**DSA Project**

**Instructions:**

* Indent your code.
* Comment your code.
* Use meaningful variable names.
* Plan your code carefully on a piece of paper before you implement it.
* Name of the program should be same as the task name. i.e. the first program should be Task\_1.cpp

# void main() is not allowed. Use int main()

* **You have to work in multiple files. i.e separate .h and .cpp files**

# You are not allowed to use system("pause")

* **You are not allowed to use any built-in functions**

# You are required to follow the naming conventions as follow:

* + **Variables:** firstName; (no underscores allowed)
  + **Function:** getName(); (no underscores allowed)
  + **ClassName:** BankAccount (no underscores allowed)

# Students are required to complete the project within the given time.

**Project Title:** **AVL Tree Implementation and Operations**

***Implement BST Tree and maintain the tree using AVL technique.***

In this project, you need to implement AVL tree techniques to efficiently manage and manipulate binary search trees. AVL trees are self-balancing binary search trees that provide efficient insertion, deletion, and search operations while maintaining balance, ensuring optimal performance in various applications.

The project will include the implementation of various functions such as insertion, deletion, searching, traversals (inorder, preorder, postorder), height calculation etc. These functions will be designed to handle AVL trees of arbitrary size. Furthermore, the project will emphasize the importance of maintaining the balance property of AVL trees and will implement mechanisms to ensure that the tree remains balanced after every operation

In this project, you need to implement the following given functions while keep maintaining the AVL properties.

* **Insertion:**

Inserting a new node into the AVL tree while maintaining the AVL property, i.e., ensuring that the balance factor of every node is either -1, 0, or 1.

* **Deletion:**

Removing a node from the AVL tree while maintaining balance. After deletion, the tree must still satisfy the AVL property.

* **Search:**

Searching for a specific key or value in the AVL tree. This operation utilizes the binary search property of the tree to efficiently find the desired node.

* **Traversals:**
* **Inorder Traversal**:

Visit the nodes of the tree in sorted order.

* **Preorder Traversal:**

Visit the current node before its children.

* **Postorder Traversal:**

Visit the children of the current node before the node itself.

* **Height Calculation:**

Calculating the height of the AVL tree, which is the maximum number of edges in any path from the root node to a leaf node. This operation is crucial for determining the balance factor of each node.

* **Balancing Factor Calculation:**

Calculating the balance factor of each node in the AVL tree. The balance factor is defined as the height of the right subtree minus the height of the left subtree. It helps identify which nodes violate the AVL property.

* **Rotation Operations:**
* **Single Rotation (Left or Right):**

Perform a rotation to restore balance in a subtree that has become unbalanced due to an insertion or deletion.

* **Double Rotation (Left-Right or Right-Left):**

Perform a combination of two single rotations to restore balance in a subtree.

* **Diameter Calculation:**

Finding the diameter of the AVL tree, which is the length of the longest path between any two nodes in the tree. This operation provides insights into the overall structure of the tree.

* **Minimum and Maximum Finding:**

Finding the node with the minimum or maximum key value in the AVL tree. These operations are efficient due to the binary search tree property.

* **Successor and Predecessor Finding:**

Finding the successor (smallest key greater than the given key) or predecessor (largest key smaller than the given key) of a node in the AVL tree.