Contribution in ds cp

To divide this project among four contributors, I'll break it into four logical sections:

1) AVL Tree Implementation (Insertion, Deletion, Search, Update) 2) Stock Price Prediction Model (Linear & Polynomial Regression, Confidence Intervals) 3) Visualization & Graph Plotting (AVL Tree Visualization & Prediction Graphs) 4) User Interface & Main Logic (Handling User Inputs & Integrating Components)

1. AVL Tree Implementation (Contributor 1) -> Implements AVL tree structure for storing stock prices. -> Supports insertion, deletion, search, and update operations.

```
#  code for contributor1
class Node:
    def __init__(self, price):
        self.price = price
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.get_height(node.left) - 
self.get_height(node.right) if node else 0

    def rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = max(self.get_height(y.left), 
self.get_height(y.right)) + 1
        x.height = max(self.get_height(x.left), 
self.get_height(x.right)) + 1
        return x

    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = max(self.get_height(x.left), 
self.get_height(x.right)) + 1
        y.height = max(self.get_height(y.left), 
self.get_height(y.right)) + 1
```

```
        return y

    def insert(self, root, price):
        if not root:
            return Node(price)
        if price < root.price:
            root.left = self.insert(root.left, price)
        elif price > root.price:
            root.right = self.insert(root.right, price)
        else:
            return root  # Duplicates not allowed

        root.height = 1 + max(self.get_height(root.left),
self.get_height(root.right))
        balance = self.get_balance(root)

        # Rotations
        if balance > 1 and price < root.left.price:
            return self.rotate_right(root)
        if balance < -1 and price > root.right.price:
            return self.rotate_left(root)
        if balance > 1 and price > root.left.price:
            root.left = self.rotate_left(root.left)
            return self.rotate_right(root)
        if balance < -1 and price < root.right.price:
            root.right = self.rotate_right(root.right)
            return self.rotate_left(root)

        return root

    def delete(self, root, price):
        if not root:
            return root
        if price < root.price:
            root.left = self.delete(root.left, price)
        elif price > root.price:
            root.right = self.delete(root.right, price)
        else:
            if not root.left or not root.right:
                return root.left if root.left else root.right
            temp = self.find_min(root.right)
            root.price = temp.price
            root.right = self.delete(root.right, temp.price)

        if not root:
            return root

        root.height = 1 + max(self.get_height(root.left),
self.get_height(root.right))
        balance = self.get_balance(root)
        if balance > 1 and self.get_balance(root.left) >= 0:
            return self.rotate_right(root)
        if balance > 1 and self.get_balance(root.left) < 0:
            root.left = self.rotate_left(root.left)
            return self.rotate_right(root)
        if balance < -1 and self.get_balance(root.right) <= 0:
            return self.rotate_left(root)
        if balance < -1 and self.get_balance(root.right) > 0:
            root.right = self.rotate_right(root.right)
            return self.rotate_left(root)

        return root

    def find_min(self, node):
        while node.left:
            node = node.left
        return node

    def search(self, root, price):
        if not root or root.price == price:
            return root
        return self.search(root.left, price) if price < root.price
else self.search(root.right, price)
```

1. Stock Price Prediction Model (Contributor 2) -> Implements linear and polynomial regression models. -> Computes prediction intervals.

```
# code for contributor2

import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from scipy import stats

class StockPredictor:
    def __init__(self, confidence_level=0.95):
        self.linear_model = LinearRegression()
        self.poly_model = LinearRegression()
        self.poly_features = PolynomialFeatures(degree=2)
        self.confidence_level = confidence_level

    def train_models(self, stock_prices):
        if len(stock_prices) < 3:
            return False

        self.X = np.array(range(len(stock_prices))).reshape(-1, 1)
        y = np.array(stock_prices)

        self.linear_model.fit(self.X, y)
        self.y_pred_linear = self.linear_model.predict(self.X)

        X_poly = self.poly_features.fit_transform(self.X)
        self.poly_model.fit(X_poly, y)
        self.y_pred_poly = self.poly_model.predict(X_poly)

        return True

    def predict(self, stock_prices, days_ahead=5):
        if not self.train_models(stock_prices):
            return None, None

        future_days = np.array(range(len(stock_prices),
len(stock_prices) + days_ahead)).reshape(-1, 1)
        linear_preds = self.linear_model.predict(future_days)
        poly_preds =
self.poly_model.predict(self.poly_features.transform(future_days))

        return linear_preds, poly_preds
```

1. Visualization & Graph Plotting (Contributor 3) -> Implements AVL tree visualization and stock price trend graphs.

```
# code for Contributor 3

import matplotlib.pyplot as plt
import networkx as nx

def draw_visualization(root, stock_prices, title="Stock Price AVL
Tree"):
    if not root:
        return

    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    time_steps = list(range(1, len(stock_prices) + 1))
    axes[0].bar(time_steps, stock_prices, color='lightblue',
alpha=0.6)
    axes[0].plot(time_steps, stock_prices, marker='o', color='red')
    axes[0].set_title("Stock Prices Over Time")

    G = nx.DiGraph()
    pos = {}

    def build_graph(node, x=0, y=0, layer=1):
        if node:
            G.add_node(node.price)
            pos[node.price] = (x, -y)
            if node.left:
                G.add_edge(node.price, node.left.price)
                build_graph(node.left, x - 1 / layer, y + 1, layer * 
1.5)
            if node.right:
                G.add_edge(node.price, node.right.price)
                build_graph(node.right, x + 1 / layer, y + 1, layer * 
1.5)

    build_graph(root)
    nx.draw(G, pos, with_labels=True, node_size=2000, 
node_color="lightblue", ax=axes[1])
    axes[1].set_title(title)
    plt.show()
```

1. User Interface & Main Logic (Contributor 4) -> Handles user input, integrates AVL tree and stock prediction

```
# code for contributor 4

from avl_tree import AVLTree, Node
from stock_predictor import StockPredictor
from visualization import draw_visualization

avl = AVLTree()
predictor = StockPredictor()
root = None
stock_prices = []

while True:
    print("\n1. Insert Price\n2. Delete Price\n3. Predict\n4. Exit")
    choice = int(input("Enter choice: "))

    if choice == 1:
        price = int(input("Enter price: "))
        root = avl.insert(root, price)
        stock_prices.append(price)
        draw_visualization(root, stock_prices)

    elif choice == 2:
        price = int(input("Enter price to delete: "))
        root = avl.delete(root, price)
        stock_prices.remove(price)
        draw_visualization(root, stock_prices)

    elif choice == 3:
        days = int(input("Days to predict: "))
        preds = predictor.predict(stock_prices, days)
        print("Predictions:", preds)

    elif choice == 4:
        break
```