1. Which type of Binary Search Tree is particularly efficient for range queries?
   1. AVL Tree
   2. Red-Black Tree
   3. Splay Tree
   4. **Interval Tree**
2. In a Binary Search Tree, which traversal technique is used to visit nodes in descending order?
   1. Inorder
   2. Preorder
   3. **Postorder**
   4. Level-order
3. Which operation is NOT typically performed using a Binary Search Tree?
   1. Finding the maximum element
   2. Finding the minimum element
   3. **Finding the median element**
   4. Sorting a random list of elements
4. In a B+ Tree, the data is stored only in:
   1. Internal nodes
   2. **Leaf nodes**
   3. Both internal and leaf nodes
   4. Root node
5. Which operation on a Red-Black Tree requires additional adjustments to maintain the properties?
   1. Insertion
   2. **Deletion**
   3. Searching
   4. Traversal
6. The B-Tree is optimized for:
   1. Searching in internal memory
   2. Sorting elements
   3. Storing elements in a queue
   4. **Searching in external storage**
7. In a Binary Search Tree, the successor of a node is:
   1. **The node with the smallest key greater than the given node**
   2. The node with the largest key smaller than the given node
   3. The left child of the given node
   4. The right child of the given node
8. Which balancing technique is used in a B-Tree to maintain balance after a node insertion?
   1. Rotation
   2. Transplant
   3. **Splitting**
   4. Splaying
9. In a Binary Search Tree, which of the following operations has the highest time complexity?
   1. Insertion
   2. **Deletion**
   3. Searching for an element
   4. Finding the maximum element
10. Which of the following is NOT a property of a Binary Search Tree?
    1. The left subtree of a node contains elements less than the node's key
    2. The right subtree of a node contains elements greater than the node's key
    3. **The left and right subtrees are balanced**
    4. There are no duplicate elements
11. In a Binary Search Tree, what is the time complexity for finding the kth smallest element?
    1. O(k)
    2. **O(log n)**
    3. O(n)
    4. O(n log n)
12. Which balancing technique is used in an Splay Tree to bring the most recently accessed node to the root?
    1. Rotation
    2. Splitting
    3. Zig-Zig operation
    4. **Splaying**
13. In a B-Tree, the minimum degree specifies:
    1. **The number of children each internal node can have**
    2. The number of keys each internal node can have
    3. The number of keys each leaf node can have
    4. The maximum height of the tree
14. Which of the following is NOT a self-balancing Binary Search Tree?
    1. Splay Tree
    2. B-Tree
    3. AVL Tree
    4. **Binary Heap**
15. What is the worst-case time complexity of AVL tree deletion?

**a) O(log n)**

b) O(n)

c) O(n^2)

d) O(1)

1. Consider the following AVL tree:

20

/ \

10 30

/ \ \

5 15 40

After deleting the node with value 40 from the above AVL tree, what will be the new root value?

a) 10

b) 15

**c) 20**

d) 30

1. When performing AVL tree deletion, which rotation(s) might be necessary to restore balance?

a) Left rotation

b) Right rotation

c) Double rotation (Left-Right or Right-Left)

**d) All of the above**

1. What will be the output of the following AVL tree deletion operation?

AVLTree tree;

tree.insert(10);

tree.insert(5);

tree.insert(15);

tree.insert(3);

tree.deleteNode(10);

tree.inOrderTraversal();

**a) 3 5 15**

b) 3 5

c) 3 15

d) 5 15

1. In AVL tree deletion, if a node has two children, what strategy is used to find its successor?

**a) Inorder successor**

b) Preorder successor

c) Postorder successor

d) Random successor

1. What will be the output of the following AVL tree deletion operation?

AVLTree tree;

tree.insert(50);

tree.insert(25);

tree.insert(75);

tree.deleteNode(25);

tree.inOrderTraversal();

a) 25 50 75

b) 25 75

**c) 50 75**

d) 25 50

1. During AVL tree deletion, how many passes through the tree are performed?

a) One pass

**b) Two passes**

c) Three passes

d) It depends on the tree's height

1. Consider the pseudo code:

int avl(binarysearchtree root):

if(not root)

return 0

left\_tree\_height = avl(left\_of\_root)

if(left\_tree\_height== -1)

return left\_tree\_height

right\_tree\_height= avl(right\_of\_root)

if(right\_tree\_height==-1)

return right\_tree\_height

Does the above code can check if a binary search tree is an AVL tree?

**a) yes**

b) no

1. After performing a single left rotation during AVL tree deletion, the height balance of the affected nodes becomes:

**a) (0, 0)**

b) (-1, 1)

c) (1, 0)

d) (0, 1)

1. AVL\_delete(Node\* root, int key):

if root is NULL:

return NULL

else if key < root->data:

root->left = AVL\_delete(root->left, key)

else if key > root->data:

root->right = AVL\_delete(root->right, key)

else:

if root->left is NULL or root->right is NULL:

temp = root->left ? root->left : root->right

root = temp

else:

temp = find\_min\_node(root->right)

root->data = temp->data

root->right = AVL\_delete(root->right, temp->data)

return root

What does the function find\_min\_node() do in this pseudo-code?

a) Finds the maximum node in the AVL tree

**b) Finds the node with the minimum value in the AVL tree**

c) Finds the node with the maximum value in the AVL tree

d) Finds the parent node of the node with the minimum value

1. What is the height difference between the left and right subtrees of an AVL node after deletion to keep it balanced?

a) -1

b) 0

**c) 1**

d) 2

1. What is the correct pseudo-code for deleting a node with key val from an AVL tree?

**a)**

**function deleteNode(root, val):**

**if root is NULL:**

**return NULL**

**if val < root.key:**

**root.left = deleteNode(root.left, val)**

**else if val > root.key:**

**root.right = deleteNode(root.right, val)**

**else:**

**// Node found, perform deletion**

**// Deletion logic goes here**

**return root**

b)

function deleteNode(root, val):

if root is NULL:

return NULL

if val < root.key:

root.right = deleteNode(root.right, val)

else if val > root.key:

root.left = deleteNode(root.left, val)

else:

// Node found, perform deletion

// Deletion logic goes here

return root

c)

function deleteNode(root, val):

if root is NULL:

return NULL

if val < root.key:

root = deleteNode(root.left, val)

else if val > root.key:

root = deleteNode(root.right, val)

else:

// Node found, perform deletion

// Deletion logic goes here

return root

d)

function deleteNode(root, val):

if root is NULL:

return NULL

if val < root.key:

root.left = deleteNode(root.left, val)

else if val > root.key:

root.right = deleteNode(root.right, val)

else:

// Node found, perform deletion

// Deletion logic goes here

1. When deleting a node in an AVL tree, how many recursive calls are made to reach the target node?

a) One recursive call

b) Two recursive calls

c) Three recursive calls

**d) It depends on the tree's height**

1. Consider the following AVL tree deletion operation pseudo-code:

AVLDelete(Node\* root, int key) {

// Implementation details for deletion

return root;

}

What does the function return?

a) The deleted node value.

**b) The new root of the AVL tree.**

c) The height of the AVL tree after deletion.

d) The number of nodes in the AVL tree after deletion.

1. In an AVL tree, what is the maximum number of rotations required to restore balance after a deletion?

a) 1

**b) 2**

c) 3

d) 4

1. deleteNode(root, key)

if root is NULL

return NULL

if key < root->data

root->left = deleteNode(root->left, key)

else if key > root->data

root->right = deleteNode(root->right, key)

else

// Deletion logic goes here

// AVL balancing code goes here

return root

What should be done in the "// Deletion logic goes here" section to delete a node with the given key?

a) Delete the node and replace it with its left child

b) Delete the node and replace it with its right child

**c) Delete the node and replace it with its in-order successor**

d) Delete the node and replace it with its parent

1. What happens to the height of an AVL tree after a successful deletion?

a) Always increases

b) Always decreases

**c) May increase or decrease**

d) Remains the same

18. deleteNode(root, key)

if root is NULL

return NULL

if key < root->data

root->left = deleteNode(root->left, key)

else if key > root->data

root->right = deleteNode(root->right, key)

else

// Deletion logic goes here

// AVL balancing code goes here

return root

What will happen if the "key" to be deleted is not found in the AVL tree?

**a) The tree remains unchanged.**

b) The tree will be restructured.

c) An error message will be displayed.

d) The tree will be converted into a binary search tree.

1. When deleting a node in an AVL tree, in which order are the rotations performed to restore balance?

**a) First rotate, then update height balance**

b) First update height balance, then rotate

c) Rotations are not performed during deletion

d) It depends on the node's value being deleted

1. Consider the following AVL tree deletion pseudo-code:

AVL\_Delete(root, key):

// Deletion logic here

return new\_root

What will be the value of 'new\_root' if the key is not found in the AVL tree?

a) NULL

b) 0

c) -1

**d) root**

1. What will be the output of the following code snippet?

Node\* root = newNode(4);

root->left = newNode(2);

root->right = newNode(6);

root->left->left = newNode(1);

root->left->right = newNode(3);

cout << root->left->right->data << endl;

1. 1
2. 2
3. **3**
4. 4
5. What will be the output of the following code snippet?

Node\* root = newNode(5);

root->left = newNode(3);

root->right = newNode(7);

root->left->left = newNode(2);

root->left->right = newNode(4);

inorderTraversal(root);

1. **2 3 4 5 7**
2. 2 3 4
3. 5 3 7
4. 5 3 2 4 7
5. What will be the output of the following code snippet?

int findHeight(Node\* root) {

if (root == NULL)

return -1;

int leftHeight = findHeight(root->left);

int rightHeight = findHeight(root->right);

return max(leftHeight, rightHeight) + 1;

}

int main() {

Node\* root = newNode(8);

root->left = newNode(3);

root->right = newNode(10);

root->left->left = newNode(1);

root->left->right = newNode(6);

root->left->right->left = newNode(4);

root->left->right->right = newNode(7);

cout << findHeight(root) << endl;

return 0;

}

1. 2
2. **3**
3. 4
4. 5