B-Trees Explained in Detail

Overview:
B-trees optimize search tree structures for storage efficiency and locality, particularly beneficial for database and memory systems where cache or disk access is expensive. Unlike binary search trees (BSTs), which store a single key per node, B-trees store multiple keys, dramatically improving locality and reducing the number of reads required for operations.

Key Concepts:
- **Locality**: Exploiting memory and cache efficiency by grouping multiple keys and pointers.
- **Branching Factor (Order)**: B-trees have a high branching factor (m > 2), meaning each node can have up to m children, significantly reducing tree height and enhancing search efficiency.
- **Nodes**: There are two types of nodes:
  - **Non-leaf (Internal) nodes**: Contain keys and pointers to children but not actual data.
  - **Leaf nodes**: Contain the actual data records and keys.

B-tree Properties:
1. **Uniform Path Length**: Every path from root to leaf is the same length.
2. **Key-Child Relationship**: If a node has n children, it contains exactly n–1 keys.
3. **Minimum Node Fill**: All nodes (except possibly the root) are at least half full.
4. **Ordered Subtrees**: Elements in subtrees fall between parent keys on either side of the subtree pointer, generalizing the binary search tree invariant.
5. **Root Constraint**: The root node must have at least two children if it’s not a leaf.

Example from Provided Image (Order-5 B-tree):
- The first image illustrates a B-tree of order 5 (each internal node can have up to 5 children). Here, nodes have multiple keys:

Structure Breakdown:
- Root Node:
  - Contains key '20', with two pointers: one to the left subtree and one to the right subtree.

- Left Internal Node:
  - Keys: 11 and 15
  - Three child pointers:
    - Leftmost pointer points to leaf node [3, 5, 7].
    - Middle pointer points to leaf node [11, 12].

    - Right pointer points to leaf node [15, 16, 19].

- Right Internal Node:
  - Keys: 25, 30, 33, and 37
  - Five child pointers:
    - [20, 22, 23], [25], [30, 31, 32], [33, 35], [37].

Insertion Operations (Illustrated in the second image):

Step 1: Insert (13)
- Insert key '13' into the leaf node [11, 12], resulting in [11, 12, 13]. No splitting is required since there's room.

Step 2: Insert (14)
- Leaf node [11, 12, 13] becomes full; inserting '14' causes splitting:
  - Left node: [11, 12]
  - Right node: [13, 14]
- Update the parent node by adding '13' as a new separating key.

Step 3: Insert (24)
- The insertion of '24' into leaf node [20, 22, 23] causes overflow, splitting into:
  - Left node: [20, 22]
  - Right node: [23, 24]
- Parent node becomes full after adding key '23', causing further split at the internal node level, promoting '30' to the root.

After these operations, the tree height increases, maintaining B-tree invariants.

Deletion Operations (General description):
- Deletion is opposite insertion; removing keys from leaves and restructuring nodes to maintain minimum fill.
- If nodes become underfilled (below half), sibling nodes are adjusted or merged, potentially reducing tree height.

Benefits of B-trees:
- Reduced number of disk reads due to high branching factor.
- Improved cache utilization by matching node size to cache lines or disk blocks.

Practical Use:
- Databases and file systems frequently use B-trees due to their balanced nature and excellent performance in data retrieval and modification operations.

Further Reading:
- Aho, Hopcroft, and Ullman, *Data Structures and Algorithms*, Chapter 11 for in-

depth theoretical and practical understanding.