Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 10

AIM: Implementation of AVL tree using C programming.

THEORY:

An AVL Tree is a type of self-balancing binary search tree (BST) where the difference in heights between the left and right subtrees of any node (called the balance factor) is at most 1. This ensures that the tree remains balanced, leading to efficient search, insertion, and deletion operations, all with a time complexity of O(log n).

In this experiment, we aim to implement the following key operations of an AVL tree:

1. Insertion: When a new node is added, the tree might become unbalanced. After each insertion, the balance factor of each node is checked. If it exceeds the allowed limit, rotations (single or double) are applied to restore balance.
2. Deletion: When a node is deleted, the balance of the tree might again be affected. After the deletion, the tree is rebalanced using rotations similar to insertion.
3. Rotations: AVL trees use two types of rotations—left rotation and right rotation. These rotations help in rebalancing the tree whenever the height difference exceeds the limit. In some cases, double rotations (left-right or right-left) are required.
4. Tree Traversal: The tree can be explored using inorder, preorder, or postorder traversal methods to retrieve the nodes in specific sequences.

Through this experiment, we explore how AVL trees maintain balance after dynamic operations like insertion and deletion, ensuring efficient performance compared to regular binary search trees.

CODE:

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

// Node structure
struct node {
   int data, ht;
   struct node* left;
   struct node* right;
};

// Function prototypes
struct node* create_node(int);
struct node* insert(struct node*, int);
```

```
struct node* delete(struct node*, int);
int height(struct node*);
struct node* search(struct node*, int);
void inorder(struct node*);
void preorder(struct node*);
void postorder(struct node*);
void show(struct node*);
int count_nodes(struct node*);

int main() {
   struct node* root = NULL;
   int choice, data;
   char cont = 'y';

   while (cont == 'y' || cont == 'Y') {
      printf("\n\n------- AVL TREE --------\n");
        printf("1. Insert\n2. Delete\n3. Search\n4. Show Tree (Inorder, Preorder, Postorder)\n5. Node
Count\n6. Exit\n");
      printf("\nEnter Your Choice: ");
      scanf("%d", &choice);

      switch(choice) {
         case 1:
            printf("\nEnter data: ");
            scanf("%d", &data);
            root = insert(root, data);
            break;
         case 2:
            printf("\nEnter data: ");
            scanf("%d", &data);
            root = delete(root, data);
            break;
         case 3:
            printf("\nEnter data to search: ");
            scanf("%d", &data);
            struct node* result = search(root, data);
            printf(result ? "\nNode found!\n" : "\nNode not found!\n");
            break;
         case 4:
            show(root);
            break;
         case 5:
            printf("\nTotal nodes: %d\n", count_nodes(root));
```

```
            break;
        case 6:
            return 0;
        default:
            printf("\nInvalid choice!\n");
      }
      printf("\nContinue? (y/n): ");
      scanf(" %c", &cont);
   }
   return 0;
}

// Create a new node
struct node* create_node(int data) {
   struct node* new_node = (struct node*) malloc(sizeof(struct node));
   new_node->data = data;
   new_node->left = new_node->right = NULL;
   new_node->ht = 1;  // New node is initially at height 1
   return new_node;
}

// Height of a node
int height(struct node* n) {
   return (n == NULL) ? 0 : n->ht;
}

// Insert a node into the AVL tree
struct node* insert(struct node* root, int data) {
   if (root == NULL) {
      return create_node(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }
   root->ht = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));
   return root;
}

// Delete a node from the AVL tree
struct node* delete(struct node* root, int data) {
   if (root == NULL) return NULL;
```

```
    if (data < root->data) {
        root->left = delete(root->left, data);
    } else if (data > root->data) {
        root->right = delete(root->right, data);
    } else {
        if (root->left == NULL || root->right == NULL) {
            struct node* temp = root->left ? root->left : root->right;
            free(root);
            return temp;
        } else {
            struct node* temp = root->right;
            while (temp->left) temp = temp->left;
            root->data = temp->data;
            root->right = delete(root->right, temp->data);
        }
    }
    return root;
}

// Search for a node
struct node* search(struct node* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }
    if (key < root->data) {
        return search(root->left, key);
    } else {
        return search(root->right, key);
    }
}

// Inorder traversal
void inorder(struct node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Preorder traversal
void preorder(struct node* root) {
    if (root == NULL) return;
```

```
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Postorder traversal
void postorder(struct node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

// Show tree in different orders
void show(struct node* root) {
    printf("\nInorder: ");
    inorder(root);
    printf("\nPreorder: ");
    preorder(root);
    printf("\nPostorder: ");
    postorder(root);
    printf("\n");
}

// Count nodes in the tree
int count_nodes(struct node* root) {
    if (root == NULL) return 0;
    return 1 + count_nodes(root->left) + count_nodes(root->right);
}
```

OUTPUT:

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 1

Enter data: 67

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 1

Enter data: 34

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 1

Enter data: 90

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 4

Inorder: 34 67 90
Preorder: 67 34 90
Postorder: 34 90 67

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 3

Enter data to search: 90

Node found!

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 5

Total nodes: 3

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 2

Enter data: 90

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 4

Inorder: 34 67
Preorder: 67 34
Postorder: 34 67

Continue? (y/n): y
```

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Show Tree (Inorder, Preorder, Postorder)
5. Node Count
6. Exit

Enter Your Choice: 6


=== Code Execution Successful ===
```

CONCLUSION:

This simplified AVL Tree program includes insertion, deletion, and search operations, along with inorder, preorder, and postorder traversals. Rotation operations were removed, and a show function was added for easier visualization of the tree structure.