AVL TREE

INSERTION:

CODE:

```
#INCLUDE <STDIO.H>

#INCLUDE <STDLIB.H>

TYPEDEF STRUCT AVLNODE {
    INT KEY;
    STRUCT AVLNODE *LEFT;
    STRUCT AVLNODE *RIGHT;
    INT HEIGHT;
} AVLNODE;

INT HEIGHT(AVLNODE *NODE) {
    IF (NODE == NULL) RETURN 0;
    RETURN NODE->HEIGHT;
}

INT GETBALANCE(AVLNODE *NODE) {
    IF (NODE == NULL) RETURN 0;
    RETURN HEIGHT(NODE->LEFT) - HEIGHT(NODE->RIGHT);
}

AVLNODE* CREATENODE(INT KEY) {
    AVLNODE *NODE = (AVLNODE *)MALLOC(SIZEOF(AVLNODE));
    NODE->KEY = KEY;
    NODE->LEFT = NULL;
    NODE->RIGHT = NULL;
    NODE->HEIGHT = 1;
    RETURN NODE;
}

AVLNODE* RIGHTROTATE(AVLNODE *Y) {
    AVLNODE *X = Y->LEFT;
    AVLNODE *T2 = X->RIGHT;
    X->RIGHT = Y;
    Y->LEFT = T2;
    Y->HEIGHT = 1 + (HEIGHT(Y->LEFT) > HEIGHT(Y->RIGHT) ? HEIGHT(Y->LEFT) : HEIGHT(Y->RIGHT));
    X->HEIGHT = 1 + (HEIGHT(X->LEFT) > HEIGHT(X->RIGHT) ? HEIGHT(X->LEFT) : HEIGHT(X->RIGHT));
```

```
    RETURN X;
}
AVLNODE* LEFTROTATE(AVLNODE *X) {
    AVLNODE *Y = X->RIGHT;
    AVLNODE *T2 = Y->LEFT;
    Y->LEFT = X;
    X->RIGHT = T2;
    X->HEIGHT = 1 + (HEIGHT(X->LEFT) > HEIGHT(X->RIGHT) ? HEIGHT(X->LEFT) : HEIGHT(X->RIGHT));
    Y->HEIGHT = 1 + (HEIGHT(Y->LEFT) > HEIGHT(Y->RIGHT) ? HEIGHT(Y->LEFT) : HEIGHT(Y->RIGHT));
    RETURN Y;
}
AVLNODE* INSERT(AVLNODE *NODE, INT KEY) {
    IF (NODE == NULL) RETURN CREATENODE(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;
NODE->HEIGHT = 1 + (HEIGHT(NODE->LEFT) > HEIGHT(NODE->RIGHT) ? 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;
```

```
}

VOID INORDER(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        INORDER(ROOT->LEFT);
        PRINTF("%D ", ROOT->KEY);
        INORDER(ROOT->RIGHT);
    }
}

VOID FREETREE(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        FREETREE(ROOT->LEFT);
        FREETREE(ROOT->RIGHT);
        FREE(ROOT);
    }
}

INT MAIN() {
    AVLNODE *ROOT = NULL;
    ROOT = INSERT(ROOT, 10);
    ROOT = INSERT(ROOT, 20);
    ROOT = INSERT(ROOT, 30);
    ROOT = INSERT(ROOT, 15);
    PRINTF("INORDER TRAVERSAL OF THE AVL TREE IS:\N");
    INORDER(ROOT);
    PRINTF("\N");
    FREETREE(ROOT);
    RETURN 0;
}
```

OUTPUT:

```
INORDER TRAVERSAL OF THE AVL TREE IS:
10 15 20 30
```

DELETION:

CODE:

```
#INCLUDE <STDIO.H>
#INCLUDE <STDLIB.H>
```

```
TYPEDEF STRUCT AVLNODE {
    INT KEY;
    STRUCT AVLNODE *LEFT;
    STRUCT AVLNODE *RIGHT;
    INT HEIGHT;
} AVLNODE;
INT HEIGHT(AVLNODE *NODE) {
    IF (NODE == NULL) RETURN 0;
    RETURN NODE->HEIGHT;
}
INT GETBALANCE(AVLNODE *NODE) {
    IF (NODE == NULL) RETURN 0;
    RETURN HEIGHT(NODE->LEFT) - HEIGHT(NODE->RIGHT);
}
AVLNODE* CREATENODE(INT KEY) {
    AVLNODE *NODE = (AVLNODE *)MALLOC(SIZEOF(AVLNODE));
    NODE->KEY = KEY;
    NODE->LEFT = NULL;
    NODE->RIGHT = NULL;
    NODE->HEIGHT = 1;
    RETURN NODE;
}
AVLNODE* RIGHTROTATE(AVLNODE *Y) {
    AVLNODE *X = Y->LEFT;
    AVLNODE *T2 = X->RIGHT;
    X->RIGHT = Y;
    Y->LEFT = T2;
    Y->HEIGHT = 1 + (HEIGHT(Y->LEFT) > HEIGHT(Y->RIGHT) ? HEIGHT(Y->LEFT) : HEIGHT(Y->RIGHT));
    X->HEIGHT = 1 + (HEIGHT(X->LEFT) > HEIGHT(X->RIGHT) ? HEIGHT(X->LEFT) : HEIGHT(X->RIGHT));
    RETURN X;
}
AVLNODE* LEFTROTATE(AVLNODE *X) {
    AVLNODE *Y = X->RIGHT;
    AVLNODE *T2 = Y->LEFT;
    Y->LEFT = X;
```

```
    X->RIGHT = T2;
    X->HEIGHT = 1 + (HEIGHT(X->LEFT) > HEIGHT(X->RIGHT) ? HEIGHT(X->LEFT) : HEIGHT(X->RIGHT));
    Y->HEIGHT = 1 + (HEIGHT(Y->LEFT) > HEIGHT(Y->RIGHT) ? HEIGHT(Y->LEFT) : HEIGHT(Y->RIGHT));
    RETURN Y;
}
AVLNODE* INSERT(AVLNODE *NODE, INT KEY) {
    IF (NODE == NULL) RETURN CREATENODE(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;
    NODE->HEIGHT = 1 + (HEIGHT(NODE->LEFT) > HEIGHT(NODE->RIGHT) ? 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;
}
AVLNODE* MINVALUENODE(AVLNODE* NODE) {
    AVLNODE* CURRENT = NODE;
    WHILE (CURRENT->LEFT != NULL)
        CURRENT = CURRENT->LEFT;
    RETURN CURRENT;
```

```
}
AVLNODE* DELETE(AVLNODE *ROOT, INT KEY) {
   IF (ROOT == NULL) RETURN ROOT;
   IF (KEY < ROOT->KEY)
      ROOT->LEFT = DELETE(ROOT->LEFT, KEY);
   ELSE IF (KEY > ROOT->KEY)
      ROOT->RIGHT = DELETE(ROOT->RIGHT, KEY);
   ELSE {
     IF ((ROOT->LEFT == NULL) || (ROOT->RIGHT == NULL)) {
       AVLNODE *TEMP = ROOT->LEFT ? ROOT->LEFT : ROOT->RIGHT;
       IF (TEMP == NULL) {
          TEMP = ROOT;
          ROOT = NULL;
       } ELSE
          *ROOT = *TEMP;
       FREE(TEMP);
     } ELSE {
       AVLNODE *TEMP = MINVALUENODE(ROOT->RIGHT);E
       ROOT->KEY = TEMP->KEY;
       ROOT->RIGHT = DELETE(ROOT->RIGHT, TEMP->KEY);
     }
}
IF (ROOT == NULL) RETURN ROOT;
ROOT->HEIGHT = 1 + (HEIGHT(ROOT->LEFT) > HEIGHT(ROOT->RIGHT) ? 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;
}
VOID INORDER(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        INORDER(ROOT->LEFT);
        PRINTF("%D ", ROOT->KEY);
        INORDER(ROOT->RIGHT);
    }
}
VOID FREETREE(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        FREETREE(ROOT->LEFT);
        FREETREE(ROOT->RIGHT);
        FREE(ROOT);
    }
}
INT MAIN() {
    AVLNODE *ROOT = NULL;
    ROOT = INSERT(ROOT, 10);
    ROOT = INSERT(ROOT, 20);
    ROOT = INSERT(ROOT, 30);
    ROOT = INSERT(ROOT, 15);
    PRINTF("INORDER TRAVERSAL BEFORE DELETION:\N");
    INORDER(ROOT);
    PRINTF("\N");
    ROOT = DELETE(ROOT, 20);
    PRINTF("INORDER TRAVERSAL AFTER DELETION:\N");
    INORDER(ROOT);
    PRINTF("\N");
    FREETREE(ROOT);
    RETURN 0;
}
```

OUTPUT:

INORDER TRAVERSAL BEFORE DELETION:

10 15 20 30

INORDER TRAVERSAL AFTER DELETION:

10 15 30

SEARCH:

CODE:

```
#INCLUDE <STDIO.H>

#INCLUDE <STDLIB.H>

TYPEDEF STRUCT AVLNODE {

   INT KEY;

   STRUCT AVLNODE *LEFT;

   STRUCT AVLNODE *RIGHT;

   INT HEIGHT;

} AVLNODE;

INT HEIGHT(AVLNODE *NODE) {

   IF (NODE == NULL) RETURN 0;

   RETURN NODE->HEIGHT;

}

AVLNODE* CREATENODE(INT KEY) {

   AVLNODE *NODE = (AVLNODE *)MALLOC(SIZEOF(AVLNODE));

   NODE->KEY = KEY;

   NODE->LEFT = NULL;

   NODE->RIGHT = NULL;

   NODE->HEIGHT = 1;

   RETURN NODE;

}

AVLNODE* RIGHTROTATE(AVLNODE *Y) {

   AVLNODE *X = Y->LEFT;

AVLNODE *T2 = X->RIGHT;

X->RIGHT = Y;

Y->LEFT = T2;

Y->HEIGHT = 1 + (HEIGHT(Y->LEFT) > HEIGHT(Y->RIGHT) ? HEIGHT(Y->LEFT) : HEIGHT(Y->RIGHT));

X->HEIGHT = 1 + (HEIGHT(X->LEFT) > HEIGHT(X->RIGHT) ? HEIGHT(X->LEFT) : HEIGHT(X->RIGHT));

RETURN X;
```

```
}

AVLNode* leftRotate(AVLNode *x) {
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
    return y;
}

AVLNode* insert(AVLNode *node, int key) {
    if (node == NULL) return createNode(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;
    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));
    int balance = height(node->left) - height(node->right);
    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;
}

AVLNode* search(AVLNode *root, int key) {
```

```
    IF (ROOT == NULL || ROOT->KEY == KEY)
        RETURN ROOT;
    IF (KEY > ROOT->KEY)
        RETURN SEARCH(ROOT->RIGHT, KEY);
    RETURN SEARCH(ROOT->LEFT, KEY);
}
VOID INORDER(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        INORDER(ROOT->LEFT);
        PRINTF("%D ", ROOT->KEY);
        INORDER(ROOT->RIGHT);
    }
}
VOID FREETREE(AVLNODE *ROOT) {
    IF (ROOT != NULL) {
        FREETREE(ROOT->LEFT);
        FREETREE(ROOT->RIGHT);
        FREE(ROOT);
    }
}
INT MAIN() {
    AVLNODE *ROOT = NULL;
    ROOT = INSERT(ROOT, 10);
    ROOT = INSERT(ROOT, 20);
    ROOT = INSERT(ROOT, 30);
    ROOT = INSERT(ROOT, 15);
    PRINTF("INORDER TRAVERSAL OF THE AVL TREE:\N");
    INORDER(ROOT);
    PRINTF("\N");
    INT KEYSTOSEARCH[] = {10, 15, 20, 25};
    FOR (INT I = 0; I < 4; I++) {
        AVLNODE *RESULT = SEARCH(ROOT, KEYSTOSEARCH[I]);
        IF (RESULT != NULL)
            PRINTF("KEY %D FOUND IN THE AVL TREE.\N", KEYSTOSEARCH[I]);
        ELSE
```

```
        printf("Key %d not found in the AVL tree.\n", keysToSearch[i]);
    }

    freeTree(root);

    return 0;
}
```

OUTPUT:

```
Inorder traversal of the AVL tree:
10 15 20 30

Key 10 found in the AVL tree.

Key 15 found in the AVL tree.

Key 20 found in the AVL tree.

Key 25 not found in the AVL tree.
```