B+ Tree Implementation Report

Introduction

This report documents the implementation of a B+ Tree data structure in Python. The B+ Tree is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. It is widely used in database systems and file systems.

The implemented B+ Tree supports the following core functionalities:

1. **Insertion**: Adding new keys while maintaining the tree's balanced property.
2. **Search**: Efficiently locating keys within the tree.
3. **Deletion**: Removing keys and rebalancing the tree (handling underflows).
4. **Display**: Traversing the leaf nodes to display all keys in ascending order.

Design & Implementation

Class Structure

The implementation adopts an Object-Oriented Programming (OOP) approach with two main classes:

1. `Node` **Class**: Represents a node in the B+ Tree.
    - `keys`: A list of keys stored in the node.
    - `children`: A list of child nodes (for internal nodes).
    - `is_leaf`: A boolean flag indicating if the node is a leaf.
    - `next`: A pointer to the next leaf node (for efficient range queries and traversal).
    - `parent`: A pointer to the parent node (to facilitate bottom-up operations like splitting and merging).
2. `BPlusTree` **Class**: Manages the tree operations.

    - `degree` (m): The order of the B+ Tree. **Note**: In this implementation, the `degree` parameter represents the maximum number of children a node can have (Order M). Consequently, the maximum number of keys in a node is $M - 1$.
    - `root`: The root node of the tree.
    - `min_keys`: The minimum number of keys a node (other than root) must contain, calculated as $\lceil m/2 \rceil - 1$.

Key Algorithms

1. Search ($O(\log n)$)

The search operation starts at the root and traverses down to the leaf level. At each internal node, it finds the correct child pointer by comparing the search key with the keys in the node. Once the leaf node is reached, a linear search is performed within the leaf's keys.

2. Insertion ($O(\log n)$)

- **Leaf Insertion**: The algorithm first locates the appropriate leaf node. The key is inserted in sorted order.
- **Splitting**: If a node exceeds the maximum number of keys (degree - 1), it splits:
    - **Leaf Split**: The leaf is split into two. The middle key is copied up to the parent. The `next` pointers are updated to maintain the linked list.
    - **Internal Node Split**: The internal node is split. The middle key is pushed up (not copied) to the parent.
- **Root Split**: If the root splits, a new root is created, increasing the tree height.

3. Deletion ($O(\log n)$)

- **Leaf Deletion**: The key is removed from the leaf.
- **Underflow Handling**: If a node has fewer than `min_keys`, the tree rebalances:
    - **Borrow (Redistribution)**: If a sibling has spare keys, a key is borrowed. For leaf nodes, the parent key is updated. For internal nodes, the parent key is rotated down.
    - **Merge (Coalesce)**: If siblings cannot lend a key, the node merges with a sibling. The separating key from the parent is moved down (for internal

nodes) or deleted (for leaf nodes). This may trigger a recursive underflow check at the parent level.

4. Display

The display function finds the left-most leaf node and traverses the `next` pointers, printing all keys in the linked list. This confirms the sorted order of the B+ Tree.

Challenges & Solutions

- **Parent Pointers**: Maintaining correct `parent` pointers during splits and merges was critical for propagating changes upwards. *Solution*: Explicitly updating `parent` attributes whenever children are moved or new nodes are created.
- **Internal vs. Leaf Logic**: Internal nodes and leaf nodes handle keys differently (e.g., copying vs. pushing up keys during splits). *Solution*: The `_split_leaf` and `_split_internal` methods were separated to handle these specific cases cleanly.
- **Underflow Complexity**: Handling all cases for deletion (borrow left, borrow right, merge left, merge right) is complex. *Solution*: The logic was broken down into helper methods `_borrow_from_left`, `_borrow_from_right`, and `_merge` to improve readability and maintainability.

Code Usage

Prerequisites

- Python 3.x

How to Run

Save the implementation in a file named `b_plus_tree.py` and run it using the Python interpreter:

```
python b_plus_tree.py
```

Example Usage

```python
# Initialize the tree with degree 4
tree = BPlusTree(degree=4)

# Insert keys
tree.insert(10)
tree.insert(20)
tree.insert(5)
tree.insert(15)

# Display all keys
print("Tree content:")
tree.display()
# Output: 5,10,15,20

# Search for a key
found = tree.search(15)
print(f"Search 15: {found}")
# Output: Search 15: True

# Delete a key
tree.delete(10)

# Display after deletion
print("After deleting 10:")
tree.display()
# Output: 5,15,20
```

Output Explanation

1. **Insertion**: Keys 5, 10, 15, 20 are inserted. The tree automatically sorts them.
2. **Display**: The output `5,10,15,20` confirms the sorted structure.
3. **Search**: Searching for 15 returns `True`, confirming successful insertion and retrieval.
4. **Deletion**: After deleting 10, the display shows `5,15,20`, verifying that the key was removed and the tree structure (and linked list) remains intact.