A) General Node Structure of the B-Tree Implementation

The B-Tree implementation, referred to as **BeeTree**, uses a clean and efficient node structure based on the BeeTreeNode class. Each node includes key components that preserve the essential properties of a B-tree.

Core Components

- **Key Storage**:
  Each node stores a dynamic list of keys, always maintained in ascending (in-order) sequence. The number of keys ranges from 1 up to a maximum of 2t−1 , where t is the minimum degree of the tree. Non-root nodes must hold at least t -1 keys to ensure structural balance.

- **Child Pointers**:
  Internal nodes contain references to child nodes, with the number of children always equal to the number of keys plus one. This structure enables efficient navigation through the tree. Leaf nodes, by contrast, do not store any child pointers.

- **Node Type Flag**:
  A Boolean flag identifies whether a node is a leaf node (which contains actual data) or an internal node (used for navigation). This distinction influences the behavior of operations like insertion, deletion, and search.

Capacity Constraints

The minimum degree ttt controls the range of keys and children each node can hold. Root nodes may have 1 to 2t – 1 keys, while non-root nodes must maintain at least t-1 keys. When a node reaches its capacity, a split operation is triggered during insertion to maintain tree balance.

Key Properties

The structure enforces important invariants:

- Keys within a node are unique and strictly sorted.
- All keys in children[i] are less than keys[i], preserving ordering.
- These properties support efficient, logarithmic search operations.

Design Benefits

This implementation leverages dynamic memory allocation to reduce overhead while ensuring cache-friendly access patterns. The design supports insertions, deletions, and traversals with logarithmic performance, while maintaining simplicity and reliability.

B) Approach for Implementing the Commands: select, rank, keysInRange, and primesInRange

All four commands in the B-tree implementation rely on a **shared strategy**: a complete **in-order traversal** using the traversel_o_in_order() function. This consistent approach emphasizes simplicity, correctness, and maintainability over maximum performance.

Core Strategy: Traversal-Based Architecture

- **Shared Foundation**:
  Every command begins with an in-order traversal, which returns all keys in ascending order. This provides a unified and reliable basis for each operation.

---

1. select(k)

- **Purpose**: Returns the k-th smallest key (1-based indexing).
- **Approach**:
  After generating the sorted key list, the k-th element is retrieved using array indexing (k-1 to convert from 1-based to 0-based). If k is out of bounds, the function returns -1.
- **Implementation Notes**:
  Simple, efficient, and leverages O(1) array access once the traversal is completed.

---

2. rank(key)

- **Purpose**: Returns the 1-based position of a given key.
- **Approach**:
  Uses Python's built-in list.index() on the sorted list to find the position, then adds 1 to convert to 1-based indexing. A try-except block handles cases where the key is not found.
- **Implementation Notes**:
  Straightforward linear search with graceful error handling.

---

3. keysInRange(lower, upper)

- **Purpose**: Returns all keys within the inclusive range [lower,upper][lower, upper][lower,upper].

- **Approach**:
  After in-order traversal, a filtering operation selects keys that fall within the specified bounds.
- **Implementation Notes**:
  Simple range filtering with early exit optimization to improve performance slightly.

---

4. primesInRange(lower, upper)

- **Purpose**: Returns all prime keys within the range [lower,upper][lower, upper][lower,upper].
- **Approach**:
  Builds upon keysInRange by adding a second layer of filtering for primality. Each key is passed through the optimized is_prime() function.
- **Implementation Notes**:
  Double filtering (range + primality), supported by a highly optimized prime-checking function.

Primality Testing Optimization

The is_prime() function uses a multi-layer strategy:

1. **Basic preconditions**
2. **Small number lookup**
3. **Special case handling**
4. **Wheel factorization**

Design Philosophy

This B-tree implementation prioritizes **correctness**, **maintainability**, and **clarity**. The in-order traversal approach ensures:

- Consistent and correct results across all commands.
- Easier debugging and verification.
- Uniform handling of edge cases.

The trade-off is **O(n)** time complexity for operations that could theoretically be improved to **O(log n)** using advanced tree augmentation techniques — a deliberate decision to favor simplicity and correctness over micro-optimization.