Project (2): -

Marmara University - Faulty of Engineering

Department of Computer Engineering

Data Structures (Autumn 2022)

Submit Date: 30/12/2022

Name : Abdelrahman Zahran

Student ID: 150120998

AVL-Tree Vs Splay Tree

Sections Of the Report: -

- Section (1): Problem Definition.
- Section (2): Implementation Details.
- Section (3): Test Cases.

AVL Trees: -

Definition: -

AVL trees are self-balancing binary search trees. They are named after their inventors, G.M. Adelson-Velsky and E.M. Landis, who published them in their 1962 paper "An Algorithm for the Organization of Information."

In an AVL tree, the height of the left and right sub-trees of any node differs by at most 1. This balance property allows AVL trees to have faster search, insert, and delete operations than other self-balancing binary search trees, such as red-black trees.

To maintain the balance property, AVL trees perform rotations, when necessary, after inserting or deleting a node. Rotations are used to move nodes around in the tree to maintain the balance property. There are two types of rotations in AVL trees: left rotations and right rotations.

Overall, AVL trees are a type of data structure that can be used to efficiently store and retrieve data in a sorted manner.

Insertion: -

Inserting a node into an AVL tree follows a similar process to inserting a node into a regular binary search tree. The new node is added as a leaf in the tree, and its value is compared to the values of the nodes it is compared to determine its position in the tree.

However, after inserting a node into an AVL tree, the balance property of the tree may be violated. If this happens, rotations are performed to restore the balance of the tree.

Here is the process for inserting a node into an AVL tree:

1. Find the correct position for the new node in the tree, as if you were inserting it into a regular binary search tree.
2. Insert the new node into the tree.
3. Check the balance of the tree starting at the node where the new node was inserted. If the balance is violated, perform the necessary rotations to restore the balance.
4. Repeat the process for each node on the path from the inserted node to the root of the tree.

Search: -

Searching for a value in an AVL tree is like searching for a value in a regular binary search tree. The value is compared to the value at the root of the tree, and if it is less than the root value, the search continues in the left subtree; if it is greater than the root value, the search continues in the right subtree. This process is repeated until the value is found or it is determined that the value is not in the tree.

Here is the process for searching for a value in an AVL tree:

1. Compare the value to be searched for to the value at the root of the tree.
2. If the value is found at the root, the search is complete.
3. If the value is less than the value at the root, search for the value in the left subtree.
4. If the value is greater than the value at the root, search for the value in the right subtree.
5. Repeat the process until the value is found or it is determined that the value is not in the tree.

Deletion: -

Deleting a node from an AVL tree involves a similar process to deleting a node from a regular binary search tree. However, after deleting a node from an AVL tree, the balance property of the tree may be violated. If this happens, rotations are performed to restore the balance of the tree.

Here is the process for deleting a node from an AVL tree:

1. Find the node to be deleted in the tree.
2. If the node has no children, simply remove it from the tree.
3. If the node has one child, remove the node and replace it with its child.
4. If the node has two children, find the successor of the node (the next larger value in the tree) and replace the node with its successor. Then delete the successor from its original position in the tree.
5. Check the balance of the tree starting at the node where the node was deleted. If the balance is violated, perform the necessary rotations to restore the balance.
6. Repeat the process for each node on the path from the deleted node to the root of the tree.

AVL - Tree code Explanation: -

```c
struct AVLTreeNode *AVLTree_Insert(struct AVLTreeNode *root, int data) {
    // Allocate memory for the new node + initialize values.
    struct AVLTreeNode *temp = (struct AVLTreeNode *) malloc(sizeof(struct
AVLTreeNode));
    temp -> data = data;
    temp -> height = 1;
    temp -> frequency = 1;
    temp -> left = NULL;
    temp -> right = NULL;

    // If the tree is empty -> make the new node the root
    if (root == NULL) {
        root = temp;
        return root;
        // If the data is less than the root node's data, insert it in the left
subtree
    } else if (temp -> data < root -> data) {
        AVLTree_NumberOfComparisons++;
        root -> left = AVLTree_Insert(root -> left, data);
        // If the data is greater than the root node's data, insert it in the right
subtree
    } else if (temp -> data > root -> data) {
        AVLTree_NumberOfComparisons++;
        root -> right = AVLTree_Insert(root -> right, data);
        // If the data is equal to the root node's data, increment the frequency of
the root node
    } else if (root -> data == temp -> data) {
        root -> frequency++;
        AVLTree_NumberOfComparisons++;
        return root;
    }
    // Balance the tree if necessary and return the root node
    root = AVLTree_Balance(root, data);
    return root;
}
```

The algorithm that explains the code for inserting a node into an AVL tree:

1. Allocate memory for the new node and initialize its values. The new node's **data** is the value being inserted, **height** is set to 1, **frequency** is set to 1, and **left** and **right** are set to NULL.
2. If the tree is empty, make the new node the root of the tree and return it.

3. If the value being inserted is less than the root node's value, insert it in the left subtree by calling the **AVLTree_Insert** function recursively on the root's left child. Increment the **AVLTree_NumberOfComparisons** counter.

4. If the value being inserted is greater than the root node's value, insert it in the right subtree by calling the **AVLTree_Insert** function recursively on the root's right child. Increment the **AVLTree_NumberOfComparisons** counter.

5. If the value being inserted is equal to the root node's value, increment the root node's **frequency** and return the root node. Increment the **AVLTree_NumberOfComparisons** counter.

6. Balance the tree, if necessary, by calling the **AVLTree_Balance** function on the root node and the value being inserted. Return the root node.

An AVL tree is a self-balancing binary search tree, which means that the tree maintains a balance property to ensure that it remains relatively balanced and has good search performance. The balance property is achieved by ensuring that the difference in height between the left and right subtrees of any node is at most 1. When a node is inserted into an AVL tree, the tree may become unbalanced, and rotations may be needed to restore the balance property.

To insert a node into an AVL tree, you can follow these steps:

1. If the root node is NULL, create a new node with the given data and set it as the root node.
2. Otherwise, compare the data of the new node to the data of the root node. If the new data is less than the root data, insert the new node into the left subtree by recursively calling the insertion function with the left child of the root node as the root. If the new data is greater than the root data, insert the new node into the right subtree by recursively calling the insertion function with the right child of the root node as the root.
3. After the new node has been inserted, check the balance factor of the root node. If the balance factor is greater than 1, a violation has occurred on the left side of the tree, and a rotation is needed to restore the balance property. If the balance factor is less than -1, a violation has occurred on the right side of the tree, and a rotation is needed to restore the balance property.
4. If a rotation is needed, determine which type of rotation is needed based on the data of the new node and the data of the root node's children. There are four types of rotations that may be needed in an AVL tree: left-left, left-right, right-left, and right-right.

- If the new data is less than the data of the root node's left child and a violation has occurred on the left side of the tree, a left-left rotation is needed.
- If the new data is greater than the data of the root node's left child and a violation has occurred on the left side of the tree, a left-right rotation is needed.
- If the new data is less than the data of the root node's right child and a violation has occurred on the right side of the tree, a right-left rotation is needed.
- If the new data is greater than the data of the root node's right child and a violation has occurred on the right side of the tree, a right-right rotation is needed.

To perform a rotation, you can use one of the rotation functions provided in the code you posted: AVLTree_LeftRotation or AVLTree_RightRotation. These functions take a node as an argument and return the new root node after the rotation has been performed.

After the rotation has been performed, update the height of the root node and continue the insertion process by returning the root node.

It's important to note that the insertion process may need to be repeated multiple times to fully balance the tree. After each insertion, the balance factor of the root node should be checked, and rotations may be needed to restore the balance property.

```
struct AVLTreeNode *AVLTree_Balance(struct AVLTreeNode *node, int data) {
    // Get the left and right children of the node
    struct AVLTreeNode *left = node -> left;
    struct AVLTreeNode *right = node -> right;

    // Initialize variables for left and right child heights and balance factor
    int left_height = 0;
    int right_height = 0;
    int BalanceFactor = 0;

    // Get the heights of the left and right children
    left_height = AVLTree_Height(node -> left);
    right_height = AVLTree_Height(node -> right);

    // Update the height of the node
    node -> height = (left_height > right_height ? left_height : right_height) + 1;

    // Calculate the balance factor of the node
    BalanceFactor = AVLTree_BalanceFactor(node);

    // If the balance factor is greater than 1, there is a violation on the left side
    if (BalanceFactor > 1) {
        // If the data to be inserted is less than the left child's data, it is a
left-left imbalance
        if (data < left -> data) {
            node = AVLTreeViolation_Left_Left_Imbalance(node);
            // If the data to be inserted is greater than the left child's data, it
is a left-right imbalance
        } else if (data > left -> data) {
            node = AVLTreeViolation_Left_Right_Imbalance(node);
        }
    // If the balance factor is less than -1, there is a violation on the right side
    } else if (BalanceFactor < -1) {
        // If the data to be inserted is less than the right child's data, it is a
right-left imbalance
        if (data < right -> data) {
            node = AVLTreeViolation_Right_Left_Imbalance(node);
            // If the data to be inserted is greater than the right child's data, it
is a right-right imbalance
        } else if (data > right -> data) {
            node = AVLTreeViolation_Right_Right_Imbalance(node);
```

```
        }
        // If the balance factor is within the acceptable range, return the node
    } else {
        return node;
    }
}
```

The algorithm that explains the code for balancing operation of the inserted node into an AVL tree:

1. Initialize variables for the left child, right child, left child height, right child height, and balance factor of the node.
2. Set the value of the left child to be the left child of the node and the value of the right child to be the right child of the node.
3. Set the value of the left child height to be the height of the left child and the value of the right child height to be the height of the right child.
4. Update the height of the node by taking the maximum of the left and right child heights and adding 1.
5. Calculate the balance factor of the node by subtracting the right child height from the left child height.
6. If the balance factor is greater than 1, there is a violation on the left side of the tree.
7. If the data to be inserted is less than the data of the left child, it is a left-left imbalance. In this case, call the AVLTreeViolation_Left_Left_Imbalance function to fix the imbalance.
8. If the data to be inserted is greater than the data of the left child, it is a left-right imbalance. In this case, call the AVLTreeViolation_Left_Right_Imbalance function to fix the imbalance.
9. If the balance factor is less than -1, there is a violation on the right side of the tree.
10. If the data to be inserted is less than the data of the right child, it is a right-left imbalance. In this case, call the AVLTreeViolation_Right_Left_Imbalance function to fix the imbalance.
11. If the data to be inserted is greater than the data of the right child, it is a right-right imbalance. In this case, call the AVLTreeViolation_Right_Right_Imbalance function to fix the imbalance.
12. If the balance factor is within the acceptable range, return the node.

In an AVL tree, balance refers to the balance property, which states that for every node in the tree, the height of the left subtree and the height of the right subtree should differ by at most 1. This ensures that the tree remains balanced and can be searched efficiently.

To maintain balance in an AVL tree, you can use rotations to restructure the tree after an insertion or deletion. There are four types of rotations that may be needed in an AVL tree: left-left, left-right, right-left, and right-right.

- If the new data is less than the data of the root node's left child and a violation has occurred on the left side of the tree, a left-left rotation is needed.
- If the new data is greater than the data of the root node's left child and a violation has occurred on the left side of the tree, a left-right rotation is needed.
- If the new data is less than the data of the root node's right child and a violation has occurred on the right side of the tree, a right-left rotation is needed.

- If the new data is greater than the data of the root node's right child and a violation has occurred on the right side of the tree, a right-right rotation is needed.

To perform a rotation, you can use one of the rotation functions provided in the code you posted: AVLTree_LeftRotation or AVLTree_RightRotation. These functions take a node as an argument and return the new root node after the rotation has been performed.

It's important to note that the balance property may be violated multiple times during an insertion or deletion, and multiple rotations may be needed to restore balance to the tree. After each rotation, the balance factor of the root node should be checked, and additional rotations may be needed if the balance factor is not within the acceptable range.

```
struct AVLTreeNode *AVLTree_Search(struct AVLTreeNode *node, int data) {
    // If the node is NULL, return
    if (node == NULL) {
        return;
        // If the data to be searched for is less than the node's data, search in the
left subtree
    } else if (node -> data < data) {
        node = node -> left;
        AVLTree_Search(node, data);
        // If the data to be searched for is greater than the node's data, search in
the right subtree
    } else if (node -> data > data) {
        node = node -> right;
        AVLTree_Search(node, data);
        // If the data to be searched for is equal to the node's data, increment the
frequency of the node and return it
    } else if (node -> data == data) {
        node -> frequency++;
        printf("Data is found!");
        return node;
    }
}
```

The algorithm that explains the code for searching operation of a node into an AVL tree:

1. If the node is NULL, return.
2. If the data to be searched for is less than the data in the node, set the value of the node to be its left child and call the AVLTree_Search function again with the new node and the search data as arguments.
3. If the data to be searched for is greater than the data in the node, set the value of the node to be its right child and call the AVLTree_Search function again with the new node and the search data as arguments.
4. If the data to be searched for is equal to the data in the node, increment the frequency of the node and print a message indicating that the data was found. Return the node.

To search for a specific data value in an AVL tree, you can use a recursive function that traverses the tree until the desired value is found or it becomes clear that the value is not present in the tree.

To search for a value in an AVL tree, you can follow these steps:

1. If the root node is NULL, return NULL to indicate that the value is not present in the tree.
2. If the value to be searched for is less than the data of the root node, search for the value in the left subtree by recursively calling the search function with the left child of the root node as the root.
3. If the value to be searched for is greater than the data of the root node, search for the value in the right subtree by recursively calling the search function with the right child of the root node as the root.
4. If the value to be searched for is equal to the data of the root node, return the root node to indicate that the value has been found.

It's important to note that the search function should not modify the tree in any way. It should simply traverse the tree and return the node containing the desired value, if it is present, or NULL if the value is not present in the tree.

```c
struct AVLTreeNode *AVLTreeViolation_Left_Left_Imbalance(struct AVLTreeNode
*node) {
    // Perform a right rotation and update the number of rotations
    AVLTree_NumberOfRotations++;
    return AVLTree_RightRotation(node);
}

struct AVLTreeNode *AVLTreeViolation_Right_Right_Imbalance(struct AVLTreeNode *node)
{
    // Perform a left rotation and update the number of rotations
    AVLTree_NumberOfRotations++;
    return AVLTree_LeftRotation(node);
}

struct AVLTreeNode *AVLTreeViolation_Left_Right_Imbalance(struct AVLTreeNode *node) {
    // Perform a left rotation on the left child and a right rotation on the node,
updating the number of rotations each time
    node -> left = AVLTree_LeftRotation(node -> left);
    AVLTree_NumberOfRotations++;
    AVLTree_NumberOfRotations++;
    return AVLTree_RightRotation(node);
}

struct AVLTreeNode *AVLTreeViolation_Right_Left_Imbalance(struct AVLTreeNode *node) {
    // Perform a right rotation on the right child and a left rotation on the node,
updating the number of rotations each time
    node -> right = AVLTree_RightRotation(node -> right);
```

```
    AVLTree_NumberOfRotations++;
    AVLTree_NumberOfRotations++;
    return AVLTree_LeftRotation(node);
}
```

The algorithm that explains the code for unbalancing cases of a node in an AVL tree:

1. AVLTreeViolation_Left_Left_Imbalance: Perform a right rotation on the node and update the number of rotations. Return the result of the rotation.
2. AVLTreeViolation_Right_Right_Imbalance: Perform a left rotation on the node and update the number of rotations. Return the result of the rotation.
3. AVLTreeViolation_Left_Right_Imbalance: Perform a left rotation on the left child of the node and a right rotation on the node. Update the number of rotations each time. Return the result of the right rotation.
4. AVLTreeViolation_Right_Left_Imbalance: Perform a right rotation on the right child of the node and a left rotation on the node. Update the number of rotations each time. Return the result of the left rotation.

In an AVL tree, an imbalance occurs when the height of the left subtree and the height of the right subtree differ by more than 1. This can happen when a new data value is inserted into the tree, or when a data value is deleted from the tree.

There are four types of imbalances that can occur in an AVL tree: left-left, left-right, right-left, and right-right. These imbalances are determined based on the location of the imbalance in the tree and the relationship between the data values of the nodes involved.

- A left-left imbalance occurs when the new data is less than the data of the root node's left child and a violation has occurred on the left side of the tree.
- A left-right imbalance occurs when the new data is greater than the data of the root node's left child and a violation has occurred on the left side of the tree.
- A right-left imbalance occurs when the new data is less than the data of the root node's right child and a violation has occurred on the right side of the tree.
- A right-right imbalance occurs when the new data is greater than the data of the root node's right child and a violation has occurred on the right side of the tree.

To restore balance to the tree, you can use one of the violation functions provided in the code you posted: AVLTreeViolation_Left_Left_Imbalance, AVLTreeViolation_Left_Right_Imbalance, AVLTreeViolation_Right_Left_Imbalance, or AVLTreeViolation_Right_Right_Imbalance. These functions take a node as an argument and return the new root node after the necessary rotation has been performed to restore balance to the tree.

```
struct AVLTreeNode *AVLTree_LeftRotation(struct AVLTreeNode *node) {
    // Initialize variables for left and right child heights
    int left_height = 0;
    int right_height = 0;

    // Perform left rotation
    struct AVLTreeNode *temp1 = node -> right;
```

```
    struct AVLTreeNode *temp2 = temp1 -> left;

    temp1 -> left = node;
    node -> right = temp2;

    // Update the heights of the rotated nodes
    left_height = AVLTree_Height(node -> left);
    right_height = AVLTree_Height(node -> right);
    node -> height = (left_height > right_height ? left_height : right_height) + 1;

    left_height = AVLTree_Height(temp1 -> left);
    right_height = AVLTree_Height(temp1 -> right);
    temp1 -> height = (left_height > right_height ? left_height : right_height) + 1;

    return temp1;
}
```

The algorithm that explains the code for Left Rotation process of a node in an AVL tree:

1. Initialize variables for the left and right child heights.
2. Create a temporary variable called "temp1" and set it to be the right child of the node.
3. Create a temporary variable called "temp2" and set it to be the left child of temp1.
4. Set the left child of temp1 to be the node and the right child of the node to be temp2.
5. Update the height of the node by calculating the maximum of the left and right child heights and adding 1.
6. Update the height of temp1 by calculating the maximum of its left and right child heights and adding 1.
7. Return temp1.

A left rotation in an AVL tree is a rotational operation that is used to restore balance to the tree when an imbalance has occurred on the right side of the tree. It involves rotating the root node of the subtree to the left, and updating the heights of the rotated nodes.

To perform a left rotation, you can use the AVLTree_LeftRotation function provided in the code you posted. This function takes a node as an argument and returns the new root node after the rotation has been performed.

Here's how the left rotation works:

1. Initialize variables for the left and right child heights.
2. Create temporary pointers to the root node's right child (temp1) and the right child's left child (temp2).
3. Set temp1 as the new root node and set temp2 as the new right child of temp1.
4. Set the root node as the new left child of temp1.
5. Update the heights of the rotated nodes.
6. Return temp1 as the new root node.

After the left rotation, the structure of the tree will have changed, and the balance property will be restored. The left rotation is often used in conjunction with other rotational operations to restore balance to an AVL tree after an insertion or deletion.

```c
struct AVLTreeNode *AVLTree_RightRotation(struct AVLTreeNode *node) {
    // Initialize variables for left and right child heights
    int left_height = 0;
    int right_height = 0;

    // Perform right rotation
    struct AVLTreeNode *temp1 = node -> left;
    struct AVLTreeNode *temp2 = temp1 -> right;

    temp1 -> right = node;
    node -> left = temp2;

    // Update the heights of the rotated nodes
    left_height = AVLTree_Height(node -> left);
    right_height = AVLTree_Height(node -> right);
    node -> height = (left_height > right_height ? left_height : right_height) + 1;

    left_height = AVLTree_Height(temp1 -> left);
    right_height = AVLTree_Height(temp1 -> right);
    temp1 -> height = (left_height > right_height ? left_height : right_height) + 1;

    return temp1;
}
```

The algorithm that explains the code for Right Rotation process of a node in an AVL tree:

1. Initialize variables for the left and right child heights.
2. Create a temporary variable called "temp1" and set it to be the left child of the node.
3. Create a temporary variable called "temp2" and set it to be the right child of temp1.
4. Set the right child of temp1 to be the node and the left child of the node to be temp2.
5. Update the height of the node by calculating the maximum of the left and right child heights and adding 1.
6. Update the height of temp1 by calculating the maximum of its left and right child heights and adding 1.
7. Return temp1.

A right rotation in an AVL tree is a rotational operation that is used to restore balance to the tree when an imbalance has occurred on the left side of the tree. It involves rotating the root node of the subtree to the right and updating the heights of the rotated nodes.

To perform a right rotation, you can use the AVLTree_RightRotation function provided in the code you posted. This function takes a node as an argument and returns the new root node after the rotation has been performed.

Here's how the right rotation works:

1. Initialize variables for the left and right child heights.
2. Create temporary pointers to the root node's left child (temp1) and the left child's right child (temp2).
3. Set temp1 as the new root node and set temp2 as the new left child of temp1.
4. Set the root node as the new right child of temp1.
5. Update the heights of the rotated nodes.
6. Return temp1 as the new root node.

After the right rotation, the structure of the tree will have changed, and the balance property will be restored. The right rotation is often used in conjunction with other rotational operations to restore balance to an AVL tree after an insertion or deletion.

```
int AVLTree_Height(struct AVLTreeNode *node) {
    // If the node is NULL, return -1
    if (node == NULL) {
        return -1;
    } else {
        // Otherwise, return the height of the node by calculating the maximum height
of the left and right children and adding 1
        int left_height = AVLTree_Height(node -> left);
        int right_height = AVLTree_Height(node -> right);
        return (left_height > right_height ? left_height : right_height) + 1;
    }
}
```

The algorithm that explains the code for obtaining height of a node in an AVL tree:

1. If the node is NULL, return -1.
2. Otherwise, calculate the height of the left child by calling the AVLTree_Height function with the left child of the node as the argument.
3. Calculate the height of the right child by calling the AVLTree_Height function with the right child of the node as the argument.
4. Return the maximum of the left and right child heights, adding 1. This will give the height of the node.

In an AVL tree, the height of a node is the number of edges from the node to the deepest leaf in the tree. The height of a tree is the height of its root node. In other words, the height of a tree is the maximum depth of any node in the tree.

To calculate the height of a node in an AVL tree, you can use the AVLTree_Height function provided in the code you posted. This function takes a node as an argument and returns the height of the node.

Here's how the AVLTree_Height function works:

1. If the node is NULL, return -1.
2. Otherwise, calculate the maximum height of the left and right children of the node by calling the AVLTree_Height function recursively on each child.
3. Return the maximum of the left and right heights, plus 1 to account for the current node.

The height of a tree is an important factor in maintaining the balance property of an AVL tree. To ensure that the tree remains balanced, the difference in height between the left and right subtrees of any node should not exceed 1. This is why it's important to update the heights of nodes after insertions and deletions, as well as after rotations.

```
int AVLTree_BalanceFactor(struct AVLTreeNode *node) {
    // calculate the balance factor by subtracting the height of the right child from
the height of the left child
    int BalanceFactor = AVLTree_Height(node -> left) - AVLTree_Height(node -> right);
    // If the node is NULL, return 0
    if (node == NULL) {
        return 0;
    } else {
        return BalanceFactor;
    }
}
```

The algorithm that explains the code for obtaining balance factor of a node in an AVL tree:

1. Calculate the balance factor by calling the AVLTree_Height function with the left child of the node as the argument and subtracting the result from the result of calling the AVLTree_Height function with the right child of the node as the argument.
2. If the node is NULL, return 0.
3. Otherwise, return the balance factor.

The balance factor of a node in an AVL tree is a measure of the balance of the tree at that node. It is calculated as the difference between the heights of the left and right children of the node. If the balance factor of a node is greater than 1, there is an imbalance on the left side of the tree. If the balance factor is less than -1, there is an imbalance on the right side of the tree.

To calculate the balance factor of a node in an AVL tree, you can use the AVLTree_BalanceFactor function provided in the code you posted. This function takes a node as an argument and returns the balance factor of the node.

Here's how the AVLTree_BalanceFactor function works:

1. If the node is NULL, return 0.
2. Otherwise, calculate the balance factor by subtracting the height of the right child from the height of the left child.
3. Return the balance factor.

The balance factor of a node is an important factor in determining whether the tree is balanced or not. If the balance factor of a node exceeds the acceptable range of -1 to 1, it indicates that the tree is unbalanced at that node and needs to be rebalanced. This is often done through rotational operations, such as left rotations and right rotations.

```c
struct AVLTreeNode *AVLTree_Preorder_Traversal(struct AVLTreeNode *root) {
    // If the root node is NULL, return NULL
    if (root == NULL) {
        return NULL;
        // Otherwise, print the root node data, traverse the left subtree, and
traverse the right subtree
    } else {
        // Allocate memory for a new node
        struct AVLTreeNode *node = malloc(sizeof(struct AVLTreeNode));
        // Print the root node data
        printf("%d ", root -> data);
        // Set the data of the new node to the root node data
        node -> data = root -> data;
        // Traverse the left subtree
        node -> left = AVLTree_Preorder_Traversal(root -> left);
        // Traverse the right subtree
        node -> right = AVLTree_Preorder_Traversal(root -> right);
        return node;
    }
}
```

The algorithm that explains the code for the process of pre-order traversal of an AVL tree:

1. If the root node is NULL, return NULL.
2. Otherwise, allocate memory for a new node.
3. Print the data of the root node.
4. Set the data of the new node to be the data of the root node.
5. Set the left child of the new node to be the result of calling the AVLTree_Preorder_Traversal function with the left child of the root node as the argument.
6. Set the right child of the new node to be the result of calling the AVLTree_Preorder_Traversal function with the right child of the root node as the argument.
7. Return the new node.

Preorder traversal is a depth-first traversal method for traversing a tree. It involves visiting the root node first, then traversing the left subtree, and finally traversing the right subtree. The order in which the nodes are visited is root, left, right. This means that the root node is processed before its children, and the left child is processed before the right child.

To implement preorder traversal, you can create a function that takes the root node of the tree as an argument and performs the following steps:

1. If the root node is NULL, return NULL.

2. Otherwise, process the root node by performing some operation on it (e.g., printing its data).
3. Recursively call the function with the left child of the root node as the argument. This will traverse the left subtree.
4. Recursively call the function with the right child of the root node as the argument. This will traverse the right subtree.

Preorder traversal can be useful for creating a copy of a tree, or for printing out the nodes of a tree in a specific order.

```
struct AVLTreeNode *AVLTree_Inorder_Traversal(struct AVLTreeNode *root) {
    // If the root node is NULL, return NULL
    if (root == NULL) {
        return NULL;
        // Otherwise, traverse the left subtree, print the root node data, and
traverse the right subtree
    } else {
        // Allocate memory for a new node
        struct AVLTreeNode *node = malloc(sizeof(struct AVLTreeNode));
        // Traverse the left subtree
        node -> left = AVLTree_Inorder_Traversal(root -> left);
        // Print the root node data
        printf("%d", root->data);
        // Set the data of the new node to the root node data
        node -> data = root->data;
        // Traverse the right subtree
        node -> right = AVLTree_Inorder_Traversal(root -> right);
        return node;
    }
}
```

The algorithm that explains the code for the process of in-order traversal of an AVL tree:

1. If the root node is NULL, return NULL.
2. Otherwise, allocate memory for a new node.
3. Set the left child of the new node to be the result of calling the AVLTree_Inorder_Traversal function with the left child of the root node as the argument.
4. Print the data of the root node.
5. Set the data of the new node to be the data of the root node.
6. Set the right child of the new node to be the result of calling the AVLTree_Inorder_Traversal function with the right child of the root node as the argument.
7. Return the new node.

In order traversal is a depth-first traversal method for traversing a tree. It involves traversing the left subtree, visiting the root node, and then traversing the right subtree. The order in which the nodes are visited is left, root, right. This means that the left child is processed before the root node, and the root node is processed before the right child.

To implement in order traversal, you can create a function that takes the root node of the tree as an argument and performs the following steps:

1. If the root node is NULL, return NULL.
2. Recursively call the function with the left child of the root node as the argument. This will traverse the left subtree.
3. Process the root node by performing some operation on it (e.g. printing its data).
4. Recursively call the function with the right child of the root node as the argument. This will traverse the right subtree.

In order traversal is often used to print out the nodes of a tree in sorted order, since the nodes are visited in ascending order. It can also be useful for creating a copy of a tree or for performing some operation on each node in the tree.

```
struct AVLTreeNode *AVLTree_Postorder_Traversal(struct AVLTreeNode *root) {
    // If the root node is NULL, return NULL
    if (root == NULL) {
        return NULL;
        // Otherwise, traverse the left subtree, traverse the right subtree, and
print the root node data
    } else {
        // Allocate memory for a new node
        struct AVLTreeNode *node = malloc(sizeof(struct AVLTreeNode));
        // Traverse the left subtree
        node -> left = AVLTree_Postorder_Traversal(root -> left);
        // Traverse the right subtree
        node -> right = AVLTree_Postorder_Traversal(root -> right);
        // Print the root node data
        printf("%d ", root -> data);
        // Set the data of the new node to the root node data
        node -> data = root -> data;
        return node;
    }
}
```

The algorithm that explains the code for the process of post-order traversal of an AVL tree:

1. If the root node is NULL, return NULL.
2. Otherwise, allocate memory for a new node.

3. Set the left child of the new node to be the result of calling the AVLTree_Postorder_Traversal function with the left child of the root node as the argument.
4. Set the right child of the new node to be the result of calling the AVLTree_Postorder_Traversal function with the right child of the root node as the argument.
5. Print the data of the root node.
6. Set the data of the new node to be the data of the root node.
7. Return the new node.

Post order traversal is a depth-first traversal method for traversing a tree. It involves traversing the left subtree, traversing the right subtree, and then visiting the root node. The order in which the nodes are visited is left, right, root. This means that the left child is processed before the right child, and the right child is processed before the root node.

To implement post order traversal, you can create a function that takes the root node of the tree as an argument and performs the following steps:

1. If the root node is NULL, return NULL.
2. Recursively call the function with the left child of the root node as the argument. This will traverse the left subtree.
3. Recursively call the function with the right child of the root node as the argument. This will traverse the right subtree.
4. Process the root node by performing some operation on it (e.g. printing its data).

Post order traversal can be useful for deleting the nodes of a tree (since the root node is processed last, it can be deleted after its children have been deleted), or for creating a copy of a tree. It can also be used to perform some operation on each node in the tree, with the operation being performed after the left and right subtrees have been processed.

Splay Trees: -

Definition: -

Splay trees are a type of self-balancing binary search tree. They are named after their inventors, Daniel Sleator and Robert Tarjan, who introduced them in their 1985 paper "Self-Adjusting Binary Search Trees".

Like other self-balancing binary search trees, splay trees maintain a balance property to ensure that the tree remains balanced and that search, insert, and delete operations can be performed efficiently. However, splay trees differ from other self-balancing binary search trees in that they use a specific type of operation called a splay operation to maintain this balance.

A splay operation is a rotation that is used to move a node up to the root of the tree. This operation is performed on a node whenever it is accessed, whether for a search, insertion, or deletion operation. The idea behind splay trees is that frequently accessed nodes will be moved closer to the root, making future accesses to those nodes faster.

Overall, splay trees are a type of data structure that can be used to efficiently store and retrieve data in a sorted manner. They are known for their fast average-case performance, although their worst-case performance can be slower than other self-balancing binary search trees.

Insertion: -

Inserting a node into a splay tree follows a similar process to inserting a node into a regular binary search tree. The new node is added as a leaf in the tree, and its value is compared to the values of the nodes it is compared to determine its position in the tree.

However, after inserting a node into a splay tree, a splay operation is performed to bring the new node to the root of the tree.

Here is the process for inserting a node into a splay tree:

1. Find the correct position for the new node in the tree, as if you were inserting it into a regular binary search tree.
2. Insert the new node into the tree.
3. Perform a splay operation on the new node to bring it to the root of the tree.

It's worth noting that the process for inserting a node into a splay tree may require multiple splay operations, depending on the structure of the tree and the position of the new node.

Search: -

Searching for a value in a splay tree is like searching for a value in a regular binary search tree. The value is compared to the value at the root of the tree, and if it is less than the root value, the search continues in the left subtree; if it is greater than the root value, the search continues in the right subtree. This process is repeated until the value is found or it is determined that the value is not in the tree.

However, after searching for a value in a splay tree, a splay operation is performed to bring the node containing the searched value to the root of the tree. This operation is performed to make it faster to access the node in the future, since it will now be closer to the root of the tree.

Here is the process for searching for a value in a splay tree:

1. Compare the value to be searched for to the value at the root of the tree.
2. If the value is found at the root, the search is complete.
3. If the value is less than the value at the root, search for the value in the left subtree.
4. If the value is greater than the value at the root, search for the value in the right subtree.
5. Repeat the process until the value is found or it is determined that the value is not in the tree.
6. If the value is found, perform a splay operation on the node containing the value to bring it to the root of the tree.

It's worth noting that if the value being searched for is not present in the splay tree, the search will continue until it is determined that the value is not in the tree. In this case, the last node accessed during the search will be splayed to the root of the tree.

Deletion: -

Deleting a node from a splay tree involves a similar process to deleting a node from a regular binary search tree. However, after deleting a node from a splay tree, a splay operation is performed to bring the parent of the deleted

node to the root of the tree. This operation is performed to make it faster to access the parent in the future, since it will now be closer to the root of the tree.

Here is the process for deleting a node from a splay tree:

1. Find the node to be deleted in the tree.
2. If the node has no children, simply remove it from the tree.
3. If the node has one child, remove the node, and replace it with its child.
4. If the node has two children, find the successor of the node (the next larger value in the tree) and replace the node with its successor. Then delete the successor from its original position in the tree.
5. Perform a splay operation on the parent of the deleted node to bring it to the root of the tree.

It's worth noting that the process for deleting a node from a splay tree may require multiple splay operations, depending on the structure of the tree and the position of the node being deleted.

Splay Tree code Explanation: -

```
struct SplayTreeNode *SplayTree_Search_Node(struct SplayTreeNode *root, int data) {
    // Return NULL if the tree is empty.
    if (root == NULL) {
        return root;
    }
    // Increment the number of comparisons performed.
    SplayTree_NumberOfComparisons++;

    // Search for the node with the given data recursively.
    if (data < root -> data) {
        return SplayTree_Search_Node(root -> left, data);
    } else if (data > root -> data) {
        return SplayTree_Search_Node(root -> right, data);
    } else if (data == root -> data) {
        // Found the node, return it.
        return root;
    }
}
```

The algorithm for the **SplayTree_Search_Node** function can be outlined as follows:

1. If the root of the tree is **NULL**, return **NULL**.
2. Increment the counter for the number of comparisons performed.

3. If the data being searched for is less than the data at the root node, search for the data recursively in the left subtree.
4. If the data being searched for is greater than the data at the root node, search for the data recursively in the right subtree.
5. If the data being searched for is equal to the data at the root node, return the root node.

This algorithm will continue to search through the tree until it either finds the node with the desired data or determines that the node is not present in the tree, at which point it will return **NULL**.

SplayTree_Search_Node is a function that searches for a node with a specific value in a splay tree. The function takes in two parameters:

1. root: a pointer to the root node of the splay tree
2. data: the value of the node to be searched for

The function first checks if the tree is empty, in which case it returns NULL. If the tree is not empty, the function increments a counter that keeps track of the number of comparisons performed. It then searches for the node with the given data recursively. If the data is less than the data of the root node, the function searches the left subtree. If the data is greater than the data of the root node, the function searches the right subtree. If the data is equal to the data of the root node, the function returns the root node. If the function reaches a leaf node without finding the value, it returns NULL.

```
int SplayTree_Search_Value(struct SplayTreeNode *node, int data) {
    // Return -1 if the tree is empty or if we have reached a leaf node without
finding the value
    if (node == NULL) {
        return -1;
        // Recursively search the left subtree.
    } else if (SplayTree_Search_Value(node -> left, data) == 1) {
        return 1;
        // Recursively search the right subtree.
    } else if (SplayTree_Search_Value(node -> right, data) == 1) {
        return 1;
        // Check if the current node contains the value.
    } else if (node -> data == data) {
        // Increment the frequency of the node.
        node -> frequency++;
        return 1;
    } else {
        return -1;
    }
}
```

The algorithm for the **SplayTree_Search_Value** function can be outlined as follows:

1. If the node is **NULL** or if the function has reached a leaf node without finding the value, return **-1**.
2. Recursively search the left subtree for the value. If the value is found, return **1**.
3. Recursively search the right subtree for the value. If the value is found, return **1**.
4. If the value is not found in the left or right subtree, check if the current node contains the value. If it does, increment the frequency of the node and return **1**.
5. If the value is not found in the current node or in either of its subtrees, return **-1**.

SplayTree_Search_Value is a recursive function that searches for a given value in a splay tree. If the value is found, the function increments the frequency of the node containing the value and returns 1. If the value is not found, the function returns -1.

The function starts by checking if the tree is empty or if it has reached a leaf node without finding the value. If either of these conditions is true, it returns to -1. If the tree is not empty and the current node does not contain the value, the function recursively searches the left or right subtree depending on whether the value is less than or greater than the current node's data, respectively. If the value is equal to the current node's data, the function increments the frequency of the node and returns 1.

```
struct SplayTreeNode *SplayTree_Insert(struct SplayTreeNode *node, int data) {
    // Find the correct place to insert the new node.
    struct SplayTreeNode *temp1 = node;
    struct SplayTreeNode *temp2 = NULL;

    // Create the new node to insert.
    struct SplayTreeNode *temp3 = (struct SplayTreeNode*)malloc(sizeof(struct
SplayTreeNode));
    temp3 -> data = data;
    temp3 -> height = 1;
    temp3 -> frequency = 1;
    temp3 -> left = NULL;
    temp3 -> right = NULL;
    temp3 -> parent = NULL;

    while (temp1 != NULL) {
        temp2 = temp1;
        if (temp1 -> data > temp3 -> data) {
            SplayTree_NumberOfComparisons++;
            temp1 = temp1 -> left;
        } else {
            SplayTree_NumberOfComparisons++;
            temp1 = temp1 -> right;
        }
    }

    // Update the parent of the new node.
```

```
    temp3 -> parent = temp2;

    // Insert the new node into the tree.
    if (temp2 == NULL) {
        // The tree is empty, the new node becomes the root.
        return temp3;
    } else if (temp2 -> data > temp3 -> data) {
        temp2 -> left = temp3;
    } else {
        temp2 -> right = temp3;
    }

    // Splay the tree to bring the new node to the root.
    return SplayTree_Splay(node, temp3);
}
```

The algorithm for the **SplayTree_Insert** function can be outlined as follows:

1. Initialize two temporary variables **temp1** and **temp2** to **node** and **NULL**, respectively.
2. Allocate memory for a new tree node and store the data being inserted in the new node. Set the height and frequency of the new node to **1**, and set its left and right pointers to **NULL**.
3. While **temp1** is not **NULL**, set **temp2** to **temp1** and update **temp1** to either its left or right child, depending on whether the data being inserted is less than or greater than the data at **temp1**.
4. Set the parent of the new node to **temp2**.
5. If **temp2** is **NULL**, set the new node to be the root of the tree and return it. If the data at **temp2** is greater than the data being inserted, set the left child of **temp2** to the new node. Otherwise, set the right child of **temp2** to the new node.
6. Splay the tree to bring the new node to the root and return the root of the tree.

SplayTree_Insert is a function that inserts a new node with a given value into a splay tree. The function first searches for the correct place to insert the new node in the tree by traversing the tree and comparing the value of the new node to the value of each node in the tree. Once the correct place to insert the new node is found, the function creates a new node and inserts it into the tree. Finally, the function splays the tree to bring the new node to the root of the tree.

The function takes as input a pointer to the root node of the splay tree and the value to be inserted into the tree. It returns a pointer to the root node of the splay tree after the insertion has been performed.

```
struct SplayTreeNode *SplayTree_Splay(struct SplayTreeNode *root, struct
SplayTreeNode *node) {
    // Keep splaying the tree until the node becomes the root.
    while (node -> parent != NULL) {
        // Handle the different splay cases.
```

```
        if (node -> parent == root) {
            // Zig or Zag case.
            if (node == node -> parent -> left) {
                SplayTree_Zig_Case(node -> parent);
            } else if (node == node -> parent -> right) {
                SplayTree_Zag_Case(node -> parent);
            }

        } else {

            struct SplayTreeNode *parent = node -> parent;
            struct SplayTreeNode *grandparent = parent -> parent;

            if (node == node -> parent -> left) {
                // Zig-Zig or Zig-Zag case.
                if (parent == parent -> parent -> left) {
                    SplayTree_Left_ZigZig_Case(parent, grandparent);

                } else if (parent == parent -> parent -> right) {
                    SplayTree_Right_ZigZag_Case(parent, grandparent);
                }
            } else if (node == node -> parent -> right) {
                // Zag-Zig or Zag-Zag case.
                if (parent == parent -> parent -> left) {
                    SplayTree_Left_ZigZag_Case(parent, grandparent);

                } else if (parent == parent -> parent -> right) {
                    SplayTree_Right_ZigZig_Case(parent, grandparent);
                }
            }
        }
    }
    // Return the root of the splay tree after splaying.
    return node;
}
```

The algorithm for the **SplayTree_Splay** function can be outlined as follows:

1. While the node being splayed has a parent, perform the following steps:
    1. If the node's parent is the root of the tree, perform a Zig or Zag splay operation on the node's parent.
    2. If the node's parent is not the root of the tree, perform a Zig-Zig, Zig-Zag, Zag-Zig, or Zag-Zag splay operation on the node's parent and grandparent, depending on the positions of the node and its parent relative to each other and to the grandparent.

2. Return the root of the splay tree after splaying.

SplayTree_Splay is a function that takes a splay tree and a node in the tree as input and performs a splay operation on the tree to bring the node to the root. This function is often used to access a node in the tree, since bringing the node to the root allows for faster access to it.

The function works by looping until the node becomes the root of the tree. Within the loop, it handles different cases of splaying the tree depending on the position of the node in the tree and its parent and grandparent nodes. The different cases are:

- Zig or Zag case: If the node is a child of the root, a single rotation is performed to bring the node to the root.
- Zig-Zig or Zig-Zag case: If the node is the left child of its parent, which is also the left child of its grandparent, two right rotations are performed to bring the node to the root. If the node is the left child of its parent, but the parent is the right child of the grandparent, a left rotation is performed on the parent and a right rotation on the grandparent.
- Zag-Zig or Zag-Zag case: If the node is the right child of its parent, which is also the right child of its grandparent, two left rotations are performed to bring the node to the root. If the node is the right child of its parent, but the parent is the left child of the grandparent, a right rotation is performed on the parent and a left rotation on the grandparent.

After the splaying operation is completed, the function returns the root of the splay tree.

```
//-----------------------------------------------------------------------------------
--------//
// SplayTree_Zig_Case: Handle a splay tree Zig case.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
void SplayTree_Zig_Case(struct SplayTreeNode *parent) {
    // Perform a right rotation.
    SplayTree_NumberOfRotations++;
    SplayTree_RightRotation(parent);
}
//-----------------------------------------------------------------------------------
--------//
// SplayTree_Zag_Case: Handle a splay tree Zag case.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
void SplayTree_Zag_Case(struct SplayTreeNode *parent) {
    // Perform a left rotation.
    SplayTree_NumberOfRotations++;
    SplayTree_LeftRotation(parent);
}
//-----------------------------------------------------------------------------------
--------//
```

```c
// SplayTree_Left_ZigZig_Case: Handle a splay tree Zig-Zig case where the parent is
the left child of the grandparent.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
// - grandparent: Pointer to the grandparent node of the node being splayed.
void SplayTree_Left_ZigZig_Case(struct SplayTreeNode *parent, struct SplayTreeNode
*grandparent) {
    // Perform two right rotations.
    SplayTree_NumberOfRotations++;
    SplayTree_RightRotation(grandparent);
    SplayTree_NumberOfRotations++;
    SplayTree_RightRotation(parent);
}
//--------------------------------------------------------------------------------
--------//
// SplayTree_Right_ZigZig_Case: Handle a splay tree Zig-Zig case where the parent is
the right child of the grandparent.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
// - grandparent: Pointer to the grandparent node of the node being splayed.
void SplayTree_Right_ZigZig_Case(struct SplayTreeNode *parent, struct SplayTreeNode
*grandparent) {
    // Perform two left rotations.
    SplayTree_NumberOfRotations++;
    SplayTree_LeftRotation(grandparent);
    SplayTree_NumberOfRotations++;
    SplayTree_LeftRotation(parent);
}
//--------------------------------------------------------------------------------
--------//
// SplayTree_Left_ZigZag_Case: Handle a splay tree Zig-Zag case where the parent is
the left child of the grandparent.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
// - grandparent: Pointer to the grandparent node of the node being splayed.
void SplayTree_Left_ZigZag_Case(struct SplayTreeNode *parent, struct SplayTreeNode
*grandparent) {
    // Perform a left rotation on the parent and a right rotation on the grandparent.
    SplayTree_NumberOfRotations++;
    SplayTree_LeftRotation(parent);
    SplayTree_NumberOfRotations++;
    SplayTree_RightRotation(grandparent);
```

```
}
//-------------------------------------------------------------------------------
--------//
// SplayTree_Right_ZigZag_Case: Handle a splay tree Zig-Zag case where the parent is
the right child of the grandparent.
//
// Parameters:
// - parent: Pointer to the parent node of the node being splayed.
// - grandparent: Pointer to the grandparent node of the node being splayed.
void SplayTree_Right_ZigZag_Case(struct SplayTreeNode *parent, struct SplayTreeNode
*grandparent) {
    // Perform a right rotation on the parent and a left rotation on the grandparent.
    SplayTree_NumberOfRotations++;
    SplayTree_RightRotation(parent);
    SplayTree_NumberOfRotations++;
    SplayTree_LeftRotation(grandparent);
}
//-------------------------------------------------------------------------------
--------//
```

The algorithms for the **SplayTree_Zig_Case**, **SplayTree_Zag_Case**, **SplayTree_Left_ZigZig_Case**, **SplayTree_Right_ZigZig_Case**, **SplayTree_Left_ZigZag_Case**, and **SplayTree_Right_ZigZag_Case** functions can be outlined as follows:

SplayTree_Zig_Case

1. Perform a right rotation on the parent node.

SplayTree_Zag_Case

1. Perform a left rotation on the parent node.

SplayTree_Left_ZigZig_Case

1. Perform a right rotation on the grandparent node.
2. Perform a right rotation on the parent node.

SplayTree_Right_ZigZig_Case

1. Perform a left rotation on the grandparent node.
2. Perform a left rotation on the parent node.

SplayTree_Left_ZigZag_Case

1. Perform a left rotation on the parent node.
2. Perform a right rotation on the grandparent node.

SplayTree_Right_ZigZag_Case

1. Perform a right rotation on the parent node.
2. Perform a left rotation on the grandparent node.

SplayTree_Zig_Case is a function that handles a "Zig" case in a splay tree. A "Zig" case occurs when the node being splayed has a parent and the parent is the root of the splay tree. In this case, the node being splayed is either the left or right child of the root, and the splay operation involves a single rotation.

The function takes a single parameter, which is a pointer to the parent node of the node being splayed. The function first performs a right rotation on the parent node, which brings the node being splayed to the root of the splay tree. This operation is performed using the SplayTree_RightRotation function, which rearranges the nodes in the splay tree to achieve the desired rotation.

Overall, the SplayTree_Zig_Case function is used to move a node to the root of a splay tree, which is often necessary to make it more easily accessible for future operations.

The SplayTree_Zag_Case function is a helper function that is used to handle a "Zag" case in a splay tree. This refers to a situation where a node that is being splayed needs to be rotated to the left in order to move it closer to the root of the tree. This function is called when the parent of the node being splayed is the root of the tree and the node being splayed is the right child of the parent.

To handle this case, the function simply performs a left rotation on the parent node. This involves re-arranging the pointers between the parent and its left and right children so that the parent becomes the left child of the node being splayed, and the node being splayed becomes the root of the subtree. The height of the parent and the node being splayed are also updated after the rotation.

This function is usually called as part of the SplayTree_Splay function, which is responsible for splaying a node in a splay tree. The SplayTree_Splay function determines which splay case (Zig, Zag, Zig-Zig, Zig-Zag, etc.) applies to the node being splayed, and calls the appropriate helper function (such as SplayTree_Zig_Case or SplayTree_Zag_Case) to handle the case. The goal of splaying a node is to move it closer to the root of the tree, which can improve the performance of certain tree operations (such as searching and inserting) by reducing the height of the tree.

SplayTree_Left_ZigZig_Case is a function that handles a specific type of splay operation in a splay tree.

In a splay tree, the splaying operation is used to bring a node that is accessed or modified to the root of the tree, so that it can be easily accessed again in the future. The splaying operation is performed by rotating the tree around the node being splayed in a specific way, depending on the location of the node in the tree.

SplayTree_Left_ZigZig_Case is used to handle the splaying of a node that is the left child of its parent, and the parent is also the left child of its parent (the grandparent of the node being splayed). This is known as a "zig-zig" splay case.

In this case, the function performs two left rotations on the tree. The first left rotation is performed on the grandparent of the node being splayed, and the second left rotation is performed on the parent of the node being splayed. This brings the node being splayed to the root of the tree.

The function takes two parameters: a pointer to the parent node of the node being splayed, and a pointer to the grandparent node of the node being splayed. These pointers are used to identify the nodes that will be rotated in the tree.

SplayTree_Right_ZigZig_Case is a function that manages a splay tree Zig-Zig case where the parent is the right child of the grandparent. In this case, the tree needs to be rotated twice to bring the node being splayed to the root.

The function takes two parameters: a pointer to the parent node of the node being splayed, and a pointer to the grandparent node of the node being splayed. It performs two left rotations, first on the grandparent node and then on the parent node, to bring the node being splayed to the root. The function returns nothing.

A left rotation is a rotation of a tree node that rearranges the tree so that the node's left child becomes the root of the subtree, and the node becomes the right child of its left child. This operation is used to move a node down the tree and make its left child the new root of the subtree. In the context of a splay tree, left rotations are used to bring nodes higher up in the tree closer to the root, so that they can be accessed more quickly.

The **SplayTree_Left_ZigZag_Case** function is a helper function that handles a special case when splaying a node in a splay tree.

In a splay tree, when a node is accessed (e.g., searched for or inserted), it is brought to the root of the tree through a series of tree rotations. The **SplayTree_Left_ZigZag_Case** function is used to handle the case where the node being splayed is the left child of its parent, and the parent is the right child of its grandparent. This case is called a "Zig-Zag" case because the shape of the tree changes from a zig-zag pattern as the node is splayed to the root.

To handle this case, the function performs a left rotation on the parent node and a right rotation on the grandparent node. This results in the node being splayed to the root of the tree.

SplayTree_Right_ZigZag_Case is a function that handles a specific type of operation called a "Zig-Zag" in a splay tree. A Zig-Zag operation is used to bring a node to the root of a splay tree. It involves two rotations: a right rotation on the parent of the node being splayed, and a left rotation on the grandparent of the node being splayed.

In this function, the node being splayed is the right child of its parent, which is the left child of its grandparent. The function first performs a right rotation on the parent node, which rearranges the nodes in the subtree rooted at the parent node so that the node being splayed becomes the root of that subtree. Then, the function performs a left rotation on the grandparent node, which rearranges the nodes in the subtree rooted at the grandparent node so that the parent node becomes the root of that subtree.

After these two rotations, the node being splayed becomes the root of the splay tree, which is the desired result. This function is used in the SplayTree_Splay function to bring a node to the root of a splay tree.

```c
struct SplayTreeNode *SplayTree_LeftRotation(struct SplayTreeNode *node) {
    // Save a reference to the right child of the node.
    struct SplayTreeNode *temp = node -> right;
    // Make the left child of the right child the new right child of the node.
    node -> right = temp -> left;

    // Update the parent of the new right child.
    if (node -> right != NULL) {
        node -> right -> parent = node;
    }

    // Update the parent of the temp node.
    temp -> parent = node -> parent;

    // Update the child of the temp node's parent to be the temp node.
    if (node -> parent != NULL) {
        if (node -> parent -> data > temp -> data) {
```

```
            node -> parent -> left = temp;
        } else {
            node -> parent -> right = temp;
        }
    }

    // Make the node the left child of the temp node.
    temp -> left = node;

    // Update the parent of the node to be the temp node.
    node -> parent = temp;

    // Return the temp node, which is now the root of the subtree.
    return temp;
}
```

The algorithm for the **SplayTree_LeftRotation** function can be outlined as follows:

1. Save a reference to the right child of the node being rotated.
2. Set the left child of the right child to be the new right child of the node being rotated.
3. Update the parent of the new right child, if it exists.
4. Update the parent of the temp node to be the parent of the node being rotated.
5. Update the child of the temp node's parent to be the temp node, if the parent exists.
6. Set the node being rotated to be the left child of the temp node.
7. Update the parent of the node being rotated to be the temp node.
8. Return the temp node, which is now the root of the subtree.

SplayTree_LeftRotation is a function that performs a left rotation on a node in a splay tree. A left rotation is a tree operation that rearranges the nodes in a binary tree such that the left child of the rotated node becomes the root of the subtree, and the rotated node becomes the right child of the new root.

Here is an example of a left rotation:

Before rotation:

A Rotated node: A

/ \ Right child: B

B C => Left child: D

/ \ Parent: NULL

D E Right child: C

Left child: E

After rotation:

B Rotated node: B

/ \ Right child: A

D A => Left child: D

/ \ Parent: NULL

C E Right child: C

Left child: E

In the function SplayTree_LeftRotation, the first step is to save a reference to the right child of the node being rotated. Then, the left child of the right child becomes the new right child of the node being rotated. The parent of the new right child is updated to be the node being rotated. The parent and child of the rotated node are updated to reflect the rotation. Finally, the rotated node becomes the left child of the right child. The function returns the right child, which is now the root of the subtree.

Splay trees are self-balancing data structures that can be used to store and efficiently retrieve data. Left and right rotations are used to maintain balance in the tree and improve the performance of search and insertion operations.

```
struct SplayTreeNode *SplayTree_RightRotation(struct SplayTreeNode *node) {
    // Save a reference to the left child of the node.
    struct  SplayTreeNode *temp = node -> left;
    // Make the right child of the left child the new left child of the node.
    node -> left = temp -> right;

    // Update the parent of the new left child.
    if (node -> left != NULL) {
        node -> left -> parent = node;
    }

    // Update the parent of the temp node.
    temp -> parent = node -> parent;

    // Update the child of the temp node's parent to be the temp node.
    if (node -> parent != NULL) {
        if (node -> parent -> data > temp -> data) {
            node -> parent -> left = temp;
        } else {
            node -> parent -> right = temp;
        }
    }

    // Make the node the right child of the temp node.
    temp -> right = node;
```

```
    // Update the parent of the node to be the temp node.
    node -> parent = temp;

    // Return the temp node, which is now the root of the subtree.
    return temp;
}
```

The algorithm for the **SplayTree_RightRotation** function can be outlined as follows:

1. Save a reference to the left child of the node being rotated.
2. Set the right child of the left child to be the new left child of the node being rotated.
3. Update the parent of the new left child, if it exists.
4. Update the parent of the temp node to be the parent of the node being rotated.
5. Update the child of the temp node's parent to be the temp node if the parent exists.
6. Set the node being rotated to be the right child of the temp node.
7. Update the parent of the node being rotated to be the temp node.
8. Return the temp node, which is now the root of the subtree.

SplayTree_RightRotation is a function that performs the right rotation on a node in a splay tree. A right rotation is a operation that rearranges the nodes in a tree such that the right child of the node becomes the root of the subtree, and the original root becomes the left child of the new root. This operation is used to balance the tree and bring frequently accessed nodes closer to the root of the tree.

To perform a right rotation, the function first saves a reference to the left child of the node (called "temp"). It then makes the right child of the left child the new left child of the node. The parent of the new left child is updated to be the node. The parent of the temporary node is updated to be the parent of the original node. The child of the temporary node's parent is updated to be the temporary node, replacing the original node. Finally, the original node is made the right child of the temporary node, and the parent of the node is updated to be the temporary node.

After the rotation, the temporary node becomes the root of the subtree and is returned by the function. The original root node becomes the left child of the temporary node. This operation helps to balance the tree and bring frequently accessed nodes closer to the root, improving the performance of search and insertion operations.

```
int SplayTree_Height(struct SplayTreeNode *key) {
    // If the tree is empty, return -1.
    if (key == NULL) {
        return -1;
    } else {
        // Calculate the height of the left and right subtrees.
        int left_height = SplayTree_Height(key -> left);
        int right_height = SplayTree_Height(key -> right);
```

```
        // Return the maximum of the left and right heights, plus 1 for the root
node.
        return (left_height > right_height ? left_height : right_height) + 1;
    }
}
```

The algorithm for the **SplayTree_Height** function can be outlined as follows:

1. If the tree is empty, return -1.
2. Calculate the height of the left and right subtrees.
3. Return the maximum of the left and right heights, plus 1 for the root node.

SplayTree_Height is a function that returns the height of a given node in a splay tree. The height of a node is the number of edges on the longest path from the node to a leaf. If the tree is empty, the function returns -1. Otherwise, it calculates the height of the left and right subtrees of the given node using recursion and returns the maximum of the left and right heights, plus 1 for the root node. The function is typically used to determine the balance of a splay tree, which can affect the performance of the tree.

```
struct SplayTreeNode *SplayTree_Preorder_Traversal(struct SplayTreeNode *root) {
    // If the tree is empty, return NULL.
    if (root == NULL) {
        return NULL;
    } else {
        // Allocate memory for a new node.
        struct SplayTreeNode *node = malloc(sizeof(struct SplayTreeNode));
        // Print the data of the root node.
        printf("%d ", root -> data);
        // Recursively traverse the left and right subtrees and assign the returned
nodes to the left and right children of the new node.
        node -> data = root-> data;
        node -> left = SplayTree_Preorder_Traversal(root -> left);
        node -> right = SplayTree_Preorder_Traversal(root -> right);

        // Return the new node, which is a copy of the root node.
        return node;
    }
}
```

The algorithm for the **SplayTree_Preorder_Traversal** function can be outlined as follows:

1. If the tree is empty, return NULL.
2. Allocate memory for a new node.
3. Print the data of the root node.

4. Recursively traverse the left and right subtrees and assign the returned nodes to the left and right children of the new node.

5. Return the new node, which is a copy of the root node.

SplayTree_Preorder_Traversal is a function that performs a preorder traversal of a splay tree and prints the data of each node in the tree. A preorder traversal visits the root node first, then recursively visits the left and right subtrees.

The function takes a pointer to the root node of the splay tree as an argument and returns a pointer to a copy of the root node. If the tree is empty, the function returns a NULL pointer.

The function first checks if the root node is NULL. If it is, the function returns NULL. If the root node is not NULL, the function allocates memory for a new node, assigns the data of the root node to the new node, and prints the data of the root node.

The function then recursively calls itself on the left and right subtrees of the root node and assigns the returned nodes to the left and right children of the new node. Finally, the function returns the new node, which is a copy of the root node.

```
struct SplayTreeNode *SplayTree_Inorder_Traversal(struct SplayTreeNode *root) {
    // If the tree is empty, return NULL.
    if (root == NULL) {
        return NULL;
    } else {
        // Allocate memory for a new node.
        struct SplayTreeNode *node = malloc(sizeof(struct SplayTreeNode));
        // Recursively traverse the left subtree and assign the returned node to the
left child of the new node.
        node -> left = SplayTree_Inorder_Traversal(root -> left);
        // Assign the data of the root node to the new node.
        node -> data = root -> data;
        // Print the data of the root node.
        printf("%d ", root -> data);
        // Recursively traverse the right subtree and assign the returned node to the
right child of the new node.
        node -> right = SplayTree_Inorder_Traversal(root -> right);

        // Return the new node, which is a copy of the root node.
        return node;
    }
}
```

The algorithm for the **SplayTree_Inorder_Traversal** function can be outlined as follows:

1. If the tree is empty, return NULL.

2. Allocate memory for a new node.

3. Recursively traverse the left subtree and assign the returned node to the left child of the new node.
4. Assign the data of the root node to the new node.
5. Print the data of the root node.
6. Recursively traverse the right subtree and assign the returned node to the right child of the new node.
7. Return the new node, which is a copy of the root node.

SplayTree_Inorder_Traversal is a recursive function that traverses a splay tree in an inorder fashion, which means that it visits the left subtree of the root node first, then the root node itself, and then the right subtree of the root node. At each node in the tree, it prints the data stored at that node and creates a new node with the same data and the left and right children as the returned nodes from recursive calls on the left and right subtrees, respectively. It then returns the new node, which is a copy of the root node.

In order traversal is a way of visiting the nodes of a tree such that for each node, the left subtree is visited first, then the node itself, and finally the right subtree. This results in the nodes being printed in ascending order (if the tree is a binary search tree).

```c
struct SplayTreeNode *SplayTree_Postorder_Traversal(struct SplayTreeNode *root) {
    // If the tree is empty, return NULL.
    if (root == NULL) {
        return NULL;
    } else {
        // Allocate memory for a new node.
        struct SplayTreeNode *node = malloc(sizeof(struct SplayTreeNode));
        // Recursively traverse the left and right subtrees and assign the returned
nodes to the left and right children of the new node
        node -> left = SplayTree_Postorder_Traversal(root -> left);
        node -> right = SplayTree_Postorder_Traversal(root -> right);

        // Assign the data of the root node to the new node.
        node -> data = root -> data;
        // Print the data of the root node.
        printf("%d ", root -> data);

        // Return the new node, which is a copy of the root node.
        return node;
    }
}
```

The SplayTree_Postorder_Traversal function is an algorithm that traverses the nodes of a splay tree in postorder. The steps are as follows:

1. Check if the tree is empty. If it is, return NULL.
2. Allocate memory for a new node.

3. Recursively traverse the left and right subtrees of the root node and assign the returned nodes to the left and right children of the new node.
4. Assign the data of the root node to the new node.
5. Print the data of the root node.
6. Return the new node, which is a copy of the root node.

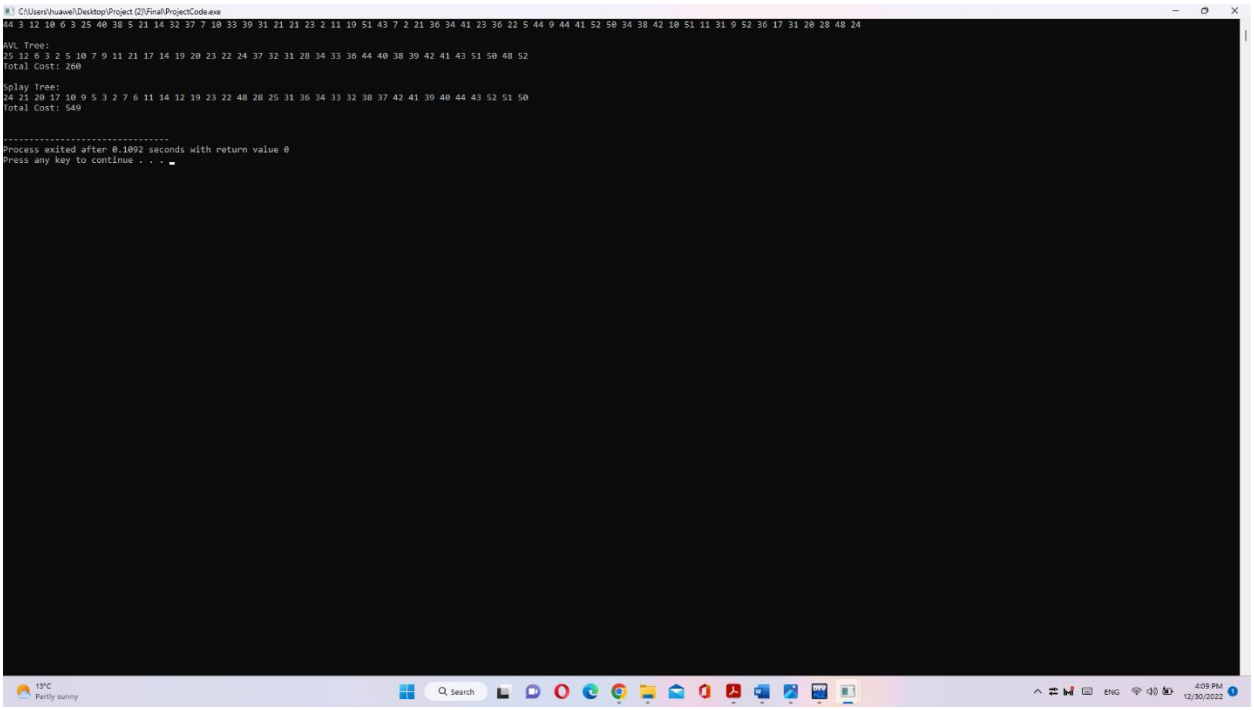
SplayTree_Postorder_Traversal is a function that traverses the nodes of a splay tree in postorder. It does this by first recursively visiting the left and right subtrees of the root node, then processing the root node itself.

The function takes a pointer to the root node of the splay tree as an input and returns a copy of the tree with the same structure, but with the data of each node printed in postorder. It does this by allocating memory for a new node, assigning the data of the root node to the new node, and recursively traversing the left and right subtrees and assigning the returned nodes to the left and right children of the new node. After this, it prints the data of the root node and returns the new node, which is a copy of the root node.

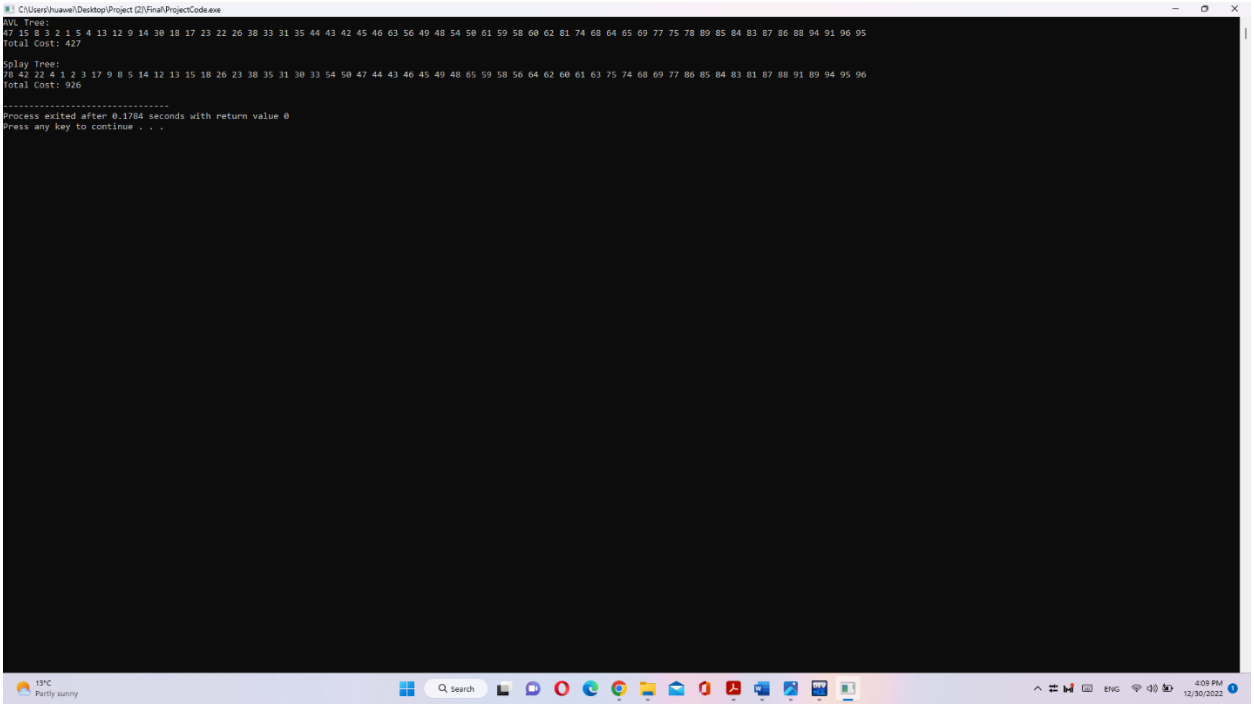
The postorder traversal of a tree is defined as visiting the left subtree, then the right subtree, and then the root node. This means that the nodes of the tree are processed in the following order: left child, right child, root. This is useful for tasks such as deleting the nodes of a tree, where the root node needs to be processed after its children have been processed.

Test Cases: -

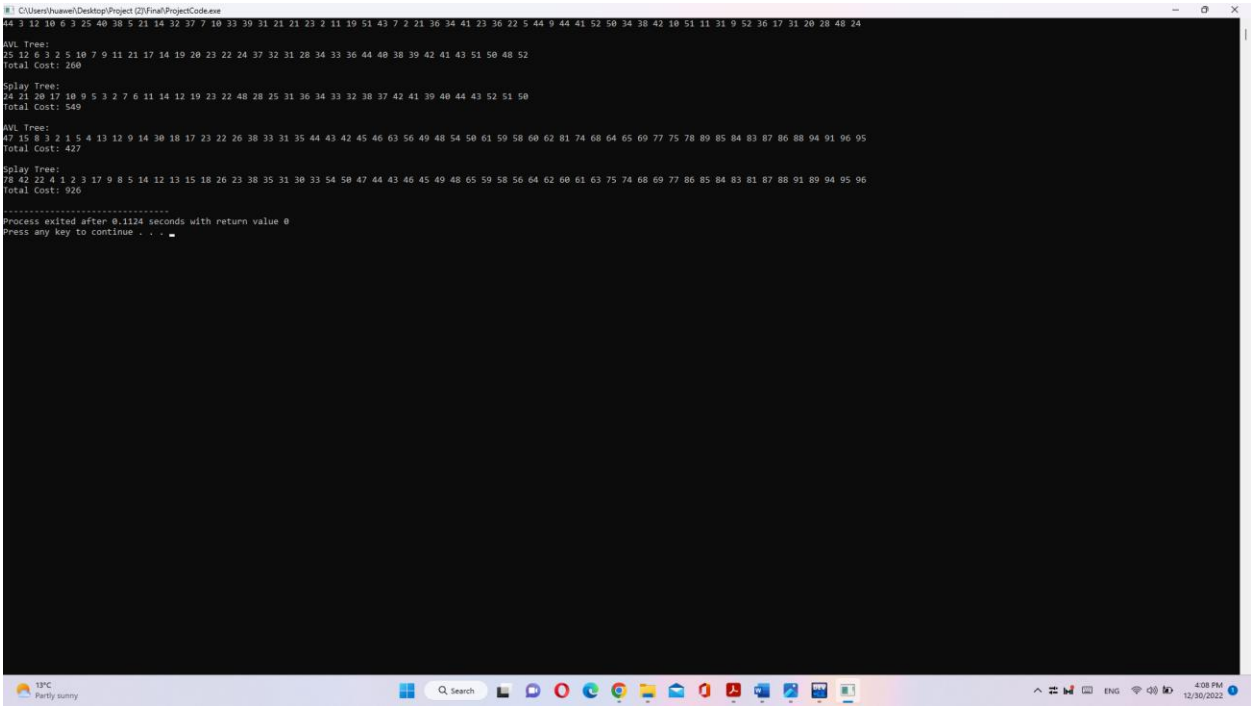
Output (1): -



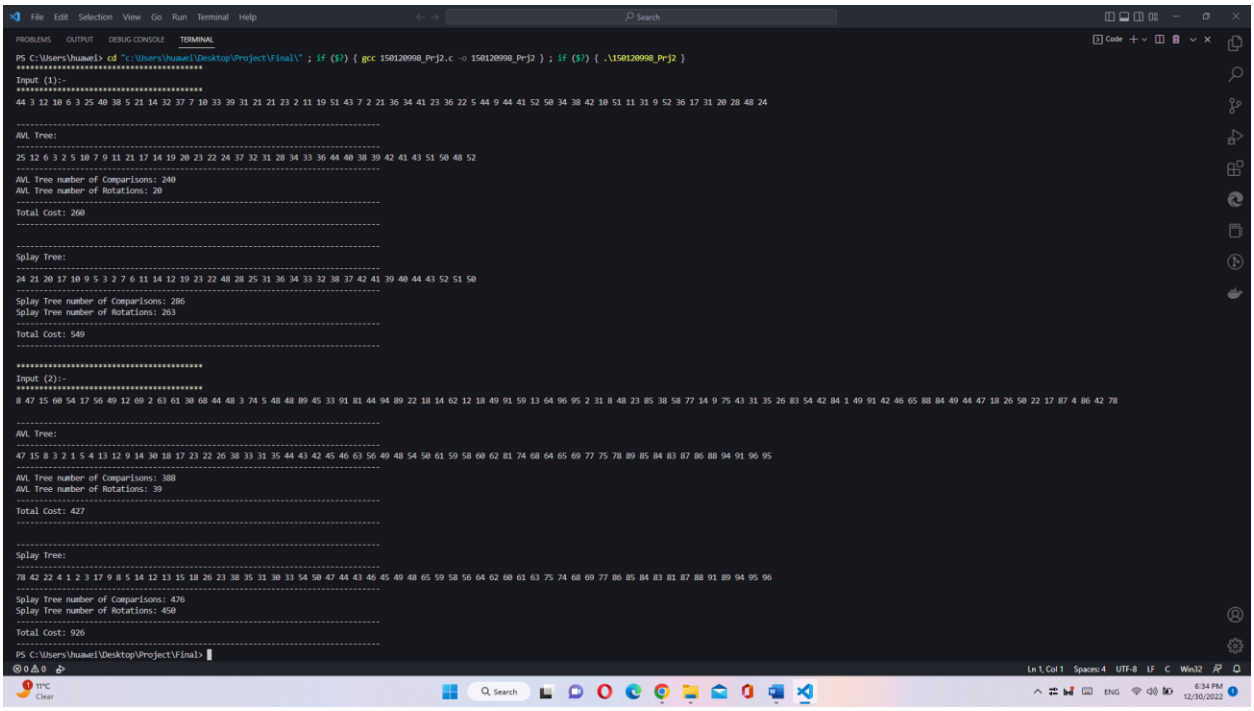
Output (2): -



Output (1) & Output (2): -



PowerShell Output: -


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\huawei> cd "c:\Users\huawei\Desktop\Project\Final\" ; if ($?) { gcc 150120998_Prj2.c -o 150120998_Prj2 } ; if ($?) { .\150120998_Prj2 }
Input (1):-
44 3 12 10 6 3 25 40 38 5 21 14 32 37 7 10 33 39 31 21 21 23 2 11 19 51 43 7 2 21 36 34 41 23 36 22 5 44 9 44 41 52 50 34 38 42 10 51 11 31 9 52 36 17 31 20 28 48 24
AVL Tree:
25 12 6 3 2 5 10 7 9 11 21 17 14 19 20 23 22 24 37 32 31 28 34 33 36 44 40 38 39 42 41 43 51 50 48 52
AVL Tree number of Comparisons: 240
AVL Tree number of Rotations: 20
Total Cost: 260
Splay Tree:
24 21 20 17 10 9 5 3 2 7 6 11 14 12 19 23 22 48 28 25 31 36 34 33 32 38 37 42 41 39 40 44 43 52 51 50
Splay Tree number of Comparisons: 286
Splay Tree number of Rotations: 263
Total Cost: 549
Input (2):-
8 47 15 60 54 17 56 49 12 69 2 63 61 30 68 44 48 3 74 5 48 48 89 45 33 91 81 44 94 89 22 18 14 62 12 18 49 91 59 13 64 96 95 2 31 8 48 23 85 38 58 77 14 9 75 43 31 35 26 83 54 42 84 1 49 91 42 46 65 88 84 49 44 47 18 26 50 22 17 87 4 86 42 78
AVL Tree:
47 15 8 3 2 1 5 4 13 12 9 14 30 18 17 23 22 26 38 33 31 35 44 43 42 45 46 63 56 49 48 54 50 61 59 58 60 62 81 74 68 64 65 69 77 75 78 89 85 84 83 87 86 88 94 91 96 95
AVL Tree number of Comparisons: 388
AVL Tree number of Rotations: 39
Total Cost: 427
Splay Tree:
78 42 22 4 1 2 3 17 9 8 5 14 12 13 15 18 26 23 38 35 31 30 33 54 50 47 44 43 46 45 49 48 65 59 58 56 64 62 60 61 63 75 74 68 69 77 86 85 84 83 81 87 88 91 89 94 95 96
Splay Tree number of Comparisons: 476
Splay Tree number of Rotations: 450
Total Cost: 926
PS C:\Users\huawei\Desktop\Project\Final>
Ln 1, Col 1 Spaces: 4 UTF-8 LF C Win32
11°C Clear
Search
ENG
6:34 PM 12/30/2022