AVL Tree Basics:

Balanced Tree: An AVL tree is a self-balancing binary search tree (BST), where the difference between the heights of the left and right subtrees of any node is at most 1.

Balance Factor: The balance factor of a node is the height of the left subtree minus the height of the right subtree.

Balance Factor of 1: Left subtree is higher than the right.

Balance Factor of -1: Right subtree is higher than the left.

Balance Factor of 0: Both subtrees are of equal height.

Imbalance occurs when the balance factor is greater than 1 or less than -1.

Insertion Process:

Insert the node as you would in a standard Binary Search Tree (BST):

Start at the root, and recursively insert the node based on whether its key is smaller or larger than the current node's key.

After insertion, check the balance factor of each ancestor node (from the inserted node to the root).

If any node's balance factor becomes greater than 1 or less than -1, the tree is imbalanced and must be rebalanced.

Imbalance Cases and Rotations: When an AVL tree becomes imbalanced, one of four rotations is performed to restore balance. The rotation type depends on the structure of the tree and the node where the imbalance occurs.

LL (Left-Left Case):

Occurs when a node's left subtree is taller than its right subtree, and the imbalance is in the left child of the left subtree.

Rotation: Perform a single right rotation (rotate the unbalanced node with its left child).

Example: When inserting a node that causes the left subtree of the left child to become taller.

RR (Right-Right Case):

Occurs when a node’s right subtree is taller than its left subtree, and the imbalance is in the right child of the right subtree.

Rotation: Perform a single left rotation (rotate the unbalanced node with its right child).

Example: When inserting a node that causes the right subtree of the right child to become taller.

LR (Left-Right Case):

Occurs when the imbalance is in the left subtree of the right child. This happens when the left child of the left subtree is shorter than its right child.

Rotation: First, perform a left rotation on the left child, and then perform a right rotation on the unbalanced node.

Example: When inserting a node that causes the left subtree of the right child to become taller.

RL (Right-Left Case):

Occurs when the imbalance is in the right subtree of the left child. This happens when the right child of the right subtree is shorter than its left child.

Rotation: First, perform a right rotation on the right child, and then perform a left rotation on the unbalanced node.

Example: When inserting a node that causes the right subtree of the left child to become taller.

Rebalancing with Rotations: After detecting an imbalance, perform the necessary rotation:

Single rotations: For LL and RR cases.

Double rotations: For LR and RL cases (first one direction, then the other).

Insertion Example Walkthrough: Here's an example of detecting and fixing imbalances:

Inserting 60:

After inserting the key 60 into the AVL tree, an imbalance occurs at node 20. The imbalance case is RR (right-heavy imbalance at node 20).

Fix: Perform a single left rotation on node 20.

Inserting 89:

After inserting the key 89, an imbalance occurs at node 60. The imbalance case is RR.

Fix: Perform a single left rotation on node 60.

Inserting 30:

After inserting 30, an imbalance occurs at node 40. The imbalance case is LR (left-heavy imbalance on the right child of node 40).

Fix: First, perform a left rotation on node 60, then perform a right rotation on node 40.

Inserting 31:

After inserting 31, an imbalance occurs at node 40. The imbalance case is LL (left-heavy imbalance at node 40).

Fix: Perform a single right rotation on node 40.

Inserting 32:

After inserting 32, an imbalance occurs at node 60. The imbalance case is LR (left-heavy imbalance on the right child of node 60).

Fix: First, perform a left rotation on node 40, then perform a right rotation on node 60.

Inserting 70:

After inserting 70, an imbalance occurs at node 40. The imbalance case is RL (right-heavy imbalance on the left child of node 40).

Fix: First, perform a right rotation on node 60, then perform a left rotation on node 40.

Balance Factors:

After each rotation, update the balance factor of the affected nodes.

Rebalancing ensures that the AVL tree maintains its height-balanced property, where the difference in height between left and right subtrees is at most 1.

Hash table notes:

Hash Table Basics: A hash table is a data structure used to store key-value pairs. It uses a hash function to compute an index (or hash code) in an array, where the value associated with the key is stored. This allows for efficient lookups, insertions, and deletions, typically in constant time.

Hash Function: A hash function takes an input (or key) and computes an index based on the size of the hash table. The result is an integer that is mapped to a specific position in the table.

Hash Function Example:

Common hash functions include:

Modulo-based hash function: h(A) = A mod table_size

Example: For a hash table of size 10, the hash function is h(A) = A mod 10.

The hash function ensures that each key is mapped to a valid index in the table, but collisions may occur when two keys hash to the same index.

Collision Resolution: Collision occurs when two keys hash to the same index. There are several methods for resolving collisions, and the two common methods are:

Separate Chaining:

In separate chaining, each index in the hash table is a linked list (or another collection) that stores all elements that hash to the same index.

How it works: If a collision occurs (i.e., two keys hash to the same index), the second key is added to the list at that index.

Example: If keys A and B both hash to index 0, then the hash table entry at index 0 contains a linked list with A and B.

Open Addressing:

Open addressing resolves collisions by finding another open index within the hash table itself, rather than using a linked list.

This method involves searching for the next available index, which can use techniques like linear probing, quadratic probing, or double hashing.

Insertion Process with Separate Chaining: The insertion of a key-value pair in a hash table with separate chaining involves the following steps:

Compute the hash index: Use the hash function to compute the index for the key.

Insert the key-value pair: If there is no collision at the computed index, insert the pair directly into the table. If there is a collision (i.e., the index already has a key-value pair), add the new pair to the linked list at that index.

Handle collisions: Collisions are handled by adding the new key-value pair to the linked list at the corresponding index.

Example Insertion Process:

Given the tuple list from your question:

ruby
Copy
Tuples: [(20:O), (40:S), (60:T), (80:R), (89:N), (70:E), (30:T), (10:N), (33:A), (31:H), (24:R), (32:E)]
Hash Function: h(A) = A mod 10
Table size: 10
Step-by-step Insertion:

(20:O):

Hash: h(20) = 20 mod 10 = 0

Insert (20:O) at index 0.

(40:S):

Hash: h(40) = 40 mod 10 = 0

Collision at index 0, so append (40:S) to the linked list at index 0.

(60:T):

Hash: h(60) = 60 mod 10 = 0

Collision at index 0, append (60:T) to the linked list at index 0.

(80:R):

Hash: h(80) = 80 mod 10 = 0

Collision at index 0, append (80:R) to the linked list at index 0.

(89:N):

Hash: h(89) = 89 mod 10 = 9

Insert (89:N) at index 9.

(70:E):

Hash: h(70) = 70 mod 10 = 0

Collision at index 0, append (70:E) to the linked list at index 0.

(30:T):

Hash: h(30) = 30 mod 10 = 0

Collision at index 0, append (30:T) to the linked list at index 0.

(10:N):

Hash: h(10) = 10 mod 10 = 0

Collision at index 0, append (10:N) to the linked list at index 0.

(33:A):

Hash: h(33) = 33 mod 10 = 3

Insert (33:A) at index 3.

(31:H):

Hash: h(31) = 31 mod 10 = 1

Insert (31:H) at index 1.

(24:R):

Hash: h(24) = 24 mod 10 = 4

Insert (24:R) at index 4.

(32:E):

Hash: h(32) = 32 mod 10 = 2

Insert (32:E) at index 2.

After all insertions, the hash table (with separate chaining) looks like this:
Index 0: [(20:O), (40:S), (60:T), (80:R), (70:E), (30:T), (10:N)]
Index 1: [(31:H)]

Index 2: [(32:E)]
Index 3: [(33:A)]
Index 4: [(24:R)]
Index 9: [(89:N)]

Key points to remember:

Separate Chaining allows multiple key-value pairs at the same index using a linked list or another structure at each table index.

The load factor (number of elements divided by the size of the hash table) affects performance. High load factors lead to increased collisions.

Resizing the hash table may be necessary when the load factor exceeds a threshold (e.g., 0.7 or higher).

Use a good hash function to ensure that the keys are evenly distributed across the table to minimize collisions.

B+ Tree Notes:

What is a B+ Tree?

A B+ tree is a self-balancing tree data structure that maintains sorted data and allows efficient insertion, deletion, and search operations.

Properties of a B+ tree:

All leaves are at the same level, ensuring uniform height.

Internal nodes store only keys (no data), while leaf nodes store keys and actual data.

Insertion and deletion of keys trigger node splitting and node merging.

Leaf nodes are linked together in a linked list to allow range queries.

Structure of a B+ Tree:

Internal Nodes:

Internal nodes only contain keys, which guide searches towards the appropriate child node.

The number of keys in an internal node is always one less than the number of child pointers.

If the internal node is full (i.e., the number of keys exceeds the maximum), it splits and promotes one key to the parent node.

Leaf Nodes:

Leaf nodes store the actual data and are connected in a linked list for sequential access.

Each leaf node can hold a maximum number of keys, as specified by the tree's order (M). In your case, each leaf node can store up to 3 keys.

Node Splitting:

When inserting a key into a node that is already full, the node splits. The splitting rule is:

Internal Nodes: When an internal node overflows, one key is moved to the parent node, and the rest stay in the newly created child node.

Leaf Nodes: When a leaf node overflows, one key is promoted to the parent node, and the other keys remain in the new leaf node.

Splitting Rule: For a node split:

Keep one element in the left node.

Move the remaining elements to the right node.

Insertion Process in B+ Trees:

Step 1: Start by inserting the new key into the appropriate leaf node.

Step 2: If the leaf node overflows (i.e., exceeds the maximum number of keys), split the node:

The middle key is promoted to the parent node.

The remaining keys are split between the old and new leaf nodes.

Step 3: If the parent node overflows, repeat the splitting process.

Example Insertion Process: Let's go through some of your insertions for better clarity.

Inserting 90:

Insert 90 in the first leaf node. This will cause the first internal node to split, and 90 will be promoted to the parent, increasing the height of the tree.

Inserting 13:

Insert 13 in the leaf node. 13 will fit without causing a split.

Inserting 74:

Insert 74 into a leaf node, causing an overflow in the node. This triggers a split and 74 is promoted to the parent.

Inserting 70:

70 will be inserted into a leaf node. This causes a split, and 70 is promoted to the parent, which increases the height of the tree by 1.

Inserting 47:

Insert 47 into the leaf node. No split occurs for this insertion.

Inserting 29:

Insert 29 into a leaf node. A split occurs, and 29 is promoted to the parent.

Inserting 50:

Insert 50 into a leaf node. This causes the node to overflow, triggering a split and promoting 50 to the parent node.

Key Points on B+ Tree Insertion:

A node split occurs when inserting a key into a node that exceeds its maximum capacity of keys. For an internal node, one key is moved up to the parent. For a leaf node, the middle key is moved up.

Height increase happens when a split propagates all the way to the root, creating a new level in the tree.

Linked leaves allow for fast range queries by traversing the leaves sequentially.

Important Rules to Remember:

M = 3 (Maximum number of keys per internal node and leaf node).

Internal nodes contain maximum of 3 keys and 4 child pointers.

Leaf nodes also contain a maximum of 3 keys.

For node splitting, 1 element stays in the left node, and 2 elements move to the newly created right node.