# B+ Trees

B-Trees and B+ Trees

A B-Tree is a self-balancing search tree designed to keep keys sorted and allow searches, sequential access, insertions, and deletions in logarithmic time. It is optimized for systems that read and write large blocks of data, like databases and file systems.

A B+ Tree is an extension of a B-Tree, where internal nodes contain only keys and leaf nodes store the actual records (or pointers to records). This structure improves range queries and sequential access.

B-Tree Properties

1. Each node can have at most m children (where m is the order of the tree).
2. A node with k children has k-1 keys.
3. All leaves are at the same depth.
4. The root has at least 2 children (unless it's a leaf).
5. Non-root nodes must have at least ⌈m/2⌉ children.
6. Keys within a node are sorted in ascending order.
7. Each key separates child subtrees, meaning:
    - Keys in the left child are less than the key.
    - Keys in the right child are greater than the key.

B+ Tree Properties

1. Internal nodes contain only keys, no data.
2. Leaf nodes store all data records, and they are linked in a doubly linked list for fast sequential access.
3. The root can be a leaf node if there are few elements.
4. Every internal node has at least ⌈m/2⌉ children (except the root).
5. Searches always end at leaf nodes, making range queries more efficient.
6. Insertions, deletions, and searches take O(log_m N) time.

B-Tree vs. B+ Tree

- B-Trees store both keys and values in internal and leaf nodes.
- B+ Trees store values only in leaf nodes, while internal nodes contain only keys for navigation.
- B+ Trees allow efficient range queries because of the linked list of leaves.
- B-Trees are better for random access, while B+ Trees are better for range scans.

B-Tree Insertion

1. Start at the root node.
2. Traverse the tree to the correct leaf.
3. Insert the key in sorted order.
4. If the node overflows (exceeds max keys), split it:
    - The middle key moves up to the parent.
    - The node splits into two, redistributing keys.
5. If the root splits, create a new root, increasing the tree height.

Example:

Insert 42 into a B-Tree of order 4 (m=4):
1. Find the correct position in the leaf node.
2. Insert 42.
3. If the node has too many keys, split and move the middle key up.

B-Tree Deletion

1. Find the key in the tree.
2. If the key is in a leaf node, remove it.
3. If the key is in an internal node, replace it with the predecessor (largest key in left subtree) or successor (smallest key in right subtree).
4. If the node has too few keys, borrow a key from a sibling or merge nodes.

Example:

Delete 42 from a B-Tree:
1. Locate 42.
2. If it's in a leaf, remove it.
3. If it's in an internal node, find the successor/predecessor to replace it.
4. If a node has too few keys, rebalance the tree.

B+ Tree Insertion

1. Start at the root node.
2. Traverse the tree to find the correct leaf node.
3. Insert the key in sorted order in the leaf.
4. If the leaf overflows, split the node and copy the smallest key of the new node to the parent (instead of moving it).
5. If an internal node overflows, split it and push the middle key to the parent.
6. If the root splits, create a new root, increasing the tree height.

Example:

Insert 35 into a B+ Tree (m=4):
1. Find the correct leaf.

2. Insert 35.
3. If the leaf overflows, split it.
4. Copy the smallest key of the new leaf to the parent.

B+ Tree Deletion

1. Find the key in the leaf node.
2. Remove it from the leaf.
3. If the leaf has too few keys, borrow a key from a sibling or merge with a sibling.
4. If an internal node has too few keys, adjust the parent pointers.
5. If the root has only one child left, it becomes the new root.

Example:

Delete 35 from a B+ Tree:
1. Locate 35 in the leaf.
2. Remove it.
3. If the node has too few keys, borrow from a sibling or merge.

B+ Tree Advantages Over B-Trees

1. Faster range queries – Leaf nodes are linked for sequential access.
2. Better disk access – Internal nodes are smaller, improving cache performance.
3. Consistent access time – All searches reach the leaf level.
4. Efficient insertions and deletions – Rebalancing is straightforward.

B-Trees vs. B+ Trees Summary

- B-Trees:
    - Keys and values stored in internal and leaf nodes.
    - Random access is fast.
    - Not ideal for range queries.
- B+ Trees:
    - Keys in internal nodes, data in leaf nodes.
    - Linked leaf nodes make range queries faster.
    - More space-efficient internal nodes.

Example Prompt

- "Insert 42 into the following B+ Tree: ..."
  Expected Response: Correctly modified tree with a split if necessary.
- "What is the inorder traversal of this B-Tree?"
  Expected Response: Sorted list of keys.
- "Delete 35 from the given B+ Tree."
  Expected Response: Correctly modified tree with adjustments to parent pointers.