Tree Data Structure

A tree is a nonlinear data structure composed of nodes with a parent-child relationship, where each node stores an element. if T is not empty, T has a special node called the root that has no parent each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.A tree may be empty or contain a root node with zero or more subtrees. The structure can be described as follows:

Root: The top node of the tree, which has no parent.

Parent: A node that has one or more child nodes.

Child: A node that has a parent node.

Leaf: A node with no children.

Subtree: A part of the tree, consisting of a node and all its descendants.

Path: A sequence of nodes connected by edges.

Level: The level of a node is the number of edges from the root to the node.

Keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

Traversing − Traversing means passing through nodes in a specific order.

Visiting − Visiting refers to checking the value of a node when control is on the node.

Binary Tree

A binary tree is a type of tree in which each node has at most two children, often referred to as the left and right child. The binary tree can be visualized as follows:

Every node contains a data element.

Each node has two pointers: one for the left child and one for the right child.

Node Definition in Python

class TreeNode:

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

self.data = data

self.left = None

self.right = None

Binary Tree Operations

Insert Operation

Inserting a node into a binary tree involves finding the correct location based on the value of the node.

def insert(root, data):

if root is None:

return TreeNode(data)

if data < root.data:

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

else:

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

return root

Search Operation

To search for an element in the binary tree, compare the value with the root node. If it’s smaller, search the left subtree; if it’s greater, search the right subtree.

def search(root, data):

if root is None or root.data == data:

return root

if data < root.data:

return search(root.left, data)

else:

return search(root.right, data)

Height of Binary Tree

The height of a binary tree is the length of the longest path from the root to a leaf.

def height(root):

if root is None:

return 0

else:

left\_height = height(root.left)

right\_height = height(root.right)

return 1 + max(left\_height, right\_height)

Inorder Traversal

In an inorder traversal, the nodes are visited in the following order: Left subtree, Root, Right subtree.

def inorder\_traversal(root):

if root:

inorder\_traversal(root.left)

print(root.data, end=" ")

inorder\_traversal(root.right)

Preorder Traversal

In a preorder traversal, the nodes are visited in the following order: Root, Left subtree, Right subtree.

def preorder\_traversal(root):

if root:

print(root.data, end=" ")

preorder\_traversal(root.left)

preorder\_traversal(root.right)

Postorder Traversal

In a postorder traversal, the nodes are visited in the following order: Left subtree, Right subtree, Root.

def postorder\_traversal(root):

if root:

postorder\_traversal(root.left)

postorder\_traversal(root.right)

print(root.data, end=" ")

Binary Search Tree (BST)

A Binary Search Tree (BST) is a special kind of binary tree where the key in the root node is greater than all keys in its left subtree and smaller than all keys in its right subtree. This property holds for all nodes in the tree.

Binary Search Tree Operations

Insert Operation

The insert operation in a BST places a node in the correct position based on its value.

def bst\_insert(root, data):

if root is None:

return TreeNode(data)

if data < root.data:

root.left = bst\_insert(root.left, data)

else:

root.right = bst\_insert(root.right, data)

return root

Search Operation

Searching for an element in a BST follows the binary search property.

def bst\_search(root, data):

if root is None or root.data == data:

return root

if data < root.data:

return bst\_search(root.left, data)

else:

return bst\_search(root.right, data)

Binary Tree Example Usage

Here’s an example of creating a binary tree, inserting nodes, and performing various operations such as search and traversal:

# Create a binary tree and insert elements

root = None

root = insert(root, 10)

root = insert(root, 5)

root = insert(root, 20)

root = insert(root, 3)

root = insert(root, 7)

# Search for a node

node = search(root, 7)

if node:

print(f"Node {node.data} found")

else:

print("Node not found")

# Traversals

print("Inorder Traversal:")

inorder\_traversal(root)

print("\nPreorder Traversal:")

preorder\_traversal(root)

print("\nPostorder Traversal:")

postorder\_traversal(root)

# Height of the tree

print(f"\nHeight of the tree: {height(root)}")

Copy Tree

To make a copy of a binary tree, a recursive function can be used:

def copy\_tree(root):

if root is None:

return None

new\_node = TreeNode(root.data)

new\_node.left = copy\_tree(root.left)

new\_node.right = copy\_tree(root.right)

return new\_node

Insert Operation - Algorithm

Whenever an element is to be inserted into a binary search tree, first locate its proper location. Start searching from the root node. If the data is less than the current node's key, search for an empty location in the left subtree and insert the data. Otherwise, search for an empty location in the right subtree and insert the data.

Algorithm:

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

// if tree is empty

if (root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while (1) {

parent = current;

// go to the left of the tree

if (data < parent->data) {

current = current->leftChild;

// insert to the left

if (current == NULL) {

parent->leftChild = tempNode;

return;

}

}

// go to the right of the tree

else {

current = current->rightChild;

// insert to the right

if (current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

---