Project Report

Project Report on: Implementation of AA Trees.
Course Code: CSE207
Course Name: Data Structure

Introduction:

A kind of balanced binary search tree known as an AA tree uses a condensed version of the red-black tree properties to maintain balance. It was named after Arne Andersson, who introduced the structure. The goal of an AA tree is to make the balancing mechanism simpler than that of a red-black tree, thus facilitating easier implementation and maintenance.

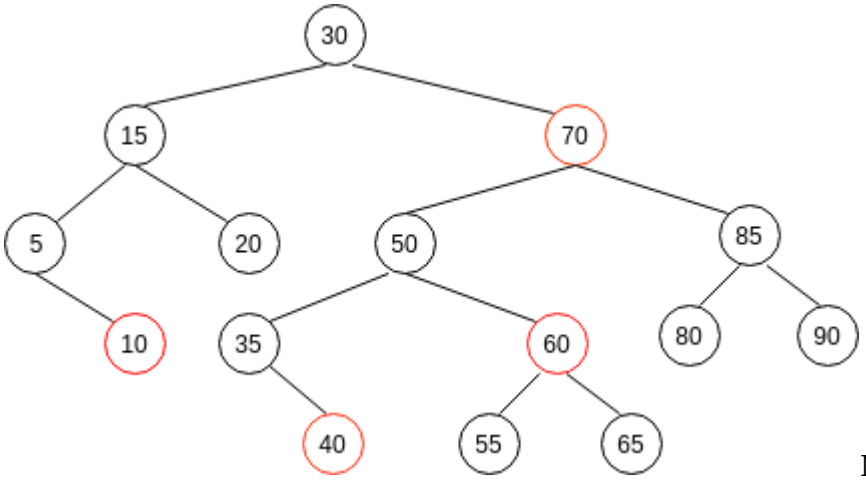
Properties:

1. Node Levels: Each node in an AA tree has a level, similar to the concept of black heights in red-black trees. The root node has the highest level, and leaves (or null nodes) are at level 0.
2. Level Invariants:

- The level of each leaf node is one.
- A right child's level is either the same as or one less compared to that of its parent.
- A left child's level is at most one less than that of its parent.
- The level of a right grandchild is always less than its grandparent's level.

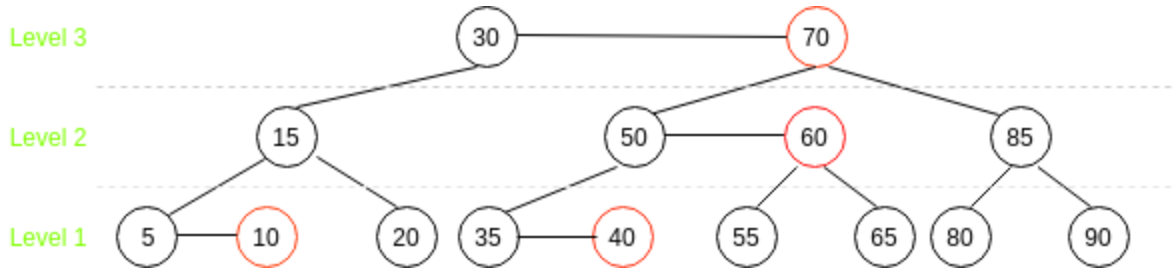
3. Balancing Operations: The tree uses two primary operations to maintain balance:
    - Skew: A right rotation to handle left horizontal links.
    - Split: A left rotation followed by an increment in level to handle consecutive right horizontal link.

Example of AA trees:



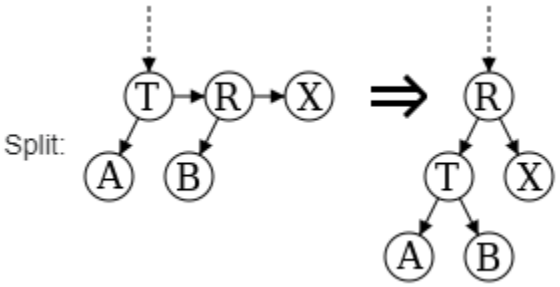
No left red children!!

After re-drawing the above AA tree with levels and horizontal links (the red nodes are shown connected through horizontal or red links), the tree looks like:

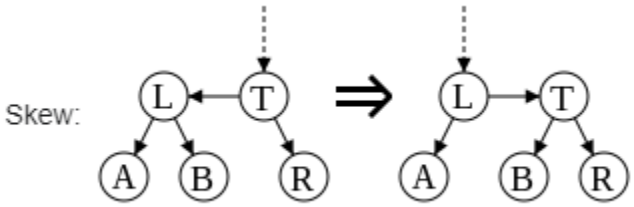


Note that all the nodes on **level 1** i.e. (5, 10, 20, 35, 40, 55, 65, 80, 90) are known as leaf nodes.

- Split Operation



- Skew Operation



- **<u>Real-Life Implementations of AA Trees</u>**

 Contact Management in Phones: AA trees can efficiently manage contact lists by keeping them balanced, ensuring quick access, insertion, and deletion operations.

 Databases: In database indexing, AA trees provide a balanced structure that supports efficient data retrieval, insertions, and updates.
 Memory Management: Operating systems use AA trees for managing memory allocation and deallocation to maintain efficient access and modification times.
 Network Routers: AA trees help in routing tables to maintain balanced and efficient pathfinding and updating mechanisms.
 File Systems: File systems use AA trees to manage directories and file locations, ensuring quick access and updates.
 Compiler Design: Compilers use AA trees for syntax trees and symbol tables, ensuring efficient parsing and lookups.

- **Why AA Tree**

When comparing AA trees with other types of self-balancing binary search trees, such as AVL trees and red-black trees, several distinctions and advantages emerge that highlight why AA trees might be chosen for certain applications.

1.The balancing operations in AA trees are simpler compared to the more complex rebalancing required in red-black and AVL trees, making the code easier to write, debug, and maintain.

2.While not as strictly balanced as AVL trees, AA trees provide a good balance with fewer rotations and simpler logic, which often leads to better practical performance in many scenarios.

Time and Space Complexity of AA Trees

- Insertion: O(log n) - Traversal and balancing operations ensure logarithmic time.
- Deletion: O(log n) - Similar to insertion, involves traversal and balancing.
- Search: O(log n) - Logarithmic due to balanced tree height.
- Skew and Split Operations: O(1) each - Constant time for each operation.
- Space Per Node: O(1) - Each node stores fixed amount of information.
- Total Space: O(n) - Linear space relative to the number of nodes.

Implementation (Algorithm):

To implement an AA tree, start by defining a node structure that includes key, value, left and right child pointers, and a level attribute. For insertion, recursively traverse the tree to find the appropriate position, insert the new node, and then perform skew and split operations to maintain balance. For deletion, locate and remove the node, adjust pointers accordingly, and then perform skew, split, and level decrease operations to rebalance the tree. Implement skew to rotate right if a left horizontal link is detected, and split to rotate left if a double right horizontal link exists. Ensure all balancing operations are integrated within the recursive insertion and deletion functions to keep the tree balanced after each modification.

Function:

1.Create Node:

"int data"The data value to be stored in the new node.Returns a pointer to the newly created node.The function uses malloc to allocate memory for a new node of type struct Node.It checks if the memory allocation is successful. If not, it prints an error message and exits the program.The fields of the new node (data, level, left, and right) are initialized.The function returns a pointer to the newly created node.Whenever a new node needs to be created, the createNode function is called, providing the desired data value.

Code:

```
struct TreeNode* createTreeNode(int value) {
   struct TreeNode* newNode = (struct TreeNode*) malloc(sizeof(struct TreeNode));
   if (newNode == NULL) {
      fprintf(stderr, "Memory allocation error\n");
      exit(EXIT_FAILURE);
   }
   newNode->value = value;
   newNode->data_level = 1;
   newNode->leftChild = NULL;
   newNode->rightChild = NULL;
   return newNode;
}
```

2.Skew and Split:

The performSkew function is designed to eliminate left horizontal links by performing a right rotation. This helps maintain the balance of the AA tree.

During skew, if the left child's level is equal to the root's level, a right rotation occurs, making the left child the new root.The performSplit function addresses right horizontal links by performing a left rotation, which helps in maintaining the tree's balance.In the split function, if the right child's right child's level is equal to the root's level, a left rotation is performed, and the root's level is incremented.Both skew and split operations are essential in the AA tree to ensure that it remains balanced after insertions and deletions, preserving the tree's efficiency for operations like search and traversal

```
struct TreeNode* performSkew(struct TreeNode* root) {
    if (root == NULL || root->leftChild == NULL)
        return root;

    if (root->leftChild->data_level == root->data_level) {
        struct TreeNode* temp = root->leftChild;
        root->leftChild = temp->rightChild;
        temp->rightChild = root;
        root = temp;
    }
    return root;
}


struct TreeNode* performSplit(struct TreeNode* root) {
    if (root == NULL || root->rightChild == NULL || root->rightChild->rightChild ==
NULL)
        return root;

    if (root->data_level == root->rightChild->rightChild->data_level) {
        struct TreeNode* temp = root->rightChild;
        root->rightChild = temp->leftChild;
        temp->leftChild = root;
        root = temp;
        root->data_level++;
    }
    return root;
```

```
}
```

3.Insert:

Inserts a new node with the specified data into the AA tree, performing skew and split operations to maintain balance.

```
struct TreeNode* addNode(struct TreeNode* root, int value) {

    if (root == NULL)
        return createTreeNode(value);

    if (value < root->value)
        root->leftChild = addNode(root->leftChild, value);
    else if (value > root->value)
        root->rightChild = addNode(root->rightChild, value);
    else {
        printf("Duplicate value, ignoring insertion.\n");
        return root;
    }



    root = performSkew(root);
    root = performSplit(root);

    return root;
}
```

4.Delete:

Deletes a node with the specified data from the AA tree, maintaining balance through skew and split operations

```
struct TreeNode* removeNode(struct TreeNode* root, int value) {
   if (root == NULL) {
      printf("Element not found, ignoring deletion.\n");
      return root;
   }

   if (value < root->value) {

      root->leftChild = removeNode(root->leftChild, value);
   } else if (value > root->value) {

      root->rightChild = removeNode(root->rightChild, value);
   } else {
            struct TreeNode* temp;
      if (root->leftChild != NULL && root->rightChild != NULL) {

         temp = root->rightChild;
         while (temp->leftChild != NULL) {
            temp = temp->leftChild;
         }
         root->value = temp->value;

         root->rightChild = removeNode(root->rightChild, temp->value);
      } else {

         temp = root;
         if (root->leftChild == NULL) {
            root = root->rightChild;
         } else if (root->rightChild == NULL) {
            root = root->leftChild;
         }
```

```
            free(temp);
        }
    }


    if (root == NULL) {
        return root;
    }


    root = performSkew(root);
    root->rightChild = performSkew(root->rightChild);
    if (root->rightChild != NULL) {
        root->rightChild->rightChild = performSkew(root->rightChild->rightChild);
    }
    root = performSplit(root);
    root->rightChild = performSplit(root->rightChild);

    return root;
}
```

5.Search:

Searches for a node with the specified data in the AA tree.

```
struct TreeNode* findNode(struct TreeNode* root, int value) {
    if (root == NULL || root->value == value)
        return root;

    if (value < root->value)
        return findNode(root->leftChild, value);
    else
        return findNode(root->rightChild, value);
}
```

6.Find Minimum and Find Maximum:

Find the nodes with the minimum and maximum values in the tree, respectively.

```
struct TreeNode* getMinimumNode(struct TreeNode* root) {
   if (root == NULL)
      return NULL;

   while (root->leftChild != NULL)
      root = root->leftChild;
   return root;
}

struct TreeNode* getMaximumNode(struct TreeNode* root) {
   if (root == NULL)
      return NULL;

   while (root->rightChild != NULL)
      root = root->rightChild;
   return root;
}
```

7.Calculate Height, Count Total Nodes, and Count Leaf Nodes:

In the implementation of an AA tree, it is essential to include utility functions that compute various properties of the tree to facilitate analysis and debugging. The longest path from the root to a leaf node is the definition of the tree's height, which is found by recursively traversing it with the calculateHeight function. This metric helps assess the tree's balance and efficiency in operations like search, insertion, and deletion. The countTotalNodes function counts the total number of nodes in the tree by recursively summing the nodes in each subtree. This provides insight into the overall size of the tree, crucial for understanding its storage requirements and performance characteristics. Lastly, the countLeafNodes function identifies and counts all leaf nodes, which are nodes without children. This count is significant for evaluating the tree's structure and can be used to infer aspects of its balance and density. These functions collectively offer a comprehensive view of the tree's structural attributes, aiding in the maintenance and optimization of the AA tree

- A function to calculate the height of the tree

```
int calculateHeight(struct TreeNode* root) {
```

```
    if (root == NULL)
        return 0;

    int leftHeight = calculateHeight(root->leftChild);
    int rightHeight = calculateHeight(root->rightChild);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}
```

- A function to determine the total number of nodes in the tree

```
int countTotalNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    return 1 + countTotalNodes(root->leftChild) + countTotalNodes(root->rightChild);
}
```

- A function to determine how many leaf nodes there are in the tree

```
int countLeafNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    if (root->leftChild == NULL && root->rightChild == NULL)
        return 1;

    return countLeafNodes(root->leftChild) + countLeafNodes(root->rightChild);
}
```

8. FreeTree:

The `freeTree` function plays a crucial role in managing memory resources within the implementation of an AA tree. As the name suggests, its purpose is to deallocate the memory allocated for all nodes in the tree, effectively releasing the entire tree from memory. This function typically operates recursively, traversing the tree in a post-order fashion to ensure that child nodes are freed before their parent nodes. By systematically deallocating memory starting from the bottom of the tree, the `freeTree` function ensures that no memory leaks occur and that all dynamically allocated memory associated with the tree is properly released back to the system. Proper memory management is essential for maintaining the efficiency and stability of

programs utilizing dynamic data structures like AA trees, making the `freeTree` function a critical component of the implementation.

```
void freeTree(struct TreeNode* root) {
   if (root == NULL)
      return;

   freeTree(root->leftChild);
   freeTree(root->rightChild);
   free(root);
}
```

9. Inorder, Preorder, and Postorder:

Perform in-order, pre-order, and post-order traversals of the tree. A preorder traversal visits the root node first, then the left subtree and finally the right subtree in a recursive manner. The order of visitation is root-left-right.In apostorder traversal, the root node is visited last, following a recursive traversal of

the left subtree and the right subtree. The order of visitation is left-right-root**.**

```
// Function for in-order traversal
void inorderTraversal(struct TreeNode* root) {
   if (root == NULL)
      return;

   inorderTraversal(root->leftChild);
   printf("%d ", root->value);
   inorderTraversal(root->rightChild);
}

// Function for pre-order traversal
void preorderTraversal(struct TreeNode* root) {
   if (root == NULL)
      return;

   printf("%d ", root->value);
   preorderTraversal(root->leftChild);
   preorderTraversal(root->rightChild);
}
```

```
// Function for post-order traversal
void postorderTraversal(struct TreeNode* root) {
   if (root == NULL)
      return;

   postorderTraversal(root->leftChild);
   postorderTraversal(root->rightChild);
   printf("%d ", root->value);
}
```

10.Main:

The `main` function orchestrates user interaction with an AA tree management program through a menu-driven interface. It continuously presents a menu of options, such as inserting, removing, searching for values, and traversing the tree, allowing users to choose the desired operation. Based on the user's selection, the `main` function calls corresponding functions to perform the requested operation on the AA tree. It also handles user input validation and ensures proper memory management, releasing allocated memory upon program termination.

11.Testing:

We have tested each operation with various input to verify the code. We have got the expected output which ensures that the AA trees properties are maintained.

Advantages:

1.Self-Balancing Tree: Designed to maintain an almost perfect balance, AA trees are self-balancing binary search trees that guarantee effective search, insertion, and deletion operations.

2.Two Key Operations: AA trees primarily utilize two operations called "skew" and "split" to maintain balance. These operations are performed during insertion and deletion to adjust the tree structure as needed.

3.Data-Level Annotation: Each node in an AA tree is annotated with a data level, representing the distance to the nearest leaf node. This data-level annotation simplifies the balancing process.

4.In-Order Property: AA trees maintain an in-order property, ensuring that the elements are stored in sorted order, facilitating efficient traversal operations.

5.Simplicity of Implementation: Compared to other self-balancing trees like AVL or red-black trees, AA trees are relatively simple to implement due to their straightforward balancing rules and fewer rotation cases.

Disadvantages:

1. **Complexity of Implementation:** Although AA trees are simpler to implement compared to some other self-balancing trees, they still require careful implementation of skew and split operations, which can be non-trivial for beginners.
2. **Memory Overhead**: AA trees typically require additional memory to store the data-level annotation for each node, increasing the memory overhead compared to simpler binary search trees. This extra memory usage can be a disadvantage in memory-constrained environments.
3. **Performance Trade-offs**: While AA trees offer relatively good performance for most operations, they may not match the performance of more complex self-balancing trees like AVL trees for certain use cases, particularly in scenarios with very large datasets or highly dynamic data.
4. **Limited Balancing**: Although AA trees maintain a near-perfect balance, they may not achieve the optimal balance achieved by AVL trees or red-black trees. This can lead to slightly slower search, insertion, and deletion operations in some cases

Conclusion:

In conclusion, the implementation of AA trees provides a valuable insight into maintaining balance in binary search trees through relatively simple operations like skew and split. This balance allows for efficient search, insertion, and deletion operations, making AA trees a practical choice for various applications. Through this implementation, we've learned the importance of balance in maintaining optimal performance in data structures, as well as the trade-offs between complexity and effectiveness. Additionally, we've gained a deeper understanding of tree structures and the significance of self-balancing mechanisms in optimizing algorithmic efficiency. Overall, the

implementation of AA trees underscores the importance of striking a balance between simplicity and effectiveness in designing data structures for real-world applications.

Source Code :

```
#include <stdio.h>
#include <stdlib.h>

struct TreeNode {
   int value;
   int data_level;
   struct TreeNode* leftChild;
   struct TreeNode* rightChild;
};


struct TreeNode* createTreeNode(int value);
struct TreeNode* performSkew(struct TreeNode* root);
struct TreeNode* performSplit(struct TreeNode* root);
struct TreeNode* addNode(struct TreeNode* root, int value);
struct TreeNode* removeNode(struct TreeNode* root, int value);
struct TreeNode* findNode(struct TreeNode* root, int value);
struct TreeNode* getMinimumNode(struct TreeNode* root);
struct TreeNode* getMaximumNode(struct TreeNode* root);
int calculateHeight(struct TreeNode* root);
int countTotalNodes(struct TreeNode* root);
int countLeafNodes(struct TreeNode* root);
void freeTree(struct TreeNode* root);
void inorderTraversal(struct TreeNode* root);
void preorderTraversal(struct TreeNode* root);
void postorderTraversal(struct TreeNode* root);
```

```
struct TreeNode* createTreeNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*) malloc(sizeof(struct TreeNode));
    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->value = value;
    newNode->data_level = 1;
    newNode->leftChild = NULL;
    newNode->rightChild = NULL;
    return newNode;
}

struct TreeNode* performSkew(struct TreeNode* root) {
    if (root == NULL || root->leftChild == NULL)
        return root;
    if (root->leftChild->data_level == root->data_level) {
        struct TreeNode* temp = root->leftChild;
        root->leftChild = temp->rightChild;
        temp->rightChild = root;
        root = temp;
    }
    return root;
}

struct TreeNode* performSplit(struct TreeNode* root) {
    if (root == NULL || root->rightChild == NULL || root->rightChild->rightChild == NULL)
        return root;
    if (root->data_level == root->rightChild->rightChild->data_level) {
        struct TreeNode* temp = root->rightChild;
        root->rightChild = temp->leftChild;
        temp->leftChild = root;
        root = temp;
        root->data_level++;
    }
    return root;
}

struct TreeNode* addNode(struct TreeNode* root, int value) {
```

```
    if (root == NULL)
        return createTreeNode(value);
    if (value < root->value)
        root->leftChild = addNode(root->leftChild, value);
    else if (value > root->value)
        root->rightChild = addNode(root->rightChild, value);
    else {
        printf("Duplicate value, ignoring insertion.\n");
        return root;
    }
    root = performSkew(root);
    root = performSplit(root);
    return root;
}

struct TreeNode* removeNode(struct TreeNode* root, int value) {
    if (root == NULL) {
        printf("Element not found, ignoring deletion.\n");
        return root;
    }
    if (value < root->value) {
        root->leftChild = removeNode(root->leftChild, value);
    } else if (value > root->value) {
        root->rightChild = removeNode(root->rightChild, value);
    } else {
        struct TreeNode* temp;
        if (root->leftChild != NULL && root->rightChild != NULL) {
            temp = root->rightChild;
            while (temp->leftChild != NULL) {
                temp = temp->leftChild;
            }
            root->value = temp->value;
            root->rightChild = removeNode(root->rightChild, temp->value);
        } else {
            temp = root;
            if (root->leftChild == NULL) {
                root = root->rightChild;
            } else if (root->rightChild == NULL) {
                root = root->leftChild;
```

```
            }
            free(temp);
        }
    }
    if (root == NULL) {
        return root;
    }
    root = performSkew(root);
    root->rightChild = performSkew(root->rightChild);
    if (root->rightChild != NULL) {
        root->rightChild->rightChild = performSkew(root->rightChild->rightChild);
    }
    root = performSplit(root);
    root->rightChild = performSplit(root->rightChild);
    return root;
}

struct TreeNode* findNode(struct TreeNode* root, int value) {
    if (root == NULL || root->value == value)
        return root;
    if (value < root->value)
        return findNode(root->leftChild, value);
    else
        return findNode(root->rightChild, value);
}

struct TreeNode* getMinimumNode(struct TreeNode* root) {
    if (root == NULL)
        return NULL;
    while (root->leftChild != NULL)
        root = root->leftChild;
    return root;
}

struct TreeNode* getMaximumNode(struct TreeNode* root) {
    if (root == NULL)
        return NULL;
    while (root->rightChild != NULL)
        root = root->rightChild;
```

```
    return root;
}

int calculateHeight(struct TreeNode* root) {
    if (root == NULL)
        return 0;
    int leftHeight = calculateHeight(root->leftChild);
    int rightHeight = calculateHeight(root->rightChild);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}

int countTotalNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;
    return 1 + countTotalNodes(root->leftChild) + countTotalNodes(root->rightChild);
}

int countLeafNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;
    if (root->leftChild == NULL && root->rightChild == NULL)
        return 1;
    return countLeafNodes(root->leftChild) + countLeafNodes(root->rightChild);
}

void freeTree(struct TreeNode* root) {
    if (root == NULL)
        return;
    freeTree(root->leftChild);
    freeTree(root->rightChild);
    free(root);
}

void inorderTraversal(struct TreeNode* root) {
    if (root == NULL)
        return;
    inorderTraversal(root->leftChild);
    printf("%d ", root->value);
    inorderTraversal(root->rightChild);
```

```
}

void preorderTraversal(struct TreeNode* root) {
    if (root == NULL)
        return;
    printf("%d ", root->value);
    preorderTraversal(root->leftChild);
    preorderTraversal(root->rightChild);
}

void postorderTraversal(struct TreeNode* root) {
    if (root == NULL)
        return;
    postorderTraversal(root->leftChild);
    postorderTraversal(root->rightChild);
    printf("%d ", root->value);
}

int main() {
    struct TreeNode* root = NULL;
    int choice, value;

    while (1) {
        printf("\nAA Tree Operations Menu:\n");
        printf("1. Insert Node\n");
        printf("2. Remove Node\n");
        printf("3. Search Node\n");
        printf("4. In-order Traversal\n");
        printf("5. Pre-order Traversal\n");
        printf("6. Post-order Traversal\n");
        printf("7. Find Minimum value Node\n");
        printf("8. Find Maximum value Node\n");
        printf("9. Calculate Height of AA Tree\n");
        printf("10. Count Total Nodes of AA Tree\n");
        printf("11. Count Leaf Nodes of AA Tree\n");
        printf("12. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
switch (choice) {
    case 1:
        printf("Enter value to insert: ");
        scanf("%d", &value);
        root = addNode(root, value);
        break;
    case 2:
        printf("Enter value to remove: ");
        scanf("%d", &value);
        root = removeNode(root, value);
        break;
    case 3:
        printf("Enter value to search: ");
        scanf("%d", &value);
        struct TreeNode* foundNode = findNode(root, value);
        if (foundNode != NULL)
            printf("Value %d found in the tree.\n", value);
        else
            printf("Value %d not found in the tree.\n", value);
        break;
    case 4:
        printf("In-order traversal: ");
        inorderTraversal(root);
        printf("\n");
        break;
    case 5:
        printf("Pre-order traversal: ");
        preorderTraversal(root);
        printf("\n");
        break;
    case 6:
        printf("Post-order traversal: ");
        postorderTraversal(root);
        printf("\n");
        break;
    case 7: {
        struct TreeNode* minNode = getMinimumNode(root);
        if (minNode != NULL)
            printf("Minimum value in the tree: %d\n", minNode->value);
```

```
            else
                printf("The tree is empty.\n");
            break;
        }
        case 8: {
            struct TreeNode* maxNode = getMaximumNode(root);
            if (maxNode != NULL)
                printf("Maximum value in the tree: %d\n", maxNode->value);
            else
                printf("The tree is empty.\n");
            break;
        }
        case 9: {
            int height = calculateHeight(root);
            printf("Height of the tree: %d\n", height);
            break;
        }
        case 10: {
            int totalNodes = countTotalNodes(root);
            printf("Total number of nodes in the tree: %d\n", totalNodes);
            break;
        }
        case 11: {
            int leafNodes = countLeafNodes(root);
            printf("Number of leaf nodes in the tree: %d\n", leafNodes);
            break;
        }
        case 12:
            freeTree(root);
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  }

  return 0;
}
```

Output :

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 1
Enter value to insert: 10
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 2
Enter value to remove: 8
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 3
Enter value to search: 2
Value 2 found in the tree.
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 4
In-order traversal: 1 3 4 5 10
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 5
Pre-order traversal: 4 1 3 10 5
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 6
Post-order traversal: 3 1 5 10 4
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 7
Minimum value in the tree: 1
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 8
Maximum value in the tree: 10
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 9
Height of the tree: 3
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 10
Total number of nodes in the tree: 5
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 11
Number of leaf nodes in the tree: 2
```

```
AA Tree Operations Menu:
1. Insert Node
2. Remove Node
3. Search Node
4. In-order Traversal
5. Pre-order Traversal
6. Post-order Traversal
7. Find Minimum value Node
8. Find Maximum value Node
9. Calculate Height of AA Tree
10. Count Total Nodes of AA Tree
11. Count Leaf Nodes of AA Tree
12. Exit
Enter your choice: 12
Exiting...
```