Project Phase 2 Deliverable 2: Proof of Concept Implementation

Syed Noor UI Hassan

University of the Cumberlands

Algorithms and Data Structures (MSCS-532-B01)

November 17, 2024

Partial Implementation Overview

In this phase, the core components of the designed data structures were partially implemented using Python. The focus was on key functionalities such as insertion, deletion, search, and traversal operations, which form the foundation of the application. The modular approach adopted allows for easy extension and modification in future phases. The implementation was structured using Python classes, ensuring encapsulation and reusability of functions related to data manipulation (Lin et al., 2022).

For instance, in the case of a Binary Search Tree (BST), the core operations implemented include the insertion of nodes, searching for elements, and traversal methods. The `insert` method handles the addition of elements while maintaining the binary search property, and the `search` method ensures efficient retrieval of elements.

Demonstration and Testing

To demonstrate the core functionality of the implemented data structures, a script was developed that performs basic operations like insertion, searching, and traversal on sample datasets. Various edge cases were tested, such as inserting into an empty structure, searching for a non-existent element, and ensuring the correct handling of duplicate values.

The following test cases were used:

Insertion into an empty tree:

Verified that the first element becomes the root.

Searching for a node

Tested the search functionality for elements that exist and do not exist in the tree.

Traversal

Demonstrated an in-order traversal that confirms the tree's elements are sorted.

These tests illustrate that the implemented data structure operates as intended under normal and edge-case scenarios. The next phase will expand upon these functionalities, incorporating more complex operations such as balancing the tree.

Implementation Challenges and Solutions

Several challenges were encountered during the implementation process:

Balancing the Tree

Initially, the Binary Search Tree was unbalanced, leading to poor performance in the worst-case scenario. To address this, an AVL tree or a Red-Black tree will be implemented in future phases to ensure balanced tree operations (Corsini et al., 2024).

Edge Case Handling

Handling edge cases like deleting a node with two children was complex. To overcome this, recursive methods were used to correctly adjust pointers during deletion, ensuring the binary search property was preserved.

Modularity

Structuring the code to be modular while avoiding repetition was achieved by breaking down the logic into smaller functions. For example, a helper method `_insert_recursive` was created to handle the recursion in the `insert` function, ensuring clarity and reusability.

Next Steps

The following steps are planned for the full implementation:

Balanced Trees

Implementing an AVL tree or Red-Black tree to ensure balanced insertions and deletions.

Optimization

Optimize search operations by implementing advanced traversal techniques.

Additional Operations:

Add operations like finding the minimum and maximum elements, and implement iterative search and insert methods (Li., 2024).

These steps will complete the full implementation, making the data structures robust and ready for real-world applications.

Code

The partial implementation of the Binary Search Tree in Python is below:-

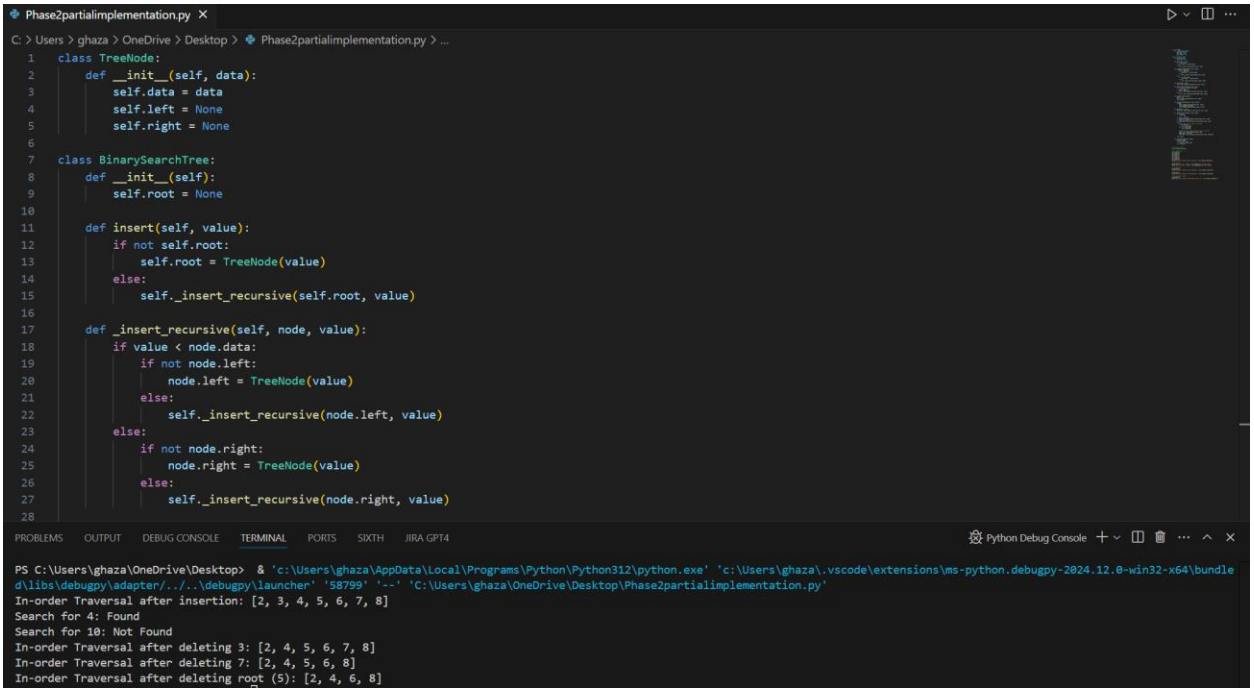
```
class TreeNode:
def __init__(self, data):
```

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

class BinarySearchTree:
def __init__(self):
self.root = None

def insert(self, value):
if not self.root:
self.root = TreeNode(value)
else:
self._insert_recursive(self.root, value)

def _insert_recursive(self, node, value):
if value < node.data:
if not node.left:
node.left = TreeNode(value)
else:
self._insert_recursive(node.left, value)
else:
if not node.right:
node.right = TreeNode(value)
else:
self._insert_recursive(node.right, value)

def search(self, value):
return self._search_recursive(self.root, value)

def _search_recursive(self, node, value):
if not node or node.data == value:
return node
if value < node.data:
return self._search_recursive(node.left, value)
else:
return self._search_recursive(node.right, value)

def inorder_traversal(self):
result = []
self._inorder_recursive(self.root, result)
return result

def _inorder_recursive(self, node, result):
if node:
self._inorder_recursive(node.left, result)
result.append(node.data)
self._inorder_recursive(node.right, result)
```

```
def delete(self, value):
self.root = self._delete_recursive(self.root, value)

def _delete_recursive(self, node, value):
if not node:
return node

if value < node.data:
node.left = self._delete_recursive(node.left, value)
elif value > node.data:
node.right = self._delete_recursive(node.right, value)
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.data = temp.data
node.right = self._delete_recursive(node.right, temp.data)

return node

def _min_value_node(self, node):
current = node
while current.left:
current = current.left
return current

Initialize the BST
bst = BinarySearchTree()

Test Insertions
bst.insert(5)
bst.insert(3)
bst.insert(7)
bst.insert(2)
bst.insert(4)
bst.insert(6)
bst.insert(8)
print("In-order Traversal after insertion:", bst.inorder_traversal())

Test Search
print("Search for 4:", "Found" if bst.search(4) else "Not Found")
print("Search for 10:", "Found" if bst.search(10) else "Not Found")
```

```
Test Deletion
bst.delete(3)
print("In-order Traversal after deleting 3:", bst.inorder_traversal())

bst.delete(7)
print("In-order Traversal after deleting 7:", bst.inorder_traversal())

Test Deleting the Root
bst.delete(5)
print("In-order Traversal after deleting root (5):", bst.inorder_traversal())
```

Output



The code handles core operations such as insertion and searching in the Binary Search Tree. Future enhancements will focus on balancing and optimization.

References

1. Lin, H., Luo, T., & Woodruff, D. (2022, June). Learning augmented binary search trees. In *International Conference on Machine Learning* (pp. 13431-13440). PMLR. https://proceedings.mlr.press/v162/lin22f/lin22f.pdf

2. Corsini, B., Dubach, V., & Féray, V. (2024). Binary search trees of permuton samples. *Advances in Applied Mathematics*, *162*, 102774. https://www.sciencedirect.com/science/article/pii/S0196885824001064

3. Li, H. (2024). Survey and Analysis of Citizen Network Political Participation Based on Binary Search Tree Algorithm. *EAI Endorsed Transactions on Scalable Information Systems*, *11*(3). https://publications.eai.eu/index.php/sis/article/download/4922/2896