**Implementation of AVL Tree**

**Aim:**

To implement and execute AVL tree and to record the output.

**Algorithm:**

**1. Right Rotation (RR)**

* Set p as the right child of T.
* Make T the left child of p.
* Set the right child of T to NULL.
* Return p.

**2. Left Rotation (LL)**

* Set p as the left child of T.
* Make T the right child of p.
* Set the left child of T to NULL.
* Return p.

**3. Right-Left Rotation (RL)**

* Set p as the right child of T.
* Set the right child of T to the left child of p.
* Set the left child of p to NULL.
* Set the right child of the new right child of T to p.
* Perform right rotation on T (RR).
* Return the result.

**4. Left-Right Rotation (LR)**

* Set p as the left child of T.
* Set the left child of T to the right child of p.
* Set the right child of p to NULL.
* Set the left child of the new left child of T to p.
* Perform left rotation on T (LL).
* Return the result.

**5. Rotation Based on Balance Factor**

* If balance factor (bf) > 1:
* If left child has left child, perform LL rotation.
* Else, perform LR rotation.
* If bf < -1:
* If right child has right child, perform RR rotation.
* Else, perform RL rotation.
* Return the rotated subtree.

**6. Calculate Height (ht)**

* If rt is NULL, return 0.
* Calculate height of left subtree (lh).
* Calculate height of right subtree (rh).
* Return 1 + max(lh, rh).

**7. Balance the AVL Tree (balance)**

* Balance left subtree.
* Balance right subtree.
* Calculate heights and balance factor.
* Rotate based on balance factor.
* Return balanced tree.

**Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node Node;

struct Node {

int data;

Node\* l;

Node\* r;

};

Node\* ins(Node\* rt, int d) {

if (rt == NULL) {

Node\* n = (Node\*)malloc(sizeof(Node));

n->data = d;

n->l = NULL;

n->r = NULL;

return n;

}

if (d < rt->data)

rt->l = ins(rt->l, d);

else if (d > rt->data)

rt->r = ins(rt->r, d);

return rt;

}

Node\* find(Node\* rt, int d) {

if (rt == NULL || rt->data == d)

return rt;

if (d < rt->data)

return find(rt->l, d);

else

return find(rt->r, d);

}

Node\* min(Node\* rt) {

Node\* cur = rt;

while (cur && cur->l != NULL)

cur = cur->l;

return cur;

}

Node\* del(Node\* rt, int d) {

if (rt == NULL)

return rt;

if (d < rt->data) {

rt->l = del(rt->l, d);

} else if (d > rt->data) {

rt->r = del(rt->r, d);

} else {

if (rt->l == NULL) {

Node\* tmp = rt->r;

free(rt);

return tmp;

} else if (rt->r == NULL) {

Node\* tmp = rt->l;

free(rt);

return tmp;

}

Node\* tmp = min(rt->r);

rt->data = tmp->data;

rt->r = del(rt->r, tmp->data);

}

return rt;

}

Node\* RR(Node\* T) {

Node\* p = T->r;

p->l = T;

T->r = NULL;

return p;

}

Node\* LL(Node\* T) {

Node\* p = T->l;

p->r = T;

T->l = NULL;

return p;

}

Node\* RL(Node\* T) {

Node\* p = T->r;

T->r = p->l;

p->l = NULL;

T->r->r = p;

return RR(T);

}

Node\* LR(Node\* T) {

Node\* p = T->l;

T->l = p->r;

p->r = NULL;

T->l->l = p;

return LL(T);

}

Node\* rot(Node\* rt, int bf) {

if (bf > 1) {

if (rt->l->l != NULL)

return LL(rt);

else

return LR(rt);

} else if (bf < -1) {

if (rt->r->r != NULL)

return RR(rt);

else

return RL(rt);

}

return rt;

}

int ht(Node\* rt) {

if (rt == NULL)

return 0;

int lh = ht(rt->l);

int rh = ht(rt->r);

return 1 + (lh > rh ? lh : rh);

}

Node\* balance(Node\* rt) {

if (rt != NULL) {

rt->l = balance(rt->l);

rt->r = balance(rt->r);

int lh = ht(rt->l);

int rh = ht(rt->r);

int bf = lh - rh;

rt = rot(rt, bf);

}

return rt;

}

void disp(Node\* rt) {

if (rt == NULL)

return;

printf("%d ", rt->data);

if (rt->l != NULL || rt->r != NULL) {

printf("{ ");

disp(rt->l);

printf(", ");

disp(rt->r);

printf("} ");

}

}

int main() {

int ch = 1, val;

Node\* rt = (Node\*)malloc(sizeof(Node));

rt->l = NULL;

rt->r = NULL;

printf("Enter the data for root node: ");

scanf("%d", &rt->data);

printf("\nOperations:\n 1) Insert\n 2) Delete\n 3) Find\n 4) Display\nPress 0 to exit\n");

while (ch) {

printf("\nEnter choice: ");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter node data to be attached: ");

scanf("%d", &val);

rt = ins(rt, val);

rt = balance(rt);

break;

case 2:

printf("Enter node data to be deleted: ");

scanf("%d", &val);

rt = del(rt, val);

rt = balance(rt);

break;

case 3:

printf("Enter node data to be searched: ");

scanf("%d", &val);

Node\* fnd = find(rt, val);

if (fnd)

printf("\nElement found at address %p\n", fnd);

else

printf("\nElement not found\n");

break;

case 4:

disp(rt);

printf("\n");

break;

case 0:

printf("Operation terminated\n");

break;

default:

printf("Invalid operation\n");

}

}

return 0;

}

**Output:**

Enter the data for root node: 10

Operations:

1) Insert

2) Delete

3) Find

4) Display

Press 0 to exit

Enter choice: 1

Enter node data to be attached: 9

Enter choice: 1

Enter node data to be attached: 20

Enter choice: 1

Enter node data to be attached: 223

Enter choice: 1

Enter node data to be attached: 26

Enter choice: 4

10 { 9 , 26 { 20 , 223 } }

Enter choice: 2

Enter node data to be deleted: 223

Enter choice: 3

Enter node data to be searched: 26

Element found at address 0x6e4b40

Enter choice: 4

10 { 9 , 26 { 20 , } }

Enter choice: 0

Operation terminated

**Result:**

Thus the code implemented and executed successfully.