**AVL TREE PROGRAMS**

**Insert,delete,search  a node in AVL tree**

**i) 18, 8,12, 5, 11,17,4**

**INSERTION**

// Utility function to get maximum of two integers

int max(int a, int b) {

    return (a > b)? a : b;

}

// Right rotate

struct Node \*rightRotate(struct Node \*y) {

    struct Node \*x = y->left;

    struct Node \*T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

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

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

    // Return new root

    return x;

}

// Left rotate

struct Node \*leftRotate(struct Node \*x) {

    struct Node \*y = x->right;

    struct Node \*T2 = y->left;

    // Perform rotation

    y->left = x;

    x->right = T2;

    // Update heights

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

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

    // Return new root

    return y;

}

// Get balance factor of node N

int getBalance(struct Node \*N) {

    if (N == NULL)

        return 0;

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

}

// Insert a node

struct Node\* insert(struct Node\* node, int key) {

    // 1. Perform the normal BST insertion

    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; // Equal keys not allowed in BST

    // 2. Update height of this ancestor node

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

    // 3. Get the balance factor of this ancestor node

    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

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

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node->left->key) {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    // Right Left Case

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

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

        return leftRotate(node);

    }

    // return the (unchanged) node pointer

    return node;

}

**DELETION**

// Find the node with the smallest key

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

    struct Node\* current = node;

    // Loop down to find the leftmost leaf

    while (current->left != NULL)

        current = current->left;

    return current;

}

// Delete a node

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

    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)

        return root;

    // If the key to be deleted is smaller than the root's key,

    // then it lies in left subtree

    if (key < root->key)

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

    // If the key to be deleted is greater than the root's key,

    // then it lies in right subtree

    else if(key > root->key)

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

    // if key is same as root's key, then this is the node to be deleted

    else {

        // node with only one child or no child

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

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

            // No child case

            if (temp == NULL) {

                temp = root;

                root = NULL;

            }

            else // One child case

                \*root = \*temp; // Copy the contents of the non-empty child

            free(temp);

        }

        else {

            // node with two children: Get the inorder successor (smallest

            // in the right subtree)

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

            // Copy the inorder successor's data to this node

            root->key = temp->key;

            // Delete the inorder successor

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

        }

    }

    // If the tree had only one node then return

    if (root == NULL)

        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

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

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether

    // this node became unbalanced)

    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case

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

        return rightRotate(root);

    // Left Right Case

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

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

        return rightRotate(root);

    }

    // Right Right Case

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

        return leftRotate(root);

    // Right Left Case

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

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

        return leftRotate(root);

    }

    return root;

}

**SEARCH**

// Search for a key in the AVL tree

struct Node\* search(struct Node\* root, int key) {

    // Base Cases: root is null or key is present at root

    if (root == NULL || root->key == key)

        return root;

    // Key is greater than root's key

    if (root->key < key)

        return search(root->right, key);

    // Key is smaller than root's key

    return search(root->left, key);

}

**2) 25,20,36, 10,22, 30,40,12,28,38,48**

**INSERTION**

// Insert a node

struct Node\* insert(struct Node\* node, int key) {

    // 1. Perform the normal BST insertion

    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; // Equal keys are not allowed in BST

    // 2. Update height of this ancestor node

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

    // 3. Get the balance factor of this ancestor node

    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

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

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node->left->key) {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    // Right Left Case

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

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

        return leftRotate(node);

    }

    // return the (unchanged) node pointer

    return node;

}

**DELETION AND SEARCH**

// Find the node with the smallest key

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

    struct Node\* current = node;

    // Loop down to find the leftmost leaf

    while (current->left != NULL)

        current = current->left;

    return current;

}

// Delete a node

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

    // STEP 1: PERFORM STANDARD BST DELETE

    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;

    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;

}

// Search for a key in the AVL tree

struct Node\* search(struct Node\* root, int key) {

    if (root == NULL || root->key == key)

        return root;

    if (root->key < key)

        return search(root->right, key);

    return search(root->left, key);

}

**3)1,2,3,4,5,6,7,8**

**INSERTION**

// Insert a node

struct Node\* insert(struct Node\* node, int key) {

    // 1. Perform the normal BST insertion

    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; // Equal keys are not allowed in BST

    // 2. Update height of this ancestor node

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

    // 3. Get the balance factor of this ancestor node

    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

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

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node->left->key) {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    // Right Left Case

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

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

        return leftRotate(node);

    }

    // return the (unchanged) node pointer

    return node;

}

**DELETION  AND SEARCH**

// Find the node with the smallest key

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

    struct Node\* current = node;

    // Loop down to find the leftmost leaf

    while (current->left != NULL)

        current = current->left;

    return current;

}

// Delete a node

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

    // STEP 1: PERFORM STANDARD BST DELETE

    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;

    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;

}

// Search for a key in the AVL tree

struct Node\* search(struct Node\* root, int key) {

    if (root == NULL || root->key == key)

        return root;

    if (root->key < key)

        return search(root->right, key);

    return search(root->left, key);

}