DAY 11

1. Avl tree

```
// AVL tree implementation in C

#include <stdio.h>

#include <stdlib.h>

// Create Node

struct Node {

  int key;

  struct Node *left;

  struct Node *right;

  int height;

};

int max(int a, int b);

// Calculate height

int height(struct Node *N) {

  if (N == NULL)

    return 0;
```

```
  return N->height;

}

int max(int a, int b) {

  return (a > b) ? a : b;

}

// Create a node

struct Node *newNode(int key) {

  struct Node *node = (struct Node *)

    malloc(sizeof(struct Node));

  node->key = key;

  node->left = NULL;

  node->right = NULL;

  node->height = 1;

  return (node);

}

// Right rotate
```

```
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;
  return x;
}
// Left rotate
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct 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;

}

// Get the balance factor

int getBalance(struct Node *N) {

  if (N == NULL)

    return 0;

  return height(N->left) - height(N->right);

}

// Insert node

struct Node *insertNode(struct Node *node, int key) {

  // Find the correct position to insertNode the node and insertNode it

  if (node == NULL)

    return (newNode(key));

  if (key < node->key)

    node->left = insertNode(node->left, key);
```

```
else if (key > node->key)
  node->right = insertNode(node->right, key);
else
  return node;
// Update the balance factor of each node and
// Balance the tree
node->height = 1 + max(height(node->left),
            height(node->right));
int balance = getBalance(node);
if (balance > 1 && key < node->left->key)
  return rightRotate(node);
if (balance < -1 && key > node->right->key)
  return leftRotate(node);
if (balance > 1 && key > node->left->key) {
  node->left = leftRotate(node->left);
  return rightRotate(node);
```

```
  }

  if (balance < -1 && key < node->right->key) {

    node->right = rightRotate(node->right);

    return leftRotate(node);

  }

  return node;

}

struct Node *minValueNode(struct Node *node) {

  struct Node *current = node;

  while (current->left != NULL)

    current = current->left;

  return current;

}

// Delete a nodes

struct Node *deleteNode(struct Node *root, int key) {

  // Find the node and delete it
```

```
if (root == NULL)

  return root;

if (key < root->key)

  root->left = deleteNode(root->left, key);

else if (key > root->key)

  root->right = deleteNode(root->right, key);

else {

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

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

    if (temp == NULL) {

      temp = root;

      root = NULL;

    } else

      *root = *temp;

    free(temp);

  } else {
```

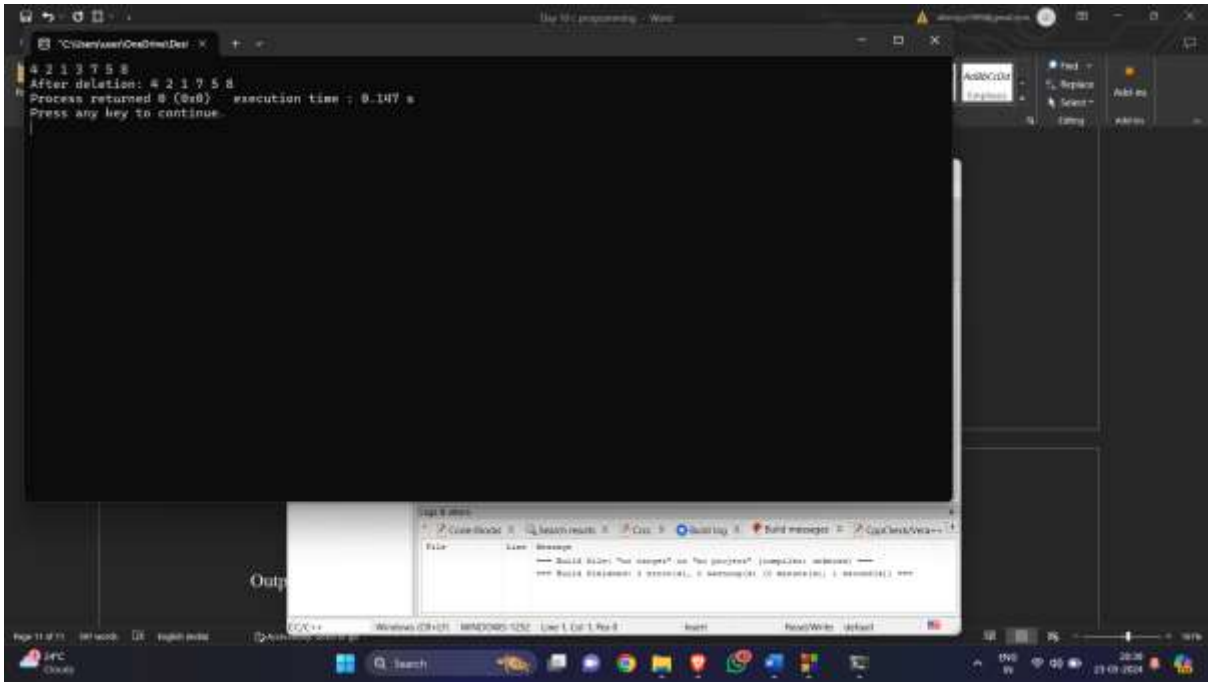
```
    struct Node *temp = minValueNode(root->right);

    root->key = temp->key;

    root->right = deleteNode(root->right, temp->key);

  }

}

if (root == NULL)

  return root;

// Update the balance factor of each node and

// balance the tree

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

            height(root->right));

int balance = getBalance(root);

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;

}

// Print the tree

void printPreOrder(struct Node *root) {

  if (root != NULL) {

    printf("%d ", root->key);

    printPreOrder(root->left);

    printPreOrder(root->right);
```

```
  }

}

int main() {

  struct Node *root = NULL;

  root = insertNode(root, 2);

  root = insertNode(root, 1);

  root = insertNode(root, 7);

  root = insertNode(root, 4);

  root = insertNode(root, 5);

  root = insertNode(root, 3);

  root = insertNode(root, 8);

  printPreOrder(root);

  root = deleteNode(root, 3);

  printf("\nAfter deletion: ");

  printPreOrder(root);

  return 0;
```

}

Output:



2. Binary tree

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// Definition of Treenode structure
typedef struct Treenode {
    int data;
    struct Treenode *left, *right;
} Treenode;
// Definition of Tree structure
typedef struct {
    Treenode *root;
} Tree;
// Function to create a new tree node
Treenode* newTreenode(int data) {
    Treenode* node = (Treenode*)malloc(sizeof(Treenode));
    node->data = data;
    node->left = node->right = NULL;
```

```
    return node;
}
// Function to calculate the height of the tree
int height(Treenode *root) {
    if (root == NULL)
        return 0;
    int left_height = height(root->left);
    int right_height = height(root->right);
    return (left_height > right_height ? left_height : right_height) + 1;
}
// Function to calculate the number of columns required to print the tree
int getcol(int h) {
    if (h == 1)
        return 1;
    return getcol(h - 1) + getcol(h - 1) + 1;
}
// Recursive function to fill the 2D array with tree data
void printTree(int **M, Treenode *root, int col, int row, int height) {
    if (root == NULL)
        return;
    M[row][col] = root->data;
    printTree(M, root->left, col - pow(2, height - 2), row + 1, height - 1);
    printTree(M, root->right, col + pow(2, height - 2), row + 1, height - 1);
}

// Function to print the tree
void TreePrinter(Tree tree) {
    int h = height(tree.root);
    int col = getcol(h);
    int **M = (int **)malloc(h * sizeof(int *));
    for (int i = 0; i < h; i++) {
        M[i] = (int *)malloc(col * sizeof(int));
```

```
        for (int j = 0; j < col; j++) {
            M[i][j] = 0; // Initialize the 2D array with 0s
        }
    }
    printTree(M, tree.root, col / 2, 0, h);
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < col; j++) {
            if (M[i][j] == 0)
                printf("  ");
            else
                printf("%d ", M[i][j]);
        }
        printf("\n");
    }
    for (int i = 0; i < h; i++) {
        free(M[i]);
    }
    free(M);
}
// Function to insert nodes in the tree
Treenode* insertLevelOrder(int arr[], Treenode* root, int i, int n) {
    if (i < n) {
        Treenode *temp = newTreenode(arr[i]);
        root = temp;
        root->left = insertLevelOrder(arr, root->left, 2 * i + 1, n);
        root->right = insertLevelOrder(arr, root->right, 2 * i + 2, n);
    }
    return root;
}
int main() {
    Tree myTree;
    myTree.root = NULL;
```

```
    int n;
    printf("Enter the number of nodes in the tree: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    printf("Enter the nodes in level order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    myTree.root = insertLevelOrder(arr, myTree.root, 0, n);
    printf("Tree structure:\n");
    TreePrinter(myTree);
    free(arr);
    return 0;
}
```

Output: