Binary Search Tree

October 13, 2023

1 Binary Search Tree (BST) Implementation in Python

The purpose of this Jupyter Notebook is to implement a Binary Search Tree (BST) in Python and demonstrate its functionality. A Binary Search Tree is a data structure that organizes its elements in a way that allows for efficient search, insertion, and deletion operations. In this notebook, we will implement two classes: `TreeNode`, representing a node in the BST, and `BinarySearchTree`, representing the BST itself.

The key characteristics of a Binary Search Tree are: - Each node has at most two children, a left child and a right child. - For each node, all elements in the left subtree are less than the node's key, and all elements in the right subtree are greater than the node's key. - This property ensures that search operations have an average time complexity of O(log n).

Let's start by implementing the classes and then test their functionality.

[ ]:
```python
class TreeNode:
    """
    A class representing a node in a Binary Search Tree.
    """
    def __init__(self, key, val):
        """
        Initializes a new TreeNode instance.

        Parameters:
        key: The key associated with the node.
        val: The value to be stored in the node.
        """
        self.key = key
        self.val = val
        self.left = None  # The left child of the node.
        self.right = None  # The right child of the node.

class BinarySearchTree:
    """
    A class representing a Binary Search Tree.
    """
    def __init__(self, key_value_list):
        """
        Initializes a balanced BinarySearchTree instance.
        Parameters:
        key_value_list: Unsorted list of key and value as tuple.
        """
        # Sort the list by key to ensure the resulting tree is balanced.
        key_value_list.sort()

        # Build a balanced Binary Search Tree from the sorted list of key-value␣
↪pairs.
        self.root = self._build_balanced_bst(key_value_list)

    def search(self, key):
        """
        Searches for a node with the specified key in the Binary Search Tree.

        Parameters:
        key: The key of the node to be found.

        Returns:
        The value of the node with the specified key, or None if the key is not␣
↪in the tree.
        """
        return self._search_recursive(self.root, key)

    def _build_balanced_bst(self, key_value_list):
        """
        Recursively constructs a balanced Binary Search Tree from a sorted list␣
↪of key-value pairs.

        Parameters:
        key_value_list: A sorted list of key-value pairs.

        Returns:
        The root of the balanced Binary Search Tree.
        """
        if not key_value_list:
            return None

        mid = len(key_value_list) // 2
        key, val = key_value_list[mid]
        node = TreeNode(key, val)

        # Recursively build the left and right subtrees, ensuring balance.
        node.left = self._build_balanced_bst(key_value_list[:mid])
        node.right = self._build_balanced_bst(key_value_list[mid + 1:])

        return node
    def _search_recursive(self, node, key):
        """
        Recursively searches for a node with the specified key in the Binary␣
↪Search Tree.

        Parameters:
        node: The current node being examined.
        key: The key of the node to be found.

        Returns:
        The value of the node with the specified key, or None if the key is not␣
↪in the tree.
        """
        if node is None:
            return None

        if key == node.key:
            return node.val
        elif key < node.key:
            return self._search_recursive(node.left, key)
        else:
            return self._search_recursive(node.right, key)

# Create a list of unsorted key-value pairs.
key_value_list = [(4, "apple"), (2, "banana"), (6, "cherry"), (1, "date"), (3,␣
↪"fig"), (5, "grape"), (7, "kiwi")]

# Create a BinarySearchTree instance.
bst = BinarySearchTree(key_value_list)

# Search for values using keys.
print(bst.search(2))  # Output should be "banana"
print(bst.search(7))  # Output should be "kiwi"
print(bst.search(9))  # Output should be None, as the key is not in the tree
```

```
banana
kiwi
None
```

2 Conclusion

In this notebook, we've successfully implemented a Binary Search Tree (BST) in Python. We defined two classes, `TreeNode` and `BinarySearchTree`, to represent the nodes and the tree itself. The tree is constructed from an unsorted list of key-value pairs, ensuring balance through sorting. The `search` operation allows us to find values associated with specific keys efficiently.

The Binary Search Tree data structure is a valuable tool for various applications, providing fast look-up and retrieval times. Understanding its principles and implementation is crucial for solving problems efficiently in computer science and programming.

This implementation is a starting point, and you can further expand it by adding features like insertion, deletion, and traversal. You can explore and experiment with more advanced applications of Binary Search Trees as well.

[ ]:

```
```