**### Binary Tree Concepts and Examples**

1. Binary Tree: A tree where each node has up to two children (0, 1, or 2). Here’s an example of a basic binary tree:

1

/ \

2 3

/ \ /

4 5 6

2. Complete Binary Tree: All levels are fully filled except possibly the last level, and the last level has all nodes as left as possible. For example:

1

/ \

2 3

/ \ /

4 5 6

3. Almost Complete Binary Tree: A binary tree where every node is as left as possible, and all levels except possibly the last are fully filled. It’s often used to refer to a heap structure.

4. Strict Binary Tree (Full Binary Tree): Every node has exactly 0 or 2 children (no node has only one child). Here’s an example:

1

/ \

2 3

/ \ / \

4 5 6 7

**### Binary Tree Representations**

1. Array Representation: Store nodes level by level in an array, where the left child of node at index `i` is at `2i + 1` and the right child at `2i + 2`.

2. Linked Representation: Each node has pointers to its left and right children.

**### Insertion and Traversal in Binary Tree**

Binary trees are typically implemented with linked representation. Insertion depends on whether it's a complete binary tree or binary search tree. Traversal can be done using pre-order, in-order, or post-order methods.

**### Binary Search Tree (BST) Implementation**

A Binary Search Tree is a type of binary tree where:

- The left subtree of any node only contains nodes with values less than the node’s value.

- The right subtree only contains nodes with values greater than the node’s value.

- Duplicate values are not allowed.

**#### Code Implementation for BST**

class Node:

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

self.item = item

self.left = None

self.right = None

class BST:

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

self.root = None

def insert(self, item):

if self.root is None:

self.root = Node(item)

else:

self.\_insert\_recursive(self.root, item)

def \_insert\_recursive(self, current\_node, item):

if item < current\_node.item:

if current\_node.left is None:

current\_node.left = Node(item)

else:

self.\_insert\_recursive(current\_node.left, item)

elif item > current\_node.item:

if current\_node.right is None:

current\_node.right = Node(item)

else:

self.\_insert\_recursive(current\_node.right, item)

def search(self, item):

return self.\_search\_recursive(self.root, item)

def \_search\_recursive(self, current\_node, item):

if current\_node is None:

return None

if current\_node.item == item:

return current\_node

elif item < current\_node.item:

return self.\_search\_recursive(current\_node.left, item)

else:

return self.\_search\_recursive(current\_node.right, item)

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.item, end=" ")

self.inorder(node.right)

def preorder(self, node):

if node:

print(node.item, end=" ")

self.preorder(node.left)

self.preorder(node.right)

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.item, end=" ")

def delete(self, item):

self.root = self.\_delete\_recursive(self.root, item)

def \_delete\_recursive(self, node, item):

if not node:

return node

if item < node.item:

node.left = self.\_delete\_recursive(node.left, item)

elif item > node.item:

node.right = self.\_delete\_recursive(node.right, item)

else:

# Node with only one child or no child

if not node.left:

return node.right

elif not node.right:

return node.left

# Node with two children: get the inorder successor (smallest in the right subtree)

temp = self.\_min\_value\_node(node.right)

node.item = temp.item

node.right = self.\_delete\_recursive(node.right, temp.item)

return node

def \_min\_value\_node(self, node):

current = node

while current.left is not None:

current = current.left

return current

# Example usage

bst = BST()

values = [10, 5, 20, 3, 7, 15, 30]

for v in values:

bst.insert(v)

print("Inorder Traversal: ", end="")

bst.inorder(bst.root) # Should print the sorted order of values

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

bst.preorder(bst.root)

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

bst.postorder(bst.root)

# Searching for a node

node = bst.search(15)

if node:

print("\nNode found with value:", node.item)

else:

print("\nNode not found.")

# Deleting a node

bst.delete(10)

print("Inorder Traversal after deletion: ", end="")

bst.inorder(bst.root)

**### Explanation of Traversals:**

- In-order Traversal: Left subtree, root, right subtree. In a BST, this gives values in sorted order.

- Pre-order Traversal: Root, left subtree, right subtree.

- Post-order Traversal: Left subtree, right subtree, root.

**### Deletion Cases in BST:**

1. No Child: Remove the node.

2. Single Child: Replace the node with its child.

3. Two Children: Replace the node with its in-order successor (smallest node in the right subtree).

This code provides a functional BST in Python, demonstrating key operations such as insertion, search, traversal, and deletion.

Assignment-20: Binary Search Tree part-1

1. Define a class Node with instance variables left, item and right. The variables left and right are used to refer left and right child node. The item variable is used to hold data item.

2. Define a class BST to implement Binary Search Tree data structure. Make\_init\_\_() method to create root instance variable to hold the reference of root node.

3. In class BST, define insert method to store new data item in the binary search tree.

4. In class BST, define a search method to find a given item in the binary search tree and returns the node reference. It returns None if search failed.

5. In class BST, define a method to implement inorder traversal.

6. In class BST, define a method to implement preorder traversal.

7. In class BST, define a method to implement postorder traversal

Deletion in BST:

--------------------------

1. In class BST, define a method to find minimum value item node.

2. In class BST, define a method to find maximum value item node.

3. In class BST, define a method to delete a node from binary search tree.

4. In class BST, define a method size to return the number of elements present in the BST

**Code Example for Binary Tree Structure**

Here’s a basic structure for binary trees in Python using the linked representation:

python

Copy code

# Node class to represent each node in the binary tree

class Node:

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

self.data = data

self.left = None # Left child

self.right = None # Right child

# Binary Tree class

class BinaryTree:

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

self.root = None

# Insert node in binary tree (Level Order)

def insert(self, data):

if not self.root:

self.root = Node(data)

else:

queue = [self.root]

while queue:

node = queue.pop(0)

if not node.left:

node.left = Node(data)

break

else:

queue.append(node.left)

if not node.right:

node.right = Node(data)

break

else:

queue.append(node.right)

# Inorder traversal

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=' ')

self.inorder(node.right)

# Preorder traversal

def preorder(self, node):

if node:

print(node.data, end=' ')

self.preorder(node.left)

self.preorder(node.right)

# Postorder traversal

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.data, end=' ')

**Binary Search Tree (BST)**

1. **What is a Binary Search Tree (BST)?**
   * A binary search tree is a binary tree in which every node's left child has a lesser value and every node's right child has a greater value, making search operations efficient.
2. **BST Properties**:
   * Values in the left subtree of a node are less than the node’s value.
   * Values in the right subtree of a node are greater than the node’s value.
   * Duplicates are not allowed (by default).

**Implementation of BST in Python**

Here’s the code to implement BST with insert, search, and traversal functions, along with methods to find minimum and maximum values.

python

Copy code

# Node class for BST

class Node:

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

self.left = None

self.right = None

self.key = key

# BST class with various operations

class BST:

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

self.root = None

# Insert method

def insert(self, key):

self.root = self.\_insert\_recursive(self.root, key)

def \_insert\_recursive(self, node, key):

if not node:

return Node(key)

if key < node.key:

node.left = self.\_insert\_recursive(node.left, key)

else:

node.right = self.\_insert\_recursive(node.right, key)

return node

# Search method

def search(self, key):

return self.\_search\_recursive(self.root, key)

def \_search\_recursive(self, node, key):

if not node or node.key == key:

return node

if key < node.key:

return self.\_search\_recursive(node.left, key)

return self.\_search\_recursive(node.right, key)

# Inorder Traversal (Left, Root, Right)

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.key, end=' ')

self.inorder(node.right)

# Preorder Traversal (Root, Left, Right)

def preorder(self, node):

if node:

print(node.key, end=' ')

self.preorder(node.left)

self.preorder(node.right)

# Postorder Traversal (Left, Right, Root)

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.key, end=' ')

# Minimum value in BST

def min\_value\_node(self):

current = self.root

while current.left:

current = current.left

return current.key

# Maximum value in BST

def max\_value\_node(self):

current = self.root

while current.right:

current = current.right

return current.key

# Deletion method

def delete(self, key):

self.root = self.\_delete\_recursive(self.root, key)

def \_delete\_recursive(self, node, key):

if not node:

return node

if key < node.key:

node.left = self.\_delete\_recursive(node.left, key)

elif key > node.key:

node.right = self.\_delete\_recursive(node.right, key)

else:

# Node with only one child or no child

if not node.left:

return node.right

elif not node.right:

return node.left

# Node with two children, get the inorder successor

temp = self.\_min\_value\_node(node.right)

node.key = temp.key

node.right = self.\_delete\_recursive(node.right, temp.key)

return node

def \_min\_value\_node(self, node):

current = node

while current.left:

current = current.left

return current

**Traversal Techniques**

* **Preorder (Root, Left, Right)**: Useful for creating a copy of the tree.
* **Inorder (Left, Root, Right)**: Gives nodes in a sorted order for BSTs.
* **Postorder (Left, Right, Root)**: Useful for deleting nodes in a tree.

**Deletion Scenarios in BST**

1. **Node with No Child**: Simply remove the node.
2. **Node with One Child**: Remove the node and replace it with its child.
3. **Node with Two Children**: Replace the node with its inorder successor (smallest node in the right subtree).

Here is the Python code for a Binary Search Tree (BST) with detailed comments explaining each section:

python

Copy code

# Node class represents each node in the Binary Search Tree

class Node:

def \_\_init\_\_(self, item=None, left=None, right=None):

# Each node has an item (data), a left child, and a right child

self.item = item

self.left = left

self.right = right

# Binary Search Tree (BST) class to manage tree operations

class BST:

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

# Initialize the tree with an empty root

self.root = None

# Insert method to add data to the BST

def insert(self, data):

# Start recursive insertion from the root

self.root = self.\_insert(self.root, data)

# Helper method for insertion

def \_insert(self, root, data):

# If current node is None, create a new node with the data

if root is None:

return Node(data)

# If data is less than current node's item, go to the left subtree

if data < root.item:

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

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

elif data > root.item:

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

# Return the unchanged node pointer

return root

# Search method to find an item in the BST

def search(self, data):

# Start recursive search from the root

return self.\_search(self.root, data)

# Helper method for search

def \_search(self, root, data):

# Base case: root is None or data matches root's item

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

return root

# If data is smaller, search in the left subtree

if data < root.item:

return self.\_search(root.left, data)

# Otherwise, search in the right subtree

else:

return self.\_search(root.right, data)

# Inorder traversal: left, root, right

def inorder(self):

result = []

# Populate result with inorder traversal of the tree

self.\_inorder(self.root, result)

return result

# Helper method for inorder traversal

def \_inorder(self, root, result):

if root:

self.\_inorder(root.left, result)

result.append(root.item) # Visit node

self.\_inorder(root.right, result)

# Preorder traversal: root, left, right

def preorder(self):

result = []

# Populate result with preorder traversal of the tree

self.\_preorder(self.root, result)

return result

# Helper method for preorder traversal

def \_preorder(self, root, result):

if root:

result.append(root.item) # Visit node

self.\_preorder(root.left, result)

self.\_preorder(root.right, result)

# Postorder traversal: left, right, root

def postorder(self):

result = []

# Populate result with postorder traversal of the tree

self.\_postorder(self.root, result)

return result

# Helper method for postorder traversal

def \_postorder(self, root, result):

if root:

self.\_postorder(root.left, result)

self.\_postorder(root.right, result)

result.append(root.item) # Visit node

# Find the minimum value in the BST, starting from a given node

def min\_value(self, temp):

current = temp

# Traverse to the leftmost node

while current.left is not None:

current = current.left

return current.item # Leftmost node is the minimum

# Find the maximum value in the BST, starting from a given node

def max\_value(self, temp):

current = temp

# Traverse to the rightmost node

while current.right is not None:

current = current.right

return current.item # Rightmost node is the maximum

# Delete a node with the given data from the BST

def delete(self, data):

# Start recursive delete from the root

self.root = self.\_delete(self.root, data)

# Helper method for delete operation

def \_delete(self, root, data):

# Base case: if root is None, return it

if root is None:

return root

# If data is less than root's item, find it in left subtree

if data < root.item:

root.left = self.\_delete(root.left, data)

# If data is greater than root's item, find it in right subtree

elif data > root.item:

root.right = self.\_delete(root.right, data)

# Node to be deleted is found

else:

# Case 1: Node has no left child

if root.left is None:

return root.right

# Case 2: Node has no right child

elif root.right is None:

return root.left

# Case 3: Node has two children

# Replace node's item with minimum item from right subtree

root.item = self.min\_value(root.right)

# Delete the duplicate node

root.right = self.\_delete(root.right, root.item)

return root

# Return the total number of nodes in the BST

def size(self):

# Inorder traversal is used to count the nodes

return len(self.inorder())

**Explanation of the Code:**

1. **Node Class**: Represents each node in the BST with item, left, and right pointers.
2. **BST Class**:
   * insert: Recursively inserts data by comparing it with current node's item.
   * search: Recursively searches for data in the tree.
   * **Traversals**:
     + inorder: Traverses nodes in left-root-right order.
     + preorder: Traverses nodes in root-left-right order.
     + postorder: Traverses nodes in left-right-root order.
   * **Finding Min and Max**: Helper functions to find the minimum and maximum item values.
   * **Delete**: Deletes a node based on three cases:
     + Node has no children (leaf).
     + Node has one child.
     + Node has two children (replaced by the smallest node in the right subtree).
   * size: Returns total nodes using inorder traversal.