1.Give a program for AVL tree insertion in c language

Program:

```
#include <stdio.h>

#include <stdlib.h>


// An AVL tree node

struct Node {

   int key;

   struct Node *left;

   struct Node *right;

   int height;

};


// A utility function to get the height of the tree

int height(struct Node *N) {

   if (N == NULL)

      return 0;

   return N->height;

}


// A utility function to get maximum of two integers

int max(int a, int b) {

   return (a > b)? a : b;

}


// A utility function to allocate a new node with the given key and NULL left and right pointers.

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;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
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;
}

// A utility function to left rotate subtree rooted with x
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);
}

// Recursive function to insert a key in the subtree rooted with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // 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 to check whether this node became unbalanced
    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;
}

// A utility function to print preorder traversal of the tree.
void preOrder(struct Node *root) {
```

```
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
    printf("\n");
}

// Driver program to test the above functions
int main() {
    struct Node *root = NULL;
    int choice, key;

    while (1) {
        printf("1. Insert\n2. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                printf("Preorder traversal after insertion: ");
                preOrder(root);
                break;
            case 2:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
```

```
    return 0;
}
```

Input:

Enter your choice: 1

Enter key to insert: 2

Output:

Preorder traversal after insertion: 2

2. Give a program for AVL tree deletion in c language

Program:

```c
#include <stdio.h>
#include <stdlib.h>

// An AVL tree node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get the height of the tree
int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get the maximum of two integers
int max(int a, int b) {
```

```
    return (a > b)? a : b;
}

// A utility function to allocate a new node with the given key and NULL left and right pointers.
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;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
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;
}

// A utility function to left rotate subtree rooted with x
```

```
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);
}

// Recursive function to insert a key in the subtree rooted with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
```

```
    node->left = insert(node->left, key);
else if (key > node->key)
    node->right = insert(node->right, key);
else // Equal keys are not allowed in BST
    return node;

// 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 to check whether this node became unbalanced
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;
}

// A utility function to find the node with the minimum key value found in that tree
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;
}

// Recursive function to delete a node with given key from subtree with given root. It returns root of the modified subtree.
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;
```

```
}

// A utility function to print preorder traversal of the tree.
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
    printf("\n");
}

// Driver program to test the above functions
int main() {
    struct Node *root = NULL;
    int choice, key;

    while (1) {
        printf("1. Insert\n2. Delete\n3. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                printf("Preorder traversal after insertion: ");
                preOrder(root);
                break;
            case 2:
                printf("Enter key to delete: ");
```

```
            scanf("%d", &key);
            root = deleteNode(root, key);
            printf("Preorder traversal after deletion: ");
            preOrder(root);
            break;
        case 3:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Input:

Enter your choice: 1

Enter key to insert: 2

Output:

Preorder traversal after insertion: 2

3. **Give a program for AVL tree search in c language.**

Program:

```
#include <stdio.h>
#include <stdlib.h>

// An AVL tree node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
```

```
};

// A utility function to get the height of the tree
int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get the maximum of two integers
int max(int a, int b) {
    return (a > b)? a : b;
}

// A utility function to allocate a new node with the given key and NULL left and right pointers.
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;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
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;
}

// A utility function to left rotate subtree rooted with x
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);
}

// Recursive function to insert a key in the subtree rooted with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // 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 to check whether this node became unbalanced
    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;
}

// A utility function to find the node with the minimum key value found in that tree
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;
}

// Recursive function to delete a node with given key from subtree with given root. It returns root of the modified subtree.
```

```
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;
}

// Function to search 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);
}

// A utility function to print preorder traversal of the tree.
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
```

```
        preOrder(root->left);
        preOrder(root->right);
    }
    printf("\n");
}

// Driver program to test the above functions
int main() {
    struct Node *root = NULL;
    int choice, key;

    while (1) {
        printf("1. Insert\n2. Delete\n3. Search\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                printf("Preorder traversal after insertion: ");
                preOrder(root);
                break;
            case 2:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                root = deleteNode(root, key);
                printf("Preorder traversal after deletion: ");
                preOrder(root);
                break;
            case 3:
```

```
            printf("Enter key to search: ");
            scanf("%d", &key);
            struct Node* result = search(root, key);
            if (result != NULL)
                printf("Key %d found in the AVL tree.\n", key);
            else
                printf("Key %d not found in the AVL tree.\n", key);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Input:

Enter your choice: 1

Enter key to insert: 4

Output:

Preorder traversal after insertion: 4