Day 17 - 10th July 2025

Name: Aravind Kasnagottu

Id:mvsnarav

Task 1:

Write Algo for AVL tree

1. **Start with an empty AVL tree.**
2. To **insert a new node**, insert it as we do in a normal BST.
3. After insertion, **update the height** of all ancestor nodes.
4. For each ancestor node, **find the balance factor**:

arduino

CopyEdit

balance = height of left subtree - height of right subtree

1. If the balance factor becomes > 1 or < -1, it means the tree is **unbalanced**, and we need to perform rotations.
2. **Check the type of imbalance:**
   * **Left-Left case:** (Node is inserted in the left child’s left subtree) ➔ Perform **Right Rotation**.
   * **Right-Right case:** (Node is inserted in the right child’s right subtree) ➔ Perform **Left Rotation**.
   * **Left-Right case:** (Node is inserted in the left child’s right subtree) ➔ Perform **Left Rotation on the left child**, then **Right Rotation** on the node.
   * **Right-Left case:** (Node is inserted in the right child’s left subtree) ➔ Perform **Right Rotation on the right child**, then **Left Rotation** on the node.
3. After rotation, the tree will be balanced again.
4. Repeat these steps for all insertions to keep the AVL tree balanced.

12.15 to 12.20

Task 2:

Write code for AVL tree

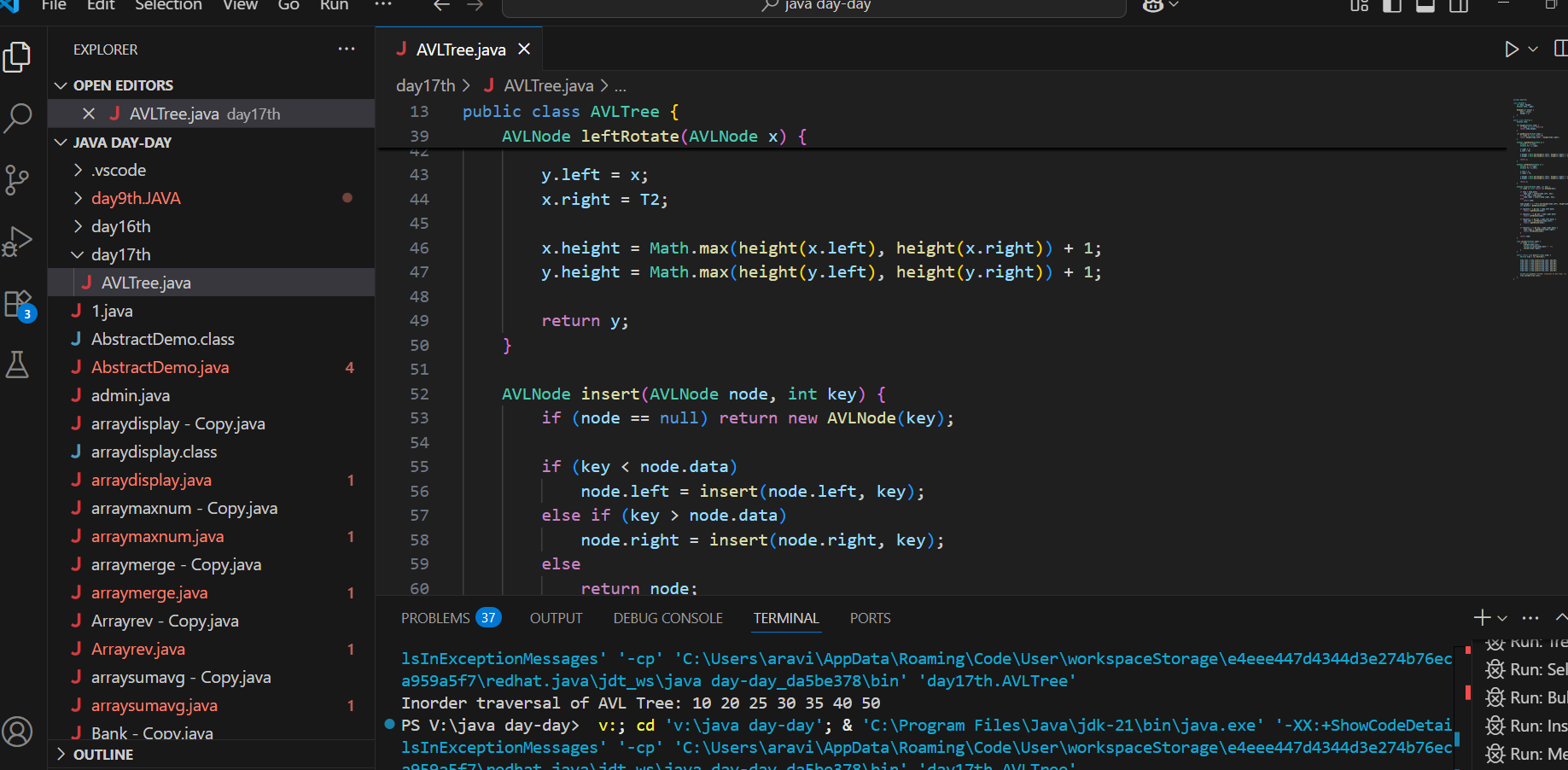
Hint: try to insert nodes

While inserting get the balance of the tree

Create 2 methods for left rotate and right rotate

Try to insert

Finally display



12.20 to 12.30

Task 3:

Write algo for Read Black tree insertion

**Start with a Red-Black Tree.**

**Insert the new node as in a normal BST** and **color the new node RED.**

After insertion, check if the tree still satisfies Red-Black properties:

* Every node is either red or black.
* The root is black.
* All leaves (NIL) are black.
* If a node is red, both its children are black.
* Every path from a node to its descendant NIL nodes has the same number of black nodes.

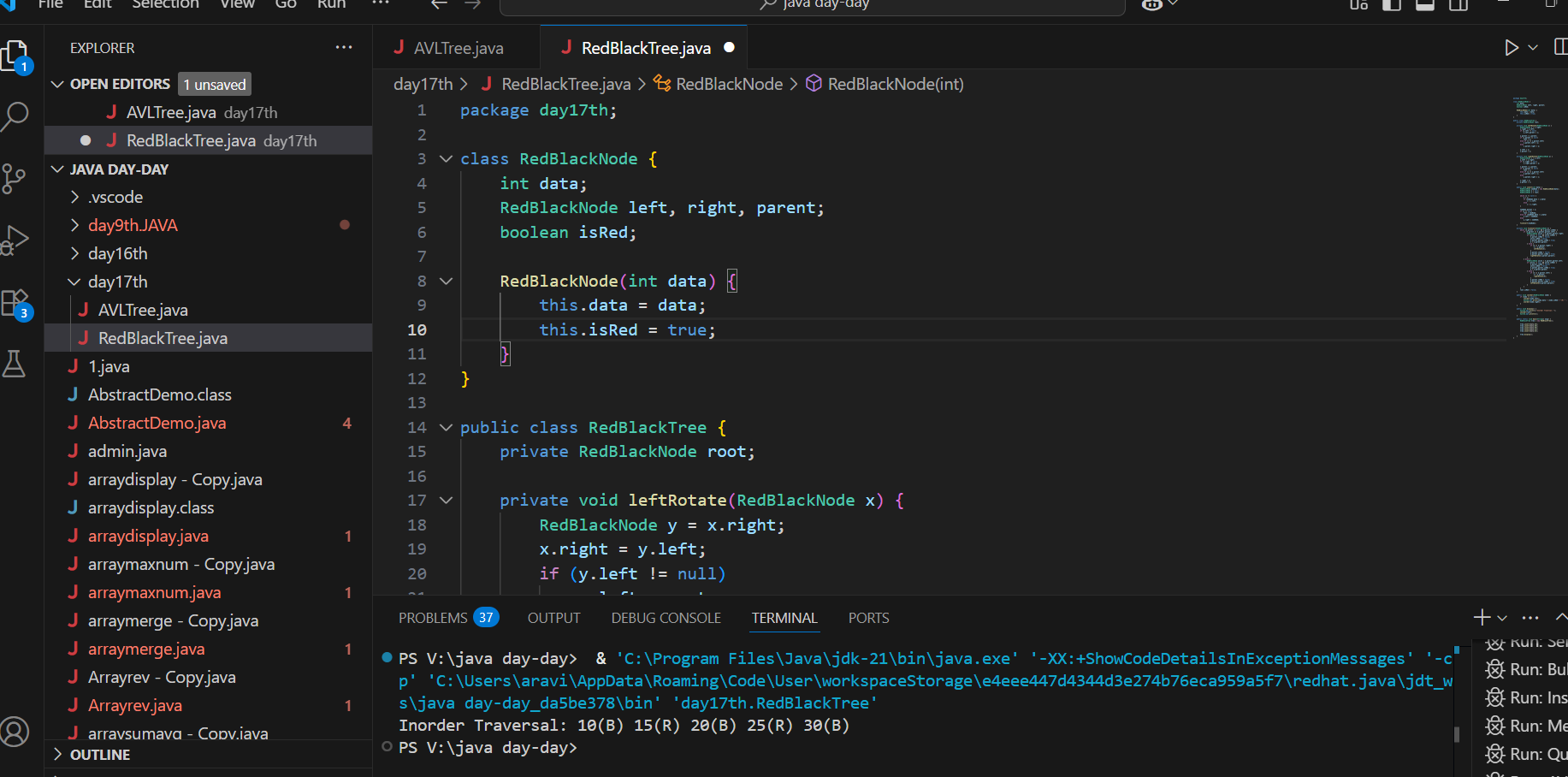
If the parent of the newly inserted node is **black**, insertion is complete (tree is still valid).

If the parent is **red**, there is a violation. Perform **fixing using these cases**:

* Perform **rotation on parent** to convert to outer case.

After all fixing, ensure that the **root is always B**

Task  4:Wap to insert an element in red black tree



5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child.

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

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);

    }

}

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Using balance factor to determine level-order traversal | 1. Adding random nodes at different depths to flatten the tree | 1. Checking balance factor to perform rotations when needed | 1. Reversing subtree links on imbalance |

DSA quiz 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| property of a priority queue differentiates it most from a regular queue implementation | It maintains a strict hierarchical structure using a self-balancing BST to enforce priority. | It allows insertion and removal only from one end, similar to a stack. | Elements are dequeued based on their priority, not their insertion order, often implemented using a binary heap. | Elements are removed based on their order of insertion rather than priority. |

Task 3:

Insert an Element - Red Black Tree −

1. Check tree is empty. If empty, then insert new node - color Black. (Because Root Node - Black in color)

2. else if Tree - not empty then insert new node as leaf node to the end and color - Red.

3. If parent of new node is Red and its neighbours(parent’s) node is also Red,

then Flip the color of the both neighbour and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbour only) i.e., Black.

4. If parent of new node is Red and its neighbours(parent’s) node is empty or NULL,

then Rotate (either Left-Left or Left-Right rotation) the new node and parent.

5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child

tell the main advantage of using a doubly linked list as shown here?

class Node {

    int data;

    Node prev, next;

    Node(int data) {

        this.data = data;

    }

}

public class DoublyList {

    Node head, tail;

     public void insertEnd(int data) {

        Node newNode = new Node(data);

        if (tail == null) {

             head = tail = newNode;

        } else {

             tail.next = newNode;

            newNode.prev = tail;

            tail = newNode;

        }

    }

}

|  |  |  |  |
| --- | --- | --- | --- |
| has simpler node deletion than singly list | allows forward traversal only using one pointer | avoids null pointers at boundaries | uses more space but enables two-way traversal |

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;

        }

    }

}

|  |  |  |  |
| --- | --- | --- | --- |
| Stable sort, w  orstcase O(n^2) | Stable sort, O(n log n) in all cases | Unstable sort with O(n^2) time for best case    14 correct out of 20 | Unstable sort with average O(n) time |

================================================================================================================================================I             Info Box

================================================================================================================================================I