Day 17

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

**Task 1:**

Write Algo for AVL tree

1. Insert node like BST
2. Update heights while going up
3. For each node, check balance factor:

- If |balance factor| > 1:

- Perform required rotation

1. Return new root (after balancing)

Solution 👍

Task 1:

AVL search Algo

1 − Create a node

2 − Check if tree is empty

3 − If tree is empty, new node is root node.

4 − not empty, perform Binary Search Tree insertion operation and check balancing factor of the node in the tree.

5 − Suppose balancing factor > apply rotations on node and resume insertion from Step 4.

12.15 to 12.20

**Task 2:**

Write code for AVL tree

Hint: try to insert nodes

While inserting get the balance of the tree

Create 2 methods for left rotate and right rotate

Try to insert

Finally display

AVL code Task 2:

import java.util.\*;

class Node {

int key, height;

Node left, right;

Node (int d) {

key = d;

height = 1;

}

}

public class AVLTree {

Node root;

int height (Node N) {

if (N == null)

return 0;

return N.height;

}

int max (int a, int b) {

return (a > b) ? a : b;

}

Node rightRotate (Node y) {

Node x = y.left;

Node T2 = x.right;

x.right = y;

y.left = T2;

y.height = max (height (y.left), height (y.right)) + 1;

x.height = max (height (x.left), height (x.right)) + 1;

return x;

}

Node leftRotate (Node x) {

Node y = x.right;

Node T2 = y.left;

y.left = x;

x.right = T2;

x.height = max (height (x.left), height (x.right)) + 1;

y.height = max (height (y.left), height (y.right)) + 1;

return y;

}

int getBalance (Node N) {

if (N == null)

return 0;

return height (N.left) - height (N.right);

}

Node insert (Node node, int key) {

if (node == null)

return (new Node (key));

if (key < node.key)

node.left = insert (node.left, key);

else if (key > node.key)

node.right = insert (node.right, key);

else

return node;

node.height = 1 + max (height (node.left), height (node.right));

int balance = getBalance (node);

if (balance > 1 && key < node.left.key)

return rightRotate (node);

if (balance < -1 && key > node.right.key)

return leftRotate (node);

if (balance > 1 && key > node.left.key) {

node.left = leftRotate (node.left);

return rightRotate (node);

}

if (balance < -1 && key < node.right.key) {

node.right = rightRotate (node.right);

return leftRotate (node);

}

return node;

}

void printTree(Node root){

if (root == null)

return;

if (root != null) {

printTree(root.left);

System.out.print(root.key + " ");

printTree(root.left);

}

}

public static void main(String args[]) {

AVLTree tree = new AVLTree();

tree.root = tree.insert(tree.root, 5);

tree.root = tree.insert(tree.root, 15);

tree.root = tree.insert(tree.root, 44);

tree.root = tree.insert(tree.root, 35);

tree.root = tree.insert(tree.root, 65);

tree.root = tree.insert(tree.root, 78);

System.out.println("AVL Tree: ");

tree.printTree(tree.root);

}

}

**PROGRAM:**

**public class Task2 {**

**// AVL Tree Node class**

**class Node {**

**int key, height;**

**Node left, right;**

**Node(int key) {**

**this.key = key;**

**height = 1;**

**}**

**}**

**Node root;**

**// Function to get height of node**

**int height(Node node) {**

**if (node == null)**

**return 0;**

**return node.height;**

**}**

**// Function to get balance factor**

**int getBalance(Node node) {**

**if (node == null)**

**return 0;**

**return height(node.left) - height(node.right);**

**}**

**// Right rotate**

**Node rightRotate(Node y) {**

**Node x = y.left;**

**Node T2 = x.right;**

**// Perform rotation**

**x.right = y;**

**y.left = T2;**

**// Update heights**

**y.height = Math.*max*(height(y.left), height(y.right)) + 1;**

**x.height = Math.*max*(height(x.left), height(x.right)) + 1;**

**return x; // New root**

**}**

**// Left rotate**

**Node leftRotate(Node x) {**

**Node y = x.right;**

**Node T2 = y.left;**

**// Perform rotation**

**y.left = x;**

**x.right = T2;**

**// Update heights**

**x.height = Math.*max*(height(x.left), height(x.right)) + 1;**

**y.height = Math.*max*(height(y.left), height(y.right)) + 1;**

**return y; // New root**

**}**

**// Insert a node into AVL tree**

**Node insert(Node node, int key) {**

**// 1. Perform standard BST insert**

**if (node == null)**

**return new Node(key);**

**if (key < node.key)**

**node.left = insert(node.left, key);**

**else if (key > node.key)**

**node.right = insert(node.right, key);**

**else**

**return node; // No duplicates**

**// 2. Update height**

**node.height = 1 + Math.*max*(height(node.left), height(node.right));**

**// 3. Get balance**

**int balance = getBalance(node);**

**// 4. Apply rotations**

**// Left Left Case**

**if (balance > 1 && key < node.left.key)**

**return rightRotate(node);**

**// Right Right Case**

**if (balance < -1 && key > node.right.key)**

**return leftRotate(node);**

**// Left Right Case**

**if (balance > 1 && key > node.left.key) {**

**node.left = leftRotate(node.left);**

**return rightRotate(node);**

**}**

**// Right Left Case**

**if (balance < -1 && key < node.right.key) {**

**node.right = rightRotate(node.right);**

**return leftRotate(node);**

**}**

**return node; // unchanged**

**}**

**// Insert wrapper**

**public void insert(int key) {**

**root = insert(root, key);**

**}**

**// In-order traversal**

**void inOrder(Node node) {**

**if (node != null) {**

**inOrder(node.left);**

**System.*out*.print(node.key + " ");**

**inOrder(node.right);**

**}**

**}**

**// Display**

**public void display() {**

**System.*out*.print("In-order traversal: ");**

**inOrder(root);**

**System.*out*.println();**

**}**

**// Main method**

**public static void main(String[] args) {**

**Task2 tree = new Task2();**

**tree.insert(30);**

**tree.insert(20);**

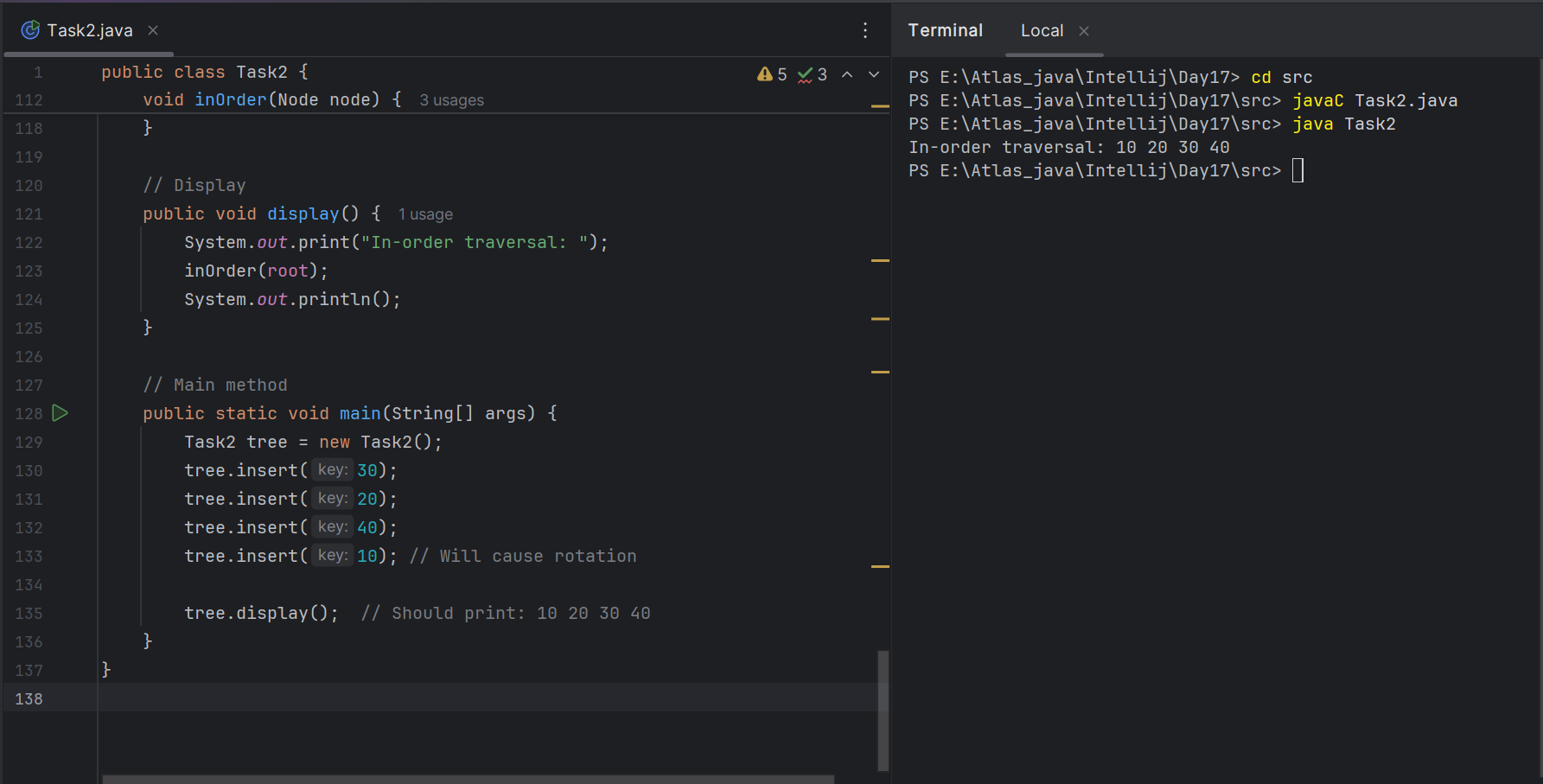
**tree.insert(40);**

**tree.insert(10); // Will cause rotation**

**tree.display(); // Should print: 10 20 30 40**

**}**

**}**

**OUTPUT:**

12.20 to 12.30

Task3:

Write algo for Read Black tree insertion

### **Insertion Algorithm**

**Step 1:** Insert like a normal Binary Search Tree

* Add the new node as you would in a normal BST
* Color the new node RED

**Step 2:** Fix Red-Black Tree properties  
 After insertion, check if any Red-Black Tree properties are violated. If yes:

* If the parent is BLACK, you're done.
* If the parent is RED, we have a violation (two REDs in a row). Handle based on uncle’s color:

**Case 1: Uncle is RED**

* Recolor: Parent and Uncle become BLACK
* Grandparent becomes RED
* Move up to grandparent and repeat fix

**Case 2: Uncle is BLACK or NULL**

* Perform rotation and recoloring based on node’s position:  
  + Left-Left → Right rotate
  + Right-Right → Left rotate
  + Left-Right or Right-Left → Double rotation (Left then Right or vice versa)

**Step 3:** Make the root BLACK

* Always ensure the root remains BLACK after fixes

**Task4:** write a pseudocode for insertion

Function Insert(value):

1. Create a new RED node with the given value

2. Insert the node like in a Binary Search Tree (BST)

3. Fix any Red-Black Tree property violations

Function FixInsert(node):

While node's parent is RED:

If parent is left child of grandparent:

Let uncle = right child of grandparent

If uncle is RED:

// Case 1 - Recoloring

parent.color = BLACK

uncle.color = BLACK

grandparent.color = RED

node = grandparent

Else:

If node is right child:

// Case 2 - Rotate left

node = parent

LeftRotate(node)

// Case 3 - Rotate right

parent.color = BLACK

grandparent.color = RED

RightRotate(grandparent)

Else (mirror image):

Let uncle = left child of grandparent

If uncle is RED:

// Case 1 - Recoloring

parent.color = BLACK

uncle.color = BLACK

grandparent.color = RED

node = grandparent

Else:

If node is left child:

// Case 2 - Rotate right

node = parent

RightRotate(node)

// Case 3 - Rotate left

parent.color = BLACK

grandparent.color = RED

LeftRotate(grandparent)

Set root.color = BLACK

Function LeftRotate(node):

// Standard left rotation logic

Function RightRotate(node):

// Standard right rotation logic

**tASK5:** Wap to insert an element in red black tree

**PROGRAM:**

**class Task5 {**

**static final int *RED* = 0;**

**static final int *BLACK* = 1;**

**class Node {**

**int data, color;**

**Node left, right, parent;**

**Node(int data) {**

**this.data = data;**

**color = *RED*; // New nodes are red by default**

**}**

**}**

**private Node root;**

**// Insert method**

**public void insert(int data) {**

**Node newNode = new Node(data);**

**root = bstInsert(root, newNode);**

**fixViolation(newNode);**

**}**

**// Standard BST insertion**

**private Node bstInsert(Node root, Node node) {**

**if (root == null)**

**return node;**

**if (node.data < root.data) {**

**root.left = bstInsert(root.left, node);**

**root.left.parent = root;**

**} else if (node.data > root.data) {**

**root.right = bstInsert(root.right, node);**

**root.right.parent = root;**

**}**

**return root;**

**}**

**// Fix Red-Black Tree properties**

**private void fixViolation(Node node) {**

**Node parent = null, grandparent = null;**

**while (node != root && node.color == *RED* && node.parent.color == *RED*) {**

**parent = node.parent;**

**grandparent = parent.parent;**

**// Parent is left child of grandparent**

**if (parent == grandparent.left) {**

**Node uncle = grandparent.right;**

**if (uncle != null && uncle.color == *RED*) {**

**// Case 1 - Recoloring**

**parent.color = *BLACK*;**

**uncle.color = *BLACK*;**

**grandparent.color = *RED*;**

**node = grandparent;**

**} else {**

**if (node == parent.right) {**

**// Case 2 - Left rotate**

**node = parent;**

**leftRotate(node);**

**}**

**// Case 3 - Right rotate**

**parent.color = *BLACK*;**

**grandparent.color = *RED*;**

**rightRotate(grandparent);**

**}**

**} else {**

**// Mirror case: Parent is right child**

**Node uncle = grandparent.left;**

**if (uncle != null && uncle.color == *RED*) {**

**// Case 1 - Recoloring**

**parent.color = *BLACK*;**

**uncle.color = *BLACK*;**

**grandparent.color = *RED*;**

**node = grandparent;**

**} else {**

**if (node == parent.left) {**

**// Case 2 - Right rotate**

**node = parent;**

**rightRotate(node);**

**}**

**// Case 3 - Left rotate**

**parent.color = *BLACK*;**

**grandparent.color = *RED*;**

**leftRotate(grandparent);**

**}**

**}**

**}**

**root.color = *BLACK*;**

**}**

**// Left rotation**

**private void leftRotate(Node node) {**

**Node rightChild = node.right;**

**node.right = rightChild.left;**

**if (rightChild.left != null)**

**rightChild.left.parent = node;**

**rightChild.parent = node.parent;**

**if (node.parent == null)**

**root = rightChild;**

**else if (node == node.parent.left)**

**node.parent.left = rightChild;**

**else**

**node.parent.right = rightChild;**

**rightChild.left = node;**

**node.parent = rightChild;**

**}**

**// Right rotation**

**private void rightRotate(Node node) {**

**Node leftChild = node.left;**

**node.left = leftChild.right;**

**if (leftChild.right != null)**

**leftChild.right.parent = node;**

**leftChild.parent = node.parent;**

**if (node.parent == null)**

**root = leftChild;**

**else if (node == node.parent.left)**

**node.parent.left = leftChild;**

**else**

**node.parent.right = leftChild;**

**leftChild.right = node;**

**node.parent = leftChild;**

**}**

**// In-order traversal for testing**

**public void inorder() {**

**inorderHelper(root);**

**System.*out*.println();**

**}**

**private void inorderHelper(Node root) {**

**if (root != null) {**

**inorderHelper(root.left);**

**System.*out*.print(root.data + "(" + (root.color == *RED* ? "R" : "B") + ") ");**

**inorderHelper(root.right);**

**}**

**}**

**// Main method for testing**

**public static void main(String[] args) {**

**Task5 tree = new Task5();**

**int[] values = {10, 20, 30, 15, 25, 5};**

**for (int val : values)**

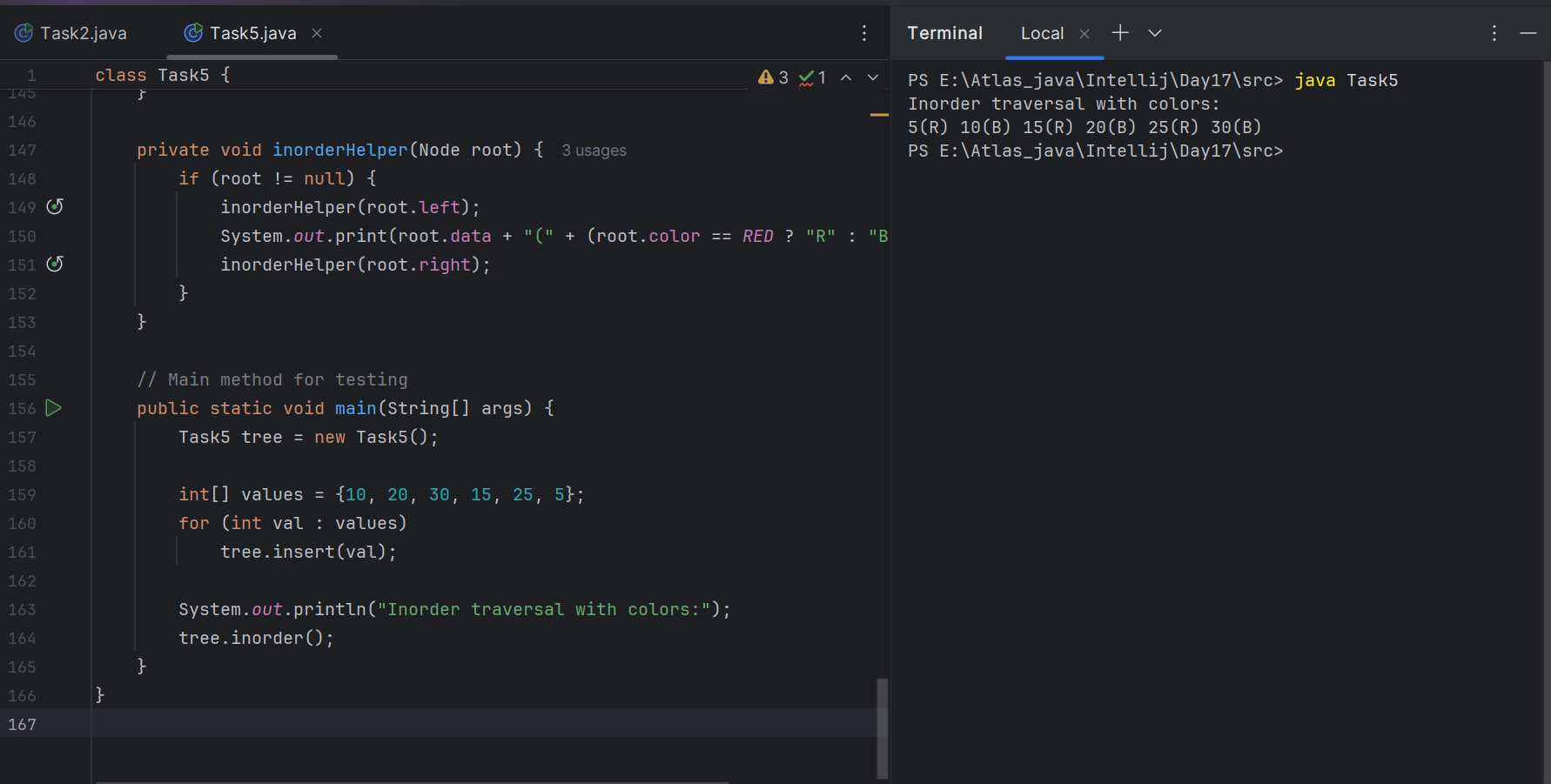
**tree.insert(val);**

**System.*out*.println("Inorder traversal with colors:");**

**tree.inorder();**

**}**

**}**

**OUTPUT:**===========================================================================================================

AVL qn:

Why are AVL trees considered balanced, and how does this impact performance?

1. They rearrange nodes to ensure maximum depth for fast access.
2. They allow unbalanced growth in left subtrees for quick insertion.
3. They maintain a balance factor (height difference) of at most 1 between subtrees to ensure O(log n) operations.
4. They replicate nodes for redundancy, enabling constant-time deletion.

What is the maximum height of an AVL tree with p nodes?  
a) p  
b) log(p)  
c) log(p)/2  
d) *p*⁄*2*

What maximum difference in heights between the leafs of a AVL tree is possible?  
a) log(n) where n is the number of nodes  
b) n where n is the number of nodes  
c) 0 or 1  
d) atmost 1 (as avl is self balanced)

Why to prefer red-black trees over AVL trees?  
a) Because red-black is more rigidly balanced  
b) AVL tree store balance factor in every node which costs space  
c) AVL tree fails at scale  
d) Red black is more efficient

in binary search tree balance what is the technique used?

class AVLTreeNode {

int val, height;

AVLTreeNode left, right;

AVLTreeNode(int v) {

val = v;

height = 1;

}

}

public class AVLTree {

public int getHeight(AVLTreeNode n) {

return (n == null) ? 0 : n.height;

}

public int getBalance(AVLTreeNode n) {

return (n == null) ? 0 : getHeight(n.left) - getHeight(n.right);

}

}

| Using balance factor to determine level-order traversal | Adding random nodes at different depths to flatten the tree | Checking balance factor to perform rotations when needed | Reversing subtree links on imbalance |
| --- | --- | --- | --- |

===========================================================================================================

DSA quiz 1:

dynamic programming optimize the Fibonacci computation. explain

public class DPFibonacci {

public int fib(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

int[] dp = new int[n + 1];

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

}

**Dynamic Programming reduces time from O(2ⁿ) to O(n)** by **caching previous results** (memoization / tabulation) and **avoiding repeated work**.

which sort and time complexity is applied in the below insertion sort code

public class Insertiont {

public void sort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

}

**Stable Sort**

Because it **preserves the relative order** of equal elements during sorting.

| **Case** | **Time Complexity** |
| --- | --- |
| **Best** | O(n) (when array is already sorted) |
| **Average** | O(n^2) (general case) |
| **Worst** | O(n^2) (when array is reverse sorted) |

**Stable sort**, **worst-case time complexity: O(n²)**

output of the reverse string function and its complexity is

public class ReverseString {

public String reverse(String input) {

char[] chars = input.toCharArray();

int left = 0, right = chars.length - 1;

while (left < right) {

char temp = chars[left];

chars[left] = chars[right];

chars[right] = temp;

left++;

right--;

}

return new String(chars);

}

public static void main(String[] args) {

ReverseString rs = new ReverseString();

System.out.println(rs.reverse("java"));

}

}

| Output: avaj, Time: O(n) due to single pass swap from ends | Output: vaaj, Time: O(n) with off-by-one index swap | Output: java, Time: O(n^2) due to nested loop |
| --- | --- | --- |

| primary purpose of using recursion in solving problems like tree traversal or factorial calculation is? | Recursion allows breaking a large problem into smaller sub-problems by calling the same function repeatedly with modified input. | Recursion introduces randomness which helps simulate complex decisions in algorithms. | Recursion eliminates the need for any stack or memory during runtime. | Recursion forces the problem to run in parallel threads, increasing computation speed. |
| --- | --- | --- | --- | --- |

| merge sort considered a stable sorting algorithm reason? | requires fewer recursive calls than unstable sorts like quicksort. | maintains the relative order of equal elements during the sorting process. | reduces time complexity by ignoring duplicate values. | rearranges all elements randomly to achieve faster execution. |
| --- | --- | --- | --- | --- |

| comparing hash tables and arrays, what is a significant functional difference regarding key access | Arrays dynamically resize to avoid collisions, while hash tables use fixed buckets. | Arrays support hashed key access, while hash tables require index-based retrieval. | Hash tables support binary searching, but arrays do not. | Hash tables map arbitrary keys to values using a hash function, whereas arrays only support integer index-based access. |
| --- | --- | --- | --- | --- |

| technique is commonly used to reduce repeated recursive calls in dynamic programming problems like Fibonacci calculation | Employing memoization to store and reuse previously computed results. | Applying nested recursion to simplify base case evaluation. | Transforming recursion into stack-based iteration for space optimization. | Using a binary search strategy to narrow down recursive branches. |
| --- | --- | --- | --- | --- |

| What is good hash function in a hash table implementation? | distribute input keys uniformly across the hash table to minimize clustering and reduce collision probability. | be complex enough to prevent reverse-engineering of keys. | produce sequential hash codes for predictable indexing and faster traversal. | generate a unique hash code for every possible key to avoid hash collisions completely. |
| --- | --- | --- | --- | --- |

dynamic programming optimize the Fibonacci computation. Explain

public class DPFibonacci {

public int fib(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

int[] dp = new int[n + 1];

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

}

| computes each Fibonacci number independently | reduces time from O(2^n) to O(n) by caching | uses recursion for faster computation | increases space complexity for no gain |
| --- | --- | --- | --- |

which sort and time complexity is applied in the below insertion sort code

public class Insertiont {

public void sort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

}