**Documentation: Binary Search Tree Analysis and Traversal**

This code represents a program that analyzes and performs traversals on a Binary Search Tree (BST) based on user input. Let's break down how it works.

**Basic Functions of the program**

1. **Importing the required module:**
   * The code begins by importing the **os** module, which provides functions for interacting with the operating system.

import os

1. **Prompting for input file name:**
   * The program prompts the user to enter the name and extension of the input file they want to process.

# Prompt the user for the name of the input file

filename = input('Input the name and extension of the file: ')

1. **Constructing the file path:**
   * The code retrieves the absolute path of the script itself using the **\_\_file\_\_** variable.
   * It then obtains the directory path of the script using **os.path.dirname()**.
   * Finally, it constructs the full file path by joining the script directory and the user-provided filename using **os.path.join()**.

# Get the path of the script

script\_path = os.path.abspath(\_\_file\_\_)

# Get the directory containing the script

script\_dir = os.path.dirname(script\_path)

# Construct the path of the input file

file\_path = os.path.join(script\_dir, filename)

1. **Checking file existence:**
   * The program checks if the file specified by the constructed file path exists using **os.path.exists()**.
   * If the file does not exist, it prints an error message and exits the program.

if not os.path.exists(file\_path):

    # If the file does not exist, print an error message and exit

    print(f'Error: File "{file\_path}" not found.')

    exit(1)

1. **Opening and reading the input file:**
   * The code opens the input file in read mode using **open(file\_path, 'r')**.
   * It reads the contents of the file using **readlines()** and stores them as a list of strings in the **lines** variable.
   * Finally, it closes the file using **close()**.

# Try to open the input file in read mode

file = open(file\_path, 'r')

# Read the contents of the file as a list of strings

lines = file.readlines()

# Close the file

file.close()

1. **Processing the input data:**
   * The program initializes an empty list called **values**.
   * It iterates over each line in the **lines** list.
   * For each line, it attempts to convert the string to an integer using **int(line)**.
   * If the conversion succeeds, the integer value is added to the **values** list.
   * If the conversion fails, indicating invalid data, it prints an error message and continues to the next line.

# Process the list of strings to extract the integer values

values = []

for line in lines:

    try:

        # Try to convert the string to an integer

        value = int(line)

    except ValueError:

        # If the conversion fails, print an error message and continue with the next line

        print(f'Error: Invalid data in input file: "{line.strip()}"')

        exit(1)

**Advance functions of the program**

1. **Defining the Node class:**
   * The **Node** class represents a node in the Binary Search Tree (BST).
   * It has three attributes:
     + **left**: Represents the left child of the node.
     + **right**: Represents the right child of the node.
     + **val**: Stores the value of the node.

class Node:

    def \_\_init\_\_(self, key):

        self.left = None

        self.right = None

        self.val = key

1. **Inserting values into the BST:**
   * The **insert()** function is used to insert values into the BST recursively.
   * It takes two parameters: **root** (the root node of the current subtree) and **key** (the value to be inserted).
   * If the **root** is **None**, it means the current position is empty, so a new node is created with the given **key** and returned.
   * If the **key** is less than the value of the **root**, it should be inserted in the left subtree.
   * If the **key** is greater than or equal to the value of the **root**, it should be inserted in the right subtree.
   * The function calls itself recursively with the appropriate child subtree until it finds the correct position for insertion.
   * Finally, it returns the modified subtree with the new node added.

def insert(root, key):

    if root is None:

        return Node(key)

    else:

        if root.val < key:

            root.right = insert(root.right, key)

        else:

            root.left = insert(root.left, key)

    return root

1. **Creating and populating the Binary Search Tree (BST):**
   * The code starts by creating an initially empty BST with the **root** variable set to **None**.
   * Then, it iterates over the **values** list obtained from reading the input file.
   * For each value in the list, it calls the **insert()** function to insert the value into the BST.
   * The **insert()** function is responsible for maintaining the structure of the BST by appropriately placing the values in the correct positions.
   * After inserting all the values, the **root** variable will be modified and will represent the root node of the populated BST.

# Create an initially empty Binary Search Tree

root = None

# Insert values into the BST

for value in values:

    root = insert(root, value)

1. **Traversing the BST:**
   * Three traversal functions are defined to traverse the BST: **preorder()**, **postorder()**, and **inorder()**.
   * Preorder Traversal (**preorder()**): Prints the value of the current node, then recursively visits the left subtree, and finally, the right subtree.
   * Postorder Traversal (**postorder()**): Recursively visits the left subtree, then the right subtree, and finally prints the value of the current node.
   * Inorder Traversal (**inorder()**): Recursively visits the left subtree, then prints the value of the current node, and finally, visits the right subtree.
   * Each function follows a recursive approach to traverse the tree and print the node values in the desired order.

def preorder(root):

    if root:

        print(root.val, end=' ')

        preorder(root.left)

        preorder(root.right)

def postorder(root):

    if root:

        postorder(root.left)

        postorder(root.right)

        print(root.val, end=' ')

def inorder(root):

    if root:

        inorder(root.left)

        print(root.val, end=' ')

        inorder(root.right)

1. **Generating a table with BST information:**
   * The **generate\_table()** function generates a table that displays various information about each node in the BST.
   * It uses helper functions to compute the depth and degree of each node and perform an inorder traversal to get the nodes in ascending order.
   * The function initializes data structures to store information about parent and sibling relationships between nodes.
   * It performs an inorder traversal of the BST to populate the **nodes** list with the nodes in ascending order.
   * For each node, it determines its parent and sibling (if they exist) and stores them in the **parent** and **sibling** dictionaries, respectively.
   * The function then prints a header row and a horizontal line for the table.
   * It iterates over the nodes and prints information about each node in the table, including its value, parent, sibling, left child, right child, degree, and depth.
   * The table provides a visual representation of the structure and relationships within the BST.

def generate\_table(root):

    # Helper function to compute the depth of a node

    def depth(node):

        d = 0

        while node != root:

            node = parent[node]

            d += 1

        return d

    # Helper function to compute the degree of a node

    def degree(node):

        d = 0

        if node.left:

            d += 1

        if node.right:

            d += 1

        return d

    # Helper function to perform an inorder traversal of the BST

    def inorder(node):

        if node:

            inorder(node.left)

            nodes.append(node)

            inorder(node.right)

    # Initialize data structures

    nodes = []

    parent = {}

    sibling = {}

    # Perform an inorder traversal of the BST to get a list of its nodes in ascending order

    inorder(root)

    # Compute the parent and sibling of each node

    for node in nodes:

        if node.left:

            parent[node.left] = node

            sibling[node.left] = node.right

        if node.right:

            parent[node.right] = node

            sibling[node.right] = node.left

    # Print a horizontal line before the header row of the table

    print()

    print()

    print('b. A table indicating information ')

    print('---------------------------------------------------------------')

    # Print the header row of the table

    print('{:<8}{:<8}{:<8}{:<12}{:<12}{:<8}{:<8}'.format('Node', 'Parent', 'Sibling', 'Left Child', 'Right Child', 'Degree', 'Depth'))

    # Print a horizontal line after the header row of the table

    print('---------------------------------------------------------------')

    # Print the information for each node in the table

    for node in nodes:

        p = parent.get(node)

        s = sibling.get(node)

        lc = node.left

        rc = node.right

        d = degree(node)

        dp = depth(node)

        print('{:<8}{:<8}{:<8}{:<12}{:<12}{:<8}{:<8}'.format(node.val, p.val if p else 'NULL', s.val if s else 'NULL', lc.val if lc else 'NULL', rc.val if rc else 'NULL', d, dp))

1. **Performing BST Traversals and Generating a Table:**
   * The code section starts by printing an empty line and the text "a. Traversals in the following sequence" to indicate the start of the traversal output.
   * It then calls the **preorder(root)** function to perform the pre-order traversal of the BST and print the values of each node in the order: root, left, right.
   * After printing the pre-order traversal, it calls **postorder(root)** to perform the post-order traversal and print the values of each node in the order: left, right, root.
   * Following that, it calls **inorder(root)** to perform the in-order traversal and print the values of each node in the order: left, root, right.
   * Each traversal is separated by newline characters (**print()**), creating a clear separation in the output.
   * After the traversals, the code calls the **generate\_table(root)** function to generate and output a table with detailed information about each node in the BST.
   * The table provides a structured view of the nodes, their relationships, and specific attributes such as parent, sibling, left child, right child, degree, and depth.

# Perform pre-order, post-order and in-order traversals of the BST and output the results

print()

print("a. Traversals in the following sequence")

print()

preorder(root)

print()

postorder(root)

print()

inorder(root)

# Generate and output a table with information about each node in the BST

generate\_table(root)