Report: B+ Tree Implementation in Python

Chaoren Wang, 122090513

December 2, 2024

1 Introduction

This report details the implementation of a B+ Tree data structure in Python. The implementation includes all core functionalities required for a B+ Tree:

- Insertion with node splitting and redistribution
- Deletion with node merging and rotation (i.e. borrowing from siblings)
- Search operations using binary search
- Optional display functionality for ASCII tree visualization

The implementation maintains the essential properties of B+ Trees, including balanced height, ordered keys, and linked leaf nodes for efficient range queries.

The code is designed to be both efficient and maintainable, with a focus on clear separation of concerns and proper documentation. Besides, logging and debugging features are also implemented for development and testing purposes. Four testcases mentioned in Lecture 8 are provided to test the correctness of the implementation.

2 Design & Implementation

2.1 Core Classes

The implementation consists of two main classes:

- `BPlusTreeNode`: Represents individual nodes in the tree, including leaf nodes and non-leaf nodes
- `BPlusTree`: Manages the tree structure and operations, providing APIs for insertion, deletion, search, and display

2.2 Key Design Choices

Node Structure Every node (including leaf nodes and non-leaf nodes) maintains lists for keys and values separately; Leaf nodes are doubly-linked for efficient range queries; Parent references are maintained for bottom-up operations (like merging and rotation).

Operation Implementation For four operations (insertion, deletion, search, and display), the implementation follows the lecture notes closely: (1) Insertion uses recursive splitting with bottom-up propagation; (2) Deletion implements both rotation and merging strategies; (3) Search utilizes binary search within nodes; (4) Optional `--plain` argument provides a plain display mode without ASCII tree format.

OOP Design The implementation is designed using object-oriented programming (OOP) techniques, which helps to keep clean code and easy to understand.

2.3 Challenges and Solutions

2.3.1 Challenge 1: Node Splitting

The main challenge was maintaining tree properties during node splits after insertion.

When inserting a new key, the node needs to be split if it is full. This involves redistributing keys and updating parent references. However, redistribution is not a trivial task, specifically when it involves updating the parent's keys.

To solve this, I implemented `node.split` and `tree._merge_into_parent` functions to handle the splitting and merging process, which is called recursively when necessary.

2.3.2 Challenge 2: Deletion Redistribution

When deleting a key, the node needs to be redistributed if it has insufficient keys. This involves borrowing keys from siblings or merging with siblings and updating the parent keys.

More challenging, when parents' keys are updated, the redistribution process needs to be recursively applied to the parent nodes until the tree meets the requirement.

To solve this, I implemented `node.remove` and `tree._update_parents` functions to handle the redistribution process. After deletion, the code will first try to borrow keys from siblings by calling `tree._left_rotate` and `tree._right_rotate`; if rotation is not possible, it will merge with siblings by `tree._left_redistribute` and `tree._right_redistribute`, and then update the parent keys by `tree._update_parents`.

3 Code Usage

The code is implemented in `bptree.py`, which is purely written in Python 3.12.7 without any external dependencies.

3.1 Basic Operations

Here's how to use the basic operations:

```
# Create a B+ Tree with order 4
tree = BPlusTree(4)

# Insert values
tree.insert(10)
tree.insert(20)
tree.insert(5)

# Search for a value
```

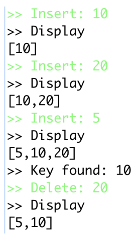
```python
tree.find(10)  # Outputs: ">> Key found: 10"

# Delete a value
tree.delete(20)

# Display the tree
tree.display()
```

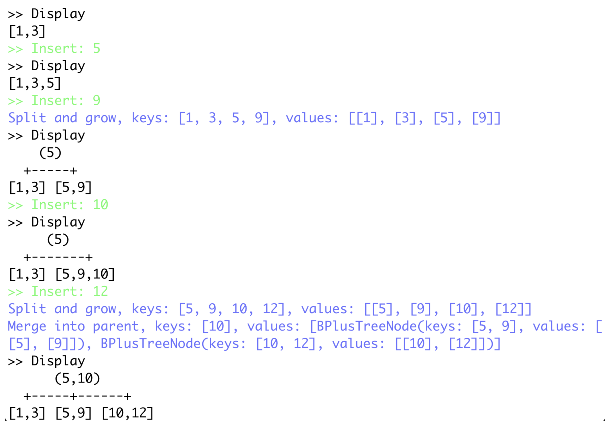
3.2 Example Output

Some example outputs are shown in Figure 1, 2, 3, 4, and 5. These outputs are produced by running with `--debug` option. For plain display mode running with `--plain` option, see Figure 6.



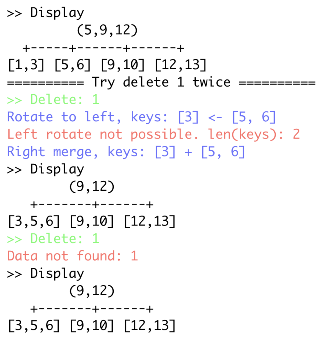
```
>> Insert: 10
>> Display
[10]
>> Insert: 20
>> Display
[10,20]
>> Insert: 5
>> Display
[5,10,20]
>> Key found: 10
>> Delete: 20
>> Display
[5,10]
```

Figure 1: Basic operations



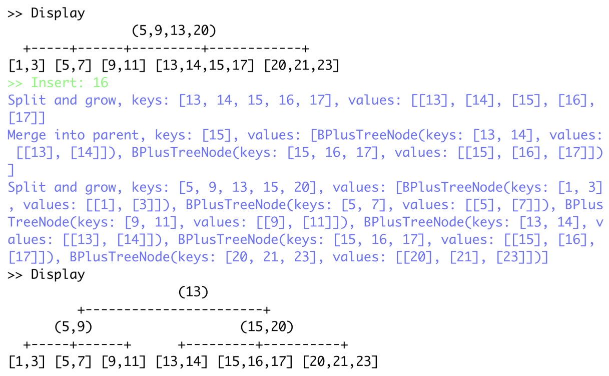
```
>> Display
[1,3]
>> Insert: 5
>> Display
[1,3,5]
>> Insert: 9
Split and grow, keys: [1, 3, 5, 9], values: [[1], [3], [5], [9]]
>> Display
    (5)
  +-----+
[1,3] [5,9]
>> Insert: 10
>> Display
     (5)
  +-------+
[1,3] [5,9,10]
>> Insert: 12
Split and grow, keys: [5, 9, 10, 12], values: [[5], [9], [10], [12]]
Merge into parent, keys: [10], values: [BPlusTreeNode(keys: [5, 9], values: [
[5], [9]]), BPlusTreeNode(keys: [10, 12], values: [[10], [12]])]
>> Display
      (5,10)
  +-----+------+
[1,3] [5,9] [10,12]
```

Figure 2: Insertion operation with split and grow



```
>> Display
         (5,9,12)
  +-----+------+------+
[1,3] [5,6] [9,10] [12,13]
========== Try delete 1 twice ==========
>> Delete: 1
Rotate to left, keys: [3] <- [5, 6]
Left rotate not possible. len(keys): 2
Right merge, keys: [3] + [5, 6]
>> Display
        (9,12)
   +-------+------+
[3,5,6] [9,10] [12,13]
>> Delete: 1
Data not found: 1
>> Display
        (9,12)
   +-------+------+
[3,5,6] [9,10] [12,13]
```

Figure 3: Deletion operation with rotation

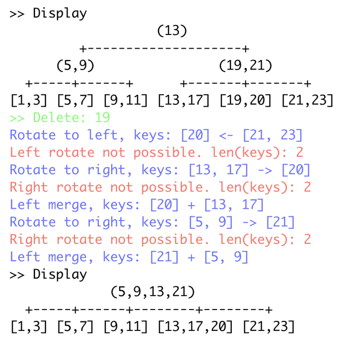


```
>> Display
                (5,9,13,20)
  +-----+------+---------+------------+
[1,3] [5,7] [9,11] [13,14,15,17] [20,21,23]
>> Insert: 16
Split and grow, keys: [13, 14, 15, 16, 17], values: [[13], [14], [15], [16],
[17]]
Merge into parent, keys: [15], values: [BPlusTreeNode(keys: [13, 14], values:
 [[13], [14]]), BPlusTreeNode(keys: [15, 16, 17], values: [[15], [16], [17]])
]
Split and grow, keys: [5, 9, 13, 15, 20], values: [BPlusTreeNode(keys: [1, 3]
, values: [[1], [3]]), BPlusTreeNode(keys: [5, 7], values: [[5], [7]]), BPlus
TreeNode(keys: [9, 11], values: [[9], [11]]), BPlusTreeNode(keys: [13, 14], v
alues: [[13], [14]]), BPlusTreeNode(keys: [15, 16, 17], values: [[15], [16],
[17]]), BPlusTreeNode(keys: [20, 21, 23], values: [[20], [21], [23]])]
>> Display
                      (13)
       +-----------------------+
      (5,9)                 (15,20)
  +-----+------+      +---------+----------+
[1,3] [5,7] [9,11] [13,14] [15,16,17] [20,21,23]
```

Figure 4: Insertion operation with merge into parent

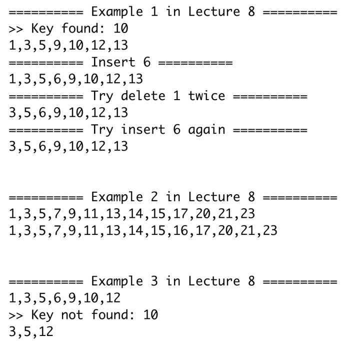
3.3 Additional Features

The implementation includes additional features: (1) Debug logging for operation tracking; (2) Command-line arguments for display options; (3) Colored output for better visualization.



```
>> Display
                   (13)
         +--------------------+
      (5,9)                (19,21)
  +-----+------+      +-------+-------+
[1,3] [5,7] [9,11] [13,17] [19,20] [21,23]
>> Delete: 19
Rotate to left, keys: [20] <- [21, 23]
Left rotate not possible. len(keys): 2
Rotate to right, keys: [13, 17] -> [20]
Right rotate not possible. len(keys): 2
Left merge, keys: [20] + [13, 17]
Rotate to right, keys: [5, 9] -> [21]
Right rotate not possible. len(keys): 2
Left merge, keys: [21] + [5, 9]
>> Display
             (5,9,13,21)
  +-----+------+--------+--------+
[1,3] [5,7] [9,11] [13,17,20] [21,23]
```

Figure 5: Deletion operation with merge



```
========== Example 1 in Lecture 8 ==========
>> Key found: 10
1,3,5,9,10,12,13
========== Insert 6 ==========
1,3,5,6,9,10,12,13
========== Try delete 1 twice ==========
3,5,6,9,10,12,13
========== Try insert 6 again ==========
3,5,6,9,10,12,13


========== Example 2 in Lecture 8 ==========
1,3,5,7,9,11,13,14,15,17,20,21,23
1,3,5,7,9,11,13,14,15,16,17,20,21,23


========== Example 3 in Lecture 8 ==========
1,3,5,6,9,10,12
>> Key not found: 10
3,5,12
```

Figure 6: Plain display mode

3.4 Running the Code

The code can be run with various options:

```
python bptree.py             # Only show tree when calling display()
python bptree.py --display   # Show tree after each operation
python bptree.py --logging   # Enable colored operation logging
python bptree.py --debug     # Enable --display and --logging
python bptree.py --plain     # Show results without ASCII format: this
    format aligns the assignment requirement, compatible with other options
```

4 Conclusion

The implemented B+ Tree maintains O(log n) complexity for all operations while preserving the core properties of B+ Trees. By designing and developing the code, I have a deeper understanding of B+ Trees and their operations, also exercised my coding skills especially in involving recursively updating after a single operation.