Disjoint Set Union)

1) Depth First Search or DFS for a Graph

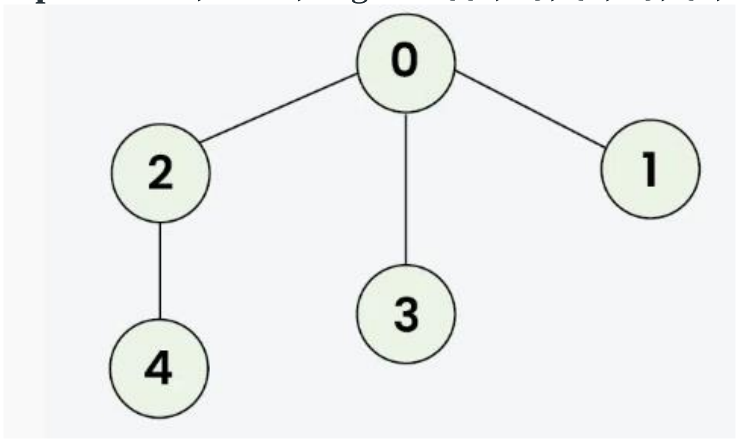
Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.
Example:
Input: V = 5, E = 5, edges = {{1, 2}, {1, 0}, {0, 2}, {2, 3}, {2, 4}}, source = 1



Output: 1 2 0 3 4
Explanation: DFS Steps:

- Start at 1: Mark as visited. Output: 1
- Move to 2: Mark as visited. Output: 2
- Move to 0: Mark as visited. Output: 0 (backtrack to 2)
- Move to 3: Mark as visited. Output: 3 (backtrack to 2)
- Move to 4: Mark as visited. Output: 4 (backtrack to 1)

Input: V = 5, E = 4, edges = {{0, 2}, {0, 3}, {0, 1}, {2, 4}}, source = 0



Output: 0 2 4 3 1
Explanation: DFS Steps:

- Start at 0: Mark as visited. Output: 0
- Move to 2: Mark as visited. Output: 2
- Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 0)
- Move to 3: Mark as visited. Output: 3 (backtrack to 0)
- Move to 1: Mark as visited. Output: 1

2) ➔BFS same as Above

3) Introduction to Disjoint Set (Union-Find Algorithm)

What is a Disjoint set data structure?

Two sets are called **disjoint sets** if they don't have any element in common, the intersection of sets is a null set.
A data structure that stores non overlapping or disjoint subset of elements is called disjoint set data structure. The disjoint set data structure supports following operations:

- Adding new sets to the disjoint set.
- Merging disjoint sets to a single disjoint set using **Union** operation.
- Finding representative of a disjoint set using **Find** operation.
- Check if two sets are disjoint or not.

Consider a situation with a number of persons and the following tasks to be performed on them:

- Add a **new friendship relation**, i.e. a person x becomes the friend of another person y i.e adding new element to a set.
- Find whether individual **x is a friend of individual y** (direct or indirect friend)

Examples:
We are given 10 individuals say, a, b, c, d, e, f, g, h, i, j
Following are relationships to be added:
a <-> b
b <-> d
c <-> f
c <-> i
j <-> e
g <-> j
Given queries like whether a is a friend of d or not. We basically need to create following 4 groups and maintain a quickly accessible connection among group items:
G1 = {a, b, d}
G2 = {c, f, i}
G3 = {e, g, j}
G4 = {h}

Find whether x and y belong to the same group or not, i.e. to find if x and y are direct/indirect friends.

Partitioning the individuals into different sets according to the groups in which they fall. This method is known as a **Disjoint set Union** which maintains a collection of **Disjoint sets** and each set is represented by one of its members.

To answer the above question two key points to be considered are:

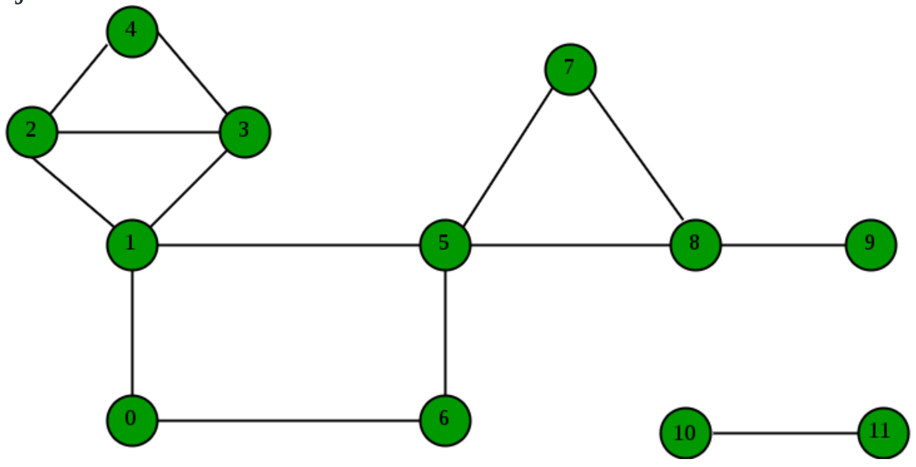
- **How to Resolve sets?** Initially, all elements belong to different sets. After working on the given relations, we select a member as a **representative**. There can be many ways to select a representative, a simple one is to select with the biggest index.
- **Check if 2 persons are in the same group?** If representatives of two individuals are the same, then they'll become friends.

Applications of Connected Component:

- Graph Theory: It is used to find subgraphs or clusters of nodes that are connected to each other.
- Computer Networks: It is used to discover clusters of nodes or devices that are linked and have similar qualities, such as bandwidth.
- Image Processing: Connected components also have usage in image processing.

Biconnected Components

A biconnected component is a maximal biconnected subgraph.
Biconnected Graph is already discussed here. In this article, we will see how to find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

Algorithm is based on Disc and Low Values discussed in Strongly Connected Components Article.
Idea is to store visited edges in a stack while DFS on a graph and keep looking for Articulation Points (highlighted in above figure). As soon as an Articulation Point u is found, all edges visited while DFS from node u onwards will form one biconnected component. When DFS completes for one connected component, all edges present in stack will form a biconnected component.
If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

Applications of Bi-Connected Component:

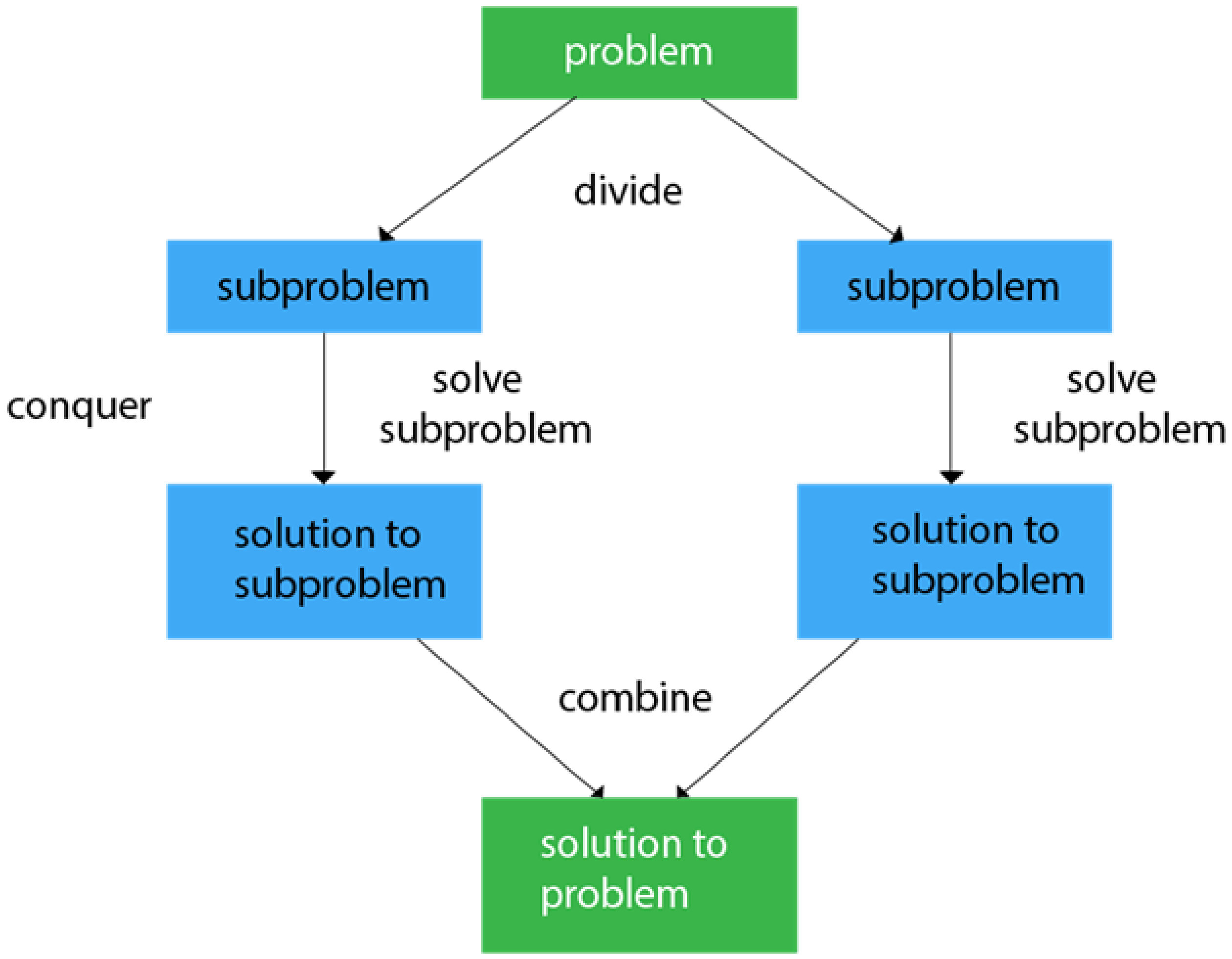
Biconnected graphs are used in the design of power grid networks. Consider the nodes as cities and the edges as electrical connections between them, you would like the network to be robust and a failure at one city should not result in a loss of power in other cities.

Divide and Conquer

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Stages of Divide and Conquer Algorithm:
Divide and Conquer Algorithm can be divided into three stages: **Divide**, **Conquer** and **Merge**.

1. Divide:

- Break down the original problem into smaller subproblems.
- Each subproblem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

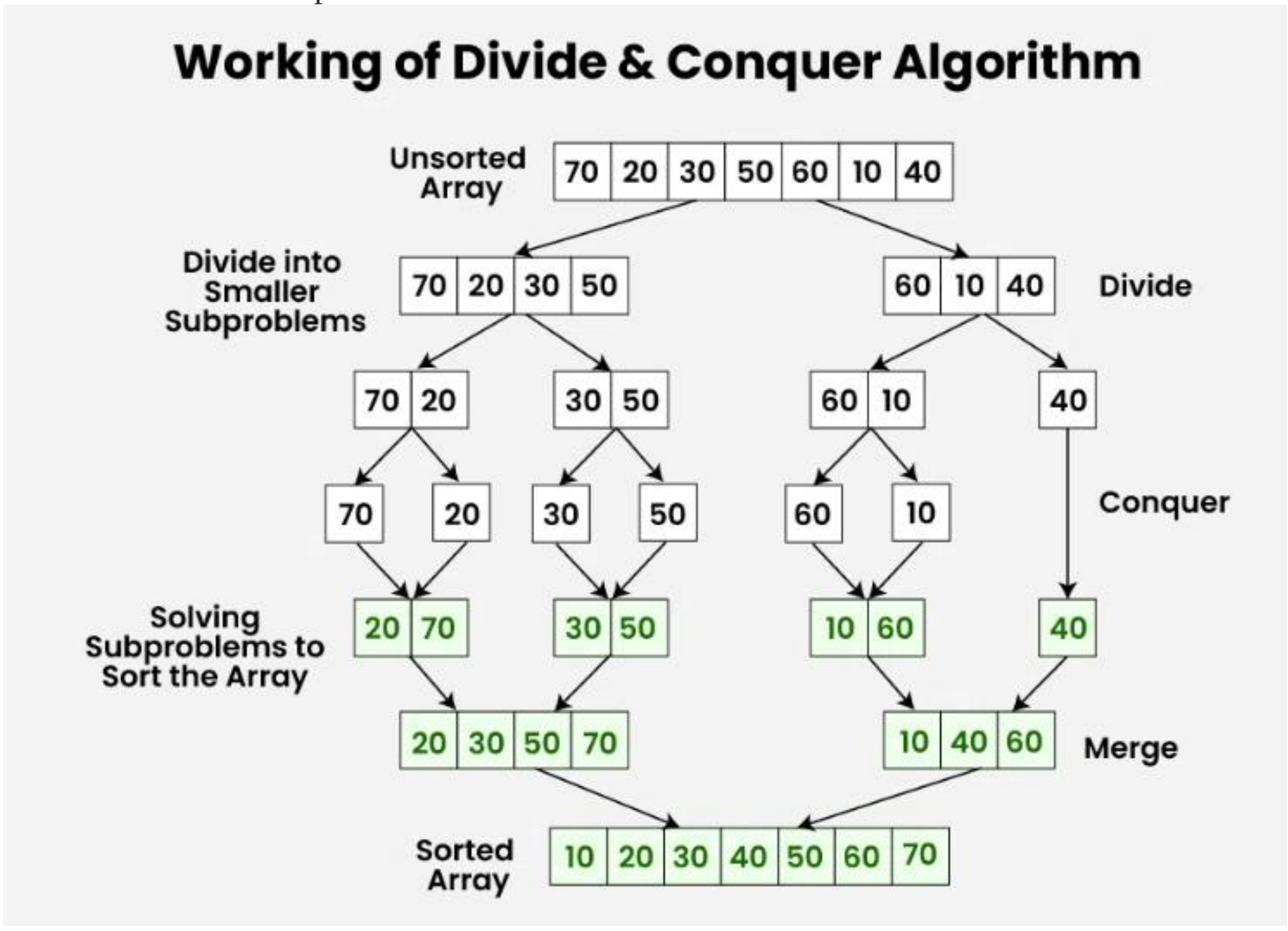
2. Conquer:

- Solve each of the smaller subproblems individually.
- If a subproblem is small enough (often referred to as the "base case"), we solve it directly without further recursion.
- The goal is to find solutions for these subproblems independently.

3. Merge:

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.

- The goal is to formulate a solution for the original problem by merging the results from the subproblems.



Applications of Divide and Conquer Algorithm:
The following are some standard algorithms that follow Divide and Conquer algorithm:

- **Quicksort** is a sorting algorithm that picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
- **Merge Sort** is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.

- **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in the x-y plane. The problem can be solved in O(n^2) time by calculating the distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(N log N) time.
- **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices needs 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.
- **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(N log N) time.
- **Karatsuba algorithm for fast multiplication** does the multiplication of two binary strings in O(n1.59) where n is the length of binary string.

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort Algorithm work?

There are mainly three steps in the algorithm.
1. Choose a pivot
2. Partition the array around pivot. After partition, it is ensured that all elements are smaller than all right and we get index of the end point of smaller elements. The left and right may not be sorted individually.
3. Recursively call for the two partitioned left and right subarrays.
4. We stop recursion when there is only one element is left.

Choice of Pivot:

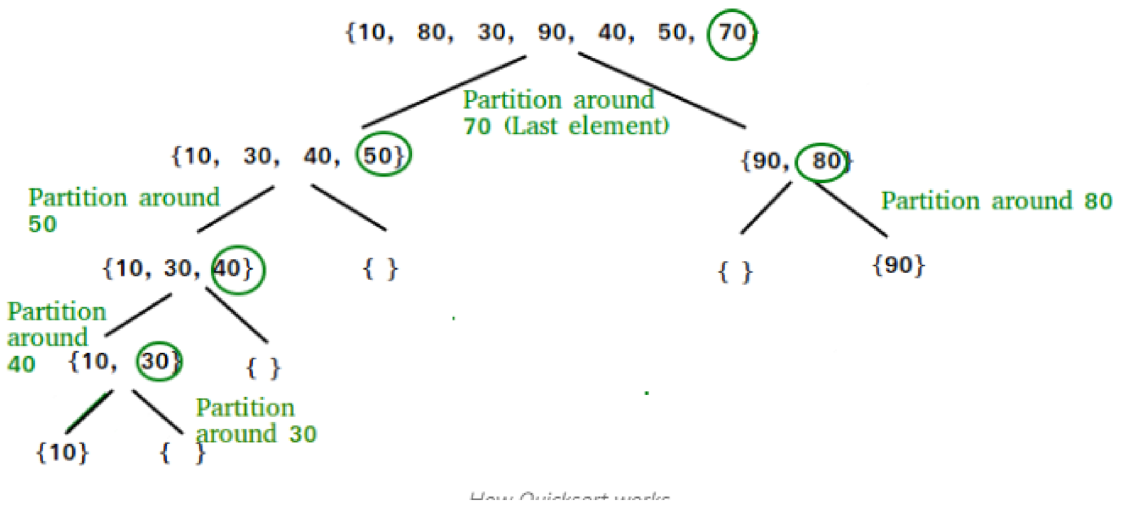
There are many different choices for picking pivots.
- Always pick the first (or last) element as a pivot. The below implementation is picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.
- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.
- Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it is low on average as median finding has high constants.

Partition Algorithm:

The key process in **quickSort** is a **partition().** There are three common algorithms to partition. All these algorithms have O(n) time complexity.

1. **Naive Partition** : Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires O(n) extra space.
2. **Lomuto Partition** : We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.
3. **Hoare's Partition** : This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned. Please refer Hoare's vs Lomuto for details.

Below diagram illustrates overview of the algorithm using last element as pivot and Lomuto partition,

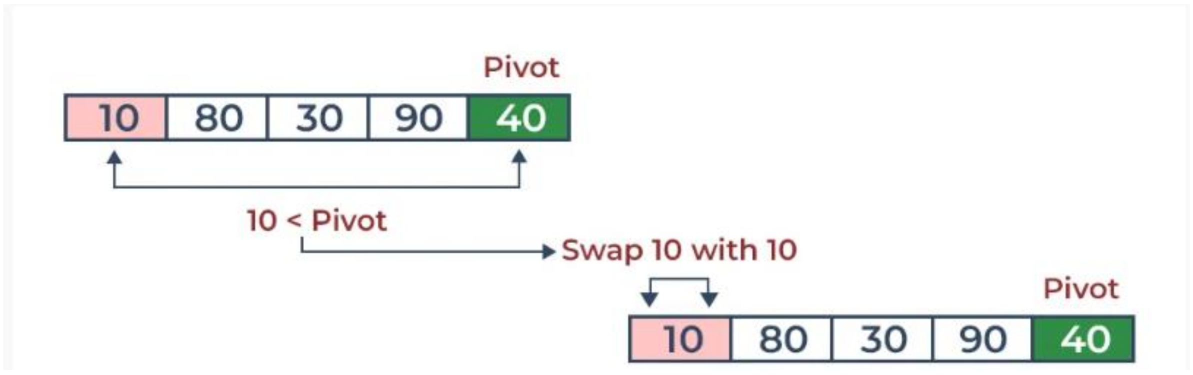


Details of the Partition Algorithm and Illustration :

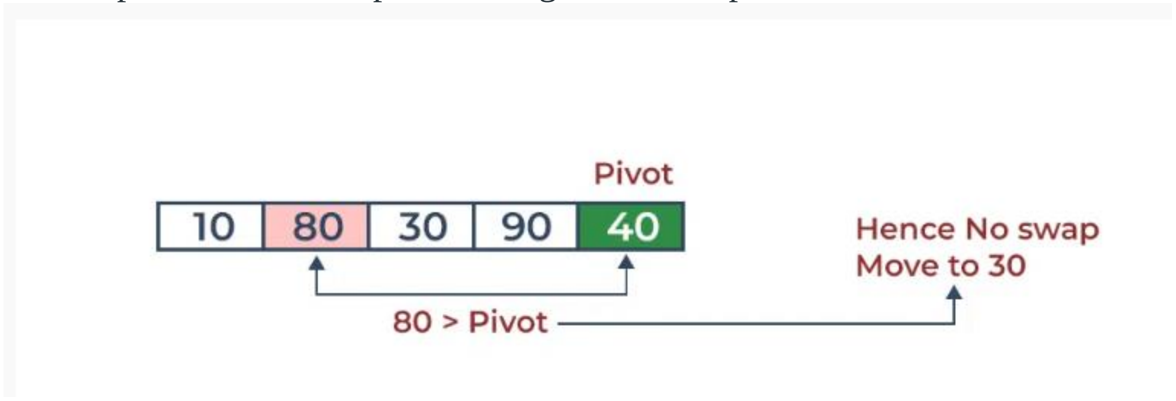
The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i** . While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:
Consider: arr[] = {10, 80, 30, 90, 40}.

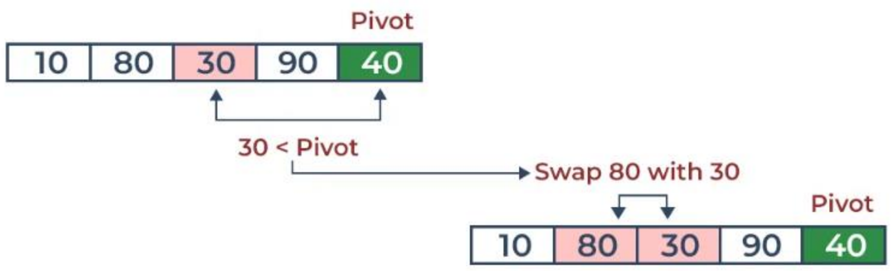
- Compare 10 with the pivot and as it is less than pivot arrange it accordingly.



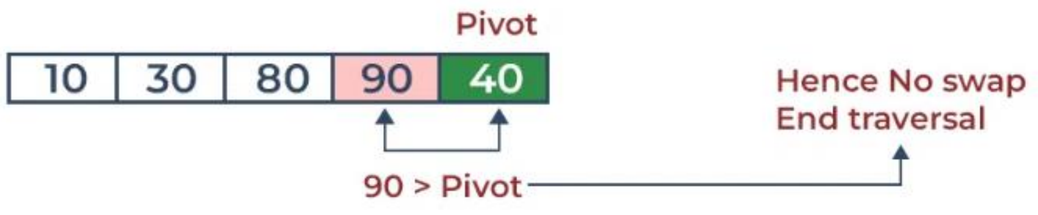
- Compare 80 with the pivot. It is greater than pivot.



- Compare 30 with pivot. It is less than pivot so arrange it accordingly.



- Compare 90 with the pivot. It is greater than the pivot.



- Arrange the pivot in its correct position.

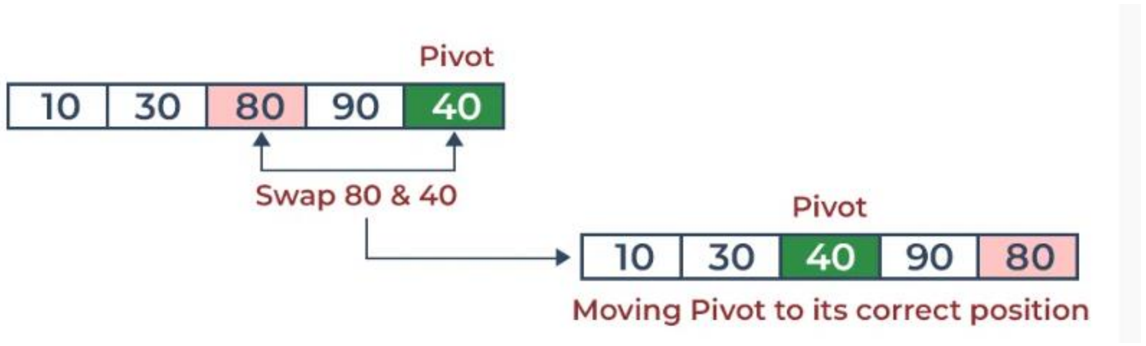
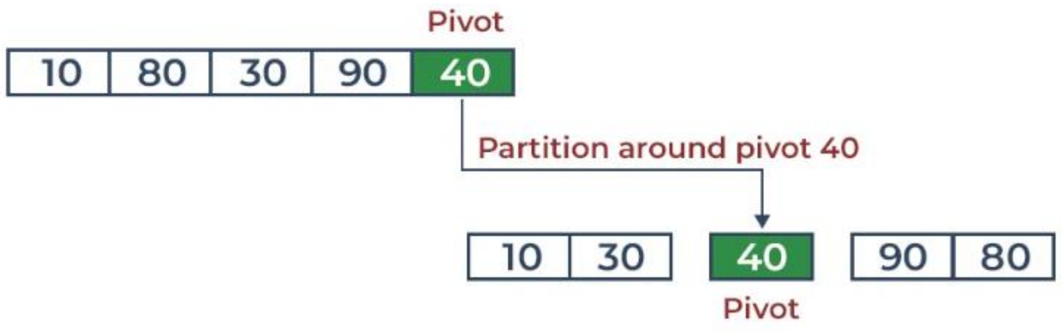
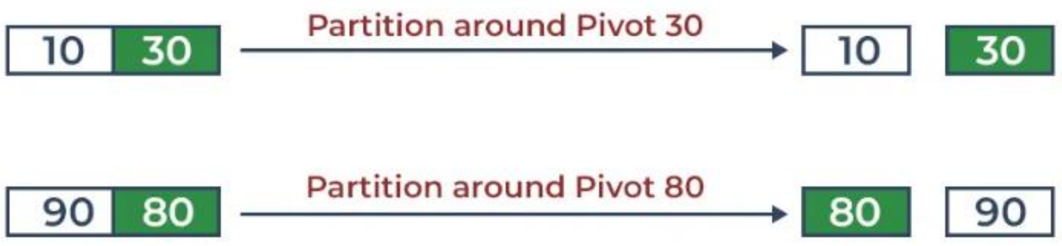


Illustration of QuickSort Algorithm:
The partition keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted. Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

- Initial partition on the main array:



Partitioning of the subarrays:



Complexity Analysis of Quick Sort :

Time Complexity:

- **Best Case** : Ω (N log (N))
  The best-case scenario for quicksort occur when the pivot chosen at the

each step divides the array into roughly equal halves.
In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

- **Average Case:** θ (N log (N))
  Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- **Worst Case:** O(N ^ 2)
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.
- **Auxiliary Space:** O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O (N).

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general purpose algorithm for large data when stability is not required.
- It is **tail recursive** and hence all the tail call optimization can be done.

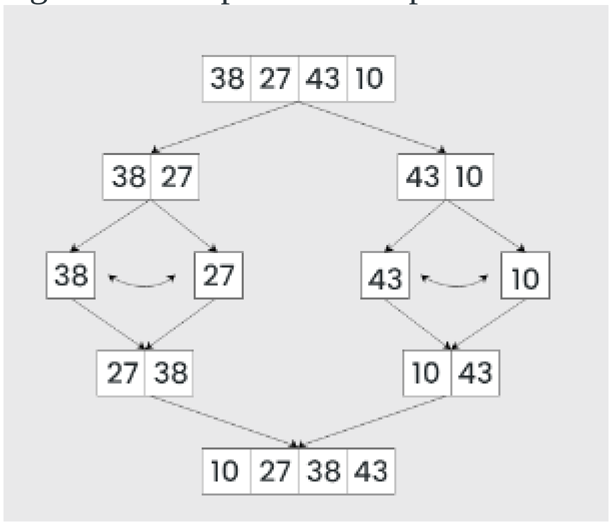
Disadvantages of Quick Sort:

- It has a worst-case time complexity of O(N 2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Merge Sort

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



How does Merge Sort work?

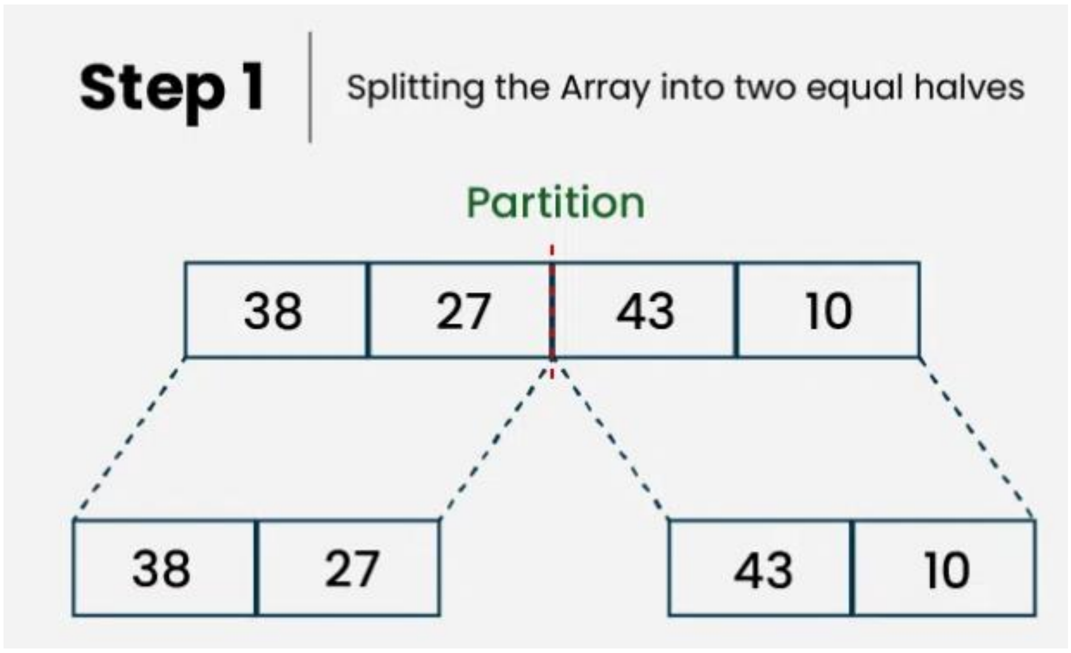
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

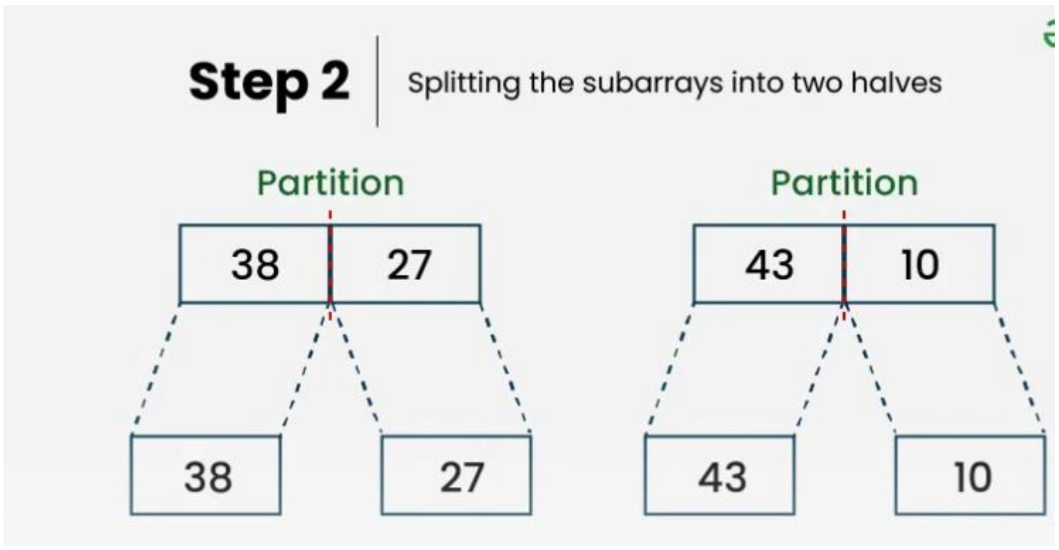
Here’s a step-by-step explanation of how merge sort works:

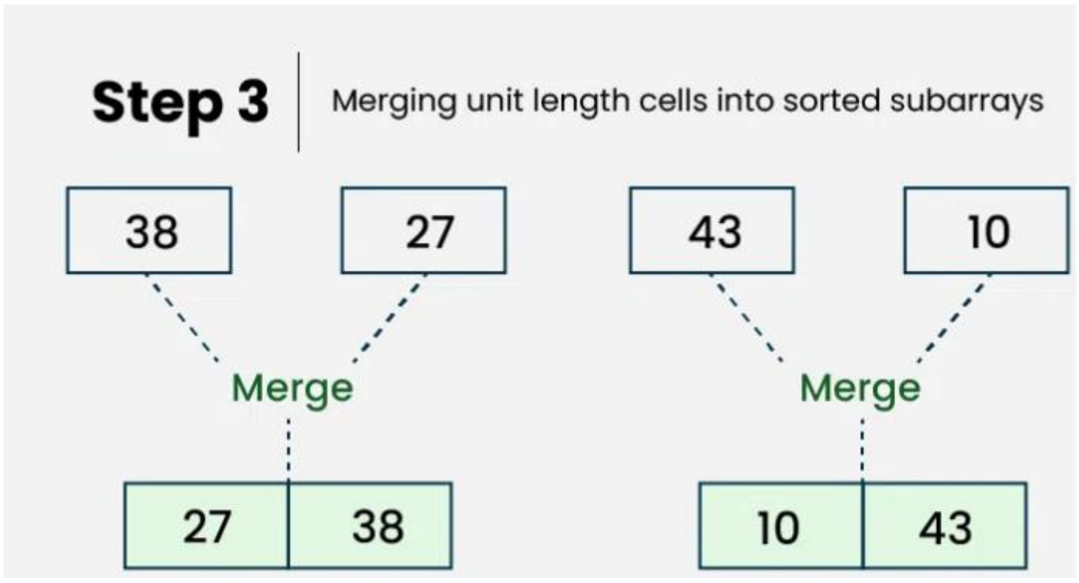
1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

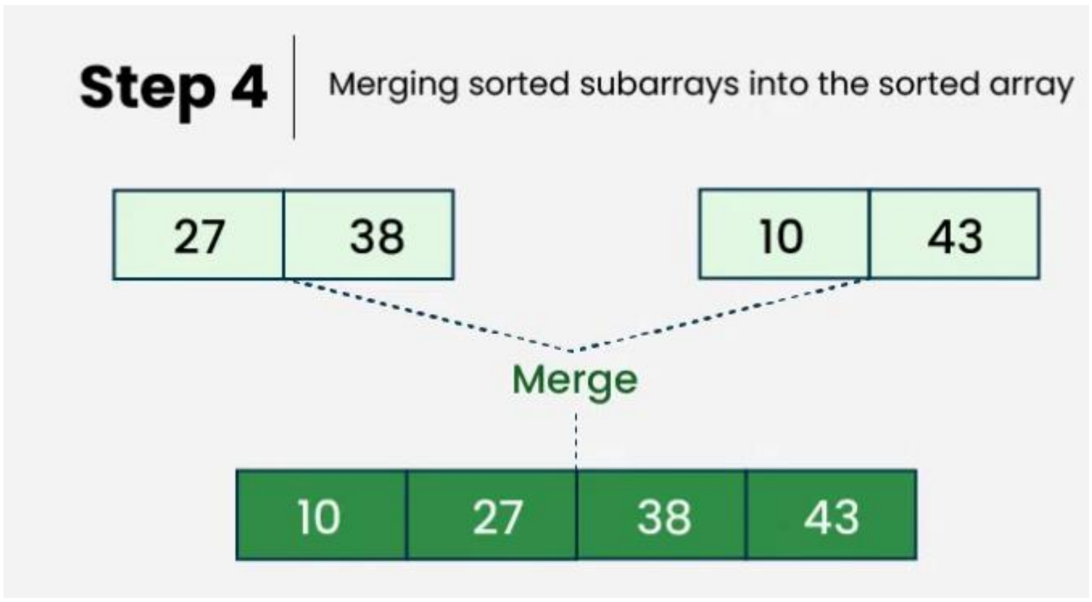
Illustration of Merge Sort:

Let’s sort the array or list **[38, 27, 43, 10]** using Merge Sort


Step 1
Splitting the Array into two equal halves
Partition
38
27
43
10
38
27
43
10


Step 2
Splitting the subarrays into two halves
Partition
Partition
38
27
43
10
38
27
43
10





Let’s look at the working of above example:

Divide:

- **[38, 27, 43, 10]** is divided into **[38, 27**] and **[43, 10]** .
- **[38, 27]** is divided into **[38]** and **[27]** .
- **[43, 10]** is divided into **[43]** and **[10]** .

Conquer:

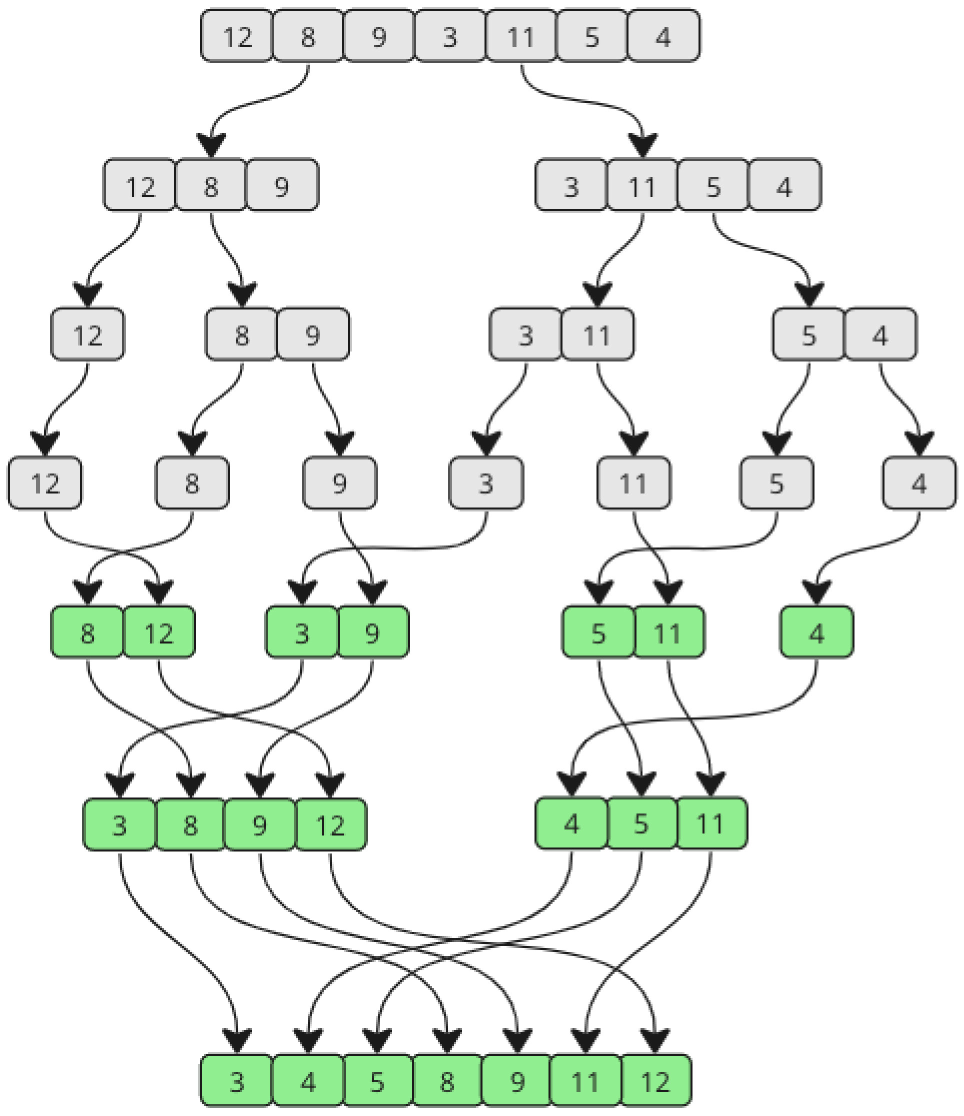
- **[38]** is already sorted.
- **[27]** is already sorted.
- **[43]** is already sorted.
- **[10]** is already sorted.

Merge:

- Merge **[38]** and **[27]** to get **[27, 38]** .
- Merge **[43]** and **[10]** to get **[10,43]** .
- Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]**

Therefore, the sorted list is **[10, 27, 38, 43]** .

Example:



Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is **$O(n^3)$**, since it takes two loops to multiply.

Strassen's method was introduced to reduce the time complexity from **$O(n^3)$** to **$O(n^{\log 7})$**.

Naive Method

First, we will discuss Naive method and its complexity. Here, we are calculating Z=***X***X × Y. Using Naive method, two matrices (**X** and **Y**) can be multiplied if the order of these matrices are **p** × **q** and **q** × **r** and the resultant matrix will be of order **p** × **r**. The following pseudocode describes the Naive multiplication −

```
Algorithm: Matrix-Multiplication (X, Y, Z)
for i = 1 to p do
  for j = 1 to r do
    Z[i,j] := 0
    for k = 1 to q do
      Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

Complexity

Here, we assume that integer operations take **O(1)** time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes **$O(n^3)$** time to execute.

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four **(n/2)×(n/2)** matrices as represented below −

Z=[IKJL]Z=[IJKL] X=[ACBD]X=[ABCD] and Y=[EGFH]Y=[EFGH]

Using Strassen's Algorithm compute the following −

M1:=(A+C)×(E+F)M1:=(A+C)×(E+F)

M2:=(B+D)×(G+H)M2:=(B+D)×(G+H)

M3:=(A−D)×(E+H)M3:=(A−D)×(E+H)

M4:=A×(F−H)M4:=A×(F−H)

M5:=(C+D)×(E)M5:=(C+D)×(E)

M6:=(A+B)×(H)M6:=(A+B)×(H)

M7:=D×(G−E)M7:=D×(G−E)

Then,

I:=M2+M3−M6−M7I:=M2+M3−M6−M7

J:=M4+M6J:=M4+M6

K:=M5+M7K:=M5+M7

L:=M1−M3−M4−M5L:=M1−M3−M4−M5

Analysis

T(n)={c7xT(n2)+dxn2ifn=1otherwisewherecanddareconstantsT(n)={cifn=17xT(n2)+dxn2otherwisewherecanddareconstants

Using this recurrence relation, we get T(n)=O(nlog7)T(n)=O(nlog7)

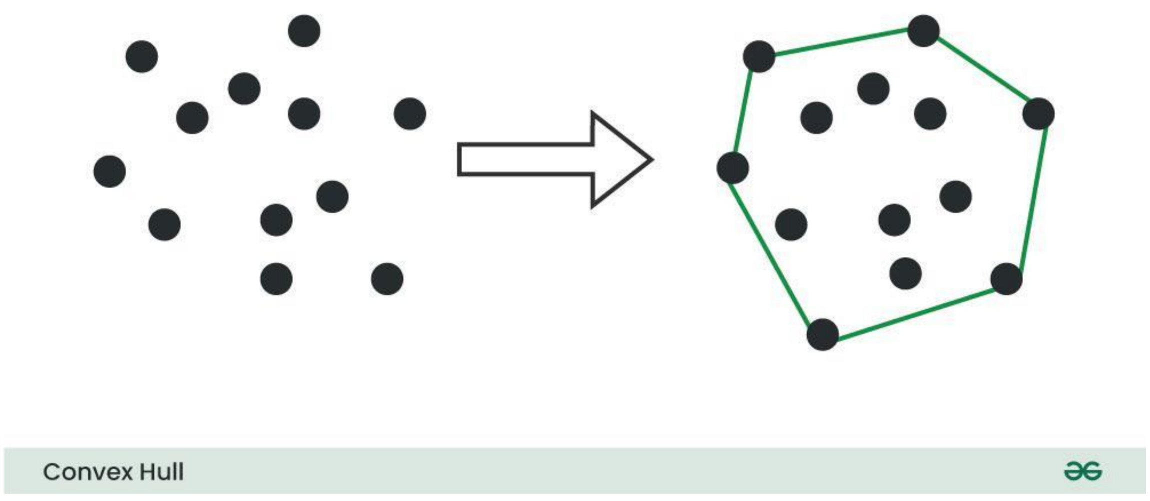
Hence, the complexity of Strassen’s matrix multiplication algorithm is O(nlog7)

Convex Hull Algorithm

The **Convex Hull Algorithm** is used to find the **convex hull** of a set of points in computational geometry. The convex hull is the smallest convex set that encloses all the points, forming a convex polygon. This algorithm is important in various applications such as image processing, route planning, and object modeling.

What is Convex Hull?

The convex hull of a set of points in a Euclidean space is the smallest convex polygon that encloses all of the points. In two dimensions (2D), the convex hull is a convex polygon, and in three dimensions (3D), it is a convex polyhedron. The below image shows a 2-D convex polygon:

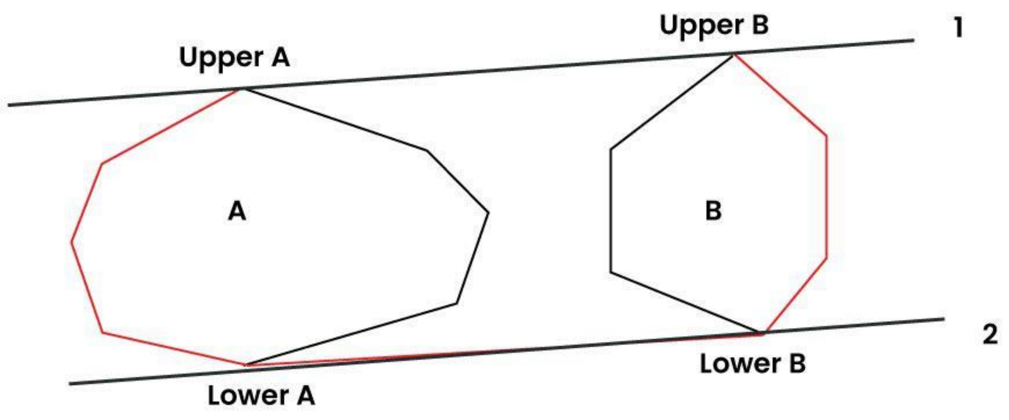


Examples:
Input: points[] = {(0, 0), (0, 4), (-4, 0), (5, 0), (0, -6), (1, 0)};
Output: (-4, 0), (5, 0), (0, -6), (0, 4)
Input: points_list = {{1, 2}, {3, 1}, {5, 6}}
Output: {{1, 2}, {3, 1}, {5, 6}}

Convex Hull using Divide and Conquer Algorithm:

Pre-requisite: Tangents between two convex polygons
Algorithm: Given the set of points for which we have to find the convex hull. Suppose we know the convex hull of the left half points and the right half points, then the problem now is to merge these two convex hulls and determine the convex hull for the complete set.

- This can be done by finding the upper and lower tangent to the right and left convex hulls. This is illustrated here Tangents between two convex polygons Let the left convex hull be **a** and the right convex hull be **b**.
- Then the lower and upper tangents are named as 1 and 2 respectively, as shown in the figure. Then the red outline shows the final convex hull.



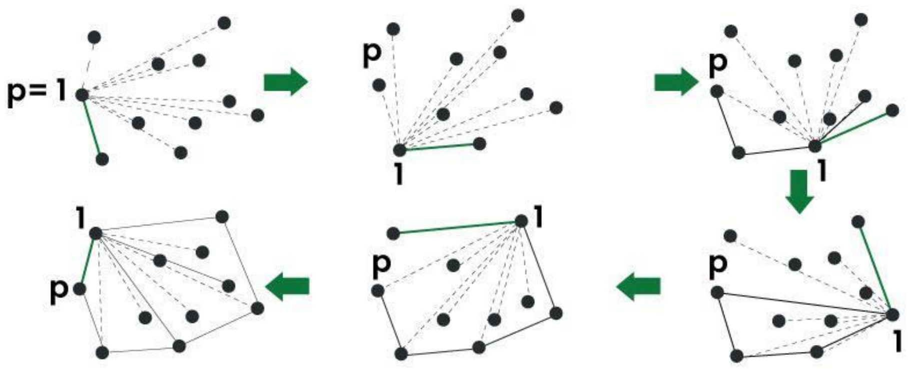
Final Convex Hull

- Now the problem remains, how to find the convex hull for the left and right half. Now recursion comes into the picture, we divide the set of points until the number of points in the set is very small, say 5, and we can find the convex hull for these points by the brute algorithm.
- The merging of these halves would result in the convex hull for the complete set of points. **Note:** We have used the brute algorithm to find the convex hull for a small number of points and it has a time complexity of O(N^3).
- But some people suggest the following, the convex hull for 3 or fewer points is the complete set of points. This is correct but the problem comes when we try to merge a left convex hull of 2 points and right convex hull of 3 points, then the program gets trapped in an infinite loop in some special cases.
- So, to get rid of this problem I directly found the convex hull for 5 or fewer points by O(N^3) algorithm, which is somewhat greater but does not affect the overall complexity of the algorithm.

Convex Hull using Jarvis' Algorithm or Wrapping**:**
Pre-requisite: How to check if two given line segments intersect?
The idea of **Jarvis's Algorithm** is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction.
The big question is, given a point p as current point, how to find the next point in output?
The idea is to use orientation() here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, q, r) = counterclockwise".

Algorithm:

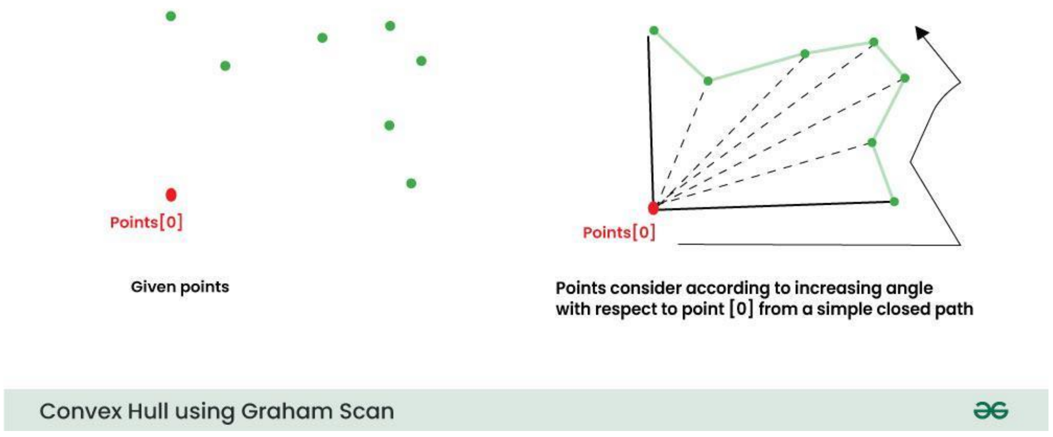
- Initialize **p** as leftmost point.
- Do following while we don't come back to the first (or leftmost) point.
  - The next point **q** is the point, such that the triplet **(p, q, r)** is counter clockwise for any other point **r.** To find this, we simply initialize **q** as next point, then we traverse through all points. For any point i, if i is more counter clockwise, i.e., **orientation(p, i, q) is** counter clockwise, then we update **q** as **i**. Our final value of **q** is going to be the most counter clockwise point.
  - **next[p] = q** (Store q as next of **p** in the output convex hull).
  - **p = q** (Set **p** as **q** for next iteration).



The exceution of jarvis March

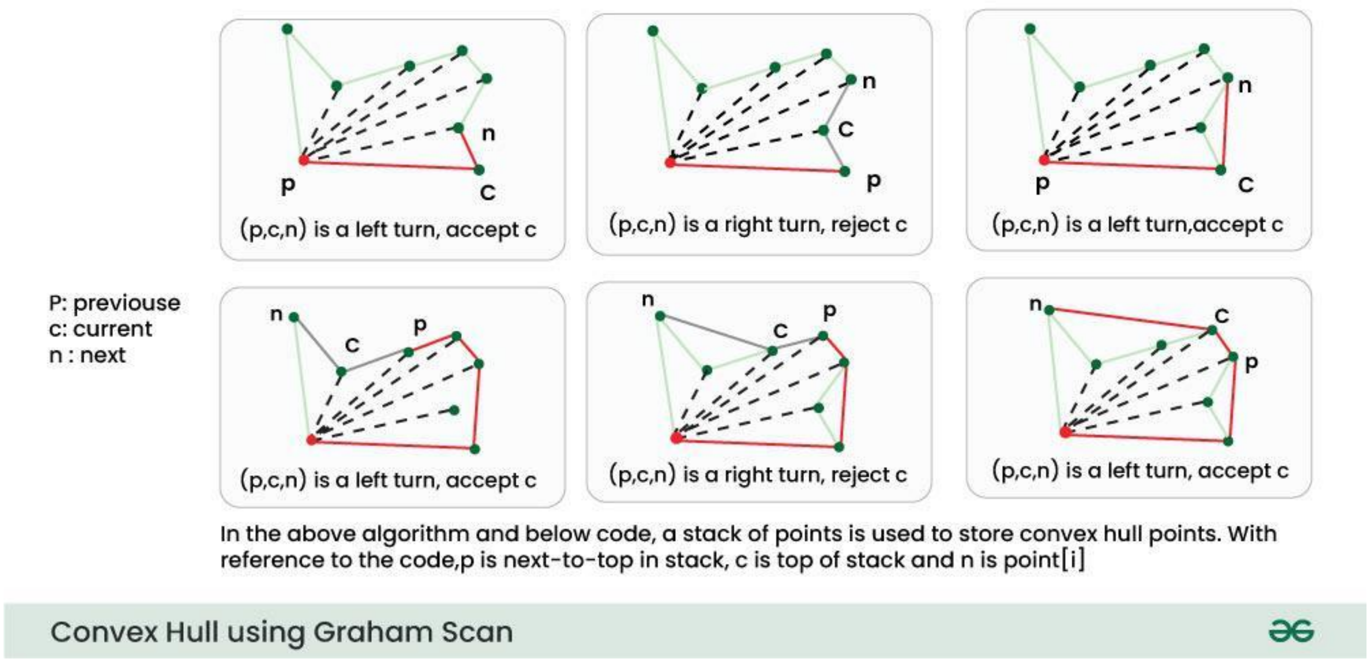
Convex Hull using Jarvis' Algorithm or Wrapping

Convex Hull using Graham Scan:
Pre-requisite: How to check if two given line segments intersect?
Algorithm: Let points[0..n-1] be the input array. Then the algorithm can be divided into two phases:
Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points be sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See the following diagram).


Points[0]
Given points
Points[0]
Points consider according to increasing angle with respect to point [0] from a simple closed path
Convex Hull using Graham Scan

What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the compare() function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be **prev(p), curr(c)** and **next(n)**. If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase.


P: previouse
c: current
n : next
p
c
n
(p,c,n) is a left turn, accept c
(p,c,n) is a right turn, reject c
(p,c,n) is a left turn,accept c
(p,c,n) is a left turn, accept c
(p,c,n) is a right turn, reject c
(p,c,n) is a left turn, accept c
In the above algorithm and below code, a stack of points is used to store convex hull points. With reference to the code,p is next-to-top in stack, c is top of stack and n is point[i]
Convex Hull using Graham Scan

Convex Hull | Monotone Chain Algorithm:
Monotone chain algorithm constructs the convex hull in **O(n * log(n))** time. We have to sort the points first and then calculate the upper and lower hulls in **O(n)** time. The points will be sorted with respect to x-coordinates (with respect to y-coordinates in case of a tie in x-coordinates), we will then find the left most point and then try to rotate in clockwise direction and find the next point and then repeat the step until we reach the rightmost point and then again rotate in the clockwise direction and find the lower hull.

Quickhull Algorithm for Convex Hull:
The **QuickHull algorith**m is a Divide and Conquer algorithm similar to QuickSort. Let **a[0…n-1]** be the input array of points. Following are the steps for finding the convex hull of these points.

Algorithm:

- Find the point with minimum x-coordinate lets say, **min_x** and similarly the point with maximum x-coordinate, **max_x**.
- Make a line joining these two points, say **L**. This line will divide the whole set into two parts. Take both the parts one by one and proceed further.
- For a part, find the point **P** with maximum distance from the line **L**. **P** forms a triangle with the points **min_x**, **max_x**. It is clear that the points residing inside this triangle can never be the part of convex hull.
- The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points **P** and **min_x** and the line joining the points **P** and **max_x** are new lines and the points residing outside the triangle is the set of points. Repeat point no. 3 till there no point left with the line. Add the end points of this point to the convex hull.