AIM: Construct an AVL tree for a given set of elements which are stored in a file. And implement insert and delete operation on the constructed tree. Write contents of tree into a new file using in-order.

```
#include <iostream>
#include <fstream>
using namespace std;

// Node structure for AVL Tree
struct Node {
   int key;
   Node* left;
   Node* right;
   int height;
};

// Function to get the height of the tree
int height(Node* N) {
   if (N == NULL)
      return 0;
   return N->height;
}

// Utility function to get the maximum of two integers
int max(int a, int b) {
   return (a > b) ? a : b;
}

// Helper function to create a new node
Node* newNode(int key) {
   Node* node = new Node();
   node->key = key;
   node->left = NULL;
   node->right = NULL;
   node->height = 1; // New node is initially added at leaf
   return(node);
```

```
}

// A utility function to right rotate subtree rooted with y
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
```

```
int getBalance(Node* N) {
   if (N == NULL)
      return 0;
   return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted with node and
returns the new root of the subtree.
Node* insert(Node* node, int key) {
   // 1. Perform the normal BST rotation
   if (node == NULL)
      return(newNode(key));

   if (key < node->key)
      node->left = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
   else // Equal keys are not allowed in BST
      return node;

   // 2. Update height of this ancestor node
   node->height = 1 + max(height(node->left), height(node->right));

   // 3. Get the balance factor of this ancestor node to check whether this
node became unbalanced
   int balance = getBalance(node);

   // If this node becomes unbalanced, then there are 4 cases

   // Left Left Case
   if (balance > 1 && key < node->left->key)
      return rightRotate(node);

   // Right Right Case
   if (balance < -1 && key > node->right->key)
      return leftRotate(node);
```

```
    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // return the (unchanged) node pointer
    return node;
}

// Function to find the node with minimum value
Node* minValueNode(Node* node) {
    Node* current = node;

    // loop down to find the leftmost leaf
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node with given key from subtree with given root. It returns root of the modified subtree.
Node* deleteNode(Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key, then it lies in left subtree
```

```
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key, then it lies in right
subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then this is the node to be deleted
    else {
        // node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            Node* temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of the non-empty child
            delete temp;
        }
        else {
            // node with two children: Get the inorder successor (smallest in the
right subtree)
            Node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
```

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

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left), height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to do inorder traversal of the tree and write it to a file
void inorder(Node* root, ofstream &outfile) {
```

```
    if (root != NULL) {
        inorder(root->left, outfile);
        outfile << root->key << " ";
        inorder(root->right, outfile);
    }
}

int main() {
    Node* root = NULL;

    // Reading elements from file
    ifstream infile("input.txt");
    int key;
    while (infile >> key) {
        root = insert(root, key);
    }
    infile.close();

    // Perform insertions and deletions as required
    // Example insertion and deletion
    root = insert(root, 30);
    root = deleteNode(root, 50);

    // Writing the contents of the tree to a file in inorder traversal
    ofstream outfile("output.txt");
    inorder(root, outfile);
    outfile.close();

    cout << "AVL Tree has been created and inorder traversal written to output.txt" << endl;

    return 0;
}
```

Explanation:

Header Files and Namespace

```
#include <iostream>  // For console input/output
#include <fstream>   // For file handling (ifstream, ofstream)
using namespace std; // To avoid prefixing std:: before cout, etc.
```

AVL Tree Node Structure

```
struct Node {
    int key;         // Data stored in the node
    Node* left;      // Pointer to left child
    Node* right;     // Pointer to right child
    int height;      // Height of the node for AVL balance calculations
};
```

Helper Functions

```
int height(Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}
```

- Returns the height of a node. If the node is `NULL`, height is 0.

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

- Returns the maximum of two integers.

```
Node* newNode(int key) {
    Node* node = new Node();  // Dynamically create a new node
    node->key = key;          // Assign the key
    node->left = NULL;        // Left and right children are NULL
    node->right = NULL;
    node->height = 1;         // A new node is a leaf, so height is 1
    return(node);
}
```

Tree Rotations

Right Rotation (for Left-Left imbalance)

```
Node* rightRotate(Node* y) {
    Node* x = y->left;       // x becomes new root
    Node* T2 = x->right;     // Temporarily hold x's right subtree

    x->right = y;            // Perform rotation
```

```
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1; // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;              // Return new root
}
```

Left Rotation (for Right-Right imbalance)

```
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

Balance Factor

```
int getBalance(Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

- Returns the balance factor of a node (left height - right height).

Insertion Function

```
Node* insert(Node* node, int key) {
    if (node == NULL)
        return(newNode(key));
```

- If tree is empty, create a new node.

```
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;  // Duplicates not allowed
```

- Standard Binary Search Tree (BST) insert.

```
    node->height = 1 + max(height(node->left), height(node->right));
```

- Update the height of the current node.

```
    int balance = getBalance(node);  // Get balance factor
```

- Calculate balance factor to check if rebalancing is needed.

Four AVL cases:

```
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);  // Left-Left case

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);   // Right-Right case

    if (balance > 1 && key > node->left->key) { // Left-Right case
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) { // Right-Left case
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;  // Return unchanged node
}
```

Find Minimum Node (for deletion)

```
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

- Finds the node with the smallest key in a subtree.

Delete Node Function

```
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;
```

- Base case: if root is NULL, return.

```
    if (key < root->key)
```

```
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
```

- Recur left or right depending on key.

```
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            Node* temp = root->left ? root->left : root->right;
```

- Found node to delete:
    - If it has only one or no child:

```
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else
                *root = *temp; // Copy contents of child
            delete temp;
        }
        else {
            Node* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
```

- If it has two children, find in-order successor, copy its value, then delete successor.

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

- If tree had only one node.

```
    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);
```

- Update height and balance after deletion.

Rebalance if needed

```
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
```

```
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
```

Inorder Traversal

```
void inorder(Node* root, ofstream &outfile) {
    if (root != NULL) {
        inorder(root->left, outfile);
        outfile << root->key << " ";
        inorder(root->right, outfile);
    }
}
```

- Recursively traverses the tree in inorder and writes keys to file.

Main Function

```
int main() {
    Node* root = NULL;
```

- Start with an empty tree.

```
    ifstream infile("input.txt");
    int key;
    while (infile >> key) {
        root = insert(root, key);
    }
    infile.close();
```

- Read keys from input.txt, insert them into the AVL tree.

```
    root = insert(root, 30);        // Example manual insertion
    root = deleteNode(root, 50);    // Example manual deletion

    ofstream outfile("output.txt");
    inorder(root, outfile);
    outfile.close();
```

- Perform in-order traversal and write to output.txt.

```
    cout << "AVL Tree has been created and inorder traversal written to output.txt" << endl;
    return 0;
}
```

Suppose input.txt contains:

10 20 30 40 50 25

These numbers are inserted one-by-one into the AVL Tree.

After inserting:

10 → 20 → 30

This causes a **Right-Right (RR)** imbalance at node 10:

```
  10
   \
    20
      \
       30
```

So a **Left Rotation** is applied at node 10:

After rotation:

```
    20
   /  \
  10   30
```

Next insert:

→ 40 → 50

This causes **Right-Right imbalance** again at node 30:

```
    20
   /  \
  10   30
         \
          40
            \
             50
```

So we apply **Left Rotation** at node 30:

After balancing:

```
    20
```

```
   /  \
 10    40
      / \
    30  50
```

Now insert:

→ 25

This causes a **Right-Left imbalance** at node 20.

Subtree before fix:

```
     20
    /  \
  10    40
       / \
     30  50
    /
   25
```

Step 1: Right Rotation at 30
Step 2: Left Rotation at 20

Final balanced tree:

```
      30
     /  \
   20    40
  /  \     \
10   25     50
```

Final In-order Traversal Output (output.txt)

In-order traversal prints values in **sorted order**:

10 20 25 30 40 50

Explanation:

- **AVL Tree**: This is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

- **Functions**:
  - **Insert**: Inserts a new key into the AVL tree, ensuring that the tree remains balanced after the insertion.
  - **Delete**: Removes a key from the AVL tree, maintaining the balance.
  - **Inorder Traversal**: Traverses the AVL tree in in-order (left-root-right) and writes the output to a file.
- **File I/O**:
  - The elements are read from an input.txt file.
  - The resultant AVL tree is stored in an output.txt file in in-order traversal.

Input and Output:

- **Input**: A file named input.txt containing space-separated integers (e.g., 10 20 30 40 50 25).
- **Output**: A file named output.txt containing the in-order traversal of the constructed AVL tree.

How to Run:

- Place the input data into a file named input.txt.
- Compile and run the C++ program.
- Check output.txt for the in-order traversal of the AVL tree.

Example

Let's assume the input.txt file contains the following integers:

Copy code

10 20 30 40 50 25

Steps:

- **Initial AVL Tree Construction**:
  - The program will read these numbers from input.txt and insert them into the AVL tree.

  - The AVL tree will balance itself during the insertion of each element.
- **Additional Insertions**:
  - The program inserts 30 into the tree (though 30 is already present, AVL trees do not allow duplicate values, so no changes).
- **Deletion**:
  - The program deletes the node with the key 50.
- **In-order Traversal**:
  - The program writes the in-order traversal of the AVL tree to output.txt.

Resulting AVL Tree Structure (after all operations):

Given the input and operations, the AVL tree might look like this:

```
    30
   /  \
  20    40
 /  \     \
10    25    50
```

After deleting 50, the tree will be:

```
    30
   /  \
  20    40
```

```
 /  \
10    25
```

Output in output.txt:

The in-order traversal of the final AVL tree will produce the following sequence:

10 20 25 30 40

So, the content of output.txt will be:

10 20 25 30 40

This is the final output of the program.

Input/Output:

- **Input**: A file named input.txt containing space-separated integers

  10 20 30 40 50 25

  Given the input and operations, the AVL tree might look like this:

```
     30
    /  \
  20    40
 /  \     \
10    25    50
```

After deleting 50, the tree will be:

```
     30
    /  \
  20    40
 /  \
10    25
```

Output in output.txt:

The in-order traversal of the final AVL tree will produce the following sequence:

10 20 25 30 40

So, the content of output.txt will be:

10 20 25 30 40

This is the final output of the program.