B+ Tree Notes

B+ Trees are specialized self-balancing tree data structures optimized for efficient storage and retrieval operations, particularly in database systems and file systems. Based on the slideshow, here are detailed notes with additional information:

Core B+ Tree Concepts

Structure

- A B+ tree is a balanced tree structure with all leaf nodes at the same level
- Internal nodes contain only keys and pointers, while leaf nodes store both keys and data values
- B+ Trees use a parameter 'm' (order) to determine the maximum number of children per node
- For an order m B+ tree:
    - Each node can have at most m children
    - Each node (except root) has at least $\lceil m/2 \rceil$ children
    - The root has at least 2 children unless it's a leaf

Key Features Not Explicitly Stated

- **Leaf-level linking**: All leaf nodes are linked together in a linked list, allowing for efficient range queries
- **Data storage**: Unlike B-Trees, all data is stored exclusively at the leaf level in B+ Trees
- **More efficient disk I/O**: Because of the higher branching factor, fewer disk reads are typically required to find data

Insertion Process

Initial State

1. The tree begins with a single node (both leaf and root)
2. Keys (like 42, 21, 63, 89) represent the keys of key-value pairs
3. The leaf node stores both keys and associated data (although data isn't shown in examples)

Node Splitting

When inserting a value causes a node to exceed its capacity (m keys):

1. For **leaf nodes**:

    - A new leaf node is created to the right of the full node
    - The keys are split: ⌈m/2⌉ keys stay in the original node, the rest move to the new node
    - The smallest key in the new leaf node is **copied** up to the parent
    - The new leaf gets inserted into the doubly-linked list of leaf nodes
2. For **internal nodes**:

    - A new internal node is created
    - The middle element is **moved** (not copied) to the parent
    - If the parent doesn't exist (as when splitting the root), a new root node is created

Specific Insertion Example

The slides walk through several insertions (42, 21, 63, 89, 35, 10, 27, 96, 30, 37) showing how:

- The tree grows in levels when necessary
- Keys are redistributed during splits
- Pointers are rearranged to maintain the B+ tree properties

Advanced Properties (Not Explicitly Covered)

Performance Characteristics

- **Search Time**: O($\log_2 N$) in the worst case, as the height of the tree is logarithmic
- **Insert/Delete Time**: O($\log_2 N$) as we need to find the position first
- **Memory Usage**: High space utilization (typically 50-100%) because of the minimum occupancy requirement

Deletion (Not Covered in Slides)

Deletion in B+ trees is more complex than insertion and requires:

1. Locating the key to delete (only at leaf level)
2. Removing the key and potentially redistributing or merging nodes if a node becomes underfilled
3. If necessary, propagating changes upward through the tree

Real-world Applications

- Database indexing (MySQL's InnoDB, PostgreSQL, etc.)
- File systems (NTFS, HFS+, etc.)
- Systems that require efficient range queries and sequential access

Variants

- B+ Trees can be optimized for different storage media
- Variants include prefix B+ trees and bulk-loaded B+ trees

Key Differences from Other Tree Structures

B+ Tree vs B-Tree

- In B-Trees, data can be stored in internal nodes
- B+ Trees store data only in leaf nodes
- B+ Trees have interconnected leaf nodes for sequential access

B+ Tree vs Red-Black Tree

- B+ Trees are optimized for disk-based storage with higher branching factors
- Red-Black trees are binary trees optimized for in-memory operations