```
from typing import Any
from indexer.trees.bst_node import BSTNode

class AVLNode(BSTNode):
    """
    AVLNode class represents a node in an AVL tree. It inherits from
BSTNode
    adds 1 additional attribute, height, used in balancing the tree.

    Attributes:
        key (Any): The key value stored in the node.
        height (int): The height of the node in the AVL tree.

    Methods:
        __init__(key: Any): Initializes a new instance of the AVLNode
class
        with the given key.
    """
    def __init__(self, key: Any):
        super().__init__(key)
        self.height: int = 1
```

—--------------------------

```
from typing import List, Optional, Any

from indexer.trees.bst_index import BinarySearchTreeIndex
from indexer.trees.avl_node import AVLNode


class AVLTreeIndex(BinarySearchTreeIndex):
    """
    An AVL Tree implementation of an index that maps a key to a list
of values.
    AVLTreeIndex inherits from BinarySearchTreeIndex meaning it
automatically
    contains all the data and functionality of BinarySearchTree.  Any
    functions below that have the same name and param list as one in
    BinaryTreeIndex overrides (replaces) the BSTIndex functionality.

    Methods:
        insert(key: Any, value: Any) -> None:
            Inserts a new node with key and value into the AVL Tree
    """
```

```
    def __init(self):
        super().__init__()
        self.root: Optional[AVLNode] = None

    def _height(self, node: Optional[AVLNode]) -> int:
        """
        Calculate the height of the given AVLNode.

        Parameters:
        - node: The AVLNode for which to calculate the height.

        Returns:
        - int: The height of the AVLNode. If the node is None,
returns 0.
        """
        # if the given node is None, return 0
        if not node:
            return 0

        # otherwise, return its height
        return node.height

    def _check_balance(self, node: AVLNode) -> int:
        """
        Calculate the balance factor (difference between height of
left and right
        subtrees) of the given AVLNode.

        Parameters:
        - node: The AVLNode for which to calculate the balance
factor.

        Returns:
        - int: The balance factor of the AVLNode. If the node is
None, returns 0.
        """
        # if the node is None, return 0
        if not node:
            return 0

        # otherwise, return the height of the left subtree - the
height of the right subtree
        else:
            return self._height(node.left) - self._height(node.right)
```

```python
    def _rotate_right(self, y: AVLNode) -> AVLNode:
        """
        Performs a right rotation on the AVL tree.

        Args:
            y (AVLNode): The node to be rotated.

        Returns:
            AVLNode: The new root of the rotated subtree.
        """
        # check whether y or y left is None, in which case the
rotation would not work
        if y is None or y.left is None:
            return y

        # set a new node with the initial left as the root of the
subtree
        rotated_node = y.left

        # save the right of the rotated node to later append to y
        T2 = rotated_node.right

        # the right of the new node is the initial node (y)
        rotated_node.right = y

        # the left of y ^ is T2
        rotated_node.right.left = T2

        # update height
        y.height = 1 + max(self._height(y.left),
self._height(y.right))
        rotated_node.height = 1 +
max(self._height(rotated_node.left),
self._height(rotated_node.right))

        return rotated_node

    def _rotate_left(self, x: AVLNode) -> AVLNode:
        """
        Rotate the given node `x` to the left.
        Args:
            x (AVLNode): The node to be rotated.
        Returns:
            AVLNode: The new root of the subtree after rotation.
```

```
        """
        # check whether x or x.right is None
        if x is None or x.right is None:
            return x

        # x.right becomes new root of subtree
        rotated_node = x.right

        # save the left of the node to become the right of the left
in the rotation
        future_lr = rotated_node.left

        # the subtree's left is the old root
        rotated_node.left = x

        # the prior left is now the right of the left
        rotated_node.left.right = future_lr

        # update height
        x.height = 1 + max(self._height(x.left),
self._height(x.right))
        rotated_node.height = 1 +
max(self._height(rotated_node.left),
self._height(rotated_node.right))

        return rotated_node

    def _insert_recursive(self, current: Optional[AVLNode], key: Any,
value: Any) -> AVLNode:
        """
        Recursively inserts a new node with the given key and value
into the AVL tree.
        Args:
            current (Optional[AVLNode]): The current node being
considered during the recursive insertion.
            key (Any): The key of the new node.
            value (Any): The value of the new node.
        Returns:
            AVLNode: The updated AVL tree with the new node inserted.
        """
        # check if current exists or if it's an empty AVL tree
        if not current:
            node = AVLNode(key)
            node.add_value(value)
            return node
```

```
        # check if the key of the node to insert is less than the
current
        if key < current.key:

            # recur on current.left
            current.left = self._insert_recursive(current.left, key,
value)

        # check if the key of the node to insert is greater than the
current key
        elif key > current.key:

            # recur on current.right
            current.right = self._insert_recursive(current.right,
key, value)

        # if key is equal to the current key, append value to list of
values
        elif key == current.key:
            current.add_value(value)

        # update height
        current.height = 1 + max(self._height(current.left),
self._height(current.right))

        # check the balance of the current node
        bal = self._check_balance(current)

        # left left case
        if bal > 1 and key < current.left.key:
            # one rotation to the right
            return self._rotate_right(current)

        # left right case
        if bal > 1 and key > current.left.key:
            # rotate to the left then the right
            current.left = self._rotate_left(current.left)
            return self._rotate_right(current)

        # right right case
        if bal < -1 and key > current.right.key:
            # one rotation to the left
            return self._rotate_left(current)
```

```
        # right left case
        if bal < -1 and key < current.right.key:
            # rotate to te right then the left
            current.right = self._rotate_right(current.right)
            return self._rotate_left(current)

        return current

    def insert(self, key: Any, value: Any) -> None:
        """
        Inserts a key-value pair into the AVL tree. If the key
exists, the
         value will be appended to the list of values in the node.

        Parameters:
            key (Any): The key to be inserted.
            value (Any): The value associated with the key.

        Returns:
            None
        """
        # if the root is none, create a node to insert with the key
value
        if self.root is None:
            self.root = AVLNode(key)
            self.root.add_value(value)

        # otherwise, call _insert_recursive to insert the key value
in its spot
        else:
            self.root = self._insert_recursive(self.root, key, value)

    def _inorder_traversal(self, current: Optional[AVLNode], result:
List[Any]) -> None:
        if current is None:
            return

        self._inorder_traversal(current.left, result)
        result.append(current.key)
        self._inorder_traversal(current.right, result)

    def get_keys(self) -> List[Any]:
        keys: List[Any] = []
        self._inorder_traversal(self.root, keys)
        return keys
```