Implementation of AVL Trees

PROGRAM:

```
# include<stdio.h>
# include<malloc.h>
# define F 0
# define T 1

typedef struct node
{
        int data;
        int bf;
        struct  node *left;
        struct  node *right;
}NODE;

NODE *INSERT(int , NODE *, int *);
void DISPLAY(NODE *, int );
NODE *Balance_Right_heavy(NODE *, int *);
NODE *Balance_Left_heavy(NODE *, int *);
NODE *DELETE(NODE *, NODE *, int *);
NODE *Delete_Element(NODE *, int , int *);

/* Function main */

void main()
{
        int h;
        int data ;
        char choice;
        NODE *Tree = (NODE *)malloc(sizeof(NODE));
        Tree =  NULL;
        clrscr();
        printf("\n IMPLEMENTATION OF AVL TREE");
        printf("\n --------------------------\n");
        //choice = getchar();
        while(choice != 'b')
        {
                fflush(stdin);
                printf("\n Input Information of the node: ");
                scanf("%d", &data);
                Tree =INSERT(data, Tree, &h);
                printf("\n Tree is:\n");
                DISPLAY(Tree, 1);
                fflush(stdin);
                printf("\n Input choice 'b' to break:");
                choice = getch();
        }
        fflush(stdin);
        while(1)
        {
                printf("\n Input choice '0' to break:");
                printf("\n Input the key value want to deleted is:");
                scanf("%d", &data);
                if (data == 0)
                    break;
                Tree = Delete_Element(Tree, data, &h);
                printf("\n Tree is:\n");
                DISPLAY(Tree, 1);
        }
}


/* Function to insert an element into tree */

NODE *  INSERT(int data, NODE *Parent, int *h)
{
        NODE *node1;
        NODE *node2;
        if(!Parent)
        {
                Parent = (NODE *) malloc(sizeof(NODE));
                Parent->data = data;
                Parent->left = NULL;
                Parent->right = NULL;
                Parent->bf = 0;
                *h = T;
                return (Parent);
        }

        if(data < Parent->data)
        {
                Parent->left =INSERT(data, Parent->left, h);
                if(*h)
                /* Left branch has grown higher */
                {
                    switch(Parent->bf)
                    {
                    case 1: /* Right heavy */
                        Parent->bf = 0;
                        *h = F;
                        break;
                    case 0: /* Balanced tree */
                        Parent->bf = -1;
                        break;
                    case -1: /* Left heavy */
                        node1 = Parent->left;
                        if(node1->bf == -1)
                        {
                            printf("\n Left to Left Rotation\n");
                            Parent->left= node1->right;
                            node1->right = Parent;
                            Parent->bf = 0;
                            Parent = node1;
                        }
                        else
                        {
                            printf("\n Left to right rotation\n");
                            node2 = node1->right;
                            node1->right = node2->left;
                            node2->left = node1;
                            Parent->left = node2->right;
                            node2->right = Parent;
                            if(node2->bf == -1)
                                Parent->bf = 1;
                            else
                                Parent->bf = 0;
                            if(node2->bf == 1)
                                node1->bf = -1;
                            else
                                node1->bf = 0;
                            Parent = node2;
                        }

                        Parent->bf = 0;
                        *h = F;
                    }
                }
        }

        if(data > Parent->data)
        {
                Parent->right =INSERT(data, Parent->right, h);
                if(*h)
                /* Right branch has grown higher */
                {
                    switch(Parent->bf)
                    {
                    case -1: /* Left heavy */
                        Parent->bf = 0;
                        *h = F;
                        break;
                    case 0: /* Balanced tree */
                        Parent->bf = 1;
                        break;

                    case 1: /* Right heavy */
                        node1 = Parent->right;
                        if(node1->bf == 1)
                        {
                            printf("\n Right to Right Rotation\n");
                            Parent->right= node1->left;
                            node1->left = Parent;
                            Parent->bf = 0;
                            Parent = node1;
                        }
                        else
                        {
                            printf("\n Right to Left Rotation\n");
                            node2 = node1->left;
                            node1->left = node2->right;
                            node2->right = node1;
                            Parent->right = node2->left;
                            node2->left = Parent;

                            if(node2->bf == 1)
                                Parent->bf = -1;
                            else
                                Parent->bf = 0;
                            if(node2->bf == -1)
                                node1->bf = 1;
                            else
                                node1->bf = 0;
                            Parent = node2;
                        }

                        Parent->bf = 0;
                        *h = F;
                    }
                }
        }
        return(Parent);
    }
```

```
NODE * Delete_Element(NODE *Parent, int data, int *h)
{
    NODE *Temp;
    if(Parent==NULL)
    {
        printf("\n Information does not exist.");
        return(Parent);
    }
    else
    {
        if (data < Parent->data )
        {
            Parent->left = Delete_Element(Parent->left, data, h);
            if(*h)
                Parent = Balance_Right_heavy(Parent, h);
        }
        else
            if(data > Parent->data)
            {
                Parent->right = Delete_Element(Parent->right, data, h);
                if(*h)
                    Parent = Balance_Left_heavy(Parent, h);
            }
            else
            {
                Temp= Parent;
                if(Temp->right == NULL)
                {
                    Parent = Temp->left;
                    *h = T;
                    free(Temp);
                }
                else
                    if(Temp->left == NULL)
                    {
                        Parent = Temp->right;
                        *h = T;
                        free(Temp);
                    }
                    else
                    {
                        Temp->left = DELETE(Temp->left, Temp,
h);
                        if(*h)
```

```
                            Parent =
Balance_Right_heavy(Parent, h);
                    }
            }
    }
    return(Parent);
}

/* Balancing Right heavy */

NODE * Balance_Right_heavy(NODE *Parent, int *h)
{
    NODE *node1, *node2;

    switch(Parent->bf)
    {
    case -1:
        Parent->bf = 0;
        break;

    case 0:
        Parent->bf = 1;
        *h= F;
        break;

    case 1: /* Rebalance */
        node1 = Parent->right;
        if(node1->bf >= 0)
        {
            printf("\n Right to Right Rotation\n");
            Parent->right= node1->left;
            node1->left = Parent;
            if(node1->bf == 0)
            {
                Parent->bf = 1;
                node1->bf = -1;
                *h = F;
            }
            else
            {
                Parent->bf = node1->bf = 0;
            }
            Parent = node1;
        }
        else
        {
```

```
                        printf("\n Right to Left Rotation\n");
                        node2 = node1->left;
                        node1->left = node2->right;
                        node2->right = node1;
                        Parent->right = node2->left;
                        node2->left = Parent;

                        if(node2->bf == 1)
                                Parent->bf = -1;
                        else
                                Parent->bf = 0;
                        if(node2->bf == -1)
                                node1->bf = 1;
                        else
                                node1->bf = 0;
                        Parent = node2;
                        node2->bf = 0;
                }
        }
        return(Parent);
}

/* Balancing Left heavy */

NODE * Balance_Left_heavy(NODE *Parent, int *h)
{
        NODE *node1, *node2;

        switch(Parent->bf)
        {
        case 1:
                Parent->bf = 0;
                break;

        case 0:
                Parent->bf = -1;
                *h= F;
                break;

        case -1: /*  Rebalance */
                node1 = Parent->left;
                if(node1->bf <= 0)
                {
                        printf("\n Left to Left Rotation\n");
                        Parent->left= node1->right;
                        node1->right = Parent;
```

```
            if(node1->bf == 0)
            {
                Parent->bf = -1;
                node1->bf = 1;
                *h = F;
            }
            else
            {
                Parent->bf = node1->bf = 0;
            }
            Parent = node1;
        }
        else
        {
            printf("\n Left to Right Rotation\n");
            node2 = node1->right;
            node1->right = node2->left;
            node2->left = node1;
            Parent->left = node2->right;
            node2->right = Parent;

            if(node2->bf == -1)
                Parent->bf = 1;
            else
                Parent->bf = 0;

            if(node2->bf == 1)
                node1->bf = -1;
            else
                node1->bf = 0;
            Parent = node2;
            node2->bf = 0;
        }
    }
    return(Parent);
}

/* Replace the node at which key is found with last right key of a left child */

NODE * DELETE(NODE *R, NODE *Temp, int *h)
{
    NODE *Dnode = R;
    if( R->right != NULL)
    {
        R->right = DELETE(R->right, Temp, h);
        if(*h)
```

```
                R = Balance_Left_heavy(R, h);
        }
        else
        {
            Dnode = R;
            Temp->data = R->data;
            R = R->left;
            free(Dnode);
            *h = T;
        }
        return(R);
}
/* Delete the key element from the tree */




/* DISPLAY function */

void DISPLAY(NODE *Tree,int Level)
{
        int i;
        if (Tree)
        {
            DISPLAY(Tree->right, Level+1);
            printf("\n");
            for (i = 0; i < Level; i++)
                printf("   ");
            printf("%d", Tree->data);
            DISPLAY(Tree->left, Level+1);
        }
}
```

OUTPUT

```
 IMPLEMENTATION OF AVL TREE
 -------------------------

 Input Information of the node:
1

 Tree is:
   1
 Input choice 'b' to break:
 Input Information of the node: 2

 Tree is:

      2
   1
 Input choice 'b' to break:
 Input Information of the node: 3

 Right to Right Rotation

 Tree is:

      3
   2
      1
 Input choice 'b' to break:
 Input Information of the node: 4

 Tree is:

         4
      3
   2
      1
 Input choice 'b' to break:
 Input Information of the node: 5

 Right to Right Rotation

 Tree is:

         5
      4
         3
   2
      1
  Input choice 'b' to break:
 Input Information of the node: 6

 Right to Right Rotation

 Tree is:
       6
     5
   4
       3
     2
       1
  Input choice 'b' to break:
  Input Information of the node: 7

  Right to Right Rotation

  Tree is:

       7
     6
       5
   4
       3
     2
       1
  Input choice 'b' to break:
  Input Information of the node: 8

  Tree is:

         8
       7
     6
       5
   4
       3
     2
       1
  Input choice 'b' to break:
 Input Information of the node: 15

 Right to Right Rotation

  Tree is:

         15
       8
         7
     6
       5
   4
```

```
      3
    2
      1
Input choice 'b' to break:
Input Information of the node: 14

Right to Right Rotation

Tree is:

      15
        14
    8
        7
      6
        5
  4
      3
    2
      1
Input choice 'b' to break:
Input Information of the node: 13

Left to Left Rotation

Tree is:

        15
      14
        13
    8
        7
      6
        5
  4
      3
    2
      1
Input choice 'b' to break:
Input Information of the node: 12

Right to Right Rotation

Tree is:

      15
    14
      13
        12
  8
        7
      6
        5
    4
        3
      2
        1
Input choice 'b' to break:
Input Information of the node: 11

Left to Left Rotation

Tree is:

      15
    14
        13
      12
        11
  8
        7
      6
        5
    4
        3
      2
        1
Input choice 'b' to break:
Input Information of the node: 10

 Left to Left Rotation

 Tree is:

         15
       14
         13
     12
       11
         10
   8
         7
      6
        5
    4
        3
      2
        1
 Input choice 'b' to break:
 Input Information of the node: 9

 Left to Left Rotation

 Tree is:

        15
      14
        13
    12
        11
      10
        9
  8
        7
      6
        5
    4
        3
      2
        1
 Input choice 'b' to break:


Input choice '0' to break:
Input the key value want to deleted is:11

Tree is:

        15
      14
        13
    12
      10
        9
  8
        7
      6
        5
```

```
     4
           3
        2
           1
Input choice '0' to break:
Input the key value want to deleted is:4

Tree is:

           15
        14
           13
     12
        10
           9
  8
           7
        6
           5
     3
        2
           1
Input choice '0' to break:
Input the key value want to deleted is:3

 Tree is:

            15
         14
            13
      12
         10
            9
   8
            7
         6
            5
      2
         1
 Input choice '0' to break:
 Input the key value want to deleted is:1

 Right to Right Rotation

 Tree is:
```

```
            15
         14
            13
      12
         10
            9
   8
         7
      6
            5
         2
 Input choice '0' to break:
 Input the key value want to deleted is:12

Tree is:

           15
        14
           13
     10
        9
  8
        7
     6
           5
        2
Input choice '0' to break:
Input the key value want to deleted is:8

Left to Right Rotation

Tree is:

           15
        14
           13
     10
        9
  7
        6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:9

Right to Right Rotation
```

```
Tree is:

        15
     14
           13
        10
  7
        6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:7

Tree is:

        15
     14
           13
        10
  6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:10

Tree is:

        15
     14
        13
  6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:13

Tree is:

        15
     14
  6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:15
```

```
Tree is:

     14
  6
     5
        2
Input choice '0' to break:
Input the key value want to deleted is:14
Left to Left Rotation

 Tree is:

      6
   5
      2
 Input choice '0' to break:
 Input the key value want to deleted is:5

 Tree is:

      6
   2
 Input choice '0' to break:
 Input the key value want to deleted is:2

 Tree is:

   6
 Input choice '0' to break:
 Input the key value want to deleted is:6

 Tree is:

 Input choice '0' to break:
 Input the key value want to deleted is:2

 Information does not exist.
 Tree is:

 Input choice '0' to break:
 Input the key value want to deleted is:0
```