Design of an Order-m B-Tree

1 B-Tree Design

The B-Tree is a self-balancing tree data structure that maintains sorted data and allows efficient insertion, deletion, and search operations. The design of the B-Tree is encapsulated within two primary classes: `BTree` and `BTreeNode`. Below is an overview of their structure and functionality. My code implementation can be found here.

1.1 BTree Class

The `BTree` class represents the B-Tree itself and manages the overall tree operations.

1.1.1 Attributes

- **m** (`int`): The order of the B-Tree, which determines the maximum number of children each node can have. Specifically, each node can contain at most $m - 1$ keys and m children.
- **root** (`BTreeNode`): A reference to the root node of the B-Tree. Initially, the root is a leaf node.

1.1.2 Methods

`search(k, x=None)` Searches for a key `k` within the subtree rooted at node `x`. If `x` is not provided, the search starts from the root node.

`insert(k)` Inserts a key `k` into the B-Tree. If the root node is full, the tree grows in height by splitting the root before proceeding with the insertion.

`delete(k)` Removes a key `k` from the B-Tree. After deletion, the tree ensures that all nodes satisfy the B-Tree properties, potentially merging or redistributing keys as necessary.

`split_child(x, i)` Splits the `i`-th child of node `x` when it becomes full. This operation ensures that the B-Tree properties are maintained by dividing the overflowing node into two and promoting a median key to the parent node.

`_insert_non_full(x, k)` A helper method used during the insertion process. It inserts a key `k` into a node `x` that is guaranteed not to be full.

`_delete(x, k)` A helper method for deleting a key `k` from the subtree rooted at node `x`. It handles various cases to maintain the B-Tree properties after deletion.

`_get_pred(x, idx)` Retrieves the predecessor of the key at index `idx` in node `x`. This is used during deletion to replace a deleted key with its predecessor.

`_get_succ(x, idx)` Retrieves the successor of the key at index `idx` in node `x`. This is used during deletion to replace a deleted key with its successor.

`_fill(x, idx)` Ensures that the child node at index `idx` of node `x` has the minimum required number of keys by borrowing from siblings or merging nodes if necessary.

`_borrow_from_prev(x, idx)` Borrows a key from the previous sibling of the child node at index `idx` of node `x` to maintain the B-Tree properties.

`_borrow_from_next(x, idx)` Borrows a key from the next sibling of the child node at index `idx` of node `x` to maintain the B-Tree properties.

`_merge(x, idx)` Merges the child node at index `idx` of node `x` with its adjacent sibling. This operation is performed when both siblings have the minimum number of keys.

1.2 BTreeNode Class

The `BTreeNode` class represents the individual nodes within the B-Tree. Each node maintains its keys and references to its children.

1.2.1 Attributes

- **m** (`int`): The order of the B-Tree, consistent with the `BTree` class.
- **keys** (`List[int]`): A list of keys stored in the node, maintained in sorted order.
- **children** (`List[BTreeNode]`): A list of child pointers. For a node with n keys, there are $n+1$ children.
- **leaf** (`bool`): A boolean flag indicating whether the node is a leaf node (i.e., it has no children).

1.2.2 Behavior

Nodes in the B-Tree adhere to the following properties to maintain balance and order:

- Each node contains a maximum of $m-1$ keys and a minimum of $\lceil m/2 \rceil - 1$ keys, except for the root node which can have fewer keys.
- All leaves appear at the same depth, ensuring the tree remains balanced.
- Keys within each node are maintained in sorted order, facilitating efficient search operations.
- Child pointers separate the key ranges, allowing the tree to direct search, insertion, and deletion operations efficiently.

1.3 Operational Overview

The `BTree` class provides high-level operations such as `search`, `insert`, and `delete`, which internally manage the complexity of maintaining the B-Tree properties. Key operations like `split_child`, `_borrow_from_prev`, `_borrow_from_next`, and `_merge` are crucial for ensuring that the tree remains balanced after insertions and deletions.

- **Search**: The `search` method navigates the tree from the root, traversing the appropriate child pointers based on key comparisons until the key is found or determined to be absent.
- **Insertion**: The `insert` method ensures that keys are added to the tree in a way that maintains order. If a node becomes full during insertion, it is split, and the median key is promoted to maintain the B-Tree properties.
- **Deletion**: The `delete` method removes a key and ensures that nodes do not underflow (i.e., have fewer keys than the minimum required). This may involve borrowing keys from siblings or merging nodes to maintain balance.

Overall, the design leverages the properties of B-Trees to provide efficient and balanced storage of keys, ensuring that operations remain performant even as the dataset grows.