# The following struct defines the node of a binary tree:

class BinaryTreeNode:

def \_\_init\_\_(self, info):

self.info = info # Stores the node's data

self.llink = None # Pointer to the left child

self.rlink = None # Pointer to the right child

# The general algorithm to find the height of a binary tree is as follows. Suppose height(p) denotes the height of the binary tree with root p.

def height(node):

if node is None:

return 0

else:

return 1 + max(height(node.llink), height(node.rlink))

# Clearly, this is a recursive algorithm. The following function implements this algorithm:

def height(node):

if node is None:

return 0

else:

return 1 + max(height(node.llink), height(node.rlink))

# COPY TREE

# Given a pointer to the root node of a binary tree,

# we next describe the function copyTree,

# which makes a copy of a given binary tree.

# This function is also useful in implementing the copy constructor and overloading the assignment operator.

def copy\_tree(other\_tree\_root):

if other\_tree\_root is None:

return None

else:

copied\_tree\_root = BinaryTreeNode(other\_tree\_root.info)

copied\_tree\_root.llink = copy\_tree(other\_tree\_root.llink)

copied\_tree\_root.rlink = copy\_tree(other\_tree\_root.rlink)

return copied\_tree\_root

# Binary Tree Operation

# Insert Operation

# Algorithm

class BinaryTreeNode:

def \_\_init\_\_(self, info):

self.info = info

self.llink = None

self.rlink = None

def insert(root, data):

# If root is None, create the root node

if root is None:

return BinaryTreeNode(data)

# Initialize the current node as root

current = root

# Loop until the correct insertion position is found

while True:

# If data is greater than the current node's info, go to the right subtree

if data > current.info:

# If there is no right child, insert here

if current.rlink is None:

current.rlink = BinaryTreeNode(data)

break

else:

current = current.rlink # Move to the right child

else:

# If there is no left child, insert here

if current.llink is None:

current.llink = BinaryTreeNode(data)

break

else:

current = current.llink # Move to the left child

return root # Return the unchanged root of the tree

# Implementation

# The implementation of insert function should look like this −

class BinaryTreeNode:

def \_\_init\_\_(self, data):

self.data = data

self.leftChild = None

self.rightChild = None

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, data):

# Create a new node

temp\_node = BinaryTreeNode(data)

# If the tree is empty, set the new node as the root

if self.root is None:

self.root = temp\_node

else:

# Initialize current and parent pointers

current = self.root

parent = None

while True:

parent = current

# Go to the left subtree

if data < parent.data:

current = current.leftChild

# If no left child, insert here

if current is None:

parent.leftChild = temp\_node

return

# Go to the right subtree

else:

current = current.rightChild

# If no right child, insert here

if current is None:

parent.rightChild = temp\_node

return

# Search Operation

class BinaryTreeNode:

def \_\_init\_\_(self, data):

self.data = data

self.leftChild = None

self.rightChild = None

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

def search(self, data):

current = self.root

print("Visiting elements: ", end="")

while current is not None:

print(current.data, end=" ")

# If the current node's data matches the search data, return the node

if current.data == data:

return current

# Go to the left subtree if data is less than the current node's data

if current.data > data:

current = current.leftChild

# Go to the right subtree if data is greater than the current node's data

else:

current = current.rightChild

# If data not found, return None

print() # For newline after visiting elements

return None

# The implementation of tree traversal

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.leftChild = None

self.rightChild = None

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, data):

tempNode = Node(data)

# If the tree is empty, set root to new node

if self.root is None:

self.root = tempNode

else:

current = self.root

parent = None

while True:

parent = current

# Insert on the left subtree

if data < parent.data:

current = current.leftChild

# Insert here if no child

if current is None:

parent.leftChild = tempNode

return

# Insert on the right subtree

else:

current = current.rightChild

# Insert here if no child

if current is None:

parent.rightChild = tempNode

return

def search(self, data):

current = self.root

print("Visiting elements: ", end="")

while current is not None:

print(current.data, end=" ")

if current.data == data:

return current

elif data < current.data:

current = current.leftChild

else:

current = current.rightChild

return None # Element not found

def pre\_order\_traversal(self, root):

if root is not None:

print(root.data, end=" ")

self.pre\_order\_traversal(root.leftChild)

self.pre\_order\_traversal(root.rightChild)

def inorder\_traversal(self, root):

if root is not None:

self.inorder\_traversal(root.leftChild)

print(root.data, end=" ")

self.inorder\_traversal(root.rightChild)

def post\_order\_traversal(self, root):

if root is not None:

self.post\_order\_traversal(root.leftChild)

self.post\_order\_traversal(root.rightChild)

print(root.data, end=" ")

# Usage

bst = BinarySearchTree()

elements = [27, 14, 35, 10, 19, 31, 42]

# Insert elements

for elem in elements:

bst.insert(elem)

# Search for an element

element\_to\_search = 31

result = bst.search(element\_to\_search)

if result is not None:

print(f"\n[{result.data}] Element found.")

else:

print(f"\n[ x ] Element not found ({element\_to\_search}).")

# Search for an element not in the tree

element\_to\_search = 15

result = bst.search(element\_to\_search)

if result is not None:

print(f"\n[{result.data}] Element found.")

else:

print(f"\n[ x ] Element not found ({element\_to\_search}).")

# Traversals

print("\nPreorder traversal: ", end="")

bst.pre\_order\_traversal(bst.root)

print("\nInorder traversal: ", end="")

bst.inorder\_traversal(bst.root)

print("\nPostorder traversal: ", end="")

bst.post\_order\_traversal(bst.root)

# Node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.leftChild = None

self.rightChild = None

# Insert Operation

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.leftChild = None

self.rightChild = None

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, data):

# Create a new node

tempNode = Node(data)

# If the tree is empty, set the root to the new node

if self.root is None:

self.root = tempNode

else:

current = self.root

parent = None

while True:

parent = current

# Go to the left of the tree

if data < parent.data:

current = current.leftChild

# Insert to the left if there is no left child

if current is None:

parent.leftChild = tempNode

return

# Go to the right of the tree

else:

current = current.rightChild

# Insert to the right if there is no right child

if current is None:

parent.rightChild = tempNode

return