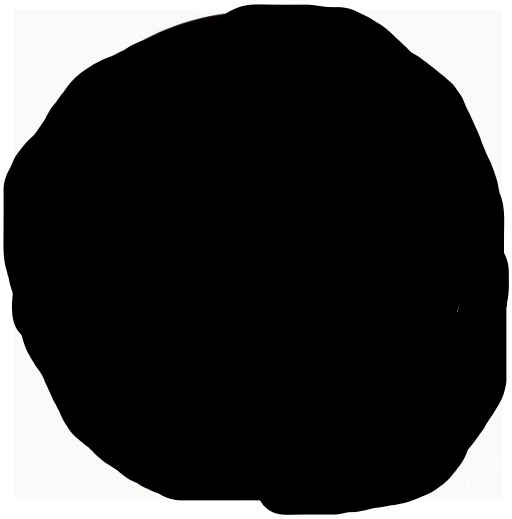


DATA STRUCTURE AND ALGORITHMS

What is an AVL Tree?

An AVL tree is a special kind of binary search tree (BST) that stays balanced. This balance helps it work faster when you search, add, or delete items. It's named after two people who invented it: Adelson-Velsky and Landis.

An AVL tree has a special rule: the height of the left and right branches of any node can only differ by 1. If this rule is broken when you add or remove something, the tree fixes itself using rotations. This balance helps the tree stay short, so operations like searching, adding, or deleting stay quick and efficient.

Why is it called a self-balancing binary search tree?

It is called self-balancing because the tree automatically adjusts its structure after every addition or removal of a node. If the tree becomes unbalanced, it uses rotations (single or double) to restore balance. This ensures that the tree remains efficient for operations like searching, adding, and deleting.

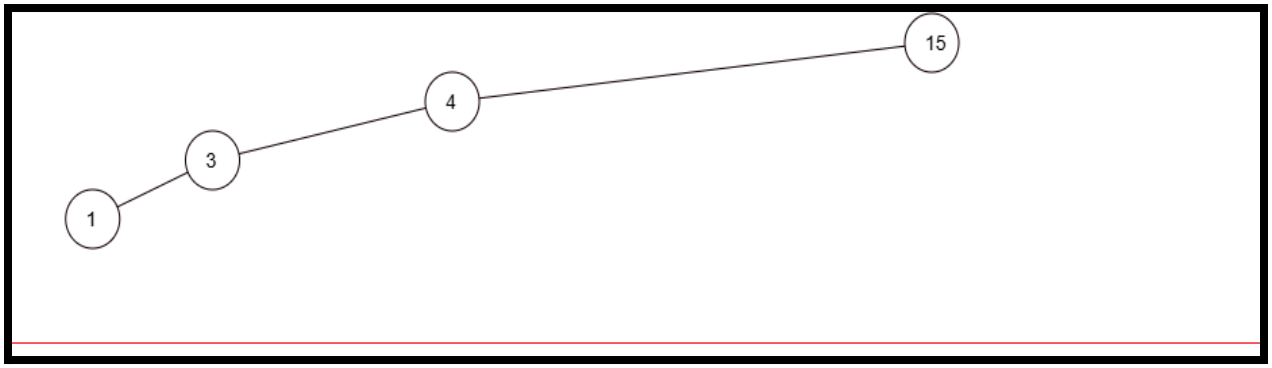
The Importance of Balancing in Binary Search Trees

Balancing is important because it prevents the tree from becoming too tall and skewed. If a binary search tree is unbalanced, operations like searching or inserting can take much longer, as their time depends on the tree's height. A balanced tree ensures operations remain fast and efficient, typically taking logarithmic time.

Key Differences Between AVL Trees and Standard Binary Search Trees (BST)

- **Balance Maintenance**: AVL trees always maintain a balance factor of -1, 0, or 1 for all nodes, while standard BSTs do not enforce any balance.
- **Efficiency**: AVL trees ensure faster performance in the worst case, while an unbalanced BST may degrade to a linked list structure, making operations slower.
- **Rotations**: AVL trees use rotations to maintain balance, whereas BSTs do not perform such operations.
- **Use Case**: AVL trees are preferred when frequent searching is needed, while standard BSTs are simpler and suitable for cases where balancing is less critical.

Illustrations: Draw diagrams to illustrate:



This is an unbalanced Binary search tree because In this BST, all nodes (,4,3,1) are on the **left side** of their parent nodes.The Right subtree of every node is empty, resulting in an unbalanced structure

This type of structure leads to performance issues, as operations may increase complexity similar to a linked list.

- The process of balancing it into an AVL Tree.

To transform the (BST) into an AVL tree, we need to perform rotations to ensure that the balance factor of each node is either -1, 0, or +1.

Step 1: Calculate Balance Factors

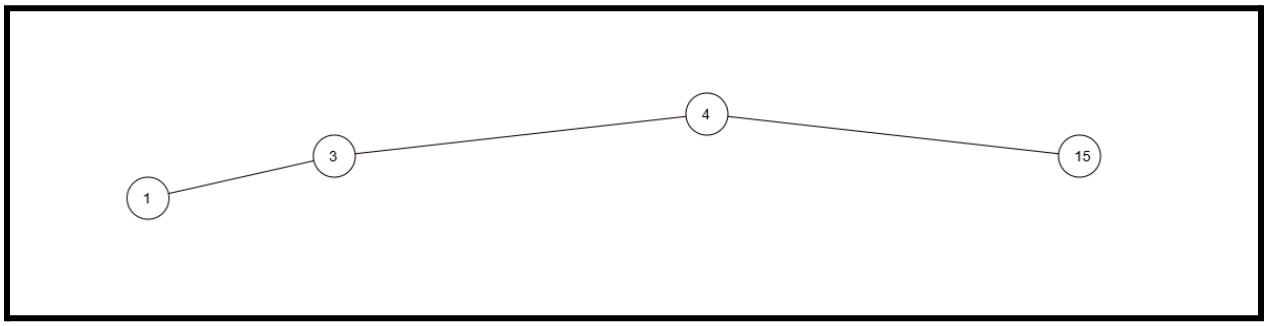
- For each node, calculate the balance factor (height of left subtree - height of right subtree).
- The balance factors for the nodes are:
    - Node 15: (left = 3) , (right =0) = 3
    - Node 4: (left = 2) , (right = 0) = 2
    - Node 3: (left = 1) , (right = 0) = 1
    - Node 1: (left = 0) , (right = 0) =0

Since the balance factor of node 15 is 3, which is greater than 1, we need to perform rotations to balance the tree

Step 2: Perform Rotations

In this case, we have a Left-Left (LL) imbalance at node 15. To fix this, we perform a right rotation around node 15.

Right Rotation around Node 15



Step 3: Recalculate Balance Factors

After the rotation, we recalculate the balance factors:

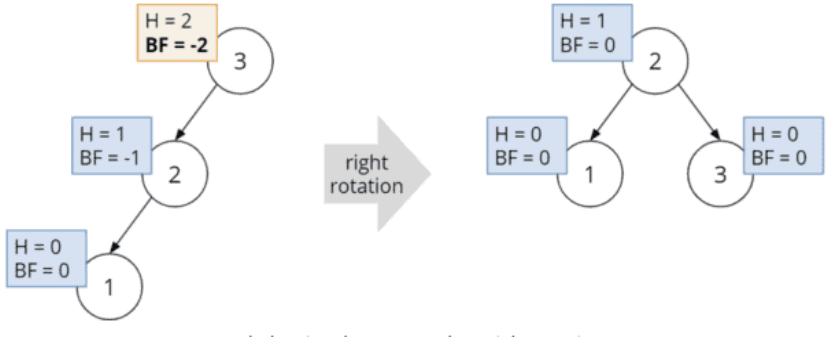
- Node 4: (left =1)   - (right = 1) = 0
- Node 3: (left = 1) - (right = 0) = 1
- Node 15:(left = 0) - (right = 0) = 0
- Node 1: (left = 0) - (right = 0) = 0

The final balanced AVL tree is: Balanced

-Rotations (single and double): Left, Right, Left-Right, and Right-Left.

There are four types of rotations: Single Left, Single Right, Left-Right, and Right-Left.

Right Rotation

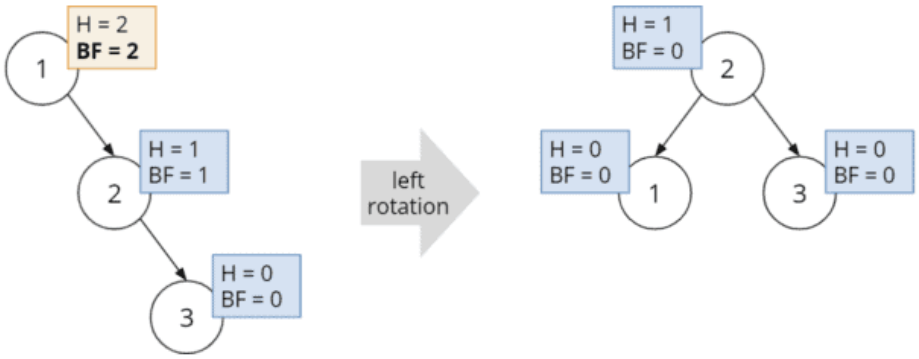


Where

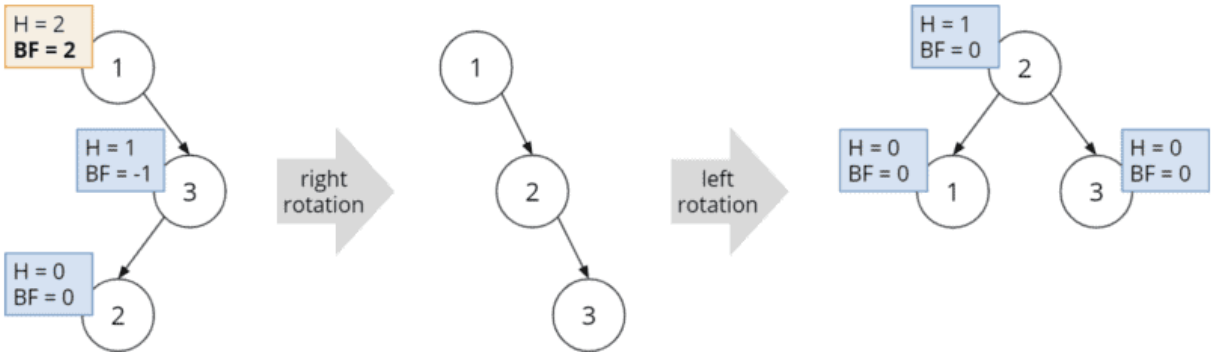
H (Height): Measures the depth of the node's subtree.

BF (Balance Factor): Indicates the balance of the node by comparing the heights of its left and right subtrees.

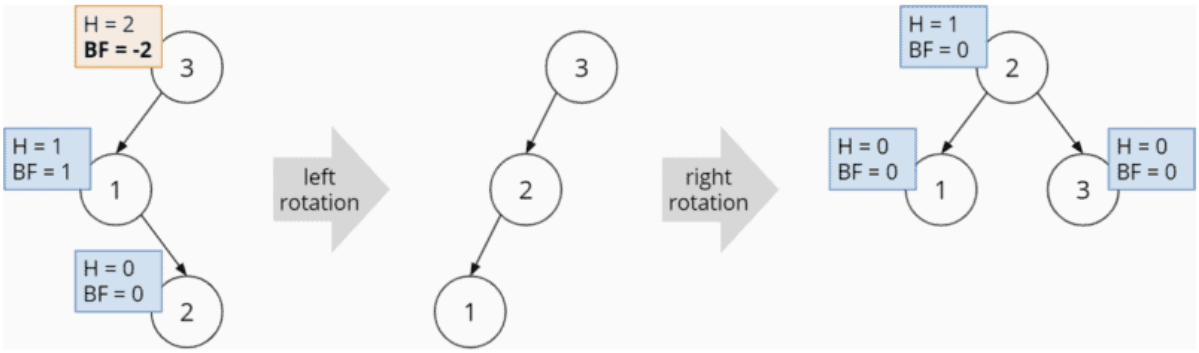
Left Rotation:



Right-Left Rotation:



Left-Right Rotation:



Here

H (Height): Measures the depth of the node's subtree.

BF (Balance Factor): Indicates the balance of the node by comparing the heights of its left and right subtrees.

Part 2: Core Implementation:

Task 1: Define the AVL Tree Node

Create a Java class for an AVL Tree node with fields for:

· - Data

· - Height

· - Left child

· - Right child

```java
class AVLTreeNode {
    int data;
    int height;
    AVLTreeNode lft;
    AVLTreeNode rght;

    public AVLTreeNode(int data) {
        this.data = data;
        this.height = 1;
    }
}
```

Store Data: The value of the node (e.g., numbers in a sorted order).

Enable Traversal: By referencing its children (left and right).

Support Tree Balancing: Through its `height` property, the balance factor of nodes can be calculated.

Task 2: Implement Basic Operations

Write methods for:

- Insertion:
- Rotation:
- Balance Factor

Insertion

```
class AVLTree {
    public AVLTreeNode root;

    public AVLTreeNode insert(AVLTreeNode node, int key) {
        if (node == null) return new AVLTreeNode(key);

        if (key < node.data) {
            node.lft = insert(node.lft, key);
        } else if (key > node.data) {
            node.rght = insert(node.rght, key);
        } else {
            return node;
        }

        node.height = 1 + Math.max(height(node.lft), height(node.rght));

        int balance = getBalance(node);

        if (balance > 1 && key < node.lft.data) {
            return rotateRight(node);
        }

        if (balance < -1 && key > node.rght.data) {
            return rotateLeft(node);
        }

        if (balance > 1 && key > node.lft.data) {
            node.lft = rotateLeft(node.lft);
            return rotateRight(node);
        }

        if (balance < -1 && key < node.rght.data) {
            node.rght = rotateRight(node.rght);
            return rotateLeft(node);
        }

        return node;
    }
```

Base Case

- If the current node is `null`, a new node is created and given key is assigned and returned.

Insert the Key

- If the key is smaller than the current node the key is inserted into the **left subtree**.
- If the key is larger than it is inserted into the **right subtree**.

Perform Rotations

- If the balance factor indicates that the tree is unbalanced, appropriate rotations are performed:
    - **Left-L Case:** Rotate the node **right**.
    - **Right-R Case:** Rotate the node **left**.
    - **Left-R Case:** Rotate the left child **left**, then the node **right**.
    - **Right-L Case:** Rotate the right child **right**, then the node **left**.

Return Updated Node

- After balancing, the updated node is returned

Rotation:

- Left rotation
- Right Rotation

In this code, the functions `rotateRight` and `rotateLeft` perform tree rotations to the maintain balance.

```
public AVLTreeNode rotateRight(AVLTreeNode y) {
    AVLTreeNode x = y.lft;
    AVLTreeNode usm = x.rght;

    x.rght = y;
    y.lft = usm;

    y.height = Math.max(height(y.lft), height(y.rght)) + 1;
    x.height = Math.max(height(x.lft), height(x.rght)) + 1;

    return x;
}

public AVLTreeNode rotateLeft(AVLTreeNode x) {
    AVLTreeNode y = x.rght;
    AVLTreeNode usm = y.lft;

    y.lft = x;
    x.rght = usm;

    x.height = Math.max(height(x.lft), height(x.rght)) + 1;
    y.height = Math.max(height(y.lft), height(y.rght)) + 1;

    return y;
}
```

- **Right Rotation**
  - **Steps**:
    1. x becomes the new root, which is the left child of y.
    2. `usm` temporarily stores the right child of x, ensuring no subtree is lost.
    3. The right child of x becomes y, and y's left child becomes `usm` (the subtree stored in `usm`).
    4. Recalculate the heights of x and y.
- **Left Rotation** .
  - **Steps**:
    1. y becomes the new root, which is the right child of x.
    2. `usm` temporarily stores the left child of y.
    3. The left child of y becomes x, and x's right child becomes `usm`.
    4. Recalculate the heights of x and y.

For Balancing

```
int height(AVLTreeNode node) {
    return node == null ? 0 : node.height;
}

int getBalance(AVLTreeNode node) {
    return node == null ? 0 : height(node.lft) - height(node.rght);
}

public void insert(int key) {
    root = insert(root, key);
}
```

`height(AVLTreeNode node)`:
This method returns the height of a given node. If `null`, it returns 0
`getBalance(AVLTreeNode node)`:
This method calculates the balance factor of a node, which is the difference between the height of the left and right subtrees. If `null`, it will also return `0`.

`insert(int key)`:
This is a public method that inserts a key into the AVL tree

Task 3: Additional Operations:

- Implement the following operations in the AVL Tree:
- **Search**
- **Deletion:**
- **Traversal:**

Implement Inorder, Preorder, and Postorder

Searching

```
public boolean search(AVLTreeNode node, int key) {
    if (node == null) return false;

    if (key < node.data) return search(node.lft, key);
    else if (key > node.data) return search(node.rght, key);
    else return true;
}

public boolean search(int key) {
    return search(root, key);
}
```

`Search(AVLTreeNode node, int key)`:

- If the current node is `null`, it means the key is not found, so it returns `false`.
- If the given key is smaller than the node's data, it searches the left child.
- If the key is larger than the node's data, it recursively searches for the right child.
- If the key matches the node's data, it returns `true`.

`search(int key)`:

- This method simplifies the call to the recursive search by passing the root node to method (`search(root, key)`). It returns the result of the search.

Deletion:

```
public AVLTreeNode delete(AVLTreeNode node, int key) {
    if (node == null) return node;

    if (key < node.data) {
        node.lft = delete(node.lft, key);
    } else if (key > node.data) {
        node.rght = delete(node.rght, key);
    } else {
        if ((node.lft == null) || (node.rght == null)) {
            AVLTreeNode temp = (node.lft != null) ? node.lft : node.rght;

            if (temp == null) {
                return null;
            } else {
                node = temp;
            }
        } else {
            AVLTreeNode temp = getMinValueNode(node.rght);
            node.data = temp.data;
            node.rght = delete(node.rght, temp.data);
        }
    }
```

`delete(AVLTreeNode node, int key)`:

This method deletes a node with the given key from the tree.

- Traversing the tree to find a given node.
- If the node has one or no child, it is replaced by its child.
- If a node has two children, it finds the node with the min value in the right subtree, replaces the current node's value with that, and then deletes the node with the min value.

```java
public void delete(int key) {
    root = delete(root, key);
}

public AVLTreeNode getMinValueNode(AVLTreeNode node) {
    AVLTreeNode current = node;
    while (current.lft != null) {
        current = current.lft;
    }
    return current;
}
```

delete(int key):

This method initiates the deletion of a node with the specified `key` by calling the `delete` function on the tree's root.

getMinValueNode(AVLTreeNode node):

This method finds and returns the node with the min value in the given subtree by continuously traversing the left child of each node

This part of the code ensures the AVL tree remains balanced after any insertion or deletion:

```
    node.height = 1 + Math.max(height(node.lft), height(node.rght));

    int balance = getBalance(node);

    if (balance > 1 && getBalance(node.lft) >= 0) {
        return rotateRight(node);
    }
    if (balance > 1 && getBalance(node.lft) < 0) {
        node.lft = rotateLeft(node.lft);
        return rotateRight(node);
    }
    if (balance < -1 && getBalance(node.rght) <= 0) {
        return rotateLeft(node);
    }
    if (balance < -1 && getBalance(node.rght) > 0) {
        node.rght = rotateRight(node.rght);
        return rotateLeft(node);
    }

    return node;
}
```

- **Recalculating height**:
  The height of the current node is updated based on the heights of its left and right children.
- **Balance factor**:
  The balance factor of the node is calculated (left subtree height - right subtree height). If the balance factor is not within the range of -1 to 1, rotations are needed to restore balance.
- **Rotations**:
  - **Right rotation**: If the left subtree is too tall (balance > 1), and the left child is balanced or left-heavy, a right rotation is performed to balance it.
  - **Left rotation**: If the right subtree is too tall (balance < -1), and the right child is balanced or right-heavy, a left rotation is performed.

- **Left-right rotation**: If the left subtree is too tall (balance > 1) and the left child is right-heavy, a left rotation on the left child is performed followed by a right rotation.
- **Right-left rotation**: If the right subtree is too tall (balance < -1) and the right child is left-heavy, a right rotation on the right child is followed by a left rotation.

Traversal:

Implement Inorder, Preorder, and Postorder

```
public AVLTreeNode getMinValueNode(AVLTreeNode node) {
    AVLTreeNode current = node;
    while (current.lft != null) {
        current = current.lft;
    }
    return current;
}

public void inOrder(AVLTreeNode node) {
    if (node != null) {
        inOrder(node.lft);
        System.out.print(node.data + " ");
        inOrder(node.rght);
    }
}

public void preOrder(AVLTreeNode node) {
    if (node != null) {
        System.out.print(node.data + " ");
        preOrder(node.lft);
        preOrder(node.rght);
    }
}

public void postOrder(AVLTreeNode node) {
    if (node != null) {
        postOrder(node.lft);
        postOrder(node.rght);
        System.out.print(node.data + " ");
    }
}
```

- **inOrder(AVLTreeNode node)**:
  This method performs an in-order traversal of the tree (left subtree, root, right subtree) and prints the node's data.
- **preOrder(AVLTreeNode node)**:
  This method performs a pre-order traversal (root, left subtree, right subtree) and prints the node's data .
- **postOrder(AVLTreeNode node)**:
  This method performs a post-order traversal (left subtree, right subtree, root) and prints the node's data.

```
public void dsplTrav() {
    System.out.println("\n In-order traversal:");
    inOrder(root);
    System.out.println("\n Pre-order traversal:");
    preOrder(root);
    System.out.println("\n Post-order traversal:");
    postOrder(root);
}
```

`dsplTrav()` method displays the results of three different tree traversals:

- **In-order traversal**: Prints nodes in ascending order (left, root, right).
- **Pre-order traversal**: Prints nodes in the order (root, left, right).
- **Post-order traversal**: Prints nodes after visiting both left and right subtrees (left, right, root).

Task 4: Testing

- Create a driver program to:
- Insert a series of elements (provided by the user or predefined). ·
- Display the tree structure after each insertion.
- Demonstrate the balancing process with rotation logs.
- Test and validate search and deletion operations.

```
public class Main {
    public static void main(String[] args) {
        AVLTree tree = new AVLTree();

         tree.insert(8);                System.out.println("Inserted " + 8);
        tree.insert(3);                 System.out.println("Inserted " +  3);
        tree.insert(9);                 System.out.println("Inserted " +  9);
        tree.insert(2);                 System.out.println("Inserted " +  2);
        tree.insert(5);                 System.out.println("Inserted " +  5);
        tree.insert(7);                 System.out.println("Inserted " +  7);
        tree.insert(4);                 System.out.println("Inserted " +  4);
         tree.dsplTrav();



        System.out.println("Searching for 5: " + tree.search(5));

        tree.delete(5);
        System.out.println("Deleted 5");
        tree.dsplTrav();
    }
}
```

In this `Main` class:

1. **Inserting values**: The `insert()` method is used to add nodes (values 8, 3, 9, 2, 5, 7, and 4) to the AVL tree. After each insertion, a message is printed showing the inserted value.
2. **Displaying traversals**: The `dsplTrav()` method is called to display the tree's in-order, pre-order, and post-order traversals after all insertions.
3. **Searching for value 5**: The `search()` method is called to check if the value 5 is present in the tree, and the result is printed.
4. **Deleting value 5**: The `delete()` method removes the node with value 5 from the tree, and the tree's structure is displayed again after the deletion.

Part 3: Analysis and Reflection:

What challenges did you face during the implementation? How did you overcome them?

1. **Understanding AVL Tree Properties:**

Understanding the AVL tree properties and concept was challenging This was overcome by studying the theory of AVL trees, reviewing various resources and examples

2. **Handling Complex Rotations and Balancing Conditions**

The four possible cases of imbalance in AVL trees (left-left, left-right, right-left, right-right) were difficult to implement correctly without causing errors. I overcomes this challenge by reading theoretical properties and implementing each one individually, testing thoroughly after each addition

These two are the hardest challenges faced during the implementation

Applications:
Research and write about real-world applications of AVL Trees.

Applications of AVL Trees

1. **E-commerce Platforms**: AVL trees can be utilized to organize and manage product catalogs, keeping them sorted by attributes like price or category for efficient retrieval.
2. **Library Management Systems**: Useful for arranging books in order based on authors' names, titles, or publication dates.
3. **Social Media Platforms**: Helpful in sorting and displaying trending posts or user feeds in an organized manner.
4. **Event Scheduling Tools**: Ideal for managing events on a timeline, enabling quick access to upcoming or past events.
5. **Financial Systems**: Used to arrange transaction records by date or transaction amount for streamlined analysis.
6. **Medical Databases**: Maintains an ordered list of patient information, such as medical history or appointment schedules, for efficient access.
7. **Gaming Resource Management**: Tracks and organizes in-game assets or resources for optimal usage during gameplay.
8. **Personal Task Management**: Helps in sorting and retrieving tasks based on priorities or deadlines.

9. **Survey Analytics**: Organizes survey data in a structured manner, sorted by response time or user demographics.
10. **Log Analysis**: Simplifies the sorting of log files, making error tracking or auditing processes faster.
11. **Weather Record Systems**: Stores historical weather data in a sorted format for easy reference and trend analysis.
12. **Network Device Management**: Keeps an ordered list of network devices, along with their statuses, to ensure smooth monitoring.
13. **Content Management Systems (CMS)**: Organizes articles or posts by publication date or popularity for better user engagement.
14. **Inventory Control Systems**: Helps in managing stock by sorting items based on quantity, category, or storage location.
15. **Data Visualization Tools**: Facilitates the graphical representation of sorted data, making it easier to analyze and understand patterns.

Learnings:

Summarize your key takeaways from this assignment.

- **Understanding AVL Trees:**

Through this assignment, I gained insights into how AVL trees maintain balance automatically after insertions or deletions by using rotations. I also learned the significance of the balance factor in keeping operations efficient and the tree well-structured.

- **Rotations and Structural Adjustments:**

Implementing and testing the four types of rotations (LL, LR, RL, RR) enhanced my understanding of how tree structures are reorganized to achieve balance after updates.

- **Recursive Problem-Solving:**

The use of recursive methods for tasks such as insertion, deletion, and traversals helped me develop better approaches to structuring recursive functions while addressing challenges like null nodes and edge cases.

References:

https://www.happycoders.eu/algorithms/avl-tree-java/

https://softwareengineering.stackexchange.com/questions/89117/avl-trees-and-the-real-world

https://www.geeksforgeeks.org/insertion-in-an-avl-tree/

https://www.javatpoint.com/avl-tree-applications

https://blog.heycoach.in/in-order-traversal-of-an-avl-tree/#:~:text=E%2Dcommerce%20Platforms%3A%20Use%20AVL,Management%20Systems%3A%20Sort%20books

https://www.happycoders.eu/algorithms/avl-tree-java/