Task 1: Lesson Review

Write a one-page reflection summarising what you have learned in this module.

This module provides a comprehensive review of binary search trees (BSTs) and their balanced variants. Trees are a special type of graph that are essential for implementing efficient insertion, deletion, and search (IDS) operations, ideally in O(logn) time complexity. BSTs embody this goal, but when unbalanced (e.g. when inserting nodes in ascending order), their time complexity drops to linear time (O(n)).

To alleviate this problem, self-balancing trees such as AVL trees and red-black trees (RBs) were developed. AVL trees have the property of maintaining height balance, ensuring that the height difference between the left and right subtrees of each node remains within ±1. When an insertion or deletion operation disrupts this balance, AVL trees employ tree rotations (single or double) to restore balance. While AVL trees are more strict in enforcing balance, they are faster to search due to their tighter height bounds.

On the other hand, red-black trees provide a less strict but more efficient balancing mechanism that associates each node with a color (red or black) and maintains five key properties (e.g., the root node is black, red nodes have black children, and all paths have the same number of black nodes). This more relaxed balancing mechanism simplifies insertion and deletion operations but slightly increases the height of the tree. Similar to AVL trees, red-black trees also use rotation operations, but these operations are selectively applied through internal rules.

In summary, this module highlights the trade-offs between search efficiency and update cost in different tree structures. Knowing when to use a basic binary search tree (BST) instead of a self-balancing variant such as an AVL tree or a red-black tree is critical to system performance.

Task 8: Compare Internal Node Structure

After constructing both a B Tree and a B+ Tree using the same keys (above):

- Count and compare the number of internal nodes.
- Which tree structure results in fewer internal nodes? Why?

How does this difference affect space usage and lookup performance in large-scale databases?

Comparison of the number of internal nodes: B-tree vs. B+ tree

B-tree and B+ tree are built using the same key set. B+ tree usually has fewer internal nodes than B-tree.

B-tree stores actual data directly in all nodes, while B+ tree stores data only in leaf nodes, and internal nodes only store key values for navigation. This structure makes the internal nodes of B+ tree more "lightweight", so that more keys can be stored.

Since the internal nodes of B+ tree do not need to be accompanied by data pointers, each node can accommodate more keys. This means that the same amount of data can be branched with fewer internal nodes in B+ tree, further reducing the height of the tree.

The structure of B+ tree is usually shorter than that of B tree, which not only reduces the traversal path, but also directly leads to fewer internal nodes required. Since the internal nodes of B-tree are responsible for data storage, their branching capacity is limited and more levels are required to cover the same data range.

Space efficiency: B+ tree has smaller internal nodes and takes up less space, which is suitable for database systems with large index scales and helps reduce overall storage costs.

Disk I/O performance: Smaller internal nodes mean that more nodes can be loaded into memory, reducing frequent access to the disk, thereby improving overall query performance.

In terms of range queries, the advantages of B+ tree are more obvious. Because its leaf nodes are connected by linked lists, data can be traversed sequentially without backtracking or re-searching the tree structure.

Task 9: Range Query in B+Tree

Using your B+ Tree from Task 6, design a range query function that returns all keys between two values, e.g., range_query("cop", "max").

- Write a pseudocode for your algorithm.
- Explain how leaf-level pointers in a B+ Tree help support this functionality.

First, since the linked list structure is formed between leaf nodes, the system can sequentially traverse all leaf nodes by accessing the key value in sequence without backtracking to the parent node or retraversing the entire tree. Once the starting point of the range query is found, the subsequent leaf nodes can be sequentially accessed through linear scanning until the query ends. The time complexity of this operation is closer to O(n).

In addition, the physical layout of leaf nodes on the disk is usually continuous, which can significantly reduce disk I/O operations, make full use of the sequential reading characteristics of the disk, and avoid costly random access.

From an implementation perspective, the linked list structure of leaf nodes also simplifies the query logic. There is no need to design a complex tree traversal algorithm, and a simple linear traversal is required to complete the range query. This not only improves code efficiency, but also reduces the possibility of system errors.

More importantly, this linked list structure ensures that all key values are not missed and can fully cover the specified query range. When a key value greater than the query end point is encountered during the scan, the system can terminate the query in advance, further improving efficiency.

Therefore, the B+ tree achieves efficient, accurate and easy-to-implement range query capabilities through the ordered pointer chain of leaf nodes.