Understood! Your goal is to create a detailed textual representation of the concepts and visualizations from your class slide decks so that your RAG can provide comprehensive and detailed responses to queries. I will help by describing the content of the slides in as much detail as possible, ensuring that the RAG can effectively explain the concepts without relying on images or visualizations.

Let's start by summarizing and elaborating on the content from the screenshots you've provided:

1. **Searching in Databases**
- **Linear Search**: This is the most basic search method where you start at the beginning of a list and check each element one by one until you find the target or reach the end of the list. In the worst case, this method has a time complexity of O(n), meaning the time it takes to search grows linearly with the number of elements.
- **SQL SELECT Statement**: In SQL, the SELECT statement is used to query data from a database. It is versatile and can be complex, depending on the conditions and joins used.

2. **Records and Collections**
- **Record**: A record is a collection of values for attributes of a single entity instance, often represented as a row in a table.
- **Collection**: A set of records of the same entity type, typically stored in a sequential order like a list.
- **Search Key**: A value for an attribute used to search for records. It can be based on one or more attributes.

3. **Lists of Records**
- **Memory Allocation**: If each record takes up \(x \) bytes of memory, then \(n \) records will require \(n \times x \) bytes.
- **Contiguously Allocated List**: All records are stored in a single, continuous block of memory. This is similar to an array.
- **Linked List**: Each record is stored with additional space for memory addresses that link to the next (and possibly previous) record. This allows for dynamic memory allocation but requires extra space for the links.

4. **Contiguous vs Linked Allocation**
- **Arrays (Contiguous Allocation)**: Faster for random access but slower for insertions, especially in the middle of the list, because elements may need to be shifted.
- **Linked Lists**: Faster for insertions anywhere in the list but slower for random access because you must traverse the list from the beginning.

5. **Binary Search**
- **Algorithm**: Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half.

- **Time Complexity**: The worst-case time complexity is O(log n), making it much faster than linear search for large datasets.

6. **Database Searching Challenges**
- **Disk Storage**: Data is often stored on disk sorted by one column (e.g., id). Searching by another column (e.g., specialVal) requires a linear scan, which is inefficient.
- **Duplication**: Storing data sorted by multiple columns would require duplicating data, which is space-inefficient.

7. **Data Structures for Fast Insert and Search**
- **Sorted Array with Binary Search**: Allows for fast searching but slow insertions because inserting into a sorted array requires shifting elements.
- **Linked List**: Allows for fast insertions but slow searching because it requires a linear scan.
- **Binary Search Tree (BST)**: A tree data structure where each node has at most two children. For any given node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater. This allows for both fast insertions and searches, with average time complexities of O(log n).

Detailed Example: Binary Search Tree (BST)
- **Structure**: A BST is a binary tree where each node has a value, and the left subtree contains only nodes with values less than the parent node, while the right subtree contains only nodes with values greater than the parent node.
- **Search Operation**: To search for a value, start at the root. If the value is less than the current node, move to the left child; if greater, move to the right child. Repeat until the value is found or a leaf node is reached.
- **Insert Operation**: To insert a value, follow the same path as a search. When a leaf node is reached, add the new value as a child of that node.

By providing these detailed descriptions, your RAG will be able to explain the concepts thoroughly, even without visual aids. If you have more PDFs or specific slides you want to elaborate on, feel free to share them!

Absolutely, I'll provide a deeper dive into the concepts and algorithms related to Binary Search Trees (BSTs) to ensure your RAG can explain these topics in great detail. Let's break down each section and expand on the content.

Binary Search Trees (BSTs)

BST Property
- A BST is a binary tree where each node has at most two children referred to as the left child and the right child.
- The **BST property** ensures that for any given node:
  - All nodes in the left subtree have keys **less than** the node's key.
  - All nodes in the right subtree have keys **greater than** the node's key.
- This property makes BSTs efficient for search, insertion, and deletion operations.

Node Structure
- Each node in a BST typically contains:
  - **Key**: The value stored in the node.
  - **Left**: A pointer to the left child.
  - **Right**: A pointer to the right child.
  - **Parent**: A pointer to the parent node (optional, but useful for certain operations).

Traversal Strategies

Inorder Traversal
- **Order**: Left subtree → Current node → Right subtree.
- **Result**: Visits nodes in **non-decreasing order** of their keys.
- **Use Case**: Ideal for retrieving sorted data from a BST.
- **Pseudocode**:
  ```python
  def inorder_walk(x):
      if x is not None:
          inorder_walk(x.left)
          print(x.key)
          inorder_walk(x.right)
  ```

Preorder Traversal
- **Order**: Current node → Left subtree → Right subtree.
- **Result**: Visits the root first, then the left subtree, and finally the right subtree.
- **Use Case**: Useful for creating a copy of the tree or prefix expression of an expression tree.
- **Pseudocode**:
  ```python
  def preorder_walk(x):
      if x is not None:
          print(x.key)
          preorder_walk(x.left)
          preorder_walk(x.right)
  ```

Postorder Traversal
- **Order**: Left subtree → Right subtree → Current node.
- **Result**: Visits the root last.
- **Use Case**: Useful for deleting the tree or postfix expression of an expression tree.
- **Pseudocode**:
  ```python
  def postorder_walk(x):
      if x is not None:
          postorder_walk(x.left)
          postorder_walk(x.right)
          print(x.key)
  ```

Operations on BSTs

Searching for a Key
- **Algorithm**:
  - Start at the root.
  - Compare the target key with the current node's key.
    - If equal, the search is successful.
    - If the target is less, move to the left child.
    - If the target is greater, move to the right child.
  - Repeat until the key is found or a `nil` node is reached.
- **Time Complexity**: O(h), where h is the height of the tree. In the worst case (unbalanced tree), h = O(n). In the best case (balanced tree), h = O(log n).

Finding Minimum and Maximum
- **Minimum**: The leftmost node in the tree.
- **Maximum**: The rightmost node in the tree.
- **Algorithm**:
  - **Minimum**:
    ```python
    def bst_minimum(x):
        while x.left is not None:
            x = x.left
        return x.key
    ```
  - **Maximum**:
    ```python
    def bst_maximum(x):
        while x.right is not None:
            x = x.right
        return x.key
    ```

Insertion
- **Algorithm**:
  - Start at the root and traverse the tree to find the appropriate position for the new key.
  - Insert the new node as a leaf.
- **Pseudocode**:
  ```python
  def bst_insert(x, z, k):
      if x is None:
          return "Error"
      y = x
      while True:
          if y.key < k:
              z = y.left
          else:
              z = y.right
          if z is None:
              break
      if y.key > k:
          y.left = z
      else:
          y.right = z
  ```

Successor and Predecessor
- **Successor**: The smallest key greater than the given key.
- **Predecessor**: The largest key less than the given key.
- **Algorithm for Successor**:
  - If the node has a right child, the successor is the minimum of the right subtree.
  - Otherwise, traverse up the tree until a node that is the left child of its parent is found.
- **Pseudocode**:
  ```python
  def bst_successor(x):
      if x.right is not None:
          y = x.right
          while y.left is not None:
              y = y.left
          return y
      else:
          y = x
          while y.parent is not None and y == y.parent.right:
              y = y.parent
          return y.parent
  ```

Deletion
- **Cases**:
  1. **No Children**: Simply remove the node.
  2. **One Child**: Replace the node with its child.
  3. **Two Children**: Replace the node with its successor and then delete the successor.
- **Algorithm**:
  ```python
  def bst_delete(T, z):
      if z.left is None or z.right is None:
          y = z
      else:
          y = bst_successor(z)
      if y.left is not None:
          x = y.left
      else:
          x = y.right
      if x is not None:
          x.parent = y.parent
      if y.parent is None:
          T.root = x
      elif y == y.parent.left:
          y.parent.left = x
      else:
          y.parent.right = x
      if y != z:
          z.key = y.key
      return y
  ```

Example Traversals
- **Inorder**: 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20.
- **Preorder**: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.
- **Postorder**: 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

By providing these detailed explanations and pseudocode, your RAG will be able to explain BST operations and traversal strategies comprehensively. If you need further elaboration or additional topics, feel free to ask!

Balancing Binary Search Trees (BSTs)

Why Balancing Matters
- **Performance Dependency**: The efficiency of BST operations (search, insert, delete) heavily depends on the tree's height. A balanced tree ensures that the height is logarithmic relative to the number of nodes, leading to optimal performance.
- **Degenerate Trees**: If a BST becomes unbalanced (e.g., degenerates into a linked list), operations degrade to O(n) time complexity, which is inefficient for large datasets.
- **Real-World Scenarios**: Sequential insertion of sorted keys (a common scenario) can lead to degenerate trees, making balancing crucial.

Perfect vs. Complete Binary Trees
- **Perfect Binary Trees**: Every level is fully filled, and the height is $\Theta(\log n)$. However, perfect trees are only possible for specific numbers of nodes (e.g., 1, 3, 7, 15).
- **Complete Binary Trees**: A more relaxed form where all levels are fully filled except possibly the last, which is filled from left to right. The height is still $\Theta(\log n)$, making it a practical goal for maintaining balance.

AVL Trees: A Balanced BST
- **AVL Property**: For every node, the heights of the left and right subtrees differ by at most 1. This ensures the tree remains balanced.
- **Rotations**: AVL trees use rotations to maintain balance after insertions and deletions. The types of rotations include:
  - **LL Rotation**: Corrects a left-heavy tree.
  - **RR Rotation**: Corrects a right-heavy tree.
  - **LR Rotation**: Corrects a left-right imbalance.
  - **RL Rotation**: Corrects a right-left imbalance.
- **Insertion and Deletion**: After inserting or deleting a node, the tree is traversed back to the root, and rotations are applied where necessary to restore the AVL property.

Asymptotic Analysis
- **Height of AVL Trees**: The height of an AVL tree with n nodes is $O(\log n)$. This ensures that search, insert, and delete operations all run in $O(\log n)$ time.
- **Proof of Height**: The minimum number of nodes in an AVL tree of height h is at least $2^{h/2}$. This leads to the conclusion that $h \leq 2 \log_2 n$, ensuring logarithmic height.

Detailed Explanation of Rotations

LL Rotation
- **Scenario**: A node becomes unbalanced due to a left-heavy left subtree.
- **Action**: Rotate the node to the right.
- **Example**:
  - Before Rotation:
    ```
        A
       /
    ```

```
    B
   /
  C
```
  - After Rotation:
```
    B
   / \
  C   A
```

RR Rotation
- **Scenario**: A node becomes unbalanced due to a right-heavy right subtree.
- **Action**: Rotate the node to the left.
- **Example**:
  - Before Rotation:
```
  A
   \
    B
     \
      C
```
  - After Rotation:
```
    B
   / \
  A   C
```

LR Rotation
- **Scenario**: A node becomes unbalanced due to a left-heavy right subtree.
- **Action**: First, perform a left rotation on the left child, then a right rotation on the node.
- **Example**:
  - Before Rotation:
```
    A
   /
  B
   \
    C
```
  - After Rotation:
```
```

```
  C
 / \
B   A
```

RL Rotation
- **Scenario**: A node becomes unbalanced due to a right-heavy left subtree.
- **Action**: First, perform a right rotation on the right child, then a left rotation on the node.
- **Example**:
  - Before Rotation:
  ```
    A
     \
      B
     /
    C
  ```
  - After Rotation:
  ```
    C
   / \
  A   B
  ```

Insertion and Deletion in AVL Trees

Insertion
1. **Standard BST Insertion**: Insert the new node as in a regular BST.
2. **Balance Check**: Traverse back to the root, checking the balance factor (height difference) at each node.
3. **Rotation**: If a node is found unbalanced, perform the appropriate rotation to restore balance.

Deletion
1. **Standard BST Deletion**: Remove the node as in a regular BST.
2. **Balance Check**: Traverse back to the root, checking the balance factor at each node.
3. **Rotation**: If a node is found unbalanced, perform the appropriate rotation(s) to restore balance. Multiple rotations may be needed.

Conclusion
Balancing BSTs, particularly through AVL trees, ensures that operations remain efficient even as the tree grows. By maintaining a balance condition and using rotations, AVL trees guarantee logarithmic height, leading to optimal performance for search, insert, and delete operations. This makes AVL trees a powerful tool for managing dynamic datasets where performance is critical.

If you need further details or additional examples, feel free to ask!

B-Trees and B+ Trees: Detailed Explanation

Introduction to B-Trees
- **Definition**: B-trees are self-balancing tree data structures that maintain sorted data and allow for efficient insertion, deletion, and search operations. They are particularly well-suited for systems that read and write large blocks of data, such as databases and file systems.
- **History**: B-trees were introduced by R. Bayer and E. McCreight in 1972. By 1979, they had become the standard for large-file access methods, replacing most other techniques except for hashing.

Properties of B-Trees
- **Height Balance**: All leaf nodes are at the same level, ensuring the tree remains balanced.
- **Branching Factor**: Each internal node (except the root) has between $\lceil m/2 \rceil$ and m children, where m is the order of the B-tree.
- **Node Capacity**: Each node can contain up to $m-1$ keys.
- **Root Node**: The root can have as few as two children if it is not a leaf.

Advantages of B-Trees
- **Shallow Trees**: Due to the high branching factor, B-trees are shallow, reducing the number of disk accesses needed to reach a record.
- **Efficient Updates**: Only the nodes along the path from the root to the target node are affected during updates, minimizing disk I/O.
- **Range Searches**: Related records (with similar key values) are stored on the same disk block, optimizing range searches.

B-Tree Operations

Search Operation
- **Process**:
  1. Start at the root node.
  2. Perform a binary search within the current node to find the appropriate child pointer.
  3. If the key is found, return the record. If not, follow the child pointer and repeat the process.
  4. If a leaf node is reached without finding the key, the search is unsuccessful.
- **Example**: Searching for key 47 in a B-tree involves traversing from the root to the appropriate leaf node.

Insertion Operation
- **Process**:

  1. Find the leaf node where the key should be inserted.
  2. If the node has space, insert the key.
  3. If the node is full, split it into two nodes and promote the middle key to the parent.
  4. If the parent becomes full, repeat the split and promotion process recursively.
- **Example**: Inserting a key into a full node of a B-tree of order 4 results in splitting the node and promoting the middle key.

Deletion Operation
- **Process**:
  1. Locate the leaf node containing the key to be deleted.
  2. If the node remains at least half full after deletion, simply remove the key.
  3. If the node underflows, borrow a key from a sibling or merge with a sibling.
  4. If merging causes the parent to underflow, repeat the process recursively.
- **Example**: Deleting a key from a B-tree may involve merging nodes to maintain the tree's properties.

B+ Trees: A Variant of B-Trees
- **Key Difference**: In B+ trees, all records are stored in the leaf nodes, while internal nodes only store keys to guide the search.
- **Leaf Node Linking**: Leaf nodes are linked together to form a doubly linked list, allowing efficient range queries and sequential access.
- **Structure**:
  - **Internal Nodes**: Store keys and pointers to child nodes.
  - **Leaf Nodes**: Store actual records or pointers to records.

B+ Tree Operations

Search Operation
- **Process**:
  1. Start at the root and traverse to the appropriate leaf node.
  2. Perform a binary search within the leaf node to find the record.
- **Example**: Searching in a B+ tree always continues to the leaf node, even if the key is found in an internal node.

Insertion Operation
- **Process**:
  1. Find the leaf node where the record should be inserted.
  2. If the node has space, insert the record.
  3. If the node is full, split it and promote the middle key to the parent.
  4. If the parent becomes full, repeat the split and promotion process.
- **Example**: Inserting into a B+ tree may involve splitting leaf nodes and promoting keys to maintain balance.

Deletion Operation

- **Process**:
  1. Locate the leaf node containing the record to be deleted.
  2. If the node remains at least half full, simply remove the record.
  3. If the node underflows, borrow a record from a sibling or merge with a sibling.
  4. If merging causes the parent to underflow, repeat the process recursively.
- **Example**: Deleting from a B+ tree may involve merging leaf nodes to maintain the tree's properties.

B* Trees: An Enhanced Variant
- **Key Difference**: B* trees aim to keep nodes at least two-thirds full by redistributing keys among siblings before splitting.
- **Advantage**: Higher space utilization and potentially fewer splits, leading to better performance.

Asymptotic Analysis
- **Time Complexity**: Search, insertion, and deletion operations in B-trees, B+ trees, and B* trees have a time complexity of $\Theta(\log n)$, where n is the number of records.
- **Space Efficiency**: B-trees and their variants are highly space-efficient, with nodes typically being at least half full.

Practical Considerations
- **Disk Access**: B-trees minimize disk I/O by keeping related records on the same disk block and maintaining a shallow tree structure.
- **Buffer Pool**: Upper levels of the tree can be kept in main memory to further reduce disk accesses.
- **High Fan-Out**: The high branching factor of B-trees ensures that the tree remains shallow, even for large datasets.

Conclusion
B-trees and their variants (B+ trees and B* trees) are fundamental data structures for managing large datasets efficiently, especially in disk-based systems. Their balanced nature, high branching factor, and efficient update operations make them ideal for applications requiring fast search, insertion, and deletion. Understanding these structures is crucial for designing and optimizing database systems and file systems.

———

B-Trees: Detailed Explanation

Introduction to B-Trees
- **Purpose**: B-trees are designed to improve locality and reduce the number of disk accesses, making them ideal for large datasets stored on disk. They are also useful for in-memory data structures due to the increasing gap between processor speed and memory access times.

- **Origin**: B-trees were initially developed for disk storage, where minimizing disk reads is crucial due to the high latency of disk access (around 5ms per read).

Properties of B-Trees
- **Order \(m \)**: A B-tree of order \(m \) is a search tree where each non-leaf node can have up to \(m \) children.
- **Structure**:
  - **Non-leaf Nodes**: Contain keys that guide the search.
  - **Leaf Nodes**: Store the actual elements of the collection.
- **Invariants**:
  1. **Uniform Path Length**: Every path from the root to a leaf has the same length.
  2. **Key-Child Relationship**: If a node has \(n \) children, it contains \(n-1 \) keys.
  3. **Node Capacity**: Every node (except the root) is at least half full.
  4. **Subtree Keys**: Elements in a subtree have keys between the keys in the parent node on either side of the subtree pointer.
  5. **Root Node**: The root has at least two children if it is not a leaf.

Advantages of B-Trees
- **High Branching Factor**: Reduces the height of the tree, minimizing the number of disk accesses.
- **Efficient Locality**: Multiple elements are stored in each node, improving cache line utilization.
- **Balanced Structure**: Ensures \(O(\log n) \) time complexity for search, insert, and delete operations.

B-Tree Operations

Lookup (Search) Operation
- **Process**:
  1. Start at the root node.
  2. Perform a linear or binary search within the current node to find the desired element or determine which child pointer to follow.
  3. Repeat the process until the element is found or a leaf node is reached.
- **Example**: Searching for an element in a B-tree involves traversing from the root to the appropriate leaf node.

Insertion Operation
- **Process**:
  1. Find the appropriate leaf node where the new element should be inserted.
  2. If the leaf node has space, insert the new element.
  3. If the leaf node is full, split it into two nodes and promote the middle key to the parent node.
  4. If the parent node becomes full, repeat the split and promotion process recursively.
  5. If the root node is split, create a new root, increasing the height of the tree by one.

- **Example**: Inserting elements into a B-tree may involve splitting nodes and promoting keys to maintain the tree's balance.

Deletion Operation
- **Process**:
  1. Locate the leaf node containing the element to be deleted.
  2. Remove the element from the leaf node.
  3. If the leaf node becomes underfull, redistribute keys from a sibling or merge with a sibling.
  4. If merging causes the parent node to underflow, repeat the redistribution or merging process recursively.
  5. If the root node has only two children and they are merged, the root is deleted, and the tree's height is reduced by one.
- **Example**: Deleting elements from a B-tree may involve merging nodes and redistributing keys to maintain the tree's properties.

Example of B-Tree Operations

Insertion Example
1. **Insert 13**: The element is inserted into a leaf node.
2. **Insert 14**: The leaf node overflows, and the node is split. A new key is added to the parent node.
3. **Insert 24**: The insertion propagates up to the root, potentially causing the root to split and increasing the tree's height.

Deletion Example
1. **Delete an Element**: The element is removed from the leaf node.
2. **Redistribute or Merge**: If the leaf node becomes underfull, keys are redistributed from a sibling, or the node is merged with a sibling.
3. **Propagate Changes**: If merging affects the parent node, the process continues up the tree, potentially reducing the tree's height.

Further Reading
- **Reference**: Aho, Hopcroft, and Ullman, *Data Structures and Algorithms*, Chapter 11.

Conclusion
B-trees are powerful data structures that provide efficient search, insertion, and deletion operations, especially for large datasets stored on disk. Their high branching factor and balanced structure ensure logarithmic time complexity, while their design optimizes locality and minimizes disk accesses. Understanding B-trees is essential for designing efficient database systems and file systems.

If you need further details or specific examples, feel free to ask!

———

B+ Trees: Insertion Process Explained

Introduction to B+ Trees
- **Structure**: B+ trees are a type of self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. They are particularly useful for databases and file systems.
- **Key Features**:
  - **Leaf Nodes**: Store actual data records or pointers to records.
  - **Internal Nodes**: Store keys that guide the search.
  - **Linked Leaves**: Leaf nodes are linked together to facilitate range queries and sequential access.

Insertion Process in B+ Trees

Initial State
- **Tree Order**: \(m = 4 \) (each node can hold up to 3 keys and 4 pointers).
- **Initial Keys**: 21, 42, 63, 89.
- **Initial Node**: The first node is both a leaf node and the root node.

Insertion Steps

1. **Insert 35**:
   - **Leaf Node Split**: The leaf node is full, so it splits into two nodes.
   - **Redistribution**: The keys are redistributed between the original node and the new node.
   - **Parent Update**: The smallest key from the new node (42) is copied up to the parent node, which is created as an internal node.

2. **Insert 10, 27, 96**:
   - **Root Node Search**: The insertion process starts at the root node. The keys of the root node are searched to determine which child node to descend to.
   - **Example**: For key 10, since \(10 < 42 \), the left child node is followed.
   - **No Splits**: None of these insertions cause a node to split.

3. **Insert 30**:
   - **Leaf Node Insertion**: The key 30 is inserted into the left-most leaf node.
   - **Leaf Node Split**: The leaf node is full, so it splits into two nodes.
   - **Redistribution**: The keys are redistributed between the original node and the new node.
   - **Parent Update**: The smallest key from the new node (27) is copied up to the parent node.

4. **Insert 37**:
   - **Leaf Node Insertion**: The key 37 is inserted into the appropriate leaf node.
   - **Leaf Node Split**: The leaf node is full, so it splits into two nodes.
   - **Internal Node Split**: The parent node (which is also the root) is full, so it splits into two nodes.

  - **New Root Creation**: The middle key (35) is moved up to form a new root node, increasing the tree's height by one.

Detailed Example of Insertion

Insert 35
- **Initial Leaf Node**: [21, 42, 63, 89]
- **After Insertion**: The leaf node splits into:
  - **Original Node**: [21, 35]
  - **New Node**: [42, 63, 89]
- **Parent Node**: [42] (new internal node)

Insert 10, 27, 96
- **Root Node**: [42]
- **Left Child**: [21, 35]
- **Right Child**: [63, 89]
- **After Insertions**:
  - **Left Child**: [10, 21, 27, 35]
  - **Right Child**: [63, 89, 96]

Insert 30
- **Leaf Node**: [10, 21, 27, 35]
- **After Insertion**: The leaf node splits into:
  - **Original Node**: [10, 21]
  - **New Node**: [27, 30, 35]
- **Parent Node**: [27, 42]

Insert 37
- **Leaf Node**: [27, 30, 35]
- **After Insertion**: The leaf node splits into:
  - **Original Node**: [27, 30]
  - **New Node**: [35, 37]
- **Parent Node**: [35, 42]
- **Root Node Split**: The parent node is full, so it splits into:
  - **New Root**: [35]
  - **Left Child**: [27, 30]
  - **Right Child**: [42, 63, 89, 96]

Conclusion
The insertion process in B+ trees involves finding the appropriate leaf node for the new key and splitting nodes as necessary to maintain the tree's balance. When a node splits, the middle key is promoted to the parent node, and this process may ripple up to the root, potentially increasing the tree's height. This ensures that the tree remains balanced and efficient for search, insert, and delete operations.

If you need further details or specific examples, feel free to ask!