Tamia Naeem
AI-004

Exercise

1. Complete implementation of AVL tree given in Example 01 by implementing Insert and Delete methods.

SOURCE CODE

```
#include <iostream>

using namespace std;

// Node class representing each node in the AVL tree

class Node {

public:

    int key;        // Key of the node

    Node* left;     // Left child

    Node* right;    // Right child

    int height;     // Height of the node for balancing


    // Constructor to initialize a new node

    Node(int k) {

        key = k;

        left = right = nullptr;

        height = 1; // New nodes are initialized with height 1

    }

};


// AVL tree class with insert, delete, and balance functions

class Tamia_004 {

private:

    Node* root; // Root of the AVL tree
```

```
// Helper function to get the height of a node
int height(Node* node) {
    return node ? node->height : 0;
}

// Helper function to calculate balance factor of a node
int balanceFactor(Node* node) {
    return node ? height(node->left) - height(node->right) : 0;
}

// Updates the height of a node based on the heights of its children
void updateHeight(Node* node) {
    node->height = 1 + max(height(node->left), height(node->right));
}

// Right rotation to balance nodes (used for Left-Left cases)
Node* rotateRight(Node* y) {
    Node* x = y->left;    // x becomes new root
    Node* T2 = x->right;  // T2 is temporarily stored

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

    // Update heights after rotation
    updateHeight(y);
    updateHeight(x);
```

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

// Left rotation to balance nodes (used for Right-Right cases)
Node* rotateLeft(Node* x) {
    Node* y = x->right;   // y becomes new root
    Node* T2 = y->left;   // T2 is temporarily stored

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

    // Update heights after rotation
    updateHeight(x);
    updateHeight(y);

    // Return the new root
    return y;
}

// Balances the given node if it becomes unbalanced
Node* balance(Node* node) {
    updateHeight(node);   // Update the node's height first
    int balance = balanceFactor(node);
```

```
    // Left heavy case
    if (balance > 1) {
        // Left-Right case
        if (balanceFactor(node->left) < 0)
            node->left = rotateLeft(node->left); // Left rotation on left child
        return rotateRight(node); // Right rotation on the current node
    }

    // Right heavy case
    if (balance < -1) {
        // Right-Left case
        if (balanceFactor(node->right) > 0)
            node->right = rotateRight(node->right); // Right rotation on right child
        return rotateLeft(node); // Left rotation on the current node
    }

    return node; // Node is balanced, no rotation needed
}

// Finds the node with the minimum key in a subtree
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
}
```

```
// Inserts a new key into the AVL tree and balances it
Node* insert(Node* node, int key) {
    // Perform standard BST insertion
    if (node == nullptr)
        return new Node(key);

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

    // Balance the node after insertion
    return balance(node);
}

// Deletes a node with the given key and balances the AVL tree
Node* deleteNode(Node* node, int key) {
    if (node == nullptr)
        return node;

    // Perform standard BST delete
    if (key < node->key)
        node->left = deleteNode(node->left, key);
    else if (key > node->key)
        node->right = deleteNode(node->right, key);
```

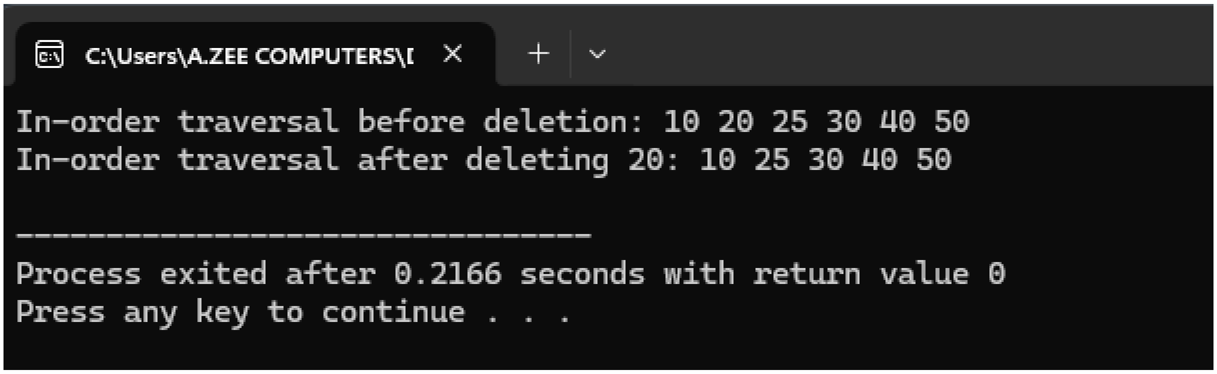
```
    else {
        // Node with only one child or no child
        if ((node->left == nullptr) || (node->right == nullptr)) {
            Node* temp = node->left ? node->left : node->right;

            // No child case
            if (temp == nullptr) {
                temp = node;
                node = nullptr;
            } else { // One child case
                *node = *temp; // Copy the contents of the non-empty child
            }
            delete temp;
        } else {
            // Node with two children: get the inorder successor
            Node* temp = minValueNode(node->right);
            node->key = temp->key; // Copy the inorder successor's key
            node->right = deleteNode(node->right, temp->key); // Delete successor
        }
    }

    if (node == nullptr)
        return node;

    // Balance the node after deletion
    return balance(node);
}
```

```
    // In-order traversal to print the AVL tree nodes
    void inOrder(Node* node) {
        if (!node) return;
        inOrder(node->left);           // Visit left subtree
        cout << node->key << " ";      // Print node's key
        inOrder(node->right);          // Visit right subtree
    }

public:
    // Constructor to initialize the AVL tree
    Tamia_004() {
        root = nullptr;
    }

    // Public method to insert a key in the AVL tree
    void insert(int key) {
        root = insert(root, key);
    }

    // Public method to delete a key from the AVL tree
    void deleteKey(int key) {
        root = deleteNode(root, key);
    }

    // Public method to perform in-order traversal
    void inOrder() {
```

```
        inOrder(root);
        cout << endl;
    }
};

// Main function to demonstrate AVL tree operations
int main() {
    Tamia_004 tree;

    // Insert nodes
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(20); // Duplicate insertion, ignored
    tree.insert(40);
    tree.insert(50);
    tree.insert(25);

    cout << "In-order traversal before deletion: ";
    tree.inOrder();

    // Delete a node and show in-order traversal again
    tree.deleteKey(20);
    cout << "In-order traversal after deleting 20: ";
    tree.inOrder();

    return 0;
```

```
}
```

OUTPUT



2. Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

```
0
/ \
-3 9
/ /
-10 5
```

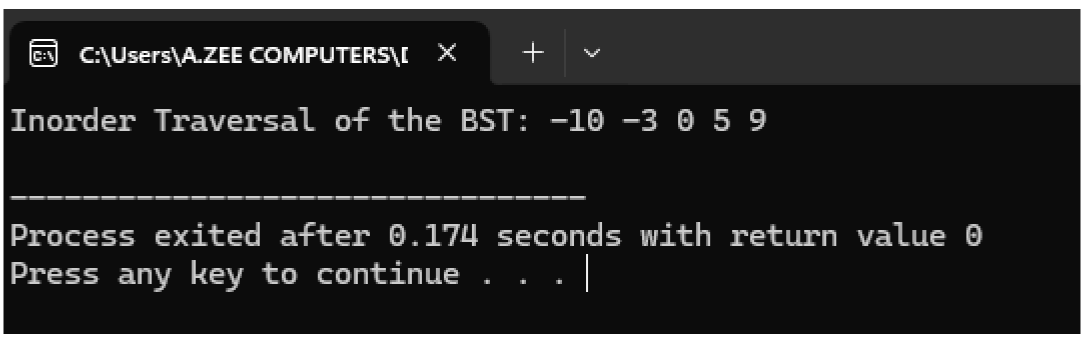
SOURCE CODE

```
#include <iostream>
#include <vector>
using namespace std;
class Node {
public:
    int value;
    Node* left;
    Node* right;
    Node(int val) {
        value = val;
```

```
        left = right = nullptr;
    }
};
class Tamia_004 {
private:
    // Helper function to convert sorted array to a height-balanced BST
    Node* sortedArrayToBST(const vector<int>& nums, int left, int right) {
        if (left > right) return nullptr;
        // Find the middle element
        int mid = left + (right - left) / 2;
        Node* node = new Node(nums[mid]);
        // Recursively build the left and right subtrees
        node->left = sortedArrayToBST(nums, left, mid - 1);
        node->right = sortedArrayToBST(nums, mid + 1, right);
        return node;
    }
    // Helper function for inorder traversal of the BST
    void inorderTraversal(Node* root) {
        if (root == nullptr) return;
        inorderTraversal(root->left);
        cout << root->value << " ";
        inorderTraversal(root->right);
    }

public:
    // Function to convert the array to BST and print inorder traversal
    void convertAndDisplayBST(const vector<int>& nums) {
```

```
        Node* root = sortedArrayToBST(nums, 0, nums.size() - 1);
        cout << "Inorder Traversal of the BST: ";
        inorderTraversal(root);
        cout << endl;
    }
};
int main() {
    vector<int> nums = {-10, -3, 0, 5, 9};
    Tamia_004 T;
    T.convertAndDisplayBST(nums);
    return 0;
}
```

OUTPUT



3. Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

-10 -> -3 -> 0 -> 5 -> 9

|

0

/ \

-3 9

/ /

-10 5

SOURCE CODE

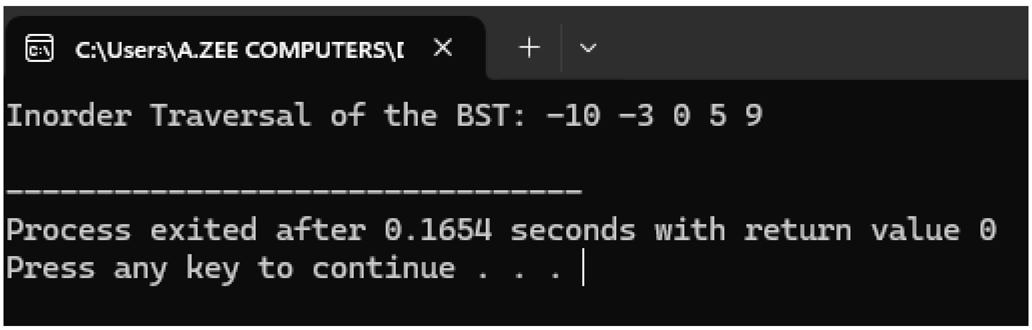
```
#include <iostream>
using namespace std;
class ListNode {
public:
  int value;
  ListNode* next;
  ListNode(int val) {
    value = val;
    next = nullptr;
  }
};
class Node {
public:
  int value;
  Node* left;
  Node* right;
  Node(int val) {
    value = val;
    left = right = nullptr;
  }
};
class Tamia_004 {
private:
  ListNode* head;
```

```
// Function to find the middle of the linked list using slow and fast pointers
ListNode* findMiddle(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
    ListNode* prev = nullptr;
    // Move slow by one step and fast by two steps to find the middle
    while (fast != nullptr && fast->next != nullptr) {
        fast = fast->next->next;
        prev = slow;
        slow = slow->next;
    }
    // Disconnect the left half of the list from the middle
    if (prev != nullptr) {
        prev->next = nullptr;
    }
    return slow;
}
// Function to convert the sorted linked list to a height-balanced BST
Node* sortedListToBST(ListNode* head) {
    if (head == nullptr) return nullptr;
    // Find the middle node (this becomes the root)
    ListNode* mid = findMiddle(head);
    // Create the root node for the BST
    Node* root = new Node(mid->value);
    // Base case: if the list has only one node, return the node as the root
    if (head == mid) return root;
    // Recursively construct the left and right subtrees
```

```
        root->left = sortedListToBST(head);  // Left half
        root->right = sortedListToBST(mid->next);  // Right half
        return root;
    }
    // Helper function to print the inorder traversal of the BST
    void inorderTraversal(Node* root) {
        if (root == nullptr) return;
        inorderTraversal(root->left);
        cout << root->value << " ";
        inorderTraversal(root->right);
    }

public:
    Tamia_004(ListNode* head) {
        this->head = head;
    }
    // Function to convert the linked list to BST and print its inorder traversal
    void convertAndDisplayBST() {
        Node* root = sortedListToBST(head);
        cout << "Inorder Traversal of the BST: ";
        inorderTraversal(root);
        cout << endl;
    }
};
int main() {
    ListNode* head = new ListNode(-10);
    head->next = new ListNode(-3);
```

```
    head->next->next = new ListNode(0);

    head->next->next->next = new ListNode(5);

    head->next->next->next->next = new ListNode(9);

    Tamia_004 T(head);

    T.convertAndDisplayBST();

    return 0;

}
```

OUTPUT



4. Given the root of a BST, find its diameter.

SOURCE CODE

```
#include <iostream>

using namespace std;

class Node {

public:

    int value;

    Node* left;

    Node* right;

    Node(int val) {

        value = val;

        left = right = nullptr;

    }

};
```

```
class Tamia_004 {
private:
   Node* root;
   // Helper function to calculate the height of a node
   int height(Node* node) {
      if (node == nullptr) return 0;
      return 1 + max(height(node->left), height(node->right));
   }
public:
   Tamia_004() {
      root = nullptr;
   }
   void setRoot(Node* node) {
      root = node;
   }
   int diameter(Node* node) {
      if (node == nullptr) return 0;

      // Get the height of left and right subtrees
      int leftHeight = height(node->left);
      int rightHeight = height(node->right);

      // Get the diameter of left and right subtrees
      int leftDiameter = diameter(node->left);
      int rightDiameter = diameter(node->right);

      // Return the maximum of the following:
```

```
        // 1. Diameter of left subtree
        // 2. Diameter of right subtree
        // 3. Height of left subtree + height of right subtree + 1
        return max(leftHeight + rightHeight + 1, max(leftDiameter, rightDiameter));
    }
};
int main() {
    Tamia_004 T;
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    // Setting root in Tamia_004 object
    T.setRoot(root);
    cout << "Diameter of the BST is: " << T.diameter(root) << endl;
    return 0;
}
```

OUTPUT