B+ trees are balanced, multi-level index structures commonly used in databases and file systems. They store keys in internal nodes and data exclusively in leaf nodes, which facilitates efficient range queries and minimizes disk accesses by maintaining a high branching factor. Unlike binary search trees or AVL trees that rely on binary comparisons, B+ trees maintain balance through node splitting and merging rather than rotations.

AVL trees are self-balancing binary search trees that strictly enforce a maximum height difference of one between the left and right subtrees at every node, ensuring O(log n) search performance. They achieve balance through rotations—single or double—after insertions or deletions. This results in more consistent performance compared to standard binary search trees, which can become skewed, though the extra rotations may introduce additional overhead.

Binary search trees (BSTs) store elements so that each node's left child is less than the node and the right child is greater, supporting efficient search, insertion, and deletion on average. However, without self-balancing mechanisms, BSTs can degenerate into skewed trees resembling linked lists, leading to worst-case linear time operations. Unlike AVL or B+ trees, basic BSTs do not perform rotations or splits to maintain balance, which can result in performance degradation for certain insertion orders.

Hash maps or hash tables use hash functions to map keys to specific indices in an array, enabling near constant-time operations for insertions, deletions, and lookups. They do not preserve order and instead rely on collision resolution techniques such as chaining or open addressing. This design prioritizes speed over the ordered traversal capabilities inherent in tree-based structures like AVL or B+ trees.

For balancing examples in AVL trees, consider the following scenarios: for a left-left imbalance, create an AVL tree with nodes [30, 20, 40] and add node 10; the extra node in the left subtree of 30 makes it too heavy, requiring a right rotation. For a left-right imbalance, create an AVL tree with nodes [30, 20, 40] and add node 25; here, node 20's right child causes the left subtree to become heavy, which is corrected by a left rotation at node 20 followed by a right rotation at node 30. For a right-right imbalance, create an AVL tree with nodes [30, 20, 40] and add node 50; the increased depth in the right subtree necessitates a left rotation at node 30. For a right-left imbalance, create an AVL tree with nodes [30, 20, 40] and add node 35; the left-heavy condition in node 40 is fixed by a right rotation at node 40 followed by a left rotation at node 30. In contrast, for B+ trees balancing is achieved by splitting rather than rotations: for instance, create a B+ tree with leaf nodes holding keys [10, 20] and [30, 40] (assuming a maximum of three keys per node), and add key 25 to the second node; this insertion causes the node to exceed its capacity and split, propagating a new key upward. Standard binary search trees, on the other hand, do not rebalance automatically when a node is added, resulting in an imbalanced tree if the insertion creates a skewed structure, and hash maps do not involve any balancing process since they depend on hash functions and collision resolution.