Assignment no:-6 SYCOA124

→1] AVL tree:-
An AVL tree is also a self balancing tree and also known as height balanced tree. Where the difference of ~~every parent nod~~ every subtree of a parent node is either (1,0,-1). If at any instance during the insertion of a new node the height factor is get disturbed the tree adjust itself and make the tree balanced.
The steps to overcome on distrubed structure are
1] calculate height factor 2] Take the parent root
3] Observe the disturbtion sequence (LL, RR, LR, RL)
4] according to disturbtion sequence run required function.

2] Operations on AVL tree:-
→(a) searching :- searching in avl tree is same as in binary tree.
(b) deletion :- It is same as a binary tree.
(c) traversal :- Traversal are same as binary tree.
(d) insertion :- one after other node is inserted if tree gets disturbed it is get balanced by the required functions.

Algorithm for searching:
Step 1:- Take the input from user.
Step 2:- Traverse the tree and compare the data at every node with input data.
Step 3:- If record is found display the msg record ~~for~~ found and break the searching function there.
Step 4:- If record not found display the msg record not found.

Time complexity:-
The searching function of avl tree has a time complexity of O(log n).

Conclusion:-
The concept of height balanced tree is implemented succesfully.

CODE:

```
#include <iostream>
using namespace std;
typedef struct node
{
    int data;
    int height;
    node *left;
    node *right;
} node;
node *root = NULL;
class tree
{
public:
    int getheight(node *n)
    {
        if (n == NULL)
            return 0;
        return n->height;
    }
    int max(int a, int b)
    {
        return (a > b) ? a : b;
    }
    node *createnode(int data)
    {
        node *nn = new node;
        nn->data = data;
        nn->left = NULL;
        nn->right = NULL;
        nn->height = 1;
        return nn;
    }
    int getbalancefactor(node *n)
    {
        if (n == NULL)
            return 0;
        return (getheight(n->left) - getheight(n->right));
    }
    node *rightrotate(node *y)
    {
        node *x = y->left;
        node *t2 = x->right;

        x->right = y;
        y->left = t2;
        y->height = max(getheight(y->right), getheight(y->left));
        x->height = max(getheight(x->right), getheight(x->left));
```

```
    return x;
}

node *leftrotate(node *x)
{
    node *y = x->right;
    node *t2 = y->left;

    y->left = x;
    x->right = t2;
    y->height = max(getheight(y->right), getheight(y->left));
    x->height = max(getheight(x->right), getheight(x->left));
    return y;
}

node *create(node *node, int data)
{
    if (node == NULL)
        return createnode(data);
    if (data < node->data)
        node->left = create(node->left, data);
    else if (data > node->data)
        node->right = create(node->right, data);
    else // Equal keys not allowed
        return node;
    int bf = getbalancefactor(node);
    if (bf > 1 && data < node->left->data)
        return rightrotate(node);

    // Right Right Case
    if (bf < -1 && data > node->right->data)
        return leftrotate(node);
    // Left Right Case
    if (bf > 1 && data > node->left->data)
    {
        node->left = leftrotate(node->left);
        return rightrotate(node);
    }

    // Right Left Case
    if (bf < -1 && data < node->right->data)
    {
        node->right = rightrotate(node->right);
        return leftrotate(node);
    }
    return node;
}
void inorder(node *root)
```

```
    {
        if (root != NULL)
        {
            inorder(root->left);
            cout << root->data << " ";
            inorder(root->right);
        }
    }

    void display()
    {
        int ch;
        node *temp = root;

            cout << "DISPLAYING ELEMENTS IN ASCENDING ORDER(inorder
traversal):";
            inorder(temp);

    }
    void search()
    {
        int d;
        cout << "ENTER DATA TO BE SEARCHED:";
        cin >> d;
        node *temp = root, *parent;
        while (temp != NULL)
        {
            if (temp->data == d)
            {
                break;
            }
            else
            {
                parent = temp;
                if (d < temp->data)
                {
                    temp = temp->left;
                }
                else
                {
                    temp = temp->right;
                }
            }
        }
        if (temp == NULL)
        {
            cout << "ELEMENT NOT FOUND\n";
        }
```

```
        else
        {
            cout << "ELEMENT FOUND\n";
        }
    }
};
int main()
{
    int ch, data;
    tree t;
    while (1)
    {
        cout << "\nENTER:\n1.CREATE\n2.DISPLAY\n3.SEARCH\n4.EXIT\nCHOICE:";
        cin >> ch;
        switch (ch)
        {
        case 1:
            cout << "ENTER NODE VALUE:";
            cin >> data;
            root = t.create(root, data);
            break;
        case 2:
            t.display();
            break;
        case 3:
            t.search();
            break;
        case 4:
            return 0;
        default:
            cout << "INVALID INPUT!!!!";
            break;
        }
    }
}
```

OUTPUT:

```
ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:1
ENTER NODE VALUE:5

ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:1
ENTER NODE VALUE:4

ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:1
ENTER NODE VALUE:6

ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
```

```
CHOICE:2
DISPLAYING ELEMENTS IN ASCENDING ORDER(inorder traversal):4 5 6
ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:1
ENTER NODE VALUE:3

ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:1
ENTER NODE VALUE:2

ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:2
DISPLAYING ELEMENTS IN ASCENDING ORDER(inorder traversal):2 3 4 5 6
ENTER:
1.CREATE
2.DISPLAY
3.SEARCH
4.EXIT
CHOICE:4
PS D:\program\secondyear>
```