UNIT – I

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.

AVL Trees – Creation, Insertion, Deletion operations and Applications

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications

……

1. Algorithm Definition

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

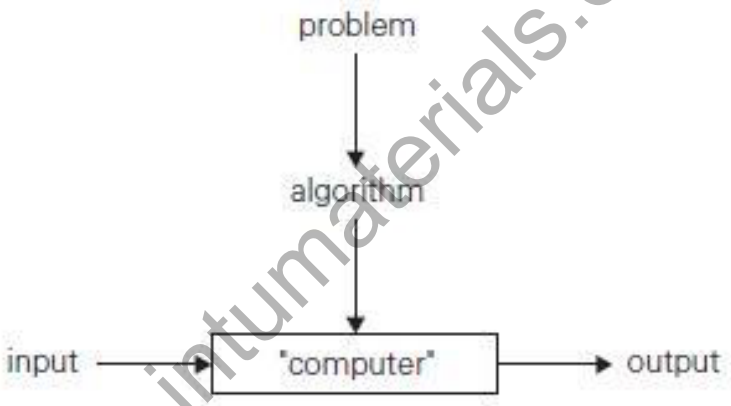
Input: Zero or more quantities are externally supplied.

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous.

Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

Effectiveness: Every instruction must be vey basic so that it can be carried out, in principle, b a person using only pencil and paper. It is not enough that each operation be definite, it also must be feasible.



Four Distinct areas of study of algorithms:

1. How to devise algorithms: Techniques–Divide & Conquer, Branch and Bound, Dynamic Programming

2. How to validate algorithms:

The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in.

3. How to analyse algorithms:

analysis algorithms or performance analysis refers to the task of determining how much computing time and storage time an algorithm required.

4. How to test a program2 phases

• Debugging - Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.

• Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

2. Asymptotic notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1: $5n^2 + 2n + 1$**
- **Algorithm 2: $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two-time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.

Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So, for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

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

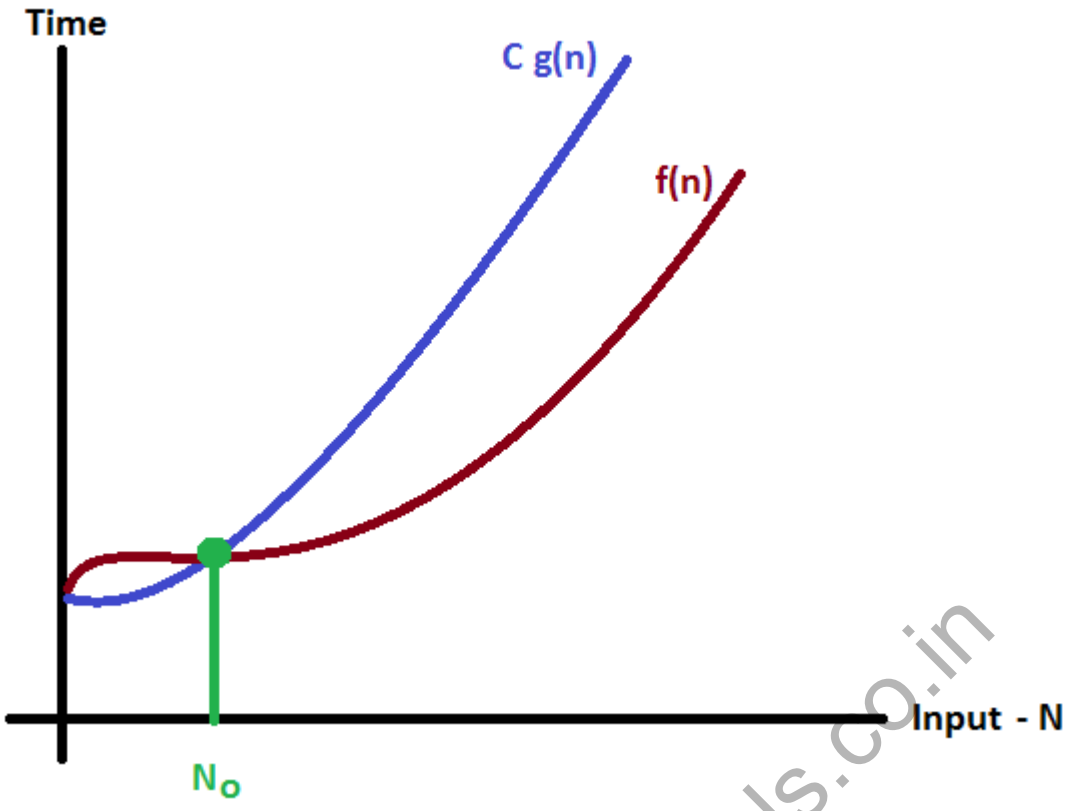
Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n
Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

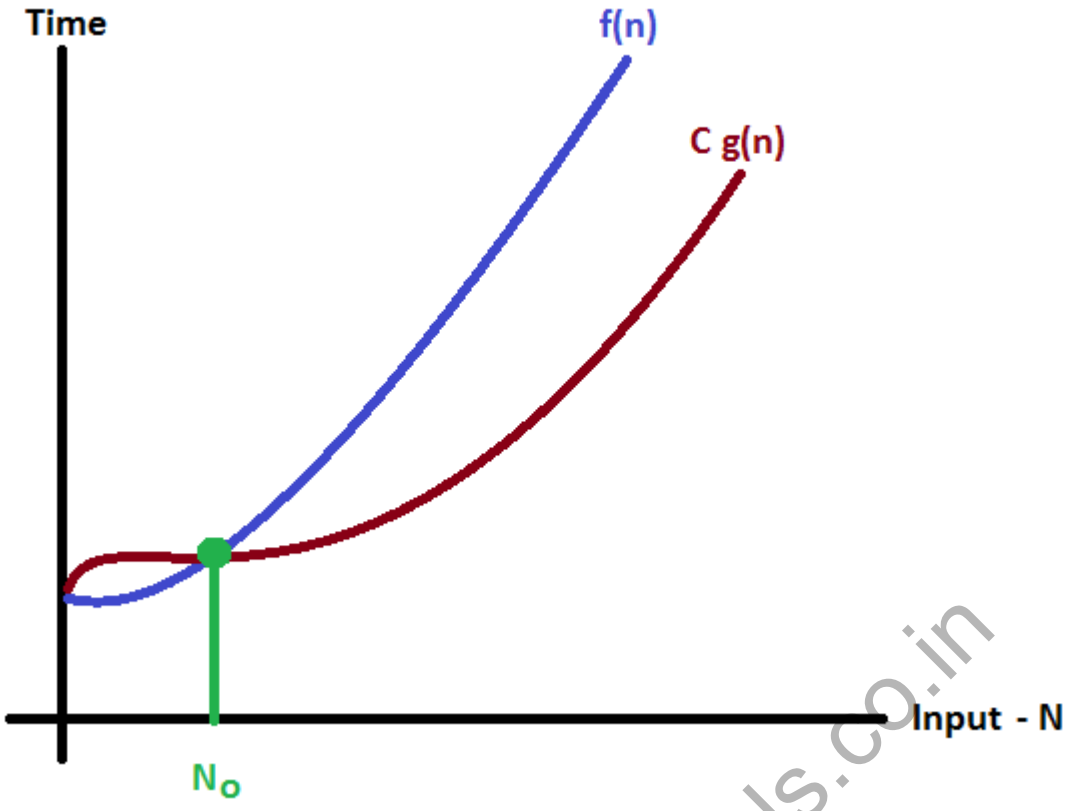
2. Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

$$f(n) = \Omega(g(n))$$

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) >= C g(n)
⇒3n + 2 >= C n
Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

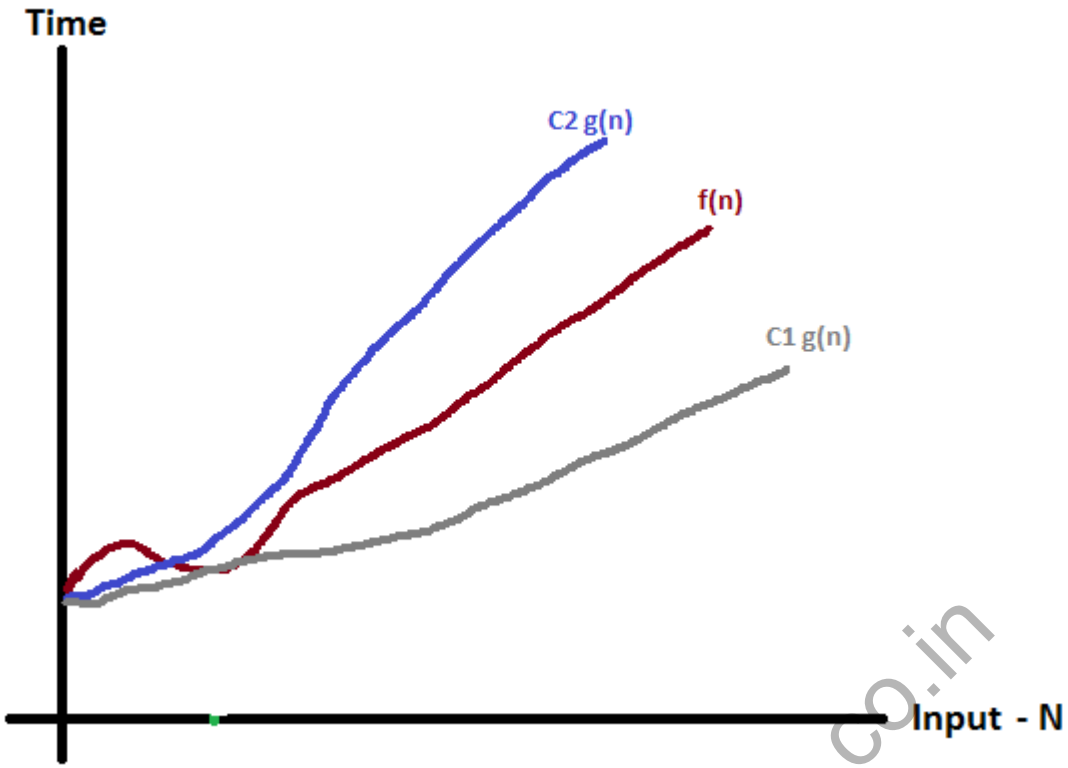
3. Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity. That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.
Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) <= C_2 g(n)

⇒C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.

By using Big - Theta notation we can represent the time compexity as follows...

3n + 2 = Θ(n)

3. Performance Analysis

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

The Running time of a program

When solving a problem, we are faced with a choice among algorithms. The basis for this can be any one of the following:

i. We would like an algorithm that is easy to understand code and debug.

ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.

2. The quality of code generated by the compiler used to create the object program.

3. The nature and speed of the instructions on the machine used to execute the program,

4. The time complexity of the algorithm underlying the program

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0

The total time will be 2n+3

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: Instruction space is the space needed to store the compiled version of the program instructions. The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

Example 2:

Algorithm sum(a,n)

{

s=0.0;

for I=1 to n do

s= s+a[I];

return s;

}

- The problem instances for this algorithm are characterized by n, the number of elements to be summed. The space needed by 'n' is one word, since it is of type integer.
- The space needed by 'a' is the space needed by variables of type array of floating-point numbers. This is at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So, we obtain Sp(sum(n)>=(n+s))[n for a[ ], one each for n, I a& s]

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases is:

1. Best Case: The minimum possible value of f(n) is called the best case.

2. Average Case: The expected value of f(n).

3. Worst Case: The maximum value of f(n) for any key possible input

Q) Introduction to Algorithm Analysis

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

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. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

1. Criteria for Algorithm Analysis

The criteria for algorithm analysis primarily include time complexity, which measures the time taken to execute an algorithm, and space complexity, which measures the total memory space required by an algorithm.

We will measure the following to analysis an algorithm

A. Correctness
B. Amount of work done
C. Amount of space used
D. Simplicity, clarity
E. Optimality

1. Correctness

- **Definition:** Does the algorithm produce the correct output for all valid inputs?
- **Example:** A sorting algorithm is correct if it consistently arranges elements in ascending or descending order for any given input array.
- **Importance:** Correctness is the fundamental requirement of any algorithm. An incorrect algorithm is useless, no matter how efficient it is.

2. Amount of Work Done (Time Complexity)

- **Definition:** How many operations does the algorithm perform as a function of the input size?
- **Example:** Bubble sort has a time complexity of O(n ^ 2), meaning the number of comparisons and swaps it makes grows quadratically with the input size.
- **Importance:** Time complexity is crucial for understanding how an algorithm scales with larger inputs. Algorithms with lower time complexity are generally more efficient.

3. Amount of Space Used (Space Complexity)

- **Definition:** How much memory does the algorithm require as a function of the input size?
- **Example:** In-place sorting algorithms, like bubble sort, have a space complexity of O(1) because they modify the input array directly and don't require additional data structures.
- **Importance:** Space complexity is important for memory-constrained environments. Algorithms that use excessive space might not be practical for large inputs.

4. Simplicity, Clarity

- **Definition:** Is the algorithm easy to understand and implement?

- **Example:** A recursive algorithm for calculating factorials might be less clear than an iterative version.
- **Importance:** Clear algorithms are easier to maintain, debug, and adapt. A complex algorithm might be more difficult to implement correctly.

5. Optimality

- **Definition:** Is the algorithm the best possible solution in terms of time or space complexity?
- **Example:** Merge sort is considered optimal for comparison-based sorting algorithms because it has a lower worst-case time complexity (O(n log n)) than most other sorting algorithms.
- **Importance:** Optimality is often a desirable goal, but it can be challenging to prove. Finding the optimal algorithm for a given problem is a significant research area.

In short, a well-designed algorithm should be:

- **Correct:** It produces the correct output.
- **Efficient:** It has reasonable time and space complexity.
- **Clear:** It is easy to understand and implement.
- **Optimal (if possible):** It is the best possible solution for the given problem.

2.Methodology & Types of analysis OR Approaches to measure time complexity.

There are two approaches to measure the time complexity of an algorithm.

1. Priori analysis
2. Posteriori analysis.

1. In **priori analysis**, before the algorithm is executed, the behavior of the algorithm is analyzed. A priori analysis concentrates on determining the order of execution of statements.
2. In **posteriori analysis**, while the algorithm is executed, the execution time is measured. Posteriori analysis gives accurate values but it is very costly.

Priori Analysis		Posteriori Analysis	
(i)	The time taken for executing the algorithm is analyzed prior to the execution of the algorithm.	(i)	The execution time taken by an algorithm is evaluated while the algorithm is being executed.
(ii)	It is also known as performance analysis that evaluates whether the code is readable or if it performs the desired functions.	(ii)	It is also known as performance measurement that measures the accuracy of the algorithm.
(iii)	It focuses on determining the order of execution of statement.	(iii)	It focuses on determining the space and time complexity of a particular algorithm.
(iv)	It provides approximate values.	(iv)	It provides accurate values.
(v)	It is less expensive than posteriori analysis.	(v)	It is very expensive analysis.

3. Frame work for analyzing recursive algorithms

When analyzing recursive algorithms, it's essential to consider the following key aspects:

1. Base Case:

- **Identification:** Clearly identify the base case(s) where the recursion terminates.
- **Correctness:** Verify that the base case(s) return the correct result.

2. Recursive Case:

- **Breakdown:** Ensure that the recursive case breaks down the problem into smaller, simpler subproblems.
- **Correctness:** Verify that the recursive calls correctly solve the subproblems.
- **Convergence:** Ensure that the recursive calls eventually reach the base case(s).

3. Time Complexity:

- **Recurrence Relation:** Express the time complexity as a recurrence relation, relating the time taken for an input size n to the time taken for smaller input sizes.
- **Solving the Recurrence:** Solve the recurrence relation using techniques like the Master Theorem, substitution method, or iteration method.

4. Space Complexity:

- **Stack Space:** Analyze the maximum depth of the recursion call stack to determine the space complexity.
- **Additional Space:** Consider any additional space used by the algorithm, such as temporary variables or data structures.

Example: Factorial Calculation

Recursive Function:

Python

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Analysis:

1. **Base Case:**
    - **Identification:** The base case is when n equals 0.
    - **Correctness:** The factorial of 0 is defined as 1, so the base case is correct.
2. **Recursive Case:**
    - **Breakdown:** The recursive case calculates the factorial of n by multiplying n with the factorial of n-1.
    - **Correctness:** The recursive call correctly solves the subproblem of calculating the factorial of n-1.
    - **Convergence:** The recursive calls eventually reach the base case when n becomes 0.
3. **Time Complexity:**
    - **Recurrence Relation:** T(n) = T(n-1) + O(1) (for the multiplication and function call)
    - **Solving:** Using the Master Theorem, the time complexity is O(n!).
4. **Space Complexity:**
    - **Stack Space:** The maximum depth of the recursion stack is n.

- **Additional Space:** O(1) for the constant space used by the function call and the multiplication operation.

4. Frame work for analyzing non-recursive algorithms

When analyzing non-recursive algorithms, focus on the following key aspects:

1. Algorithm Structure:

- **Loop Structure:** Identify the main loops and their nesting levels.
- **Data Structures:** Analyze the data structures used and their operations.

2. Time Complexity:

- **Loop Iterations:** Count the number of iterations in each loop and their dependence on the input size.
- **Operation Cost:** Estimate the cost of operations within the loops, such as comparisons, assignments, or arithmetic operations.
- **Overall Complexity:** Determine the overall time complexity based on the dominant loop and operation costs.

3. Space Complexity:

- **Data Structures:** Analyze the space requirements of the data structures used.
- **Auxiliary Space:** Consider any additional space used by the algorithm, such as temporary variables or data structures.

Example: Bubble Sort

Algorithm:

Python

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1],    arr[j]
```

Analysis:

1. **Algorithm Structure:**
    - **Loops:** Two nested loops. The outer loop iterates n times, and the inner loop iterates n-i-1 times for each outer loop iteration.
    - **Data Structures:** An array is used to store the elements.
2. **Time Complexity:**
    - **Loop Iterations:** The outer loop iterates n times, and the inner loop iterates n-i-1 times for each outer loop iteration.
    - **Operation Cost:** The comparison and potential swap operations within the inner loop have a constant cost.

- **Overall Complexity:** The total number of comparisons is O(n^2), and the total number of swaps depends on the initial order of the elements but is also O(n^2) in the worst case. Therefore, the time complexity of bubble sort is O(n^2).

3. **Space Complexity:**
    - **Data Structures:** The array itself requires O(n) space.
    - **Auxiliary Space:** The algorithm uses a constant amount of additional space for variables like i, j, and the temporary variables for swapping.

……..

Q) How to measure the performance of an algorithm?

In algorithm analysis, the best, worst, and average cases refer to the different scenarios under which an algorithm's performance can vary, impacting its efficiency. The best case represents the fastest possible runtime, the worst case represents the slowest possible runtime, and the average case represents the runtime expected for a typical input.

Best Case:

- The input that leads to the algorithm completing its task in the least amount of time.
- Represents the most optimal scenario, where the algorithm can achieve the fastest possible performance.
- For example, in a searching algorithm, the best case might be when the desired item is found at the beginning of the list.

Worst Case:

- The input that leads to the algorithm taking the most time to complete its task.
- Represents the least favourable scenario, where the algorithm may encounter the highest possible number of operations.
- For example, in a sorting algorithm, the worst case might be an input that is already sorted in reverse order.

Average Case:

- The average runtime of the algorithm across various inputs, assuming a typical or random input distribution.
- Represents the expected runtime under normal conditions, where the algorithm performs neither its best nor worst.
- For example, in a searching algorithm, the average case might be when the desired item is found approximately in the middle of the list.

AVL Trees – Creation, Insertion, Deletion operations and Applications

……

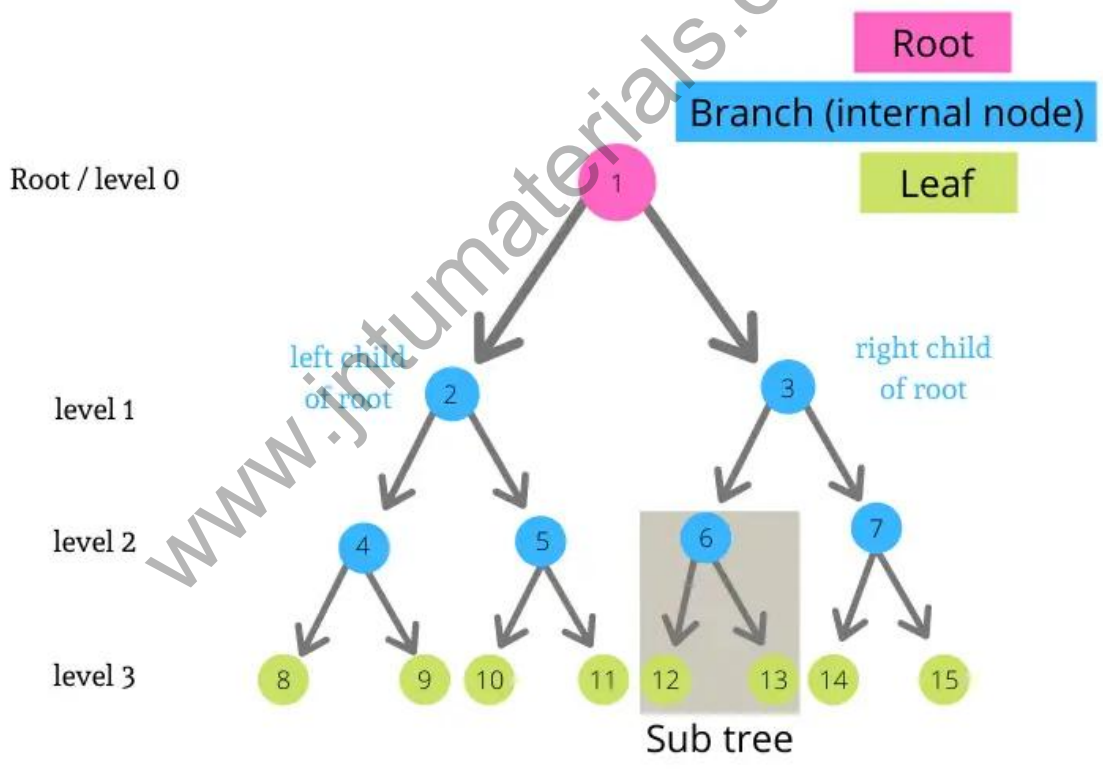
Introduction

1.Height of a tree in data structure

In a tree data structure, the height refers to the maximum number of edges between the root node and any leaf node in the tree. It's essentially the length of the longest path from the root to a leaf. The height of a node is the number of edges on the longest path from that node to a leaf.

Here's a more detailed explanation:

- **Root Node:** The root node is the topmost node in the tree, and it's considered to be at level 0.
- **Leaf Node:** A leaf node is a node with no children.
- **Depth of a Node:** The depth of a node is the number of edges from the node to the root.
- **Height of a Tree:** The height of a tree is the maximum depth of any node in the tree, which is also the number of edges on the longest path from the root to a leaf.



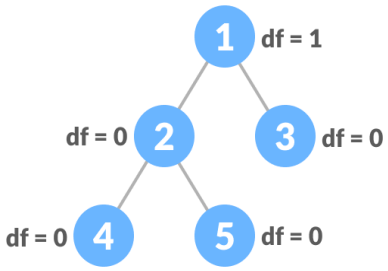
Tree Data Structure

Height of a binary tree is the maximum depth of the tree. As you can see in the above diagram the height of it is 4 because it contains 4 levels of depth.

2. Balanced Trees

A balanced tree in data structures is a binary search tree (BST) where the heights of the left and right subtrees of any node differ by at most one. This ensures that the tree maintains a logarithmic height, enabling efficient search, insertion, and deletion operations.

Key Characteristics of Balanced Trees:



a) Self-Balancing:

They automatically rebalance themselves after insertions or deletions to maintain the height constraint.

b) Logarithmic Height:

The height of a balanced tree grows logarithmically with the number of nodes, making operations efficient.

c) Efficient Operations:

Search, insertion, and deletion operations have a time complexity of O(log n), where n is the number of nodes.

Examples:

AVL trees, Red-Black trees, and B-trees are examples of balanced tree implementations.

.....

1.AVL Trees

AVL tree in data structures is a popular self-balancing binary search tree where the difference between the heights of left and right subtrees for any node does not exceed unity. It was introduced by Georgy Adelson-Velsky and Evgenii Landis in 1962, hence the name AVL. It automatically adjusts its structure to maintain the minimum possible height after any operation with the help of a balance factor for each node.

Balance Factor in AVL Tree in Data Structures

The balance factor of a node in an AVL Tree is a numerical value that represents the difference in height between the left and right subtrees of that node. It is the extra information used to determine the tree's balance. The balance factor is calculated as follows:

Balance Factor = height(left subtree) - height(right subtree)

1. If the balance factor of any node = 1, the left sub-tree is one level higher than the right sub-tree i.e. the given node is left-heavy.
2. If the balance factor of any node = 0, the left sub-tree and right sub-tree are of equal height.

For leaf nodes, the balance factor is 0 as they do not contain any subtrees.

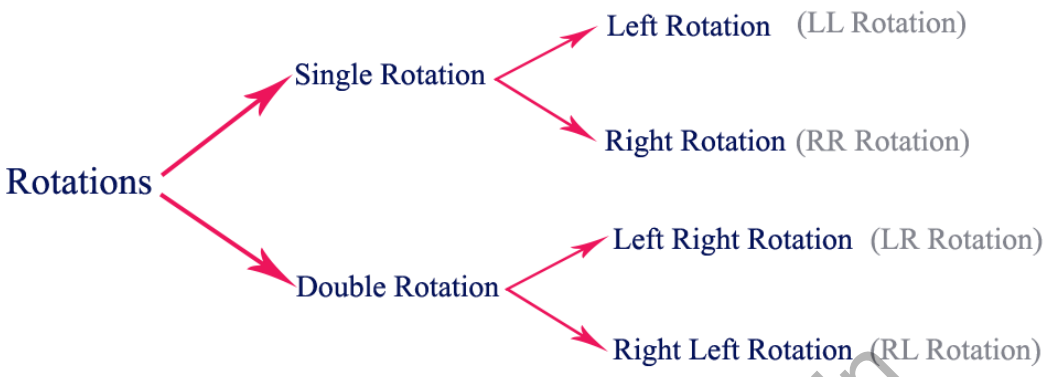
3. If the balance factor of any node = -1, the left sub-tree is one level lower than the right sub-tree i.e. the given node is right-heavy.

Hence, in this way, the self-balancing property of an AVL tree is maintained by the balance factor. Thus, we can find an unbalanced node in the tree and locate where the height-affecting operation was performed that caused the imbalance of the tree.

AVL Tree Rotations

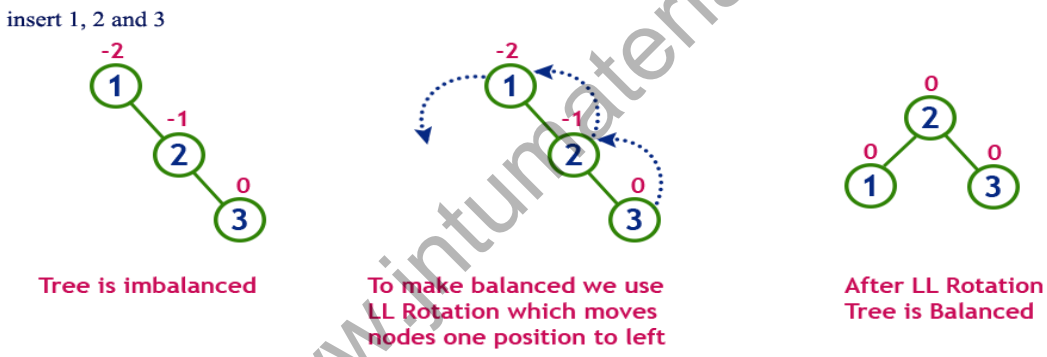
AVL tree rotation is a fundamental operation used in self-balancing binary search trees, specifically in AVL trees. Due to any operations like insertion or deletion, if any node of an AVL tree becomes unbalanced, specific tree rotations are performed to restore the balance.

The tree rotations involve rearranging the tree structure without changing the order of elements. The positions of the nodes of a subtree are interchanged. There are four types of AVL rotations:



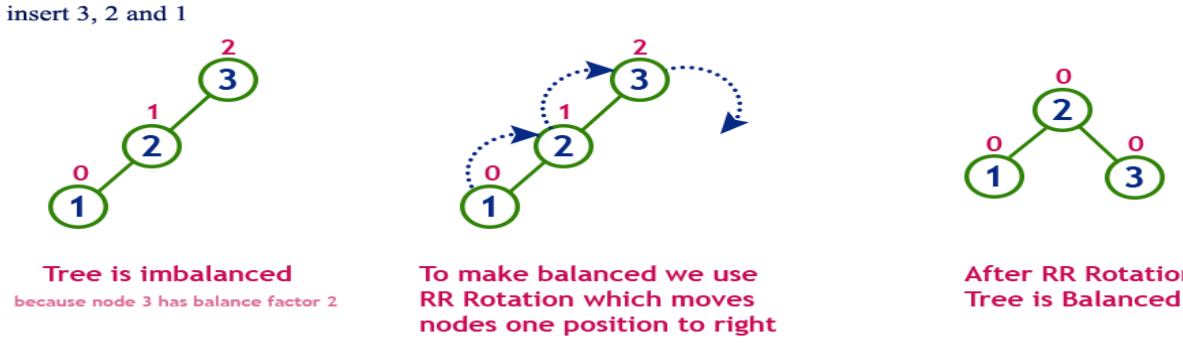
Single Left Rotation (LL Rotation)

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



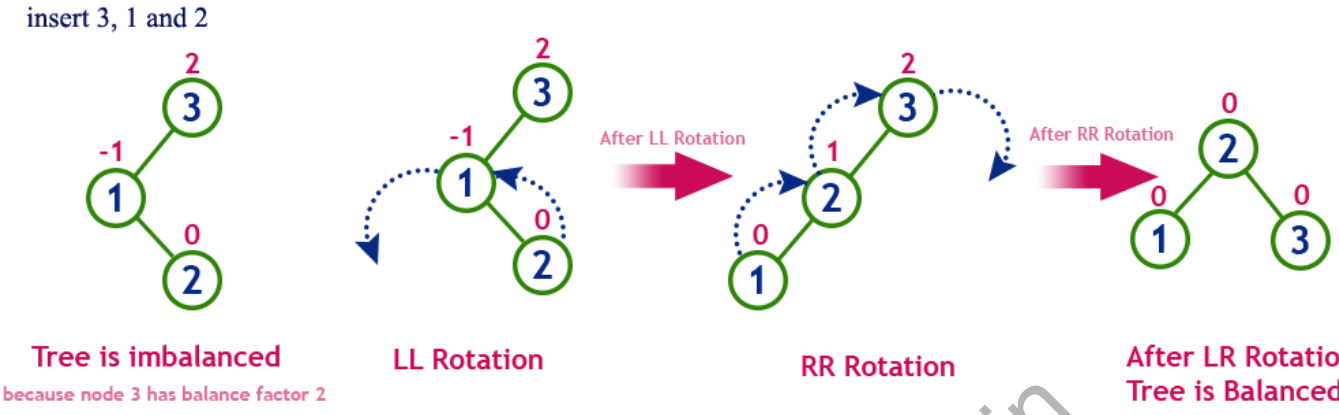
Single Right Rotation (RR Rotation)

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



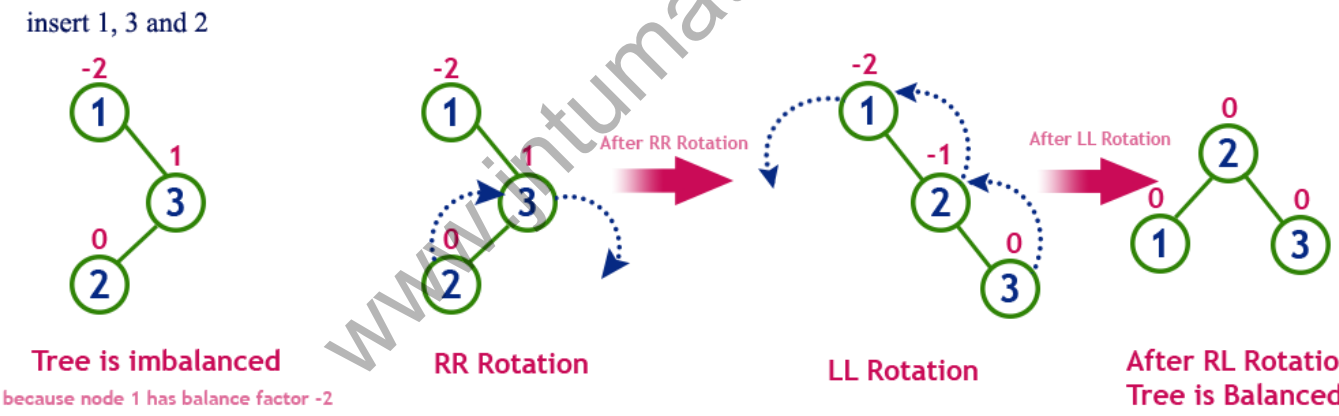
Left Right Rotation (LR Rotation)

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



Right Left Rotation (RL Rotation)

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



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

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

- **Step 1 -** Read the search element from the user.

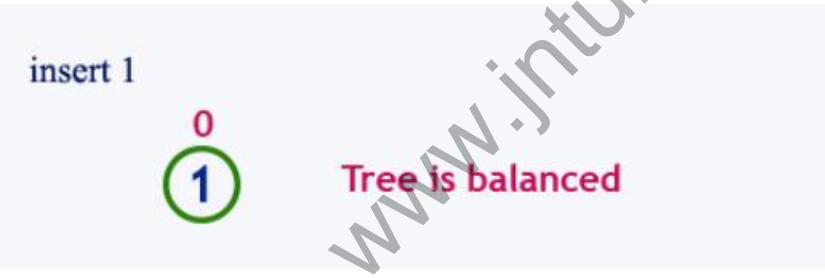
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

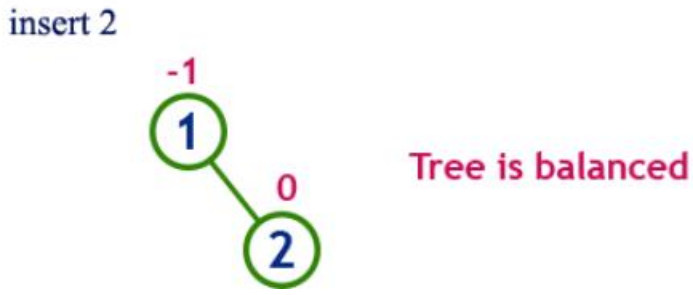
Insertion Operation in AVL Tree

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

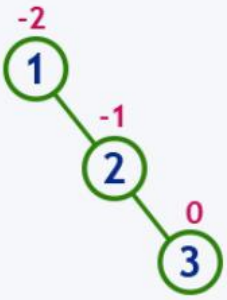
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

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

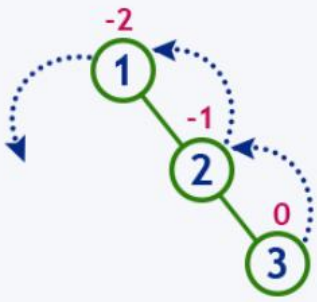




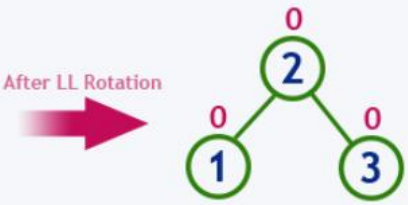
insert 3



Tree is imbalanced

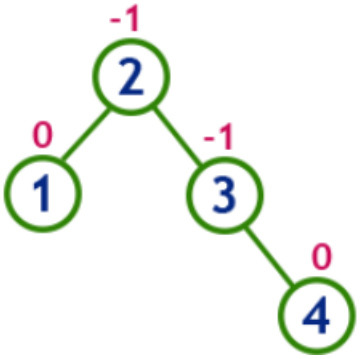


LL Rotation



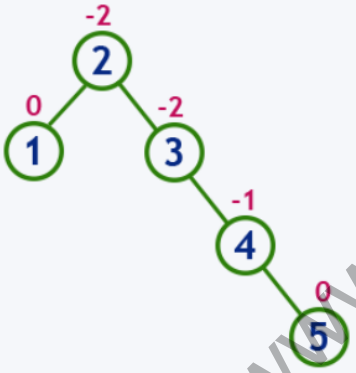
Tree is balanced

insert 4



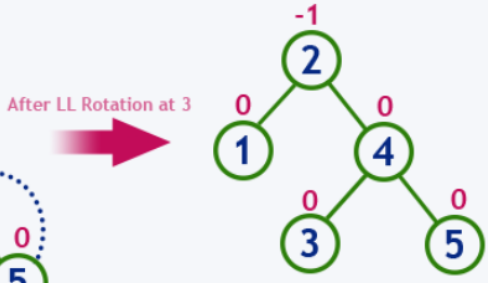
Tree is balanced

insert 5



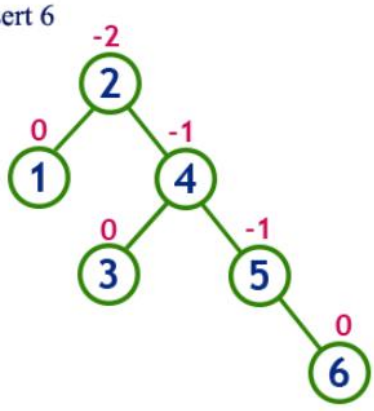
Tree is imbalanced

LL Rotation at 3

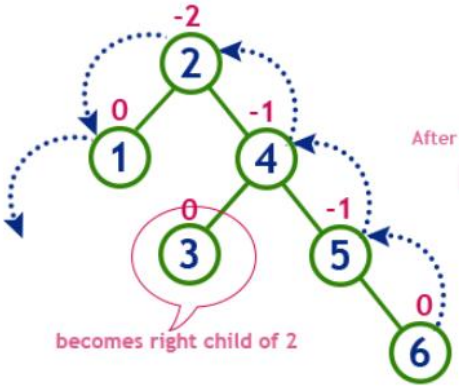


Tree is balanced

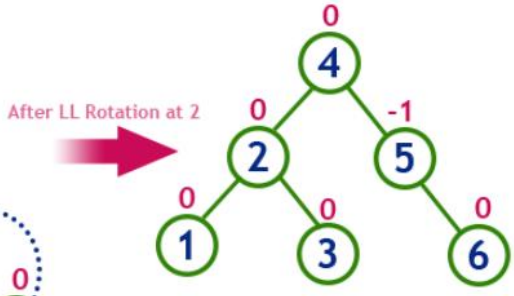
insert 6



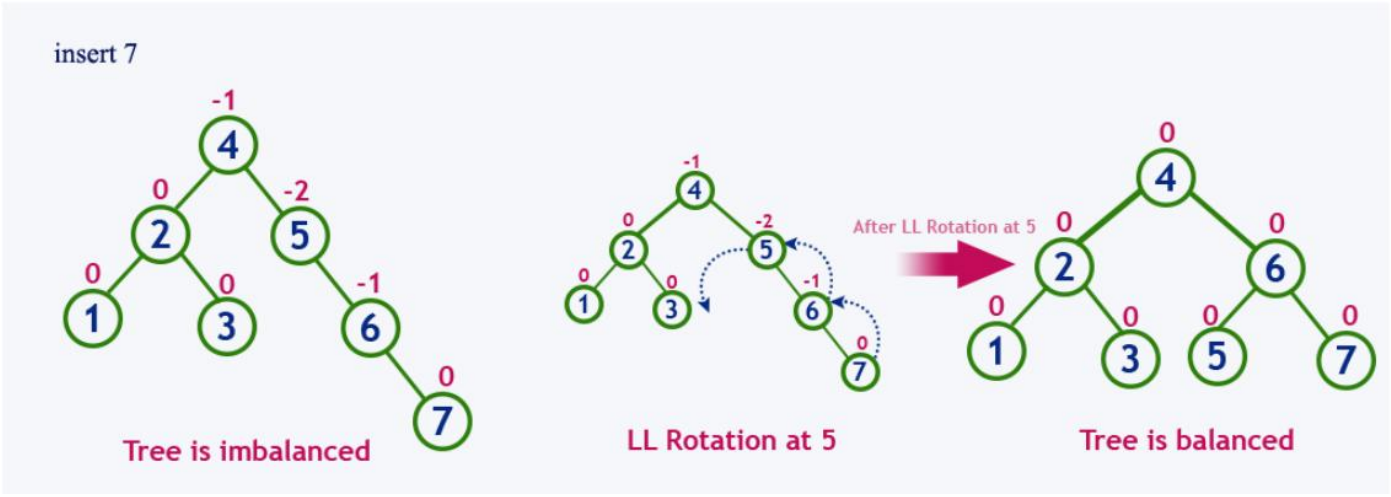
Tree is imbalanced

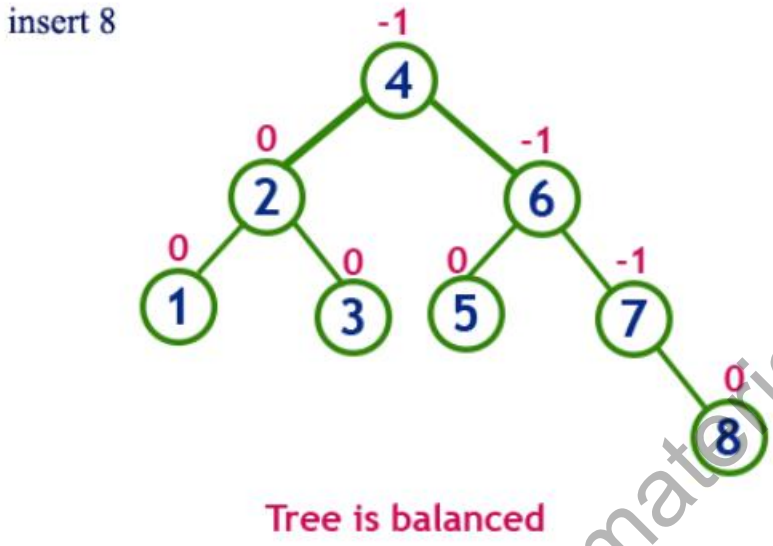


LL Rotation at 2



Tree is balanced





Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Applications of AVL Trees

- **Databases and File Systems**: For indexing and maintaining sorted data with quick lookups.
- **Memory Management**: Used in dynamic memory allocation (e.g., buddy memory allocation).
- **Network Routing Algorithms**: For quick IP lookup operations.
- **Compiler Design**: Symbol table implementations.
- **Gaming and Real-time systems**: Where fast lookups and updates are needed.

Advantages of AVL Trees

- Faster lookups than simple BSTs due to balance guarantee.
- Ensures **O(log n)** time complexity for insert, delete, and search.

Disadvantages

- Slightly more complex due to rotation logic.
- More pointer manipulation and memory usage (extra space for height or balance factor).

……

Q) Difference between Binary Search Tree and AVL Tree

Feature	Binary Search Tree (BST)	AVL Tree
Definition	A hierarchical data structure where left < root < right	A self-balancing BST where balance factor is maintained at each node
Balance Condition	No specific balance condition	Balance factor (height difference of left and right subtree) is -1, 0, or +1
Balancing Mechanism	Not self-balancing; can become skewed	Automatically balances using rotations (LL, RR, LR, RL)
Rotations	Not required	Required to maintain balance after insertions/deletions
Search Time Complexity	Best: O(log n), Worst: O(n) (in skewed trees)	Always O(log n) due to guaranteed balance
Insertion/Deletion Cost	O(h), h may be up to n	O(log n), with additional cost for rotations
Height of Tree	Can be up to n in worst case	Always maintained at O(log n)
Use Case	Simple and fast to implement when balanced input is expected	Used when consistently fast search, insert, delete are required

CHAPTER-III

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications

……

Introduction

1. Binary Tree

A binary tree is a hierarchical data structure where each node has at most two children, referred to as the left child and the right child.

Types of Binary Trees

- **Full Binary Tree:** Each node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are completely filled except possibly the last level and all nodes in the last level are as far left as possible.
- **Perfect Binary Tree:** All internal nodes have two children and all leaves are at the same level.
- **Balanced Binary Tree:** The height difference between the left and right subtrees of every node is at most 1.
- **Skewed Binary Tree:** All nodes except one have only one child, forming a linear chain.
- **Binary Search Tree (BST):** The value of each node is greater than or equal to all values in its left subtree and less than or equal to all values in its right subtree.

2. Heap Tree

A Heap is a complete binary tree where each node satisfies the heap property:

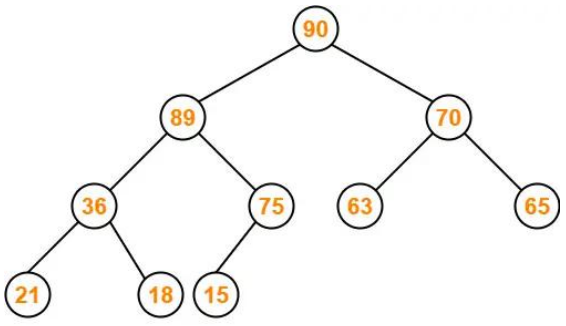
- **Max Heap**: Parent node ≥ its children
- **Min Heap**: Parent node ≤ its children

These are used to implement priority queues, which support fast insertion and deletion of the element with the highest (or lowest) priority.

1. Max Heap-

- Max Heap conforms to the above properties of heap.
- In max heap, every node contains greater or equal value element than its child nodes.
- Thus, root node contains the largest value element.

Example-

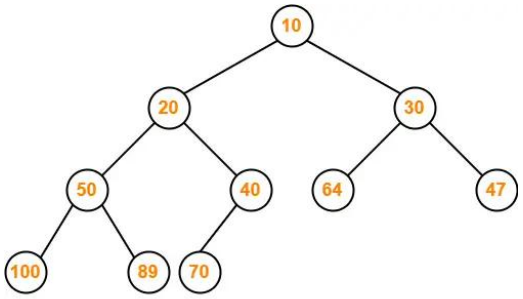


Max Heap Example

2. Min Heap-

- Min Heap conforms to the above properties of heap.
- In min heap, every node contains lesser value element than its child nodes.
- Thus, root node contains the smallest value element.

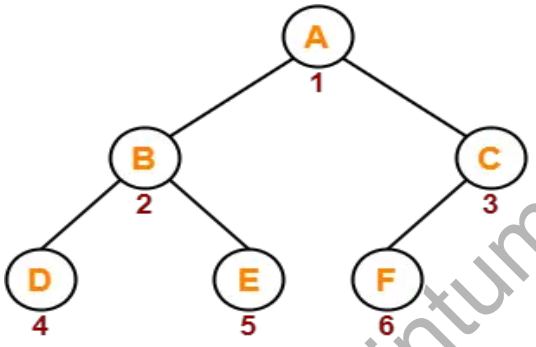
Example



Min Heap Example

Array Representation of Binary Heap-

A binary heap is typically represented as an array.



Binary Heap

Array Representation

For a node present at index 'i' of the array Arr-

If indexing starts with 0,

- Its parent node will be present at array location = Arr [i/2]
- Its left child node will be present at array location = Arr [2i+1]
- Its right child node will be present at array location = Arr [2i+2]

If indexing starts with 1,

- Its parent node will be present at array location = Arr [⌊i/2⌋]
- Its left child node will be present at array location = Arr [2i]
- Its right child node will be present at array location = Arr [2i+1]

Important Notes-

Note-01:

- Level order traversal technique may be used to achieve the array representation of a heap tree.
- Array representation of a heap never contains any empty indices in between.
- However, array representation of a binary tree may contain some empty indices in between.

Note-02:

Given an array representation of a binary heap,

- If all the elements are in descending order, then heap is definitely a max heap.
- If all the elements are not in descending order, then it may or may not be a max heap.
- If all the elements are in ascending order, then heap is definitely a min heap.
- If all the elements are not in ascending order, then it may or may not be a min heap.

Note-03:

- In max heap, every node contains greater or equal value element than all its descendants.
- In min heap, every node contains smaller value element that all its descendants.

Problems-

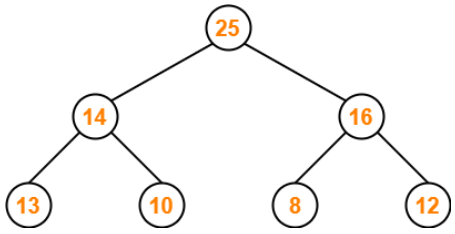
Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap? (GATE CS 2009)

1. 25, 14, 16, 13, 10, 8, 12
2. 25, 12, 16, 13, 10, 8, 14
3. 25, 14, 12, 13, 10, 8, 16
4. 25, 14, 13, 16, 10, 8, 12

Solutions-

Option-1: 25, 14, 16, 13, 10, 8, 12

The given array representation may be converted into the following structure-



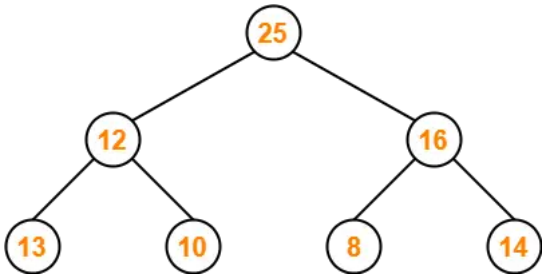
Clearly,

- It is a complete binary tree.
- Every node contains a greater value element than its child nodes.

Thus, the given array represents a max heap.

Option-2: 25, 12, 16, 13, 10, 8, 14-

The given array representation may be converted into the following structure-



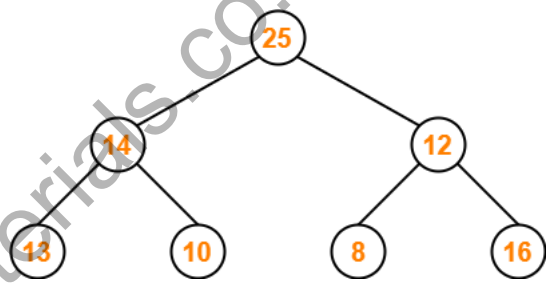
Clearly,

- It is a complete binary tree.
- Every node does not contain a greater value element than its child nodes. (Node 12)
- So, it is not a max heap.
- Every node does not contain a smaller value element than its child nodes.
- So, it is not a min heap.

Thus, the given array does not represent a heap.

Option-3: 25, 14, 12, 13, 10, 8, 16-

The given array representation may be converted into the following structure-



Clearly,

- It is a complete binary tree.
- Every node does not contain a greater value element than its child nodes. (Node 12)
- So, it is not a max heap.
- Every node does not contain a smaller value element than its child nodes.
- So, it is not a min heap.

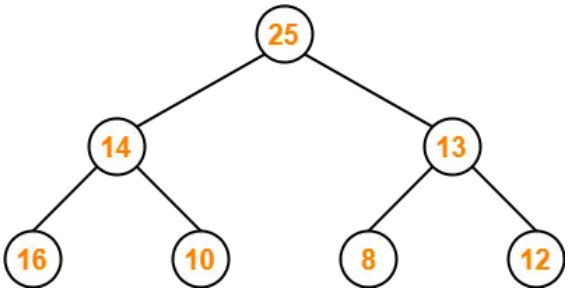
Thus, the given array does not represent a heap.

Option-4: 25, 14, 13, 16, 10, 8, 12-

The given array representation may be converted into the following structure-

Clearly,

- It is a complete binary tree.
- Every node does not contain a greater value element than its child nodes. (Node 14)
- So, it is not a max heap.
- Every node does not contain a smaller value element than its child nodes.
- So, it is not a min heap.



Thus, the given array does not represents a heap.

3. Heap Operations

The most basic and commonly performed operations on a heap are

Max Heap Operations-

We will discuss the construction of a max heap and how following operations are performed on a max heap-

- Finding Maximum Operation
- Insertion Operation
- Deletion Operation

Max Heap Construction-

Given an array of elements, the steps involved in constructing a max heap are

Step-01:

Convert the given array of elements into an almost complete binary tree.

Step-02:

Ensure that the tree is a max heap.

- Check that every non-leaf node contains a greater or equal value element than its child nodes.
- If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
- Start checking from a non-leaf node with the highest index (bottom to top and right to left).

Finding Maximum Operation-

- In max heap, the root node always contains the maximum value element.
- So, we directly display the root node value as maximum value in max heap.

Insertion Operation-

Insertion Operation is performed to insert an element in the heap tree.

The steps involved in inserting an element are

Step-01:

Insert the new element as a next leaf node from left to right.

Step-02:

Ensure that the tree remains a max heap.

- Check that every non-leaf node contains a greater or equal value element than its child nodes.
- If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
- Start checking from a non-leaf node with the highest index (bottom to top and right to left).

Deletion Operation-

Deletion Operation is performed to delete a particular element from the heap tree.

When it comes to deleting a node from the heap tree, following two cases are possible-

Case-01: Deletion Of Last Node-

- This case is pretty simple.
- Just remove / disconnect the last leaf node from the heap tree.

Case-02: Deletion Of Some Other Node-

- This case is little bit difficult.
- Deleting a node other than the last node disturbs the heap properties.

The steps involved in deleting such a node are-

Step-01:

- Delete the desired element from the heap tree.
- Pluck the last node and put in place of the deleted node.

Step-02:

Ensure that the tree remains a max heap.

- Check that every non-leaf node contains a greater or equal value element than its child nodes.
- If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
- Start checking from a non-leaf node with the highest index (bottom to top and right to left).

Example:

Construct a max heap for the given array of elements-

1, 5, 6, 8, 12, 14, 16

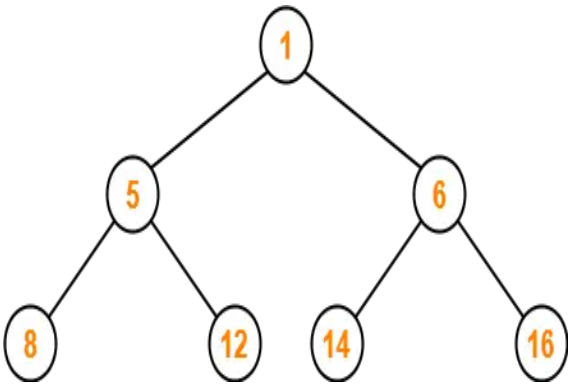
Solution-

Step-01:

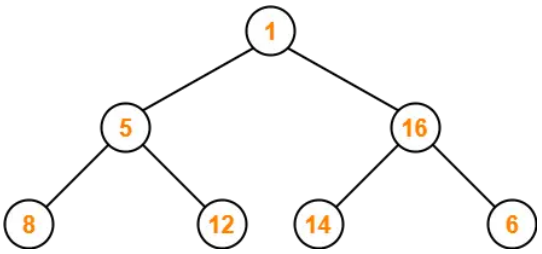
We convert the given array of elements into an almost complete binary tree-

Step-02:

- We ensure that the tree is a max heap.
- Node 6 contains greater element in its right child node.
- So, we swap node 6 and node 16.



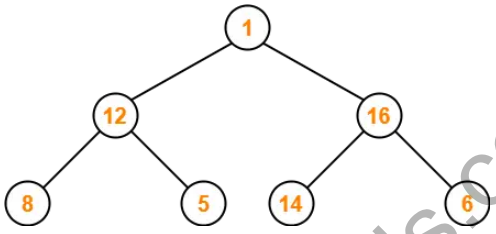
The resulting tree is-



Step-03:

- Node 5 contains greater element in its right child node.
- So, we swap node 5 and node 12.

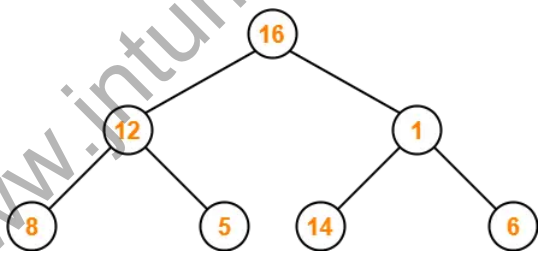
The resulting tree is-



Step-04:

- Node 1 contains greater element in its right child node.
- So, we swap node 1 and node 16.

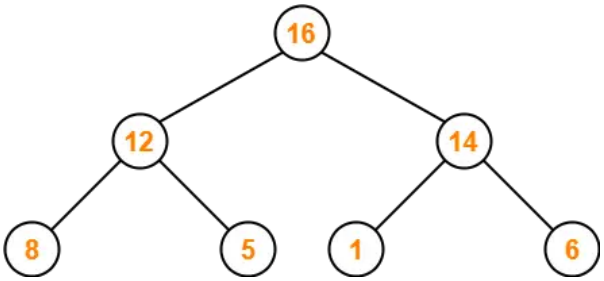
The resulting tree is-



Step-05:

- Node 1 contains greater element in its left child node.
- So, we swap node 1 and node 14.

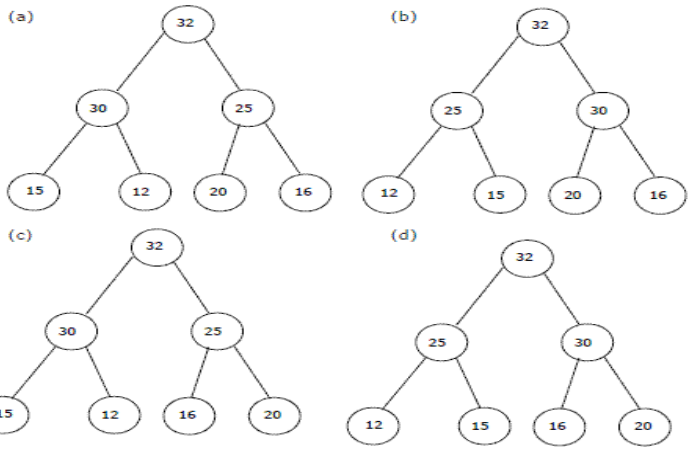
The resulting tree is-



This is the required max heap for the given array of elements.

Previously Asked GATE Questions on Binary Heap

Question 1: The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a Max Heap. The resultant Max Heap is.



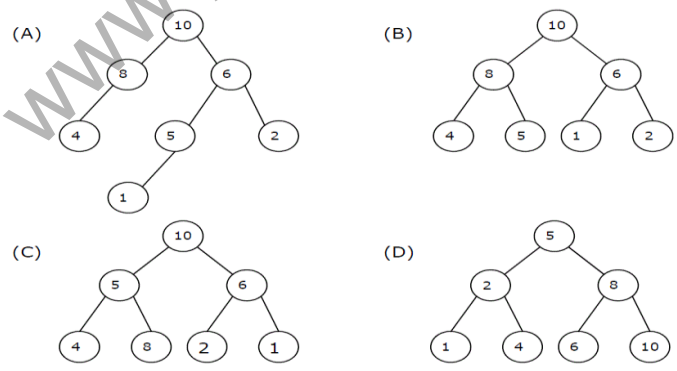
Question 2: In a heap with n elements with the smallest element at the root, the 7th smallest element can be found in time (GATE CS 2003)

a) Θ(n log n)
b) Θ(n)
c) Θ(log n)
d) Θ(1)

Question 3: The number of elements that can be sorted in Θ(logn) time using heap sort is

(a) Θ(1)
(b) Θ(sqrt(logn))
(c) Θ(Log n/(Log Log n))
(d) Θ(Log n)

Question 4: A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?



Question 5: A 3-ary max heap is like a binary max heap, but instead of 2 children, nodes have 3 children. A 3-ary heap can be represented by an array as follows: The root is stored in the first location, a[0], nodes in the next level, from left to right, is stored from a[1] to a[3]. The nodes from the second level of the tree from left to right are stored from a[4] location onward. An item x can be inserted into a 3-ary heap containing n items by placing x in the location a[n] and pushing it up the tree to satisfy the heap property.

Which one of the following is a valid sequence of elements in an array representing 3-ary max heap?
(a) 1, 3, 5, 6, 8, 9
(b) 9, 6, 3, 1, 8, 5

(c) 9, 3, 6, 8, 5, 1
(d) 9, 5, 6, 8, 3, 1

Question 6: Suppose the elements 7, 2, 10 and 4 are inserted, in that order, into the valid 3- ary max heap found in the above question, Which one of the following is the sequence of items in the array representing the resultant heap?

(a) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4
(b) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
(c) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3
(d) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

Question 7: Consider the process of inserting an element into a Max Heap, where the Max Heap is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is:

(a) Θ(logn)
(b) Θ(LogLogn)
(c) Θ(n)
(d) Θ(nLogn)

Question 8: In a binary max heap containing n numbers, the smallest element can be found in time

(GATE CS 2006)

(a) 0(n)
(b) O(logn)
(c) 0(loglogn)
(d) 0(1)

Question 9: A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is: 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

(a) 10, 8, 7, 3, 2, 1, 5
(b) 10, 8, 7, 2, 3, 1, 5
(c) 10, 8, 7, 1, 2, 3, 5
(d) 10, 8, 7, 5, 3, 2, 1

Question 10: Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4.

Array Index	1	2	3	4	5	6	7	8	9
Value	40	30	20	10	15	16	17	8	7

Now consider that a value 35 is inserted into this heap. After insertion, the new heap is

(a) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35
(b) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
(c) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15
(d) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

UNIT-I Questions

Part A – 2 Marks Questions (Short Answer Type)

1. Define Time Complexity with an example.
2. What is Space Complexity? How is it calculated?
3. List the types of Asymptotic Notations and their meanings.
4. Write the Big O notation for Binary Search and Selection Sort.
5. What is an AVL Tree?
6. Define Balance Factor in an AVL Tree.
7. Mention two rotations used in AVL Trees.
8. What is a Min Heap and a Max Heap?
9. Define Heap Property with an example.
10. List any two applications of Priority Queues.
11. What is the time complexity of insertion in a heap?
12. Define a complete binary tree.
13. Differentiate between AVL Tree and Binary Search Tree (BST).
14. What is the purpose of heapify operation in a heap?

Part B – 14 Marks Questions (Essay/Problem Type)

1. Explain the different asymptotic notations (Big O, Theta, Omega) with graphical and code examples.
2. Discuss space and time complexity analysis with suitable examples.
3. Explain the creation, insertion, and deletion operations in AVL Trees with examples and rotation cases (LL, RR, LR, RL).
4. Illustrate the AVL Tree insertion process for the following sequence: 30, 20, 10, 25, 40, 50. Show rotation steps.
5. What are Min and Max Heaps? Explain the insert and delete operations with heap trees and array representation.
6. Construct a Min Heap for the elements: 10, 15, 20, 17, 25. Show all intermediate steps.
7. Write an algorithm to implement a priority queue using a max heap. Explain its time complexity.
8. Compare and contrast AVL Trees and Heaps in terms of structure, purpose, and operations with examples.