**NAME** :- Manish Shashikant Jadhav

**UID** :- 2023301005.

**BRANCH** :- Comps -B. **BRANCH:** B.

**EXPERIMENT 6:** Implement an ADT for storing an AVL Tree and performing given operations on it.

**SUBJECT** :- DS (DATA STRUCTURES).

**CODE** :-

#include <stdio.h>

#include <stdlib.h>

struct AVLNode {

    int data;

    struct AVLNode\* left;

    struct AVLNode\* right;

    int height;

};

int max(int a, int b) {

    return (a > b) ? a : b;

}

int getHeight(struct AVLNode\* node) {

    if (node == NULL) {

        return 0;

    }

    return node->height;

}

int getBalance(struct AVLNode\* node) {

    if (node == NULL) {

        return 0;

    }

    return getHeight(node->left) - getHeight(node->right);

}

struct AVLNode\* createNode(int data) {

    struct AVLNode\* newNode = (struct AVLNode\*)malloc(sizeof(struct AVLNode));

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    newNode->height = 1;  // New node is initially added at leaf, so its height is 1

    return newNode;

}

struct AVLNode\* rotateRight(struct AVLNode\* y) {

    struct AVLNode\* x = y->left;

    struct AVLNode\* T2 = x->right;

    x->right = y;

    y->left = T2;

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

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

    return x;

}

struct AVLNode\* rotateLeft(struct AVLNode\* x) {

    struct AVLNode\* y = x->right;

    struct AVLNode\* T2 = y->left;

    y->left = x;

    x->right = T2;

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

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

    return y;

}

struct AVLNode\* insert(struct AVLNode\* root, int data) {

    if (root == NULL) {

        return createNode(data);

    }

    if (data < root->data) {

        root->left = insert(root->left, data);

    } else if (data > root->data) {

        root->right = insert(root->right, data);

    } else {

        return root; // Duplicate data not allowed

    }

    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    int balance = getBalance(root);

    if (balance > 1 && data < root->left->data) {

        return rotateRight(root);

    }

    if (balance < -1 && data > root->right->data) {

        return rotateLeft(root);

    }

    if (balance > 1 && data > root->left->data) {

        root->left = rotateLeft(root->left);

        return rotateRight(root);

    }

    if (balance < -1 && data < root->right->data) {

        root->right = rotateRight(root->right);

        return rotateLeft(root);

    }

    return root;

}

struct AVLNode\* minValueNode(struct AVLNode\* node) {

    struct AVLNode\* current = node;

    while (current->left != NULL) {

        current = current->left;

    }

    return current;

}

struct AVLNode\* delete(struct AVLNode\* root, int data) {

    if (root == NULL) {

        return root;

    }

    if (data < root->data) {

        root->left = delete(root->left, data);

    } else if (data > root->data) {

        root->right = delete(root->right, data);

    } else {

        if (root->left == NULL || root->right == NULL) {

            struct AVLNode\* temp = root->left ? root->left : root->right;

            if (temp == NULL) {

                temp = root;

                root = NULL;

            } else {

                \*root = \*temp;

            }

            free(temp);

        } else {

            struct AVLNode\* temp = minValueNode(root->right);

            root->data = temp->data;

            root->right = delete(root->right, temp->data);

        }

    }

    if (root == NULL) {

        return root;

    }

    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0) {

        return rotateRight(root);

    }

    if (balance > 1 && getBalance(root->left) < 0) {

        root->left = rotateLeft(root->left);

        return rotateRight(root);

    }

    if (balance < -1 && getBalance(root->right) <= 0) {

        return rotateLeft(root);

    }

    if (balance < -1 && getBalance(root->right) > 0) {

        root->right = rotateRight(root->right);

        return rotateLeft(root);

    }

    return root;

}

void displayAVLTree(struct AVLNode\* root) {

    if (root != NULL) {

        displayAVLTree(root->left);

        printf("%d ", root->data);

        displayAVLTree(root->right);

    }

}

void freeAVLTree(struct AVLNode\* root) {

    if (root == NULL) {

        return;

    }

    freeAVLTree(root->left);

    freeAVLTree(root->right);

    free(root);

}

int main() {

    struct AVLNode\* root = NULL;

    root = insert(root, 28);

    root = insert(root, 9);

    root = insert(root, 13);

    root = insert(root, 2);

    root = insert(root, 5);

    root = insert(root, 19);

    root = insert(root, 10);

    printf("The structure of AVL Trees with given values is: ");

    displayAVLTree(root);

    printf("\n");

    printf("Height of the AVL Tree is: %d\n", getHeight(root));

    root = delete(root, 5);

    printf("AVL Tree after deleting 5 from the above Tree: ");

    displayAVLTree(root);

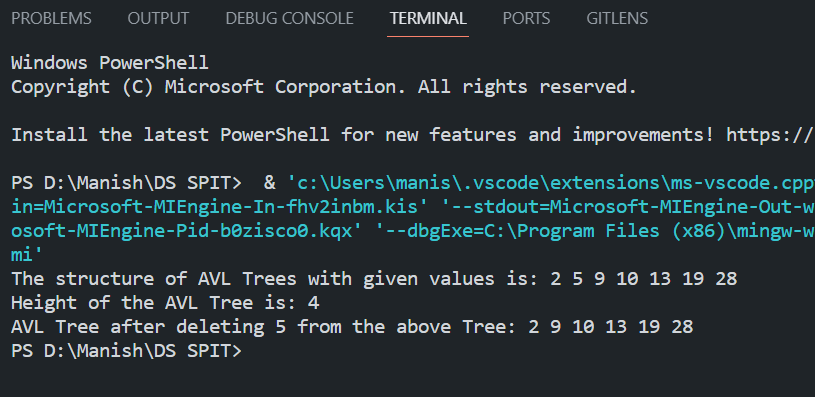
    printf("\n");

    freeAVLTree(root);

    return 0;

}

**Output:**



**Algorithm:**

**Structures:**

- Define a structure for AVLNode with fields: `data` (integer), `left` (pointer to AVLNode), `right` (pointer to AVLNode), and `height` (integer).

**Helper Functions:**

- `max(a, b)`: Returns the maximum of two integers `a` and `b`.

- `getHeight(node)`: Returns the height of the AVL tree starting from the given `node`.

- `getBalance(node)`: Returns the balance factor of the AVL tree at the given `node`.

**AVLNode Creation:**

- Create a function `createNode(data)` that allocates memory for an AVLNode, initializes its data, left, and right pointers to NULL, and sets its height to 1.

**Right Rotation:**

- Create a function `rotateRight(y)` that performs a right rotation at node `y`. It returns the new root of the rotated subtree.

**Left Rotation:**

- Create a function `rotateLeft(x)` that performs a left rotation at node `x`. It returns the new root of the rotated subtree.

**Insertion:**

- Create a function `insert(root, data)` for inserting a new node with `data` into the AVL tree rooted at `root`.

- If `root` is NULL, create a new node with the given data and return it.

- If `data` is less than `root->data`, insert it in the left subtree and update the height.

- If `data` is greater than `root->data`, insert it in the right subtree and update the height.

- Update the height of the current node.

- Calculate the balance factor of the current node.

- Perform appropriate rotations to balance the tree, if necessary.

- Return the new root of the subtree.

**Find Minimum Node:**

- Create a function `minValueNode(node)` that finds and returns the node with the minimum value in the given subtree rooted at `node`.

**Deletion:**

- Create a function `delete(root, data)` to delete a node with `data` from the AVL tree rooted at `root`.

- If `root` is NULL, return `root`.

- If `data` is less than `root->data`, delete it from the left subtree.

- If `data` is greater than `root->data`, delete it from the right subtree.

- If the node to be deleted has one or no child, replace it with the non-empty child or NULL.

- If the node to be deleted has two children, replace it with the in-order successor (node with the minimum value in the right subtree) and delete the in-order successor.

- Update the height of the current node.

- Calculate the balance factor of the current node.

- Perform appropriate rotations to balance the tree, if necessary.

- Return the new root of the subtree.

**Display AVL Tree:**

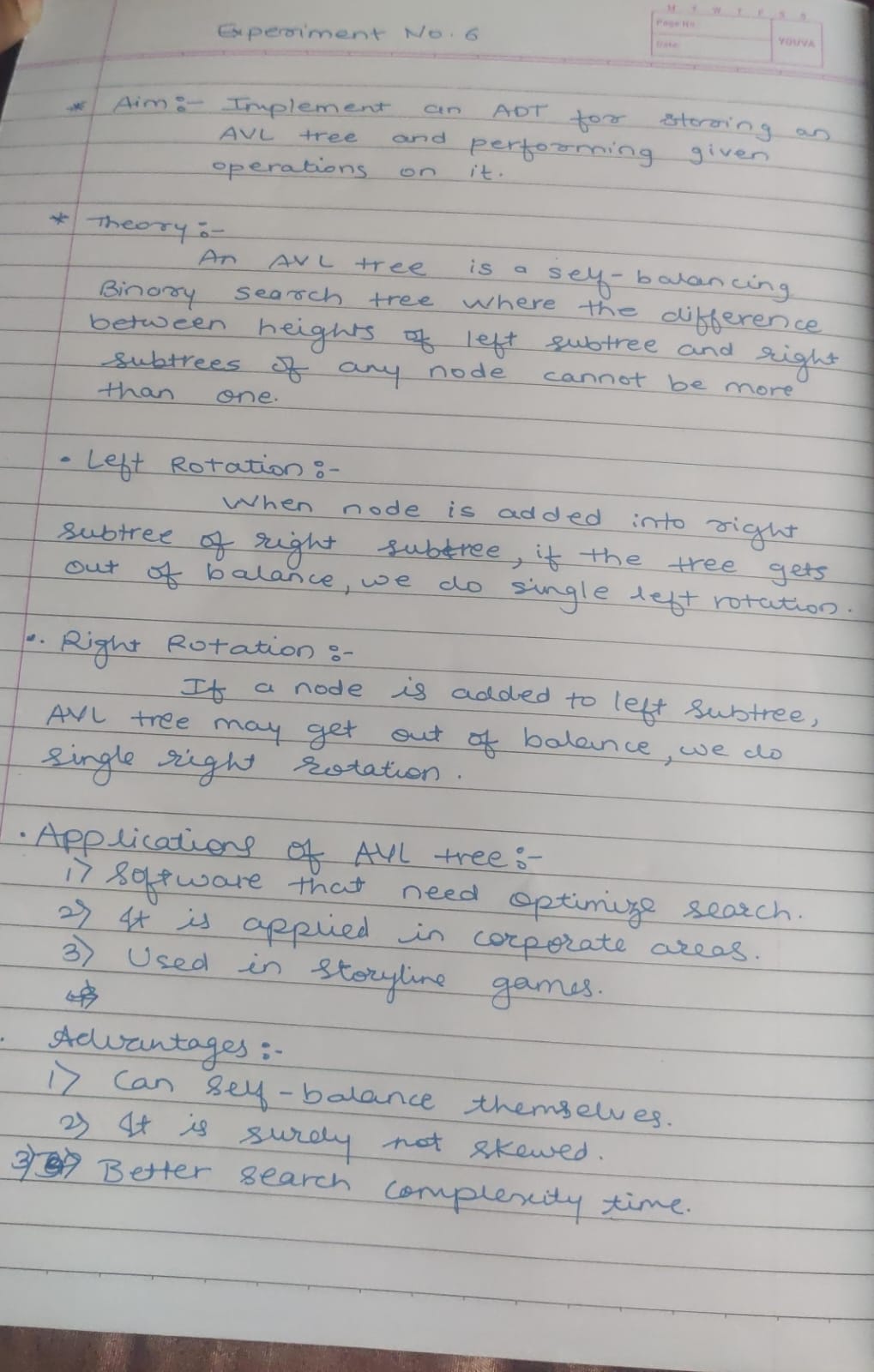
- Create a function `displayAVLTree(root)` to display the AVL tree in in-order traversal.

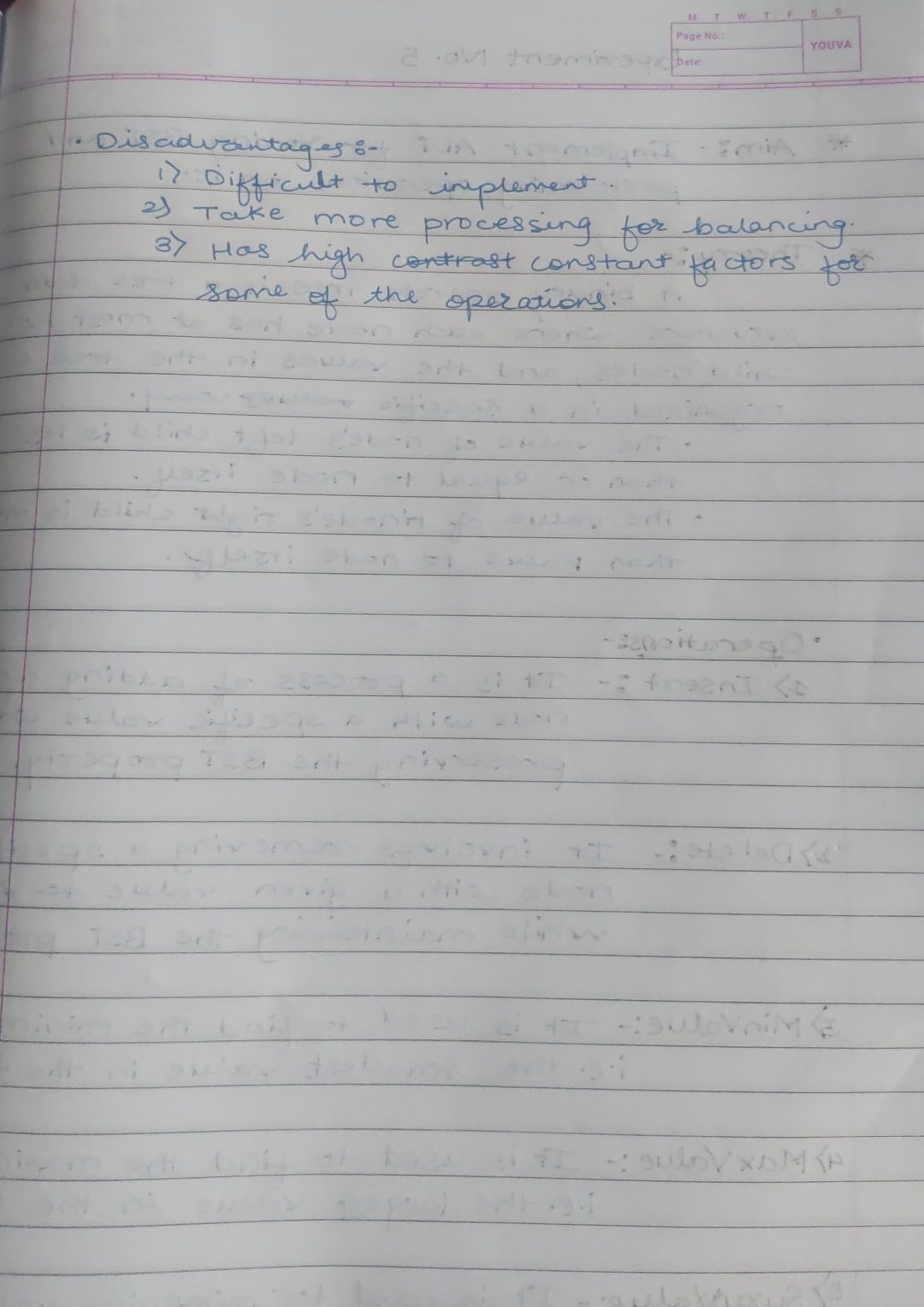
**Free AVL Tree:**

- Create a function `freeAVLTree(root)` to free the memory allocated for the AVL tree using post-order traversal.

**Main Function:**

- In the `main` function, initialize the AVL tree, insert nodes, display the tree, get the height, delete nodes, and free the memory.





**Conclusion:**

Hence, by completing this experiment I came to know about implement an ADT for storing an AVL Tree and performing given operations on it.