Homework # 02 Solution

Section A: Multiple Choice Questions (MCQs)

Q1)Which traversal of a Binary Search Tree (BST) results in keys in ascending order?

A) Preorder

B) Inorder

C) Postorder

D) Level order

Q2)A B+ tree supports rapid random access and sequential access because:

A) Only random access is efficient

B) Leaves are linked together and tree is balanced

C) Only sequential access is efficient

D) It uses hashing for access

Q3)What is the balance factor in an AVL tree?

A) Height(left subtree) + Height(right subtree)

B) Height(right subtree) − Height(left subtree)

C) |Height(left subtree) − Height(right subtree)| ≤ 1

D) Height(left subtree) − Height(right subtree)

Q4)If you perform a preorder traversal on a BST containing keys {10, 5, 20, 3, 7}, which sequence do you get?

A) 3, 5, 7, 10, 20

B) 10, 5, 3, 7, 20

C) 20, 10, 5, 7, 3

D) 5, 3, 7, 10, 20

Q5)When inserting a new node into an AVL tree, at most how many rotations are required to restore balance?

A) O(n)

B) 2

C) $\log_2(n)$

D) Unlimited

Q6) In a B+ tree, where are the actual data records (or pointers to them) stored?
A) In the internal nodes only

B) In both internal and leaf nodes

C) Only in the leaf nodes

D) Only in the root node

Q7) What **is the primary reason for preferring a B+ tree over a BST for indexing a large database stored on disk?**

A) B+ trees are always perfectly balanced.

B) B+ tree operations have a better

Big-O complexity in terms of CPU time.

C) B+ trees are designed to minimize the number of disk I/O operations.

D) B+ trees can store more distinct keys than a BST of the same height.

Q8) An AVL tree is a self-balancing version of which fundamental data structure?

A) B-Tree
B) Binary Search Tree
C) Red-Black Tree
D) Heap

Q9) If a node in an AVL tree has a balance factor of +2, what does this signify according to the

formula
A) The tree is critically left-heavy at that node.
B) The tree is critically right-heavy at that node.
C) The tree is perfectly balanced.
D) The node is a leaf node.

Q10) What is the worst-case time complexity for search, insert, and delete operations in a balanced AVL tree?

A) O(n)

B) O(log n)

C) O(n log n)

D) O(1)

Q11) Which of the following is true for a Binary Search Tree (BST)?
a) The left child is always smaller than the root, and the right child is always larger.
b) The left and right subtrees are always balanced.
c) Inorder traversal of a BST gives elements in descending order.
d) The root must always be the smallest element.

Q12) In a BST, the time complexity of searching for an element in the average case is:
a) O(1)
b) O(log n)
c) O(n)
d) O(n log n)

Q13) When deleting a node with two children in a BST, the node is usually replaced with:
a) Its left child
b) Its right child
c) Its inorder successor or predecessor
d) The root node

Q14) Which traversal of a BST always results in a sorted sequence of elements?
a) Preorder
b) Postorder
c) Inorder
d) Level-order

Q15) Consider the BST formed by inserting the numbers 15, 10, 20, 8, 12. What is the postorder traversal of this tree?
a) 8, 12, 10, 20, 15
b) 12, 8, 10, 20, 15
c) 8, 10, 12, 20, 15
d) 20, 12, 10, 8, 15

Q16) The worst-case height of a BST with n nodes is:
a) O(log n)
b) O(n)
c) O(n log n)
d) O(√n)

Q17) Which of the following sequences cannot represent the inorder traversal of a BST?
a) 10, 20, 30, 40, 50

b) 50, 40, 30, 20, 10
c) 20, 30, 25, 40, 50
d) 5, 10, 15, 20, 25

Q18) A BST is balanced if:
a) Every node has at most two children
b) The left subtree and right subtree of every node differ in height by at most 1
c) The inorder traversal results in a sorted sequence
d) The root is the median of all elements

Q19) If you build a BST by inserting 1, 2, 3, 4, 5 in that order, what structure do you get?
a) A complete binary tree
b) A skewed tree (right-skewed)
c) A skewed tree (left-skewed)
d) A perfectly balanced tree

Q20) If two BSTs contain the same set of elements, then:
a) Their inorder traversals must be the same
b) Their structures must be identical
c) Both a and b
d) None of the above

Q21) If a BST has height h, the minimum number of nodes it can have is:
a) h
b) 2^h
c) h + 1
d) h

Q22)An AVL tree is a type of:
a) Binary Search Tree
b) Heap
c) Graph
d) B-tree

Q23) The balance factor of a node in an AVL tree is defined as:
a) Height of right subtree – Height of left subtree
b) Height of left subtree – Height of right subtree
c) Total number of nodes in the tree
d) Depth of the node

Q24) What are the possible values of the balance factor in a valid AVL tree?
a) -2, -1, 0, 1, 2
b) -1, 0, 1
c) 0 only
d) Any integer

Q25) The height of an AVL tree with *n* nodes is always:
a) O(n)
b) O(log n)
c) O(√n)
d) O(1)

Q26)Which of the following is true for an AVL tree?
a) It is always a complete binary tree
b) It is always a full binary tree
c) It is a height-balanced binary search tree
d) It always has equal number of nodes in left and right subtrees

Q27)Inserting a new node into an AVL tree may require:
a) At most 1 rotation
b) At most 2 rotations
c) At most log n rotations
d) Unlimited rotations

Q28)What is the worst-case height H of an AVL tree with n nodes?
a) h = O(n)
b) h = O(log n)
c) h = O(log log n)
d) h = O(√n)

Q29)The time complexity of search, insertion, and deletion in an AVL tree is:
a) O(1)
b) O(h)
c) O(log n)
d) O(n)

Q30) What is the minimum possible height of an AVL tree with 1000 nodes?
a) ≈ 9
b) ≈ 10
c) ≈ 11
d) ≈ 12

Q31)Which imbalance occurs when a node is inserted into the left subtree of the left child of a node?
a) LL imbalance
b) RR imbalance
c) LR imbalance
d) RL imbalance

Q32)Which rotation is required to fix an LL imbalance?
a) Right rotation
b) Left rotation

c) Left-Right rotation
d) Right-Left rotation

Q33)After an insertion, the balance factor of a node becomes +2 and the balance factor of its left child is +1. What rotation is required?
a) Single left rotation
b) Single right rotation
c) Left-Right rotation
d) Right-Left rotation

Q34)After an insertion, the balance factor of a node becomes +2 and the balance factor of its left child is -1. What rotation is required?
a) Single left rotation
b) Single right rotation
c) Left-Right rotation
d) Right-Left rotation

Q35)Suppose a B+ Tree of order m has n keys. What is the maximum possible number of leaf nodes?
a) n
b) n/m
c) n/(m - 1)
d) log_m(n)

Q36) In a B+ Tree of order m, the minimum number of keys in a non-root internal node is:
a) $\lceil m/2 \rceil$ - 1
b) $\lfloor m/2 \rfloor$
c) m/2
d) $\lceil m/2 \rceil$

Section B: Short Questions:

Question No1:

```
        15
       /  \
     10    20
    / \      \
   8   12      25
```

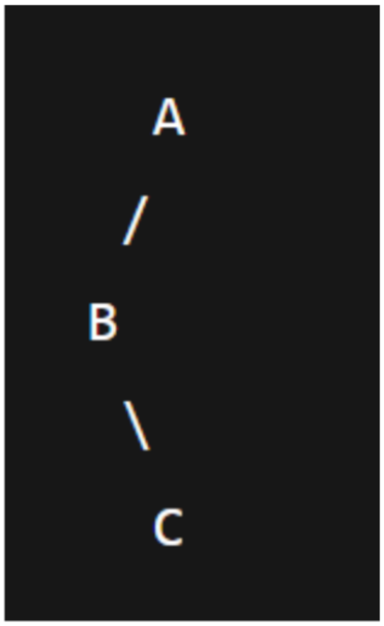
Height of BST: 2

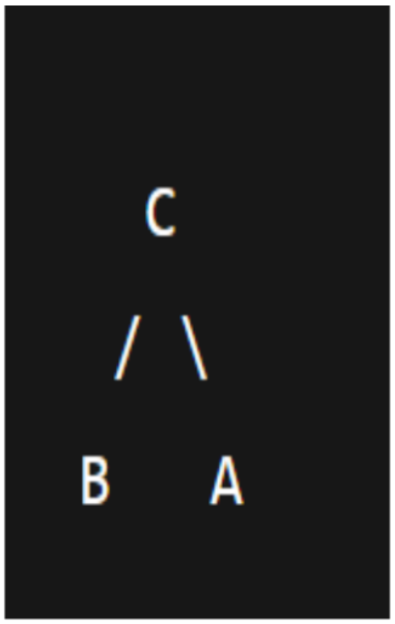
Question No2:

a)Case = Left-Right (LR) imbalance

b)Fix = Double Rotation: Left on child, then Right on parent

Before: After:





Question No3:

i)Maximum keys in a non-root internal node = children − 1 = 4 − 1 = 3.

Minimum children for a non-root internal node = $\lceil m/2 \rceil$ = $\lceil 4/2 \rceil$ = 2.

Minimum keys = minimum children − 1 = 2 − 1 = 1.

ii)B+ trees are highly efficient for range queries because all actual records are stored in the leaf nodes, and these leaves are linked together in sequence. Once the first key in the range is located using a single O(log n) search, the remaining keys can be retrieved by simply traversing the linked leaves sequentially. This avoids repeated tree lookups and minimizes disk I/O, making B+ trees far better suited for range queries than AVL trees, which lack sequential leaf links and would require multiple random searches.

Question No4:

Question No4:

```
void printGreater(Node* root, int k) {

    if (!root) return;
    if (root->data > k) {
        cout << root->data << " ";
        printGreater(root->left, k);
        printGreater(root->right, k);
    } else {
        printGreater(root->right, k);
    }
}
```

Question No 5:

a) The most suitable data structure is a B+ tree.

b) Explanation:

Efficient Range Queries: The key requirement is retrieving all items in a range. B+ trees are perfect for this because all data pointers are in the leaf nodes, and these leaves are connected sequentially like a linked list. This allows for a single search to find the start of the range, followed by a simple, fast traversal along the leaf nodes.

Optimized for Disk I/O: Since the database is on disk, minimizing disk access is crucial. B+ trees have a high branching factor (many children per node), which keeps the tree very short and wide. A shorter tree means fewer disk reads are needed to traverse from the root to the leaves, making it much faster than a tall, skinny tree like a BST or AVL tree for disk-based data.

c) Algorithm:

Perform a search on the B+ tree to locate the leaf node containing the minPrice.

Once at the leaf node, scan through its keys and retrieve the data for all products whose prices are within the range [minPrice, maxPrice].

Follow the sequential pointer to the next leaf node and repeat step 2.

Continue this process until you encounter a price greater than maxPrice.

Question No6:

a) The best choice is an AVL tree.

b) Justification:

Guaranteed Performance: The primary need is for fast lookups, insertions, and deletions (O(log n)). A standard BST risks degenerating into a linear structure (becoming like a linked list) if usernames are inserted in a somewhat sorted order, leading to O(n) performance, which is too slow. An AVL tree automatically balances itself, guaranteeing that all operations remain O(log n).

In-Memory Operation: Since the data is in memory, the disk I/O advantage of a B+ tree is irrelevant. The overhead of maintaining the complex node structure of a B+ tree is unnecessary and less efficient for in-memory operations compared to the simpler structure of an AVL tree.

c)

```
struct Node {
    string username;
    Node *left, *right;
    int height;
};

bool doesUserExist(Node* root, string username) {
    if (!root) {
        return false; // User not found
    }

    if (username == root->username) {
        return true; // User found
    }
    if (username < root->username) {
        return doesUserExist(root->left, username); // Search in left subtree
    } else {
        return doesUserExist(root->right, username); // Search in right subtree
    }
}
```

Question No7:

a) The most appropriate data structure is an AVL tree.

b) Justification:

Guaranteed Performance: A standard BST could become unbalanced if scores are inserted in a sorted or nearly sorted manner, leading to worst-case O(n) performance, which is unacceptable for a real-time system. An AVL tree's self-balancing nature guarantees that insertions, updates (delete + insert), and lookups will always be O(log n).

In-Memory Optimization: A B+ tree is designed for minimizing disk I/O and is overly complex for a purely in-memory task. An AVL tree has less overhead per node and is better suited for main memory operations.

c) Algorithm for Top 10 Players:
To get the highest scores, you need to retrieve keys in descending order. This can be done efficiently with a reverse inorder traversal:

Start at the root node.

Recursively traverse the right subtree (which contains all higher scores).

Visit and record the current node's data.

Recursively traverse the left subtree.

Stop the traversal after you have collected 10 players.

```
void getTopPlayers(Node* root, vector<pair<int, string>>& result, int limit = 10)
{
    if (!root || result.size() >= limit) return;

    // Traverse right subtree first (higher scores)
    getTopPlayers(root->right, result, limit);

    // Record current node
    if (result.size() < limit) {
        result.push_back({root->score, root->player});
    }

    // Traverse left subtree
    getTopPlayers(root->left, result, limit);
}
```

Question No8:

a) A B+ tree is the ideal data structure for this index.

b) Explanation:

Disk I/O Efficiency: B+ trees are short and wide due to their high branching factor. This minimizes the number of disk reads required to find a specific date, which is the main performance bottleneck for disk-based data structures.

Excellent for Range Queries: The core feature is searching for a range of dates. B+ trees are perfect for this because their leaf nodes are linked together sequentially. The system can perform one O(log n) search to find the leaf with the start date and then simply follow the pointers through the linked leaves to read all the entries in the date range, which is extremely fast and minimizes disk seeks.

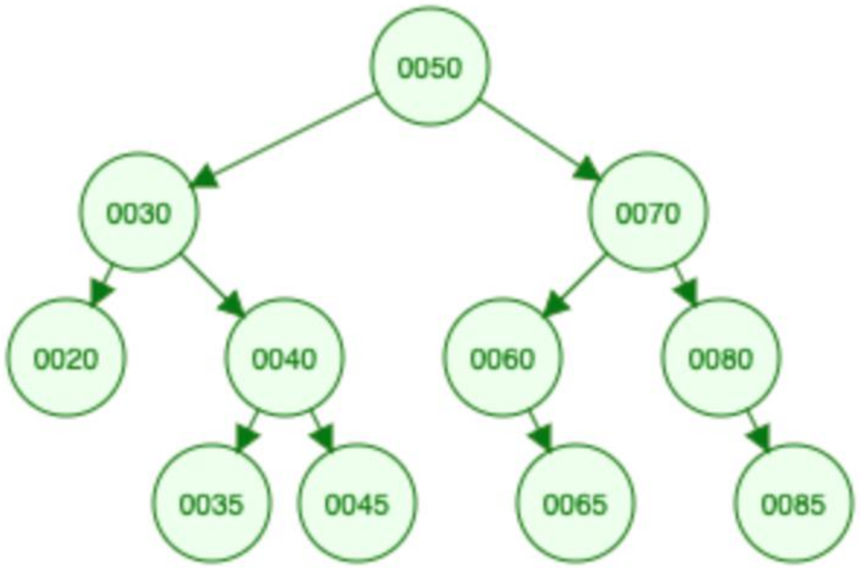
c) This is the Left-Right (LR) Case. It requires a double rotation to fix:

A Left Rotation on the left child.

A Right Rotation on the parent node (the one that was originally imbalanced).

Question No9:

a)



Part b)

1. 3

2. 5 (20, 35, 45, 65, 85)

3. 6

4. A complete binary tree must have all levels filled except possibly the last, and last filled from left to right. The structure violates this completeness.
Therefore, no, it is not complete.

Part c)

1. [50,70,60]

2. [50, 30]

3. [20, 40, 35, 45]

4. 40

5. 45

6. 60 has no left subtree, so predecessor is the nearest ancestor for which 60 is in the ancestor's right subtree.
That ancestor is 50, so predecessor = 50.

Question No10:

a)Preorder (Root, Left, Right): root is processed first so **0 nodes** before the root.

Inorder (Left, Root, Right): all nodes in left subtree are processed before root so **500 nodes.**

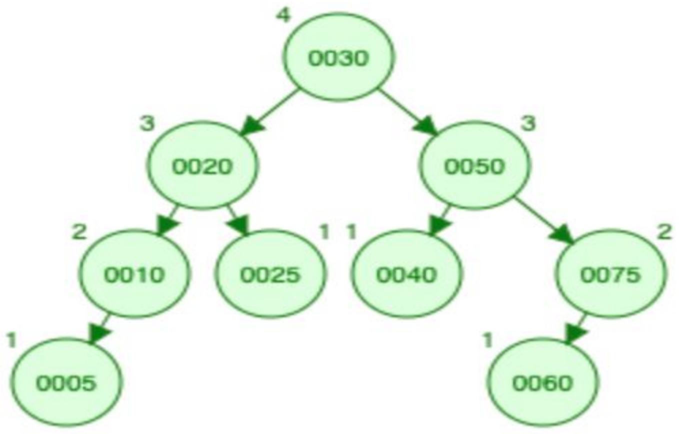
Postorder (Left, Right, Root): all nodes in left and right subtrees are processed before root so 500 + 200 = 700 nodes.

2b)R S Q U T P

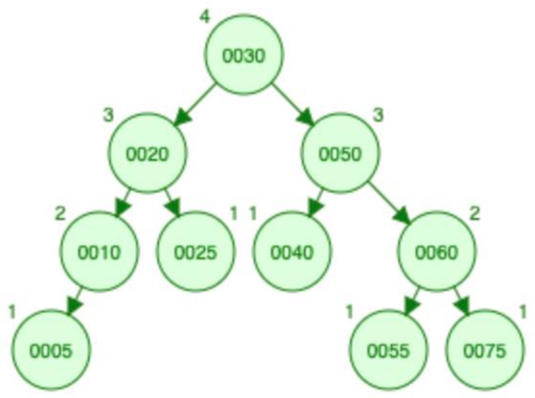
2c) Nodes at **level 2** are: 20, 45, 70, 90 so total **4 nodes**.

Question No11:

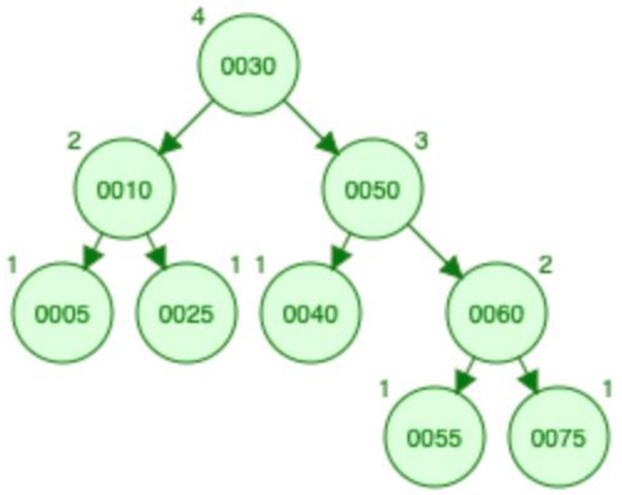
a) **Insert([30, 20, 40, 10, 25, 50, 5, 75, 60])**



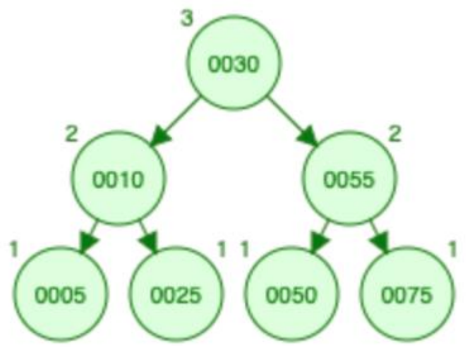
a) b) Insert(55)



c)Delete(20)



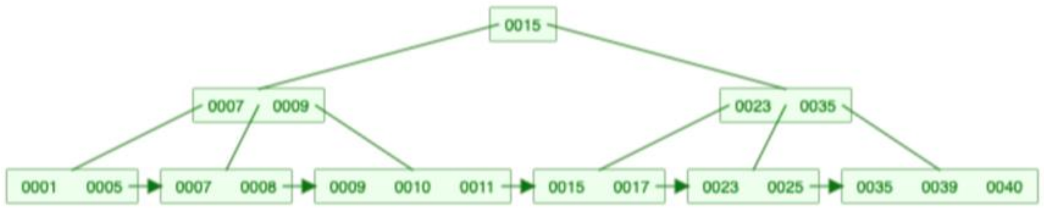
d) Delete (60, 40)



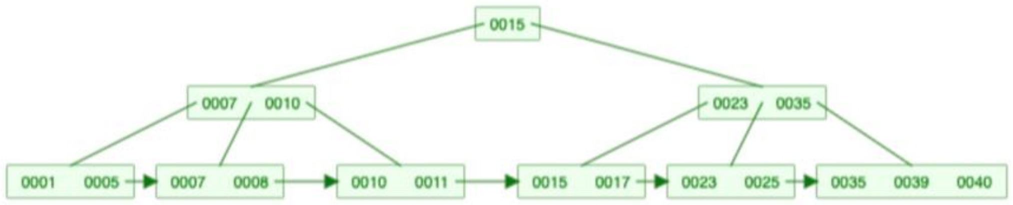
Question No12:

Insert([7, 10, 1, 23, 5, 15, 17, 9, 11, 39, 35, 8, 40, 25])

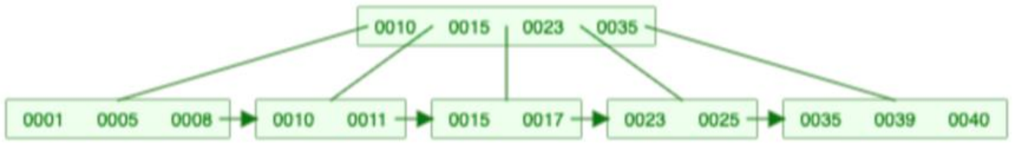
Answer:



Delete(9)



Delete(7)


0010 0015 0023 0035
0001 0005 0008
0010 0011
0015 0017
0023 0025
0035 0039 0040