DAY 10

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)
```

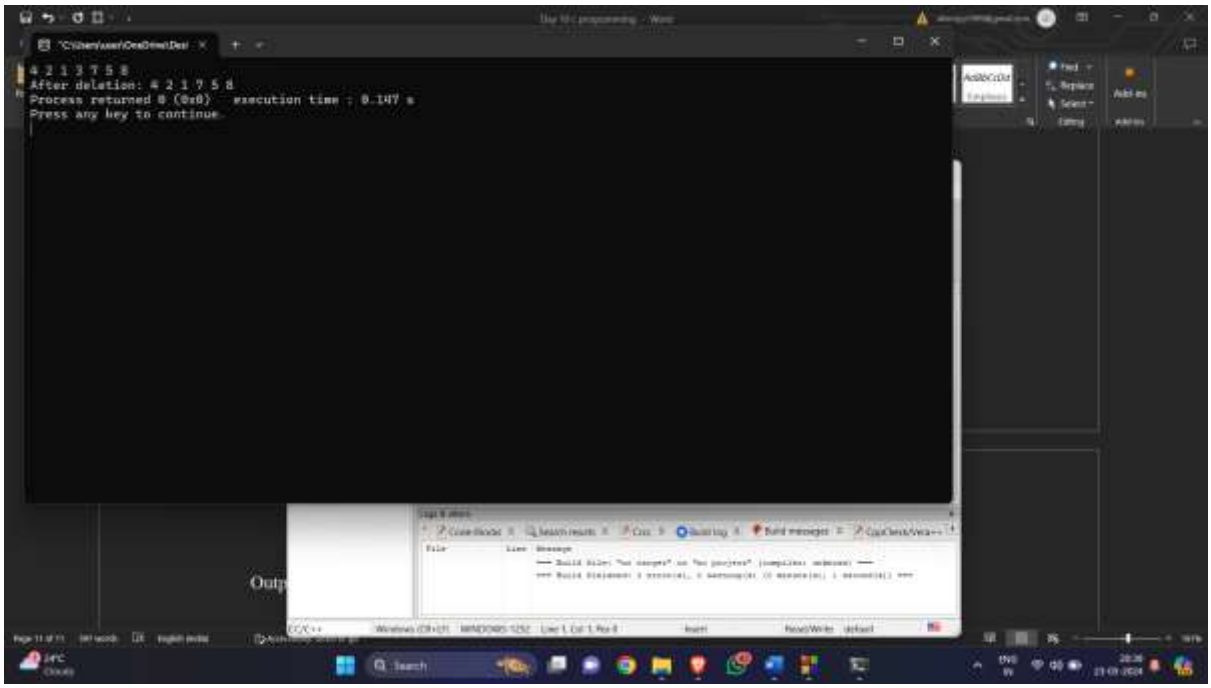
```
  return root;

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:


4 2 1 3 7 5 8
After deletion: 4 2 1 7 5 8
Process returned 0 (0x0)   execution time : 0.147 s
Press any key to continue.