B and B+ Trees

We've climbed through the branches of various trees, from simple Binary Trees to the self-balancing AVL and Red-Black Trees. These trees are fantastic for organizing data when it fits nicely in your computer's **main memory (RAM)**. Their O(logN) operations are super fast because accessing any part of RAM is quick.

But what happens when you have **gigabytes or terabytes of data**? Data that's so massive it can't all fit into RAM at once? This kind of data is typically stored on **disk drives (HDDs/SSDs)**. The problem is, accessing data from disk is *thousands to millions of times slower* than accessing RAM. When you read from disk, you don't just read one byte; you read a whole chunk (a "block" or "page") at a time.

This is where **B-Trees** and **B+ Trees** come into their own! They are specialized tree data structures designed to minimize disk access operations, making them perfect for databases and file systems.

🌳 B-Trees: Designed for Disk Efficiency

A **B-Tree** (often pronounced "B-tree," not "bee-tree") is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The "B" often stands for "balanced" or "Bayer" (after one of its inventors).

The key difference from binary trees is that B-tree nodes can have **many children** (more than 2). This makes them "fat" and "short," which is ideal for disk-based storage.

The Core Idea: Minimizing Disk I/O

When you read a node from disk, you're reading a whole block of data. A B-tree is designed so that **each node corresponds to a disk block**. By having many children (and thus many keys) in a single node, you reduce the height of the tree significantly. A shorter tree means fewer disk reads to find your data!

Imagine:

- A Binary Tree might have a height of 30 for a large dataset, requiring 30 disk reads.
- A B-Tree for the same dataset might have a height of only 3 or 4, requiring just 3 or 4 disk reads. This is a massive performance improvement!

Parameters: Order of the B-Tree

A B-tree is defined by its **order (m)**, which dictates the minimum and maximum number of children a node can have.

- **Order m:** A B-tree of order m has the following properties:
    - Each node (except the root and leaves) has between ceil(m/2) and m children.
    - Each node (except the root) has between ceil(m/2) - 1 and m - 1 keys.
    - The root has at least 2 children (unless it's a leaf node itself, i.e., the tree has only one node).
    - All leaves are at the same level.

📜 Properties of a B-Tree

1. **All leaves are at the same level (same depth).** This is crucial for guaranteed O(logN) worst-case performance.
2. **Each node has a specific number of keys and children.** For a B-tree of order m:
    - A node has k keys, where ceil(m/2) - 1 <= k <= m - 1.
    - A node has k + 1 children, where ceil(m/2) <= k + 1 <= m.
    - The keys within a node are sorted in ascending order.
    - The children pointers C1, C2, ..., Ck+1 point to subtrees. All keys in subtree Ci are between key Ki-1 and key Ki.
3. **The root node:**
    - If the tree is not empty, the root has at least 1 key.
    - If the root is not a leaf, it has at least 2 children.
4. **Black-Height Property (Implicit):** Similar to Red-Black trees, all paths from the root to a leaf have the same number of nodes.

Visual Example (B-Tree of order m=3, so 1 or 2 keys per node, 2 or 3 children):

```
          [ 50 ]
         /      \
      [ 20 40 ]   [ 70 90 ]
     /  |  \  /  |  \
  [10] [25 35] [45] [60] [80] [100]
```

In this example:

- Root [50] has 1 key, 2 children.
- Internal nodes [20 40] and [70 90] have 2 keys, 3 children.
- Leaf nodes [10], [25 35], etc., have 1 or 2 keys and no children.

- All leaves are at the same level.

🚀 Basic Operations in a B-Tree

Operations in a B-tree are designed to minimize disk access, often involving reading and writing entire nodes (disk blocks).

1. search(key): Finding a Key

- **Concept:** Start at the root. Within the current node, perform a binary search to find the key or the appropriate child pointer to follow. If the key is found in the current node, you're done. Otherwise, follow the child pointer to the next node (which might involve a disk read). Repeat until the key is found or a null child pointer is encountered (key not found).
- **Time Complexity:** O(log_mN) where m is the order of the tree. Since m can be very large (e.g., hundreds or thousands), this is effectively very few disk reads.

2. insert(key): Adding a New Key

Insertion in a B-tree is always done at a **leaf node**.

1. **Search to Leaf:** Search for the appropriate leaf node where the new key should be inserted.
2. **Insert Key:** If the leaf node has space (fewer than m-1 keys), simply insert the key into the sorted list of keys within that node.
3. **Handle Overflow (Splitting):** If the leaf node is full (already has m-1 keys), inserting a new key causes an **overflow**.
    - The node is split into two new nodes.
    - The middle key is **promoted** to the parent node.
    - The keys smaller than the promoted key go into the left new node, and keys larger go into the right new node.
    - This promotion might cause the parent node to overflow, leading to a cascading split up the tree, potentially even splitting the root (which increases the tree's height).

3. delete(key): Removing a Key

Deletion is the most complex operation in a B-tree, requiring careful handling to maintain the B-tree properties.

1. **Search for Key:** Find the node containing the key to be deleted.
2. **Case 1: Key is in a Leaf Node.**
    - Remove the key.
    - **Handle Underflow:** If the leaf node now has fewer than ceil(m/2) - 1 keys (underflow), it needs to be rebalanced.

        - **Redistribution (Borrowing):** Try to borrow a key from a sibling node that has more than the minimum number of keys. This involves moving a key from the sibling, promoting a key from the parent, and demoting a key from the parent.
        - **Merging:** If borrowing is not possible (siblings are also at minimum keys), merge the underflowed node with a sibling and pull down a key from the parent. This might cause the parent to underflow, leading to cascading merges up the tree.
3. **Case 2: Key is in an Internal Node.**
    - Find the key's **inorder predecessor** (largest key in the left child's subtree) or **inorder successor** (smallest key in the right child's subtree).
    - Replace the key to be deleted with the successor/predecessor.
    - Then, recursively delete the successor/predecessor from its original position (which will always be in a leaf node, reducing it to Case 1).
    - After deletion, check for underflow and perform redistribution or merging as needed, propagating up the tree.

➕ B+ Trees: An Evolution for Range Queries

B+ Trees are a variation of B-trees, widely used in database systems (like SQL databases) and file systems. They are optimized for **range queries** and **sequential access**.

Key Differences from B-Trees:

1. **All data is stored in Leaf Nodes:**
    - Internal nodes (non-leaf nodes) only store **keys** (or indices) to guide the search. They do *not* store actual data records.
    - All actual data records (or pointers to them) are stored exclusively in the **leaf nodes**.
2. **Leaf Nodes are Linked:**
    - All leaf nodes are linked together in a **singly linked list** from left to right. This allows for very efficient sequential scanning of data (e.g., "find all records between X and Y").
3. **Duplicate Keys in Internal Nodes:**
    - Keys in internal nodes can be duplicated in leaf nodes. An internal node's key acts as a separator or routing key.
    - The key in an internal node is essentially the smallest key in the right child's subtree.

Visual Example (B+ Tree of order m=3):

```
                [ 50 ]
              /        \
           [ 20 40 ]   [ 70 90 ]
          /  |  \   /   |  \
(Linked List) --> [10] --> [20 25 30] --> [40 45] --> [50 60] --> [70 80] --> [90 100]
--> (NULL)
```

In this example:

- Internal nodes [20 40] and [70 90] only contain keys.
- All data records (or pointers to them) are in the leaf nodes.
- Leaf nodes [10], [20 25 30], etc., are linked horizontally.

Basic Operations in a B+ Tree:

- **search(key):**
    - Traverse down the tree, following internal node keys, until you reach the appropriate **leaf node**.
    - Then, search for the key within that leaf node.
    - **Range Search:** Once you find the starting key in a leaf, you can easily traverse the linked list of leaf nodes to find all keys within a range.
- **insert(key):**
    - Always insert into a **leaf node**.
    - If a leaf node overflows, it splits, and the **middle key is copied (not promoted)** to the parent. The copied key remains in the right new leaf node.
    - Internal node splits are similar to B-trees (middle key promoted).
- **delete(key):**
    - Always delete from a **leaf node**.
    - If a leaf node underflows, it tries to borrow from a sibling. If not possible, merge with a sibling.
    - If a merge occurs, the key that was guiding to the merged nodes in the parent might need to be removed from the parent. This can cause cascading underflow in internal nodes.