Option #1: Binary Search Tree

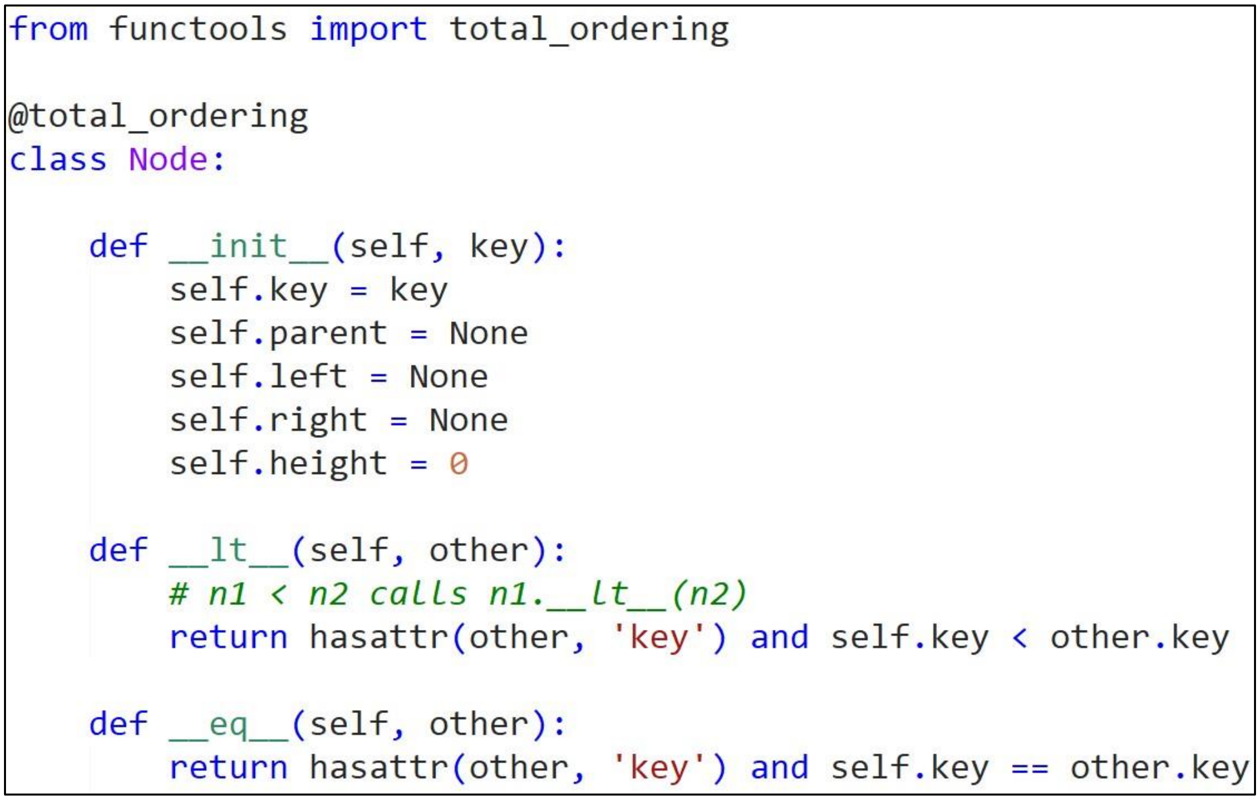
Scott Miner

Colorado State University – Global Campus

Abstract

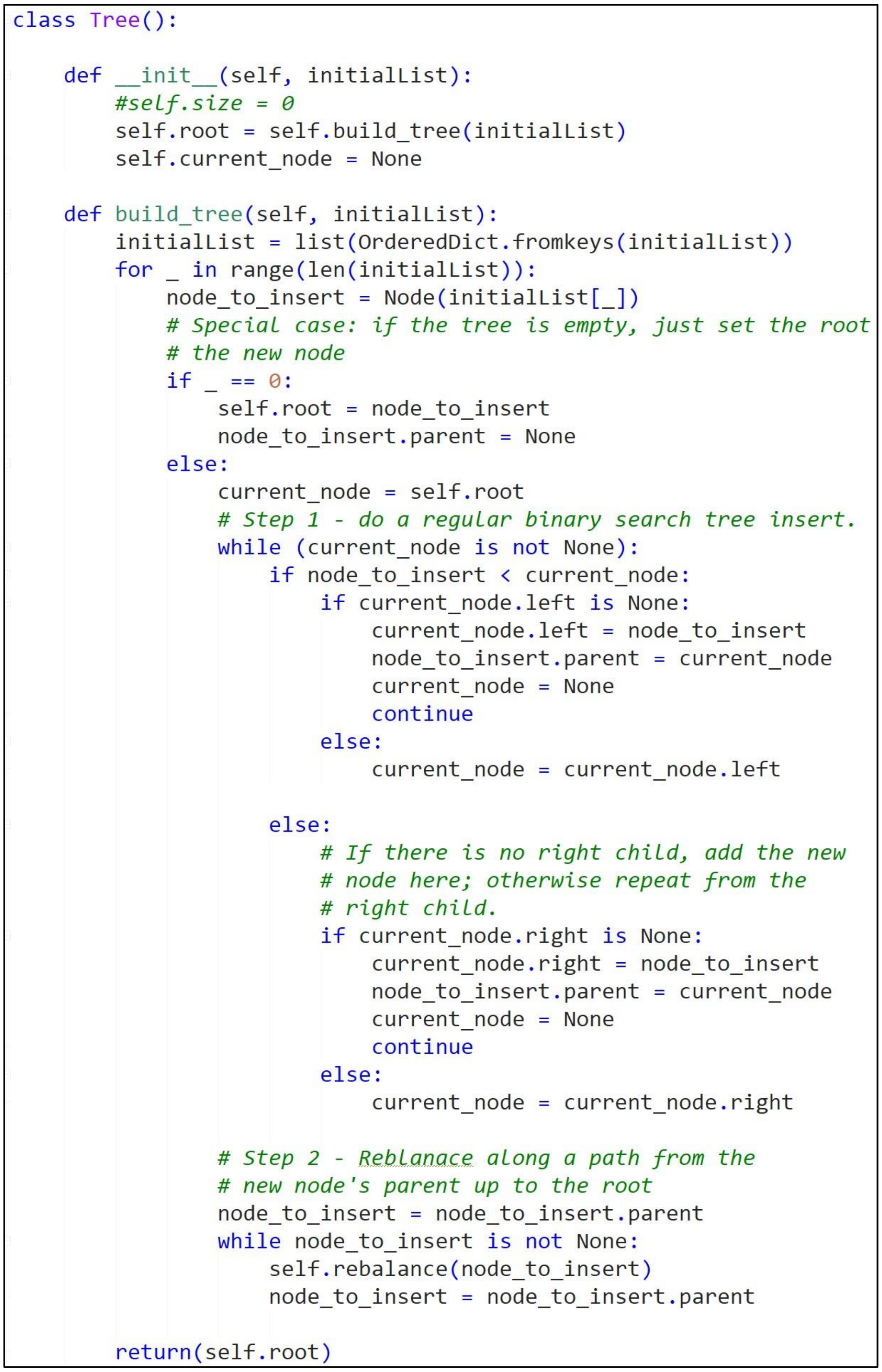
This paper introduces an efficient algorithm for constructing and maintaining a balanced binary search tree (BST) or AVL tree from a given list of items. The proposed algorithm sorts the input items according to the ordering property of the BST while ensuring the insertion of only unique values. The implementation provides insert() and delete() methods that facilitate the addition and removal of nodes within the tree while preserving its balance. The algorithm demonstrates the successful execution of these methods in a program, maintaining a balanced BST structure throughout various insertions and deletions.

The paper also offers an overview of BSTs and AVL trees, highlighting the importance of maintaining tree height using rebalancing operations closely linked with update operations. By implementing such rebalancing strategies, the AVL tree can significantly reduce the runtime complexity of BST update operations from O(N) to O(log N). This work presents a comprehensive approach to constructing and managing balanced BSTs, enhancing performance and efficiency in various applications.



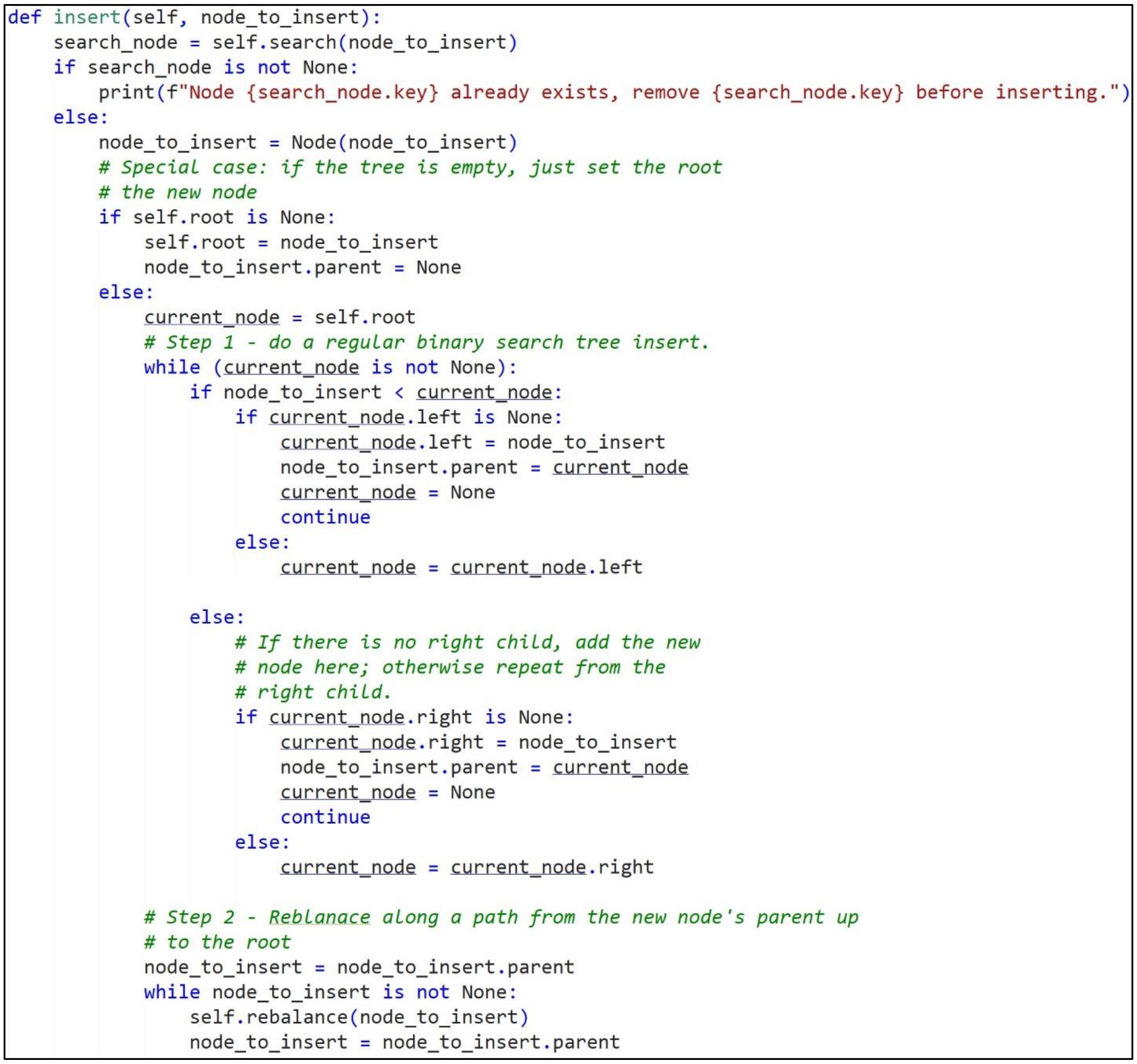
```python
from functools import total_ordering

@total_ordering
class Node:

    def __init__(self, key):
        self.key = key
        self.parent = None
        self.left = None
        self.right = None
        self.height = 0

    def __lt__(self, other):
        # n1 < n2 calls n1.__lt__(n2)
        return hasattr(other, 'key') and self.key < other.key

    def __eq__(self, other):
        return hasattr(other, 'key') and self.key == other.key
```

Figure 1. Node class constructor with overridden comparison operators, enabling effortless node comparisons based on key values.



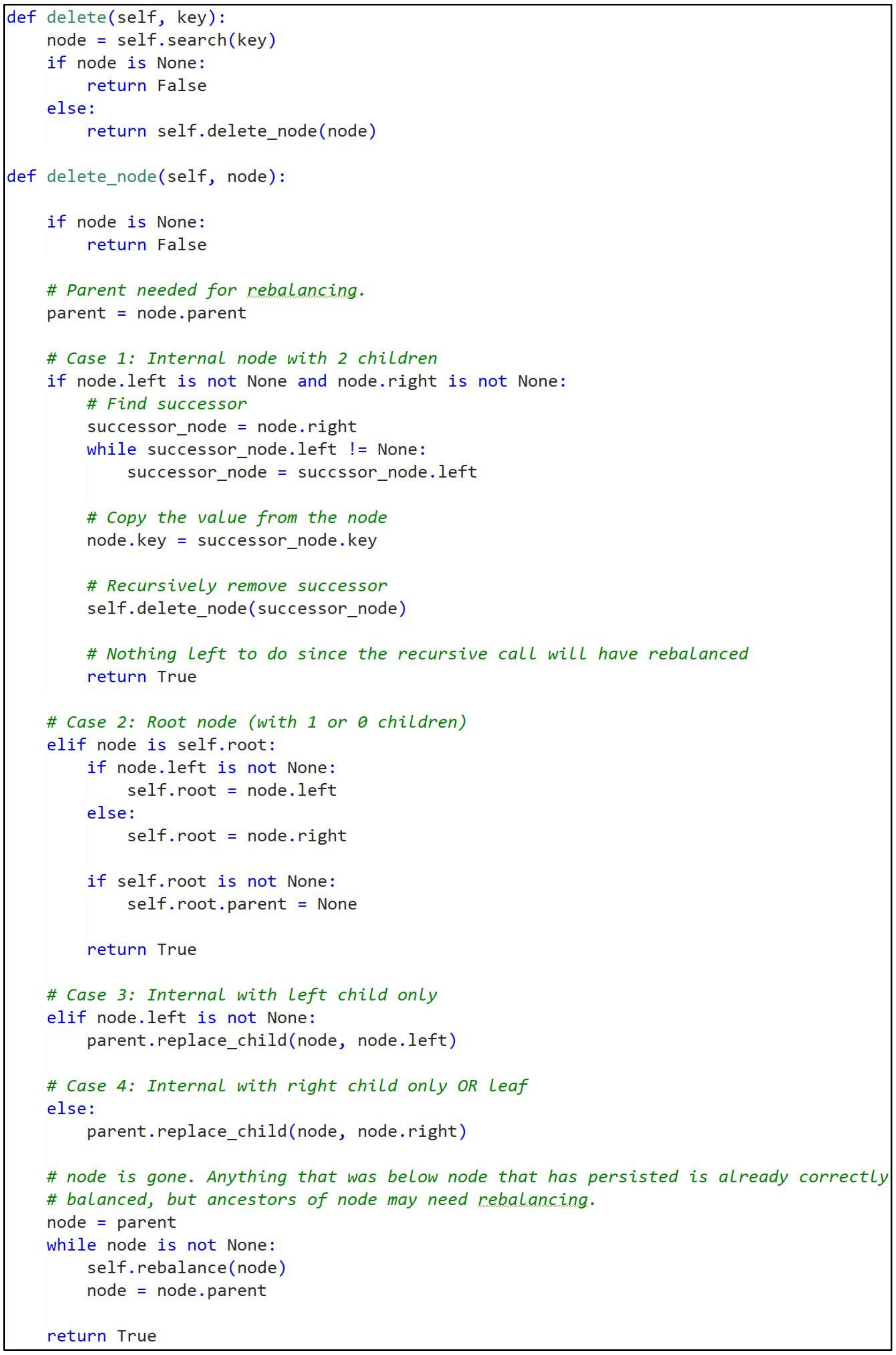
```
class Tree():

    def __init__(self, initialList):
        #self.size = 0
        self.root = self.build_tree(initialList)
        self.current_node = None

    def build_tree(self, initialList):
        initialList = list(OrderedDict.fromkeys(initialList))
        for _ in range(len(initialList)):
            node_to_insert = Node(initialList[_])
            # Special case: if the tree is empty, just set the root
            # the new node
            if _ == 0:
                self.root = node_to_insert
                node_to_insert.parent = None
            else:
                current_node = self.root
                # Step 1 - do a regular binary search tree insert.
                while (current_node is not None):
                    if node_to_insert < current_node:
                        if current_node.left is None:
                            current_node.left = node_to_insert
                            node_to_insert.parent = current_node
                            current_node = None
                            continue
                        else:
                            current_node = current_node.left

                    else:
                        # If there is no right child, add the new
                        # node here; otherwise repeat from the
                        # right child.
                        if current_node.right is None:
                            current_node.right = node_to_insert
                            node_to_insert.parent = current_node
                            current_node = None
                            continue
                        else:
                            current_node = current_node.right

                # Step 2 - Reblanace along a path from the
                # new node's parent up to the root
                node_to_insert = node_to_insert.parent
                while node_to_insert is not None:
                    self.rebalance(node_to_insert)
                    node_to_insert = node_to_insert.parent

        return(self.root)
```

Figure 2. Tree class initialization and root attribute, utilizing the build_tree() method to construct the balanced binary search tree.



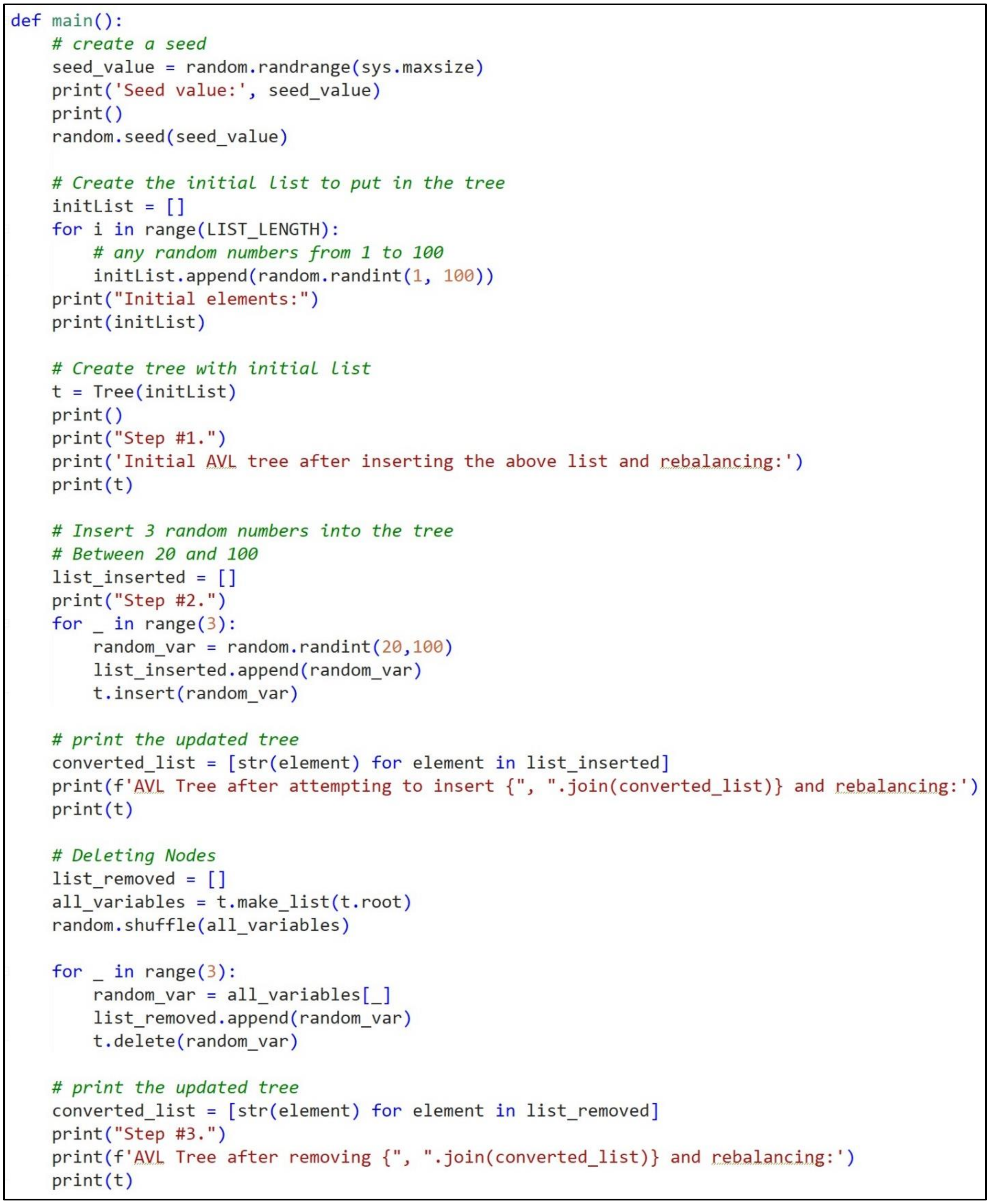
```python
def insert(self, node_to_insert):
    search_node = self.search(node_to_insert)
    if search_node is not None:
        print(f"Node {search_node.key} already exists, remove {search_node.key} before inserting.")
    else:
        node_to_insert = Node(node_to_insert)
        # Special case: if the tree is empty, just set the root
        # the new node
        if self.root is None:
            self.root = node_to_insert
            node_to_insert.parent = None
        else:
            current_node = self.root
            # Step 1 - do a regular binary search tree insert.
            while (current_node is not None):
                if node_to_insert < current_node:
                    if current_node.left is None:
                        current_node.left = node_to_insert
                        node_to_insert.parent = current_node
                        current_node = None
                        continue
                    else:
                        current_node = current_node.left

                else:
                    # If there is no right child, add the new
                    # node here; otherwise repeat from the
                    # right child.
                    if current_node.right is None:
                        current_node.right = node_to_insert
                        node_to_insert.parent = current_node
                        current_node = None
                        continue
                    else:
                        current_node = current_node.right

            # Step 2 - Reblanace along a path from the new node's parent up
            # to the root
            node_to_insert = node_to_insert.parent
            while node_to_insert is not None:
                self.rebalance(node_to_insert)
                node_to_insert = node_to_insert.parent
```

Figure 3. Insert() method implementation for adding nodes to the binary search tree while maintaining balance.



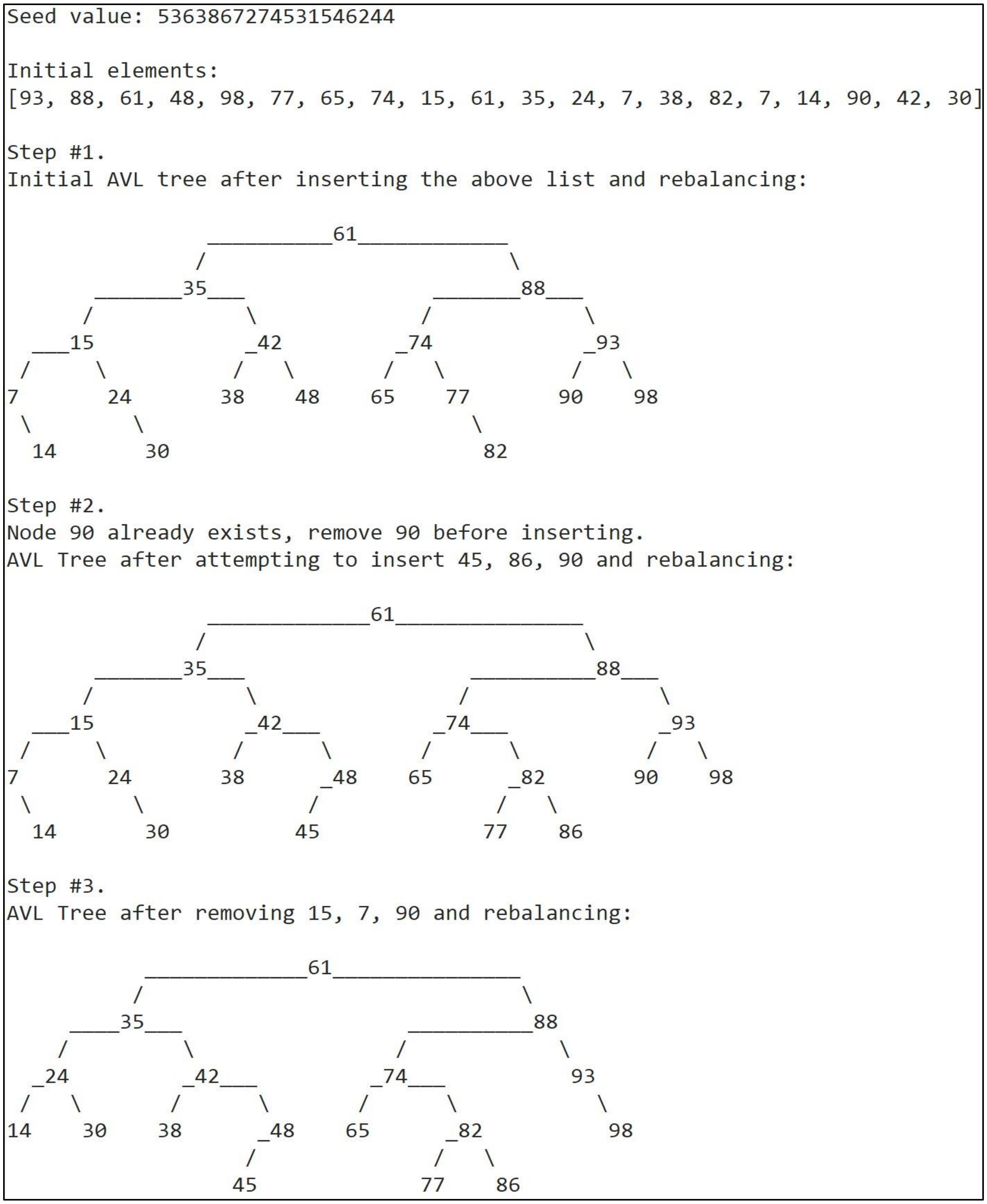
```
def delete(self, key):
    node = self.search(key)
    if node is None:
        return False
    else:
        return self.delete_node(node)

def delete_node(self, node):

    if node is None:
        return False

    # Parent needed for rebalancing.
    parent = node.parent

    # Case 1: Internal node with 2 children
    if node.left is not None and node.right is not None:
        # Find successor
        successor_node = node.right
        while successor_node.left != None:
            successor_node = succssor_node.left

        # Copy the value from the node
        node.key = successor_node.key

        # Recursively remove successor
        self.delete_node(successor_node)

        # Nothing left to do since the recursive call will have rebalanced
        return True

    # Case 2: Root node (with 1 or 0 children)
    elif node is self.root:
        if node.left is not None:
            self.root = node.left
        else:
            self.root = node.right

        if self.root is not None:
            self.root.parent = None

        return True

    # Case 3: Internal with left child only
    elif node.left is not None:
        parent.replace_child(node, node.left)

    # Case 4: Internal with right child only OR leaf
    else:
        parent.replace_child(node, node.right)

    # node is gone. Anything that was below node that has persisted is already correctly
    # balanced, but ancestors of node may need rebalancing.
    node = parent
    while node is not None:
        self.rebalance(node)
        node = node.parent

    return True
```

Figure 4. Delete() and delete_node() methods for removing nodes from the binary search tree and ensuring proper tree balance.



```python
def main():
    # create a seed
    seed_value = random.randrange(sys.maxsize)
    print('Seed value:', seed_value)
    print()
    random.seed(seed_value)

    # Create the initial list to put in the tree
    initList = []
    for i in range(LIST_LENGTH):
        # any random numbers from 1 to 100
        initList.append(random.randint(1, 100))
    print("Initial elements:")
    print(initList)

    # Create tree with initial list
    t = Tree(initList)
    print()
    print("Step #1.")
    print('Initial AVL tree after inserting the above list and rebalancing:')
    print(t)

    # Insert 3 random numbers into the tree
    # Between 20 and 100
    list_inserted = []
    print("Step #2.")
    for _ in range(3):
        random_var = random.randint(20,100)
        list_inserted.append(random_var)
        t.insert(random_var)

    # print the updated tree
    converted_list = [str(element) for element in list_inserted]
    print(f'AVL Tree after attempting to insert {", ".join(converted_list)} and rebalancing:')
    print(t)

    # Deleting Nodes
    list_removed = []
    all_variables = t.make_list(t.root)
    random.shuffle(all_variables)

    for _ in range(3):
        random_var = all_variables[_]
        list_removed.append(random_var)
        t.delete(random_var)

    # print the updated tree
    converted_list = [str(element) for element in list_removed]
    print("Step #3.")
    print(f'AVL Tree after removing {", ".join(converted_list)} and rebalancing:')
    print(t)
```

Figure 5. Main() function designed for testing the binary search tree program with insertion and deletion operations.



```
Seed value: 5363867274531546244

Initial elements:
[93, 88, 61, 48, 98, 77, 65, 74, 15, 61, 35, 24, 7, 38, 82, 7, 14, 90, 42, 30]

Step #1.
Initial AVL tree after inserting the above list and rebalancing:

                __________61____________
               /                        \
       _______35___               _______88___
      /            \             /            \
  ___15            _42         _74            _93
 /     \          /   \       /   \          /   \
7       24       38    48    65    77       90    98
 \        \                          \
  14       30                         82

Step #2.
Node 90 already exists, remove 90 before inserting.
AVL Tree after attempting to insert 45, 86, 90 and rebalancing:

                _____________61_______________
               /                              \
       _______35___                  __________88___
      /            \                /               \
  ___15            _42___         _74___            _93
 /     \          /      \       /      \          /   \
7       24       38      _48    65      _82       90    98
 \        \             /              /   \
  14       30          45             77    86

Step #3.
AVL Tree after removing 15, 7, 90 and rebalancing:

           _____________61_______________
          /                              \
     ____35___                  __________88
    /         \                /            \
  _24         _42___         _74___          93
 /   \       /      \       /      \           \
14    30    38      _48    65      _82          98
                   /              /   \
                  45             77    86
```

Figure 6. Program output highlighting the balanced binary search tree after insertion and deletion operations.

Table of Contents

Option #1: Binary Search Tree

This paper presents the construction of a simple binary search tree (BST) using a Node class and a Tree class.

Node Class

The Node class contains attributes for a node's key value, parent node, left and right children, and height. The constructor for the Node class overrides the comparison operators to allow for the easy comparison of nodes based on their key values (Figure 1).

Tree Class and build_tree() Method

The Tree class accepts an array when initialized and contains an attribute, root, which uses the return value of the build_tree() method (Figure 2). The build_tree() method first removes any duplicate values from the initial list, as BSTs can only contain unique values. Then, the method iterates through each list value, inserting each key per the BST's ordering property.

Ordering Property of BST

The ordering property of a BST is defined as follows: the keys of any node's left subtree are ≤ the node's key and the keys of any node's right subtree are ≥ the node's key (Lysecky & Vahid, 2019). The runtime complexity of the BST insertion algorithm varies between O(log N) in the best case and O(N) in the worst case.

AVL Trees and Rebalancing

An AVL tree is a BST with a height property and rebalancing operations that maintain a tree's balance factor after the insertion and deletion of nodes (Lysecky & Vahid, 2019; Bronson *et al.*, 2010). AVL trees are self-balancing BSTs named after their inventors, Adelson-Velsky and Landis (Tsakalidis, 1985).

In an AVL tree, the heights of the left and right subtrees below any node differ by at most one (e.g., -1, 0, or 1) (Davis, 1987). The build_tree() method in Figure 2 traverses a path from a newly inserted node's parent up to the tree's root, rebalancing each node along the way for those with balance factors of 2 or -2. After inserting the elements and balancing the BST, the build_tree() method returns the tree's root to the calling function.

Insert and Delete Methods

Search Function

The insert() and delete() methods accept values to insert into and delete from the BST (Figures 3 and 4). The delete() method begins by calling the search() function to locate a node to delete. The insertion algorithm also calls this function and, if the algorithm encounters a node already in the tree, outputs a message to the user conveying the node must be deleted before it can be inserted. After the deletion algorithm removes a node from an AVL tree, it traverses a path from the parent of the removed node up to the tree's root, rebalancing all the removed node's ancestors along the way if needed (Lysecky & Vahid, 2019).

Runtime and Space Complexity

The worst-case scenario requires visiting all levels of the BST to find a node to delete and then traversing back up to the tree's root to rebalance it. The algorithm visits one node per level and makes, at most, two rotations per node to rebalance the tree, making the runtime complexity of the AVL tree removal algorithm O(log N). The insertion and deletion algorithms use a fixed number of pointers to perform these operations, making the space complexity O(1).

Program Implementation and Testing

Building and Balancing the BST

The main() function of the program (Figure 5) tests the build_tree(), insert(), and delete() functions by generating a list of 20 random integers between 1 and 100 with replacement. The program then creates an instance of the Tree class by passing the randomly generated list of integers as an argument to the Tree class's constructor. The constructor calls the build_tree() method with the list of integers and returns the root node to the calling function. Step 1 of Figure 6 shows this newly constructed, balanced BST.

Inserting and Deleting Nodes

Step 2 of the main() function inserts three random values between 20 and 100 into the BST. If the data structure already contains any of these values, the algorithm outputs an informative message to the user, as shown in Figure 6, since BSTs can only contain unique values. Step 3 of the main() function generates a list of all node keys in the BST, shuffles the list, and randomly removes the first three values from the BST. Figure 6 shows the rebalanced BST after removing the integers 15, 7, and 90.

Conclusion

This paper presented an algorithm for constructing a balanced binary search tree (BST) or AVL tree from a list of items. The algorithm sorted all array items according to the ordering property of the BST and ensured that only unique values could be inserted into the BST. The insert() and delete() methods of the algorithm allowed for the insertion and removal of nodes from the BST, with successful execution demonstrated in the program screenshots.

After inserting or removing a node, the algorithm traversed a path from the node's parent to the tree's root, calculating the balance factor at each node and performing any needed rotations for balance factors of 2 or -2. This paper also provided an overview of BSTs and AVL trees. By maintaining the height of BSTs using rebalancing operations tightly coupled with update operations, AVL trees can reduce the runtime complexity of BST update operations from O(N) to O(log N).

References

Bronson, N., Casper, J., Chafi, H., & Olukotun, K. (2010). A Practical Concurrent Binary Search Tree. In *ACM SIGPLAN Notices* (Vol. 45, p. 268). https://doi.org/10.1145/1693453.1693488

Davis, I. J. (1987). A locally correctable AVL tree. *Digest of Papers: The Seventeenth International Symposium on Fault-Tolerant Computing*, 85–88.

Larsen, K. S. (1994). AVL trees with relaxed balance. *Proceedings of 8th International Parallel Processing Symposium*, 888–893.

Lysecky, R., & Vahid, F. (2019). *Data Structures Essential: Pseudocode with Python Examples*. Zyante Inc. (zyBooks.com).

Tsakalidis, A. K. (1985). AVL-trees for localized search. *Information and Control*, *67*(1–3), 173–194.