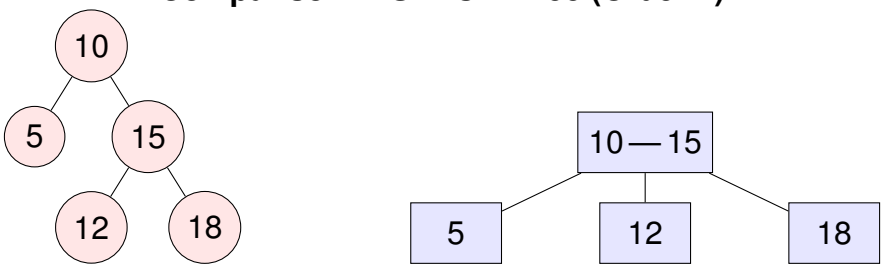
# B-Trees

**Motivating Example**
Suppose you are designing a database system that must efficiently store and retrieve millions of records. Using a simple binary search tree (BST) would result in a very tall tree, leading to slow disk access. A B-Tree, with its multi-way branching, keeps the tree shallow and minimizes the number of disk reads required for any operation.

**Comparison: BST vs B-Tree (Order 4)**




*Left: BST (tall, many levels).* *Right: B-Tree (shallow, fewer levels).*

## A Introduction to B-Trees

B-Trees are balanced search trees designed for systems that read and write large blocks of data, such as databases and filesystems. Unlike binary search trees (BSTs), B-Trees can have more than two children per node, resulting in a much shallower tree. This shallowness is crucial for minimizing disk reads, as each node can store many keys and children, matching the size of a disk block.

**How B-Trees Differ from Other Trees:**

- **Multi-way branching:** Nodes can have many children (not just two).
- **Shallower height:** Fewer levels than BSTs for the same number of keys.
- **Disk optimization:** Nodes are sized to fit disk blocks, reducing the number of disk accesses.
- **Balanced:** All leaves are at the same depth, ensuring $O(\log n)$ operations.

**B-Tree Node Example**

8 — 15 — 23

*A B-Tree node can store multiple keys, unlike a BST node.*

## B Keys in B-Trees

A **key** in a B-Tree is a value used to organize and search for data. Each node contains multiple keys, which are always kept in sorted order. The keys divide the range of values among the node's children.

**How keys are assigned/created:**

- When inserting, the key is placed in the appropriate position in a leaf node.
- If the node overflows (too many keys), it is split: the middle key is promoted to the parent, and the remaining keys are split between two new nodes.
- This process may propagate up to the root, increasing the tree's height.

**Example:**

- Insert 8, 15, 23 into an empty B-Tree of order 4:

  - After 8: [8]
  - After 15: [8 — 15]
  - After 23: [8 — 15 — 23]
- Insert 42: Node overflows, split into two nodes and promote 15:
  - Root: [15]
  - Children: [8] [23 — 42]

Disk Optimization in B-Trees
B-Trees are designed to minimize disk reads and writes. Each node is sized to match the disk block size, so reading or writing a node requires only one disk access. By storing many keys per node, B-Trees keep the tree shallow, reducing the number of disk accesses needed for search, insertion, or deletion. This makes B-Trees ideal for databases and filesystems, where disk I/O is the main performance bottleneck.

## C Typical Implementation Strategies

- **Node Structure:** Each node contains multiple keys and child pointers. Keys are kept sorted within each node.
- **Insertion:** Insert the key in the appropriate leaf. If the node overflows, split it and promote the middle key to the parent. Splitting may propagate up to the root.
- **Deletion:** Remove the key. If a node underflows, borrow a key from a sibling or merge nodes, possibly propagating changes up the tree.
- **Search:** Traverse from the root, choosing the child whose interval contains the key, until the key is found or a leaf is reached.
- **Disk Optimization:** B-Trees are designed to minimize disk reads by maximizing the number of keys per node (matching disk block size).

## D Properties of B-Trees

A B-Tree of order $m$ has the following properties:

- Every node has at most $m$ children.
- Every internal node (except root) has at least $\lceil m/2 \rceil$ children.
- The root has at least two children if it is not a leaf.
- All leaves appear at the same level.
- A non-leaf node with $k$ children contains $k - 1$ keys.

**Practice Exercise:**

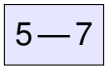
Suppose you have a B-Tree of order 4 (each node can have up to 3 keys). Insert the keys 8, 15, 23, 4, 42, 16, 7 in order. Draw the tree after each insertion and explain any node splits or promotions that occur.

## E Step-by-Step Example: Building a B-Tree of Order 3

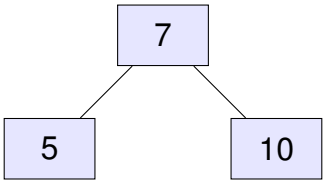
Below is a step-by-step visualization of building a B-Tree of order 3 (each node can have up to 2 keys and 3 children) by inserting the keys 5, 7, 10, 12, 15, 20, 22, 30. Each step shows the tree after an insertion and explains any node splits.

**Step 1: Insert 5**

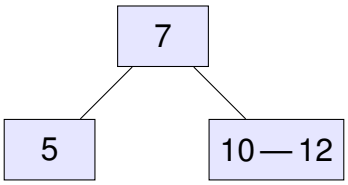

**Step 2: Insert 7**



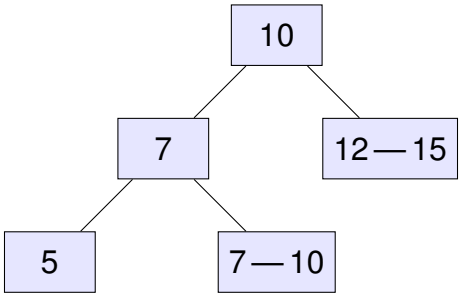

**Step 3: Insert 10 (Node full, split required)**



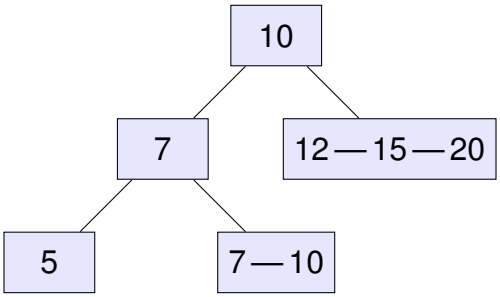

**Step 4: Insert 12**



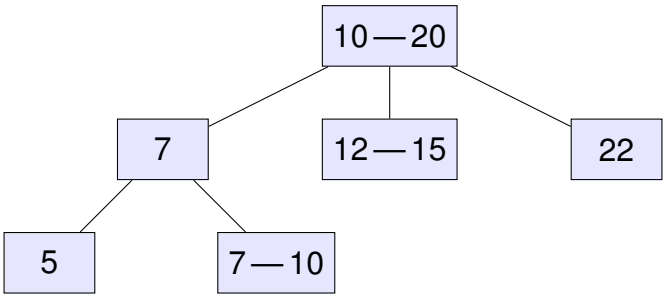

**Step 5: Insert 15 (Node full, split required)**



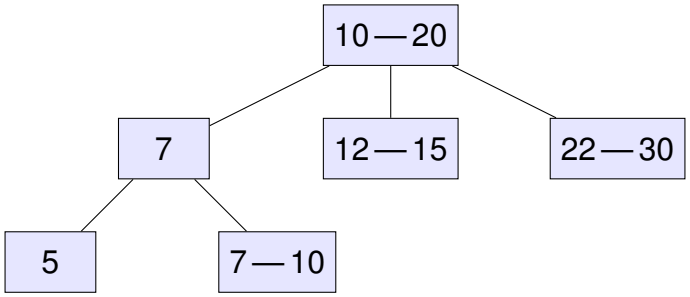

**Step 6: Insert 20**



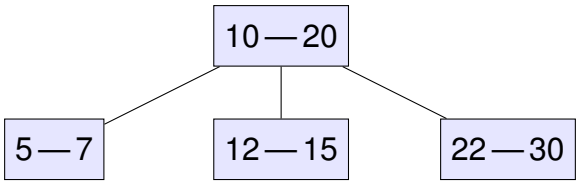

**Step 7: Insert 22 (Node full, split required)**



**Step 8: Insert 30**




**Final Step: After all insertions and splits**




**Practice Exercise:**

Given the B-Tree of order 3 built from the keys 5, 7, 10, 12, 15, 20, 22, 30, describe what happens if you delete the key 10. Show the resulting tree and explain any merges or redistributions.

## F Use Cases

- Database indexing (e.g., MySQL, PostgreSQL)
- Filesystems (e.g., NTFS, HFS+)
- Any application requiring efficient, balanced, disk-friendly search trees

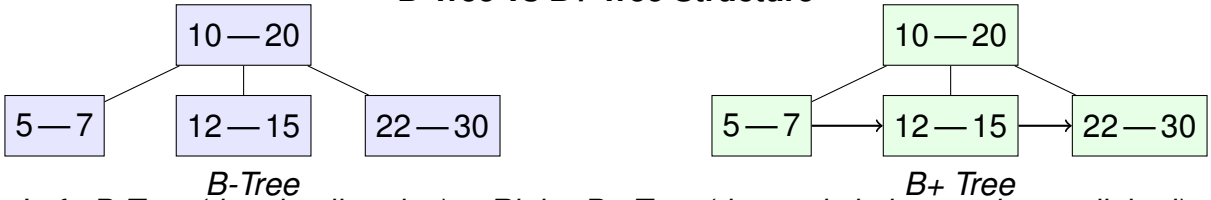
## G B+ Trees

B+ Trees are a variant of B-Trees commonly used in database and file system indexing. Like B-Trees, B+ Trees are balanced, multi-way search trees, but they have important differences that make them especially efficient for range queries and disk-based storage.

Key Differences: B-Tree vs B+ Tree

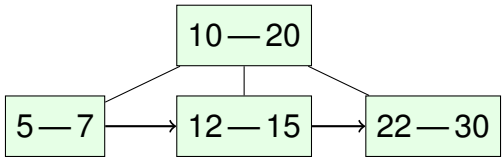
- **Data Storage:** In a B-Tree, both internal and leaf nodes store keys and associated data. In a B+ Tree, only the leaf nodes store the actual data (or pointers to data); internal nodes store only keys for navigation.
- **Leaf Node Linking:** In a B+ Tree, all leaf nodes are linked together in a linked list, enabling fast range queries and ordered traversal.
- **Search Path:** In a B+ Tree, all searches for data must reach the leaf level, while in a B-Tree, data may be found in internal nodes.
- **Efficiency:** B+ Trees are more efficient for range queries and full scans, as all data is stored in the leaves and can be traversed sequentially.

**B-Tree vs B+ Tree Structure**




*Left: B-Tree (data in all nodes). Right: B+ Tree (data only in leaves, leaves linked).*

B+ Tree Example
Consider inserting the keys 5, 7, 10, 12, 15, 20, 22, 30 into a B+ Tree of order 3 (max 2 keys per node):




*All data is stored in the leaves, and leaves are linked for fast range queries.*

Summary Table: B-Tree vs B+ Tree

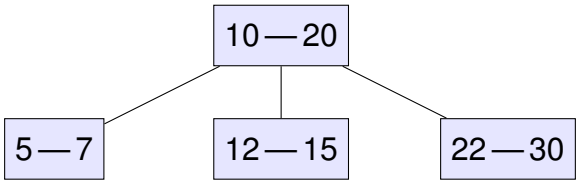
| | **B-Tree** | **B+ Tree** |
|---|---|---|
| Data in internal nodes | Yes | No |
| Data in leaves | Yes | Yes |
| Leaf nodes linked | No | Yes |
| Range queries | Moderate | Fast |
| All data at leaves | No | Yes |

## H How Balance is Maintained in B-Trees

A key property of B-Trees is that they remain balanced at all times, ensuring that all leaf nodes are at the same depth. This balance is maintained through the following mechanisms:

- **Node Splitting (on Insertion):** When a node becomes full (i.e., it contains the maximum number of keys), it is split into two nodes. The middle key is promoted to the parent node. If the parent is also full, this process may propagate up to the root, potentially increasing the tree's height by one. This ensures that no node ever exceeds the allowed number of keys, and the tree remains balanced.
- **Node Merging and Redistribution (on Deletion):** When a key is deleted and a node falls below the minimum number of keys, the tree restores balance by either borrowing a key from a sibling (redistribution) or merging with a sibling. If merging or redistribution is needed at the root and the root becomes empty, the height of the tree decreases by one. This process ensures that all nodes (except the root) maintain at least the minimum number of keys, and all leaves remain at the same level.
- **Consistent Leaf Depth:** All insertions and deletions are performed in a way that preserves the property that all leaves are at the same depth, so the tree never becomes skewed.

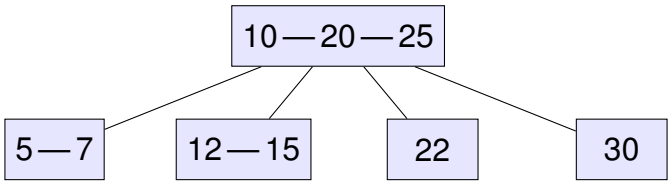
**Detailed Example: Maintaining Balance During Insertion and Deletion**
**Insertion Example:** Suppose we have a B-Tree of order 3 (each node can have at most 2 keys). We start with the following tree:




Now, insert the key 25:

- 25 belongs in the rightmost leaf node (22 — 30).
- After insertion: 22 — 25 — 30 (node overflows, as it now has 3 keys).

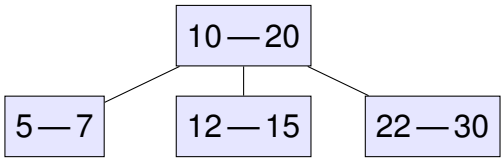
- Split the node: 25 is promoted to the parent, and the node splits into two: 22 and 30.



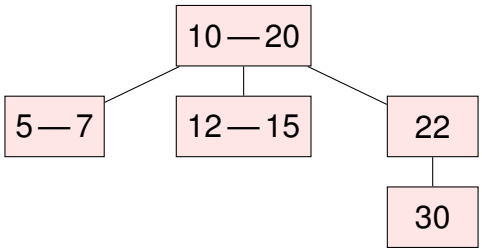

*The parent node now has 3 keys (10 — 20 — 25), which is the maximum allowed. If another key is inserted and causes the parent to overflow, it will also split, and the process may propagate up to the root. Throughout, all leaves remain at the same depth.*

Balanced vs. Unbalanced Tree: Visual Comparison
A B-Tree is always balanced by design, but to illustrate the importance, compare a balanced B-Tree with an unbalanced (invalid) tree:

**Balanced B-Tree**

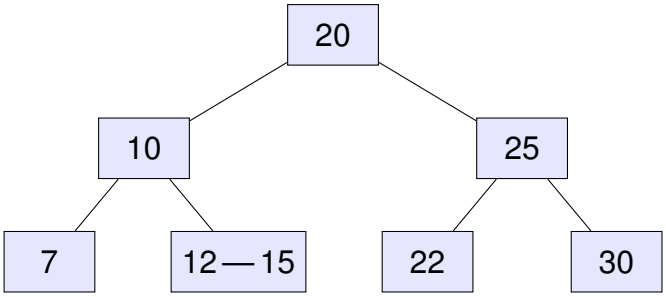



**Unbalanced Tree (Not a B-Tree)**

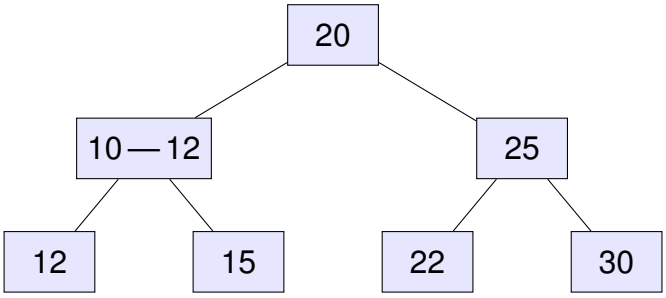



**Deletion Example:** Now, delete the key 5 from the leftmost leaf:

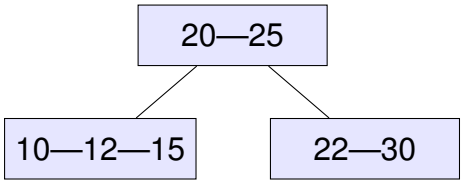
- After deletion, the leftmost leaf has only one key (7), which is the minimum allowed for a leaf in a B-Tree of order 3.




- Now, delete 7 as well. The node becomes empty (underflows). To fix this, we can borrow a key from the sibling (12 — 15) or merge with it.
- If we borrow, we take 12 from the sibling and move it to the left node, resulting in:



- If we merge instead, we combine the parent key (10) with the sibling (12 — 15) and the underflowed node, resulting in a single node (10 — 12 — 15) as the only child of the root (20). If the root is left with only one child after the merge, the merged node becomes the new root and the tree height decreases by one:




*The tree remains balanced: all leaves are still at the same depth. If the root ever has only one child after a merge, the root is removed and the child becomes the new root, reducing the tree's height by one.*

**Summary:** These balancing operations (splitting on insertion, merging/redistribution on deletion) guarantee that the B-Tree remains height-balanced, ensuring $O(\log n)$ search, insertion, and deletion.