Binary Search Tree

Created by	j jubink
Created time	@February 26, 2025 10:15 PM
Tags	

🌲 Binary Search Tree (BST) – Beginner Level Solutions

Each problem below includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 1. Insert a Node into a BST

Algorithm

1. If the tree is empty, create a new node and return it as the root.
2. If the key is smaller than the root, recursively insert it in the left subtree.
3. If the key is greater than the root, recursively insert it in the right subtree.
4. Return the updated tree.

Python Code

```
python
CopyEdit
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

```python
class BST:
    def __init__(self):
        self.root = None

    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        return root

    def inorder(self, root):  # Inorder traversal (LNR)
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

# Usage
tree = BST()
root = None
root = tree.insert(root, 50)
root = tree.insert(root, 30)
root = tree.insert(root, 70)
root = tree.insert(root, 20)
root = tree.insert(root, 40)
root = tree.insert(root, 60)
root = tree.insert(root, 80)

print("Inorder traversal of BST:")
tree.inorder(root)
```

Output

```
yaml
CopyEdit
Inorder traversal of BST:
20 30 40 50 60 70 80
```

🔷 2. Search for a Node in a BST

Algorithm

1. If the root is `None`, return `False` (element not found).
2. If the key matches the root, return `True` (element found).
3. If the key is smaller, search recursively in the left subtree.
4. If the key is greater, search recursively in the right subtree.

Python Code

```
python
CopyEdit
def search(root, key):
    if root is None or root.key == key:
        return root is not None  # Return True if found, else False
    if key < root.key:
        return search(root.left, key)
    return search(root.right, key)

# Usage
print("Searching for 40:", search(root, 40))  # True
print("Searching for 90:", search(root, 90))  # False
```

Output

```
yaml
CopyEdit
Searching for 40: True
Searching for 90: False
```

🔷 3. Find the Minimum and Maximum Value in a BST

Algorithm

📌 **For Minimum Value:**

1. Start from the root and keep moving left until `left` is `None`.
2. The last node encountered is the minimum value.

📌 **For Maximum Value:**

1. Start from the root and keep moving right until `right` is `None`.
2. The last node encountered is the maximum value.

Python Code

```
python
CopyEdit
def find_min(root):
    while root.left:
        root = root.left
    return root.key

def find_max(root):
    while root.right:
        root = root.right
    return root.key
```

```
# Usage
print("Minimum value in BST:", find_min(root))  # 20
print("Maximum value in BST:", find_max(root))  # 80
```

Output

```
yaml
CopyEdit
Minimum value in BST: 20
Maximum value in BST: 80
```

🔷 4. Find the Height of a BST

Algorithm

1. If the tree is empty, return `1`.
2. Compute the height of the left and right subtrees recursively.
3. The height of the tree is `1 + max(left_height, right_height)`.

Python Code

```
python
CopyEdit
def find_height(root):
    if root is None:
        return -1
    return 1 + max(find_height(root.left), find_height(root.right))

# Usage
print("Height of BST:", find_height(root))  # 2
```

Output

```
css
CopyEdit
Height of BST: 2
```

🔷 5. Find the Total Number of Nodes in a BST

Algorithm

1. If the tree is empty, return `0`.
2. Recursively count nodes in the left and right subtrees.
3. Total nodes = `1 + left_count + right_count`.

Python Code

```
python
CopyEdit
def count_nodes(root):
    if root is None:
        return 0
    return 1 + count_nodes(root.left) + count_nodes(root.right)

# Usage
print("Total nodes in BST:", count_nodes(root))  # 7
```

Output

```
yaml
CopyEdit
```

```
Total nodes in BST: 7
```

🔷 6. Inorder, Preorder, and Postorder Traversal of a BST

Algorithm

✅ **Inorder (LNR):** Left → Root → Right

✅ **Preorder (NLR):** Root → Left → Right

✅ **Postorder (LRN):** Left → Right → Root

Python Code

```
python
CopyEdit
def inorder(root):  # Left, Root, Right
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def preorder(root):  # Root, Left, Right
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

def postorder(root):  # Left, Right, Root
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")
```

```
# Usage
print("\nInorder traversal:")
inorder(root)

print("\nPreorder traversal:")
preorder(root)

print("\nPostorder traversal:")
postorder(root)
```

Output

```
yaml
CopyEdit
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
```

🔷 7. Check if a Given Tree is a BST

Algorithm

1. Check if the left subtree contains only values **less** than the root.
2. Check if the right subtree contains only values **greater** than the root.
3. Recursively check left and right subtrees.

Python Code

```
python
CopyEdit
def is_bst(root, min_val=float('-inf'), max_val=float('inf')):
    if root is None:
```

```
        return True
    if not (min_val < root.key < max_val):
        return False
    return is_bst(root.left, min_val, root.key) and is_bst(root.right, root.key, max_
val)

# Usage
print("Is the tree a BST?", is_bst(root))  # True
```

Output

```
vbnet
CopyEdit
Is the tree a BST? True
```

🌲 Binary Search Tree (BST) – Intermediate Level Solutions

Each problem includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 1. Find the k-th Smallest/Largest Element in a BST

Algorithm for k-th Smallest Element

1. Perform an **inorder traversal** (since it gives elements in sorted order).
2. Keep a **counter** to track how many elements are visited.
3. Return the k-th element when the counter reaches `k`.

Python Code

```
python
CopyEdit
def kth_smallest(root, k, counter=[0]):
    if root:
        left = kth_smallest(root.left, k, counter)
        if left is not None:
            return left
        counter[0] += 1
        if counter[0] == k:
            return root.key
        return kth_smallest(root.right, k, counter)
    return None

# Usage
print("3rd smallest element:", kth_smallest(root, 3))  # 40
```

Output

```
yaml
CopyEdit
3rd smallest element: 40
```

🔷 2. Delete a Node from a BST

Algorithm

1. If the node is **a leaf node**, delete it directly.
2. If the node has **one child**, replace it with its child.
3. If the node has **two children**, find its **inorder successor** (smallest in right subtree), copy its value, and delete the successor.

Python Code

```
python
CopyEdit
def delete_node(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = delete_node(root.left, key)
    elif key > root.key:
        root.right = delete_node(root.right, key)
    else:  # Found node to delete
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        temp = find_min(root.right)  # Inorder successor
        root.key = temp
        root.right = delete_node(root.right, temp)
    return root

# Usage
root = delete_node(root, 50)
print("BST after deleting 50:")
tree.inorder(root)
```

🔷 3. Check if a BST is Balanced

Algorithm

1. Calculate the height of left and right subtrees.
2. If the **absolute difference** between them is more than 1, return `False`.
3. Check recursively for left and right subtrees.

Python Code

```
python
CopyEdit
def is_balanced(root):
    def height(root):
        if root is None:
            return 0
        left_height = height(root.left)
        right_height = height(root.right)
        if abs(left_height - right_height) > 1:
            return -1
        return 1 + max(left_height, right_height)

    return height(root) != -1

# Usage
print("Is BST balanced?", is_balanced(root))  # True or False
```

🔷 4. Find the Distance Between Two Nodes in a BST

Algorithm

1. Find the **Lowest Common Ancestor (LCA)** of the two nodes.
2. Compute the **distance** from LCA to both nodes.
3. Distance = `distance_from_LCA(n1) + distance_from_LCA(n2)`.

Python Code

```
python
CopyEdit
def lca(root, n1, n2):
    if not root:
```

```
        return None
    if root.key > n1 and root.key > n2:
        return lca(root.left, n1, n2)
    if root.key < n1 and root.key < n2:
        return lca(root.right, n1, n2)
    return root

def find_distance(root, key):
    if root.key == key:
        return 0
    elif key < root.key:
        return 1 + find_distance(root.left, key)
    else:
        return 1 + find_distance(root.right, key)

def find_distance_between_nodes(root, n1, n2):
    ancestor = lca(root, n1, n2)
    return find_distance(ancestor, n1) + find_distance(ancestor, n2)

# Usage
print("Distance between 20 and 60:", find_distance_between_nodes(root, 20,
60))
```

🔷 5. Convert a Sorted Array into a Balanced BST

Algorithm

1. Select the **middle element** of the array as the root.
2. Recursively construct the left subtree from the left half of the array.
3. Recursively construct the right subtree from the right half of the array.

Python Code

```
python
CopyEdit
def sorted_array_to_bst(arr):
    if not arr:
        return None
    mid = len(arr) // 2
    root = Node(arr[mid])
    root.left = sorted_array_to_bst(arr[:mid])
    root.right = sorted_array_to_bst(arr[mid+1:])
    return root

# Usage
sorted_array = [10, 20, 30, 40, 50, 60, 70]
balanced_bst = sorted_array_to_bst(sorted_array)
tree.inorder(balanced_bst)
```

🔷 6. Convert a BST into a Sorted Doubly Linked List

Algorithm

1. Perform an **inorder traversal** to process elements in sorted order.
2. Modify left and right pointers to create a doubly linked list.

Python Code

```
python
CopyEdit
def bst_to_dll(root):
    if not root:
        return None
    prev = None
    head = None
```

```
    def inorder(node):
        nonlocal prev, head
        if not node:
            return
        inorder(node.left)
        if prev:
            prev.right = node
            node.left = prev
        else:
            head = node
        prev = node
        inorder(node.right)

    inorder(root)
    return head

# Usage
dll_head = bst_to_dll(root)
```

🔷 7. Print All Paths from Root to Leaf in a BST

Algorithm

1. Use **DFS** to traverse the tree.
2. Maintain a **path list** to store the current path.
3. When reaching a **leaf node**, print the path.

Python Code

```
python
CopyEdit
def print_paths(root, path=[]):
    if root is None:
```

```
        return
    path.append(root.key)
    if root.left is None and root.right is None:
        print(" → ".join(map(str, path)))
    else:
        print_paths(root.left, path[:])
        print_paths(root.right, path[:])

# Usage
print("Root to Leaf Paths:")
print_paths(root)
```

🔷 8. Merge Two BSTs into a Single BST

Algorithm

1. Convert both BSTs into sorted lists using inorder traversal.
2. Merge the two sorted lists.
3. Construct a balanced BST from the merged list.

Python Code

```
python
CopyEdit
def merge_sorted_lists(list1, list2):
    return sorted(list1 + list2)

def inorder_to_list(root, result=[]):
    if root:
        inorder_to_list(root.left, result)
        result.append(root.key)
        inorder_to_list(root.right, result)
    return result
```

```
def merge_bsts(root1, root2):
    list1 = inorder_to_list(root1, [])
    list2 = inorder_to_list(root2, [])
    merged_list = merge_sorted_lists(list1, list2)
    return sorted_array_to_bst(merged_list)
```

🌲 Binary Search Tree (BST) – Advanced Level Solutions

Each problem includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 1. Construct a BST from its Given Preorder Traversal

Algorithm

1. The **first element** in the preorder traversal is always the root.
2. Recursively divide the remaining elements into the **left subtree** (values smaller than root) and **right subtree** (values greater than root).
3. Insert nodes in the BST following the **BST insertion rules**.

Python Code

```
python
CopyEdit
import sys

class Node:
    def __init__(self, key):
        self.key = key
```

```
        self.left = None
        self.right = None

def construct_bst_preorder(preorder):
    def build_bst(bound=float('inf')):
        nonlocal index
        if index == len(preorder) or preorder[index] > bound:
            return None
        root = Node(preorder[index])
        index += 1
        root.left = build_bst(root.key)
        root.right = build_bst(bound)
        return root

    index = 0
    return build_bst()

# Usage
preorder = [50, 30, 20, 40, 70, 60, 80]
bst_root = construct_bst_preorder(preorder)
```

🔷 2. Find the Largest BST Subtree in a Binary Tree

Algorithm

1. Use a **recursive function** that checks if a subtree is a BST.
2. For each subtree, maintain:
    - `min_val`, `max_val` (to check BST property).
    - `size` (to track the number of nodes).
3. If a subtree is a BST, update the maximum BST size found.

Python Code

```
python
CopyEdit
class BSTInfo:
    def __init__(self, is_bst, size, min_val, max_val):
        self.is_bst = is_bst
        self.size = size
        self.min_val = min_val
        self.max_val = max_val

def largest_bst(root):
    def helper(node):
        if not node:
            return BSTInfo(True, 0, float('inf'), float('-inf'))
        left = helper(node.left)
        right = helper(node.right)
        if left.is_bst and right.is_bst and left.max_val < node.key < right.min_val:
            return BSTInfo(True, left.size + right.size + 1, min(left.min_val, node.ke
y), max(right.max_val, node.key))
        return BSTInfo(False, max(left.size, right.size), 0, 0)

    return helper(root).size

# Usage
print("Size of the largest BST subtree:", largest_bst(root))
```

🔷 3. Find the Closest Value to a Given Key in a BST

Algorithm

1. Start at the **root** and maintain a variable `closest`.
2. At each node, update `closest` if the current node is closer to the target than the previous `closest`.
3. If the target is smaller, move left; if larger, move right.

4. Stop when a leaf node is reached.

Python Code

```
python
CopyEdit
def find_closest_value(root, target):
    closest = root.key

    while root:
        if abs(root.key - target) < abs(closest - target):
            closest = root.key
        root = root.left if target < root.key else root.right

    return closest

# Usage
print("Closest value to 55:", find_closest_value(root, 55))
```

🔷 4. Find the Sum of Leaf Nodes in a BST

Algorithm

1. Perform a **recursive traversal** of the tree.
2. If a node has **no children**, add its value to the sum.
3. Recursively sum up values from left and right subtrees.

Python Code

```
python
CopyEdit
def sum_of_leaf_nodes(root):
    if not root:
        return 0
```

```
    if not root.left and not root.right:
        return root.key
    return sum_of_leaf_nodes(root.left) + sum_of_leaf_nodes(root.right)

# Usage
print("Sum of all leaf nodes:", sum_of_leaf_nodes(root))
```

🔷 5. Find the Longest Increasing Sequence in a BST

Algorithm

1. Perform an **inorder traversal** while tracking the longest increasing sequence.
2. If the current node is greater than the previous, increase the sequence length.
3. If not, reset the sequence length and update the maximum found.

Python Code

```
python
CopyEdit
def longest_increasing_bst(root):
    def inorder(node, prev, length, max_length):
        if not node:
            return max_length
        max_length = inorder(node.left, prev, length, max_length)
        if prev[0] and node.key > prev[0].key:
            length += 1
        else:
            length = 1
        max_length = max(max_length, length)
        prev[0] = node
        return inorder(node.right, prev, length, max_length)

    return inorder(root, [None], 0, 0)
```

```
# Usage
print("Length of the longest increasing sequence:", longest_increasing_bst(root))
```