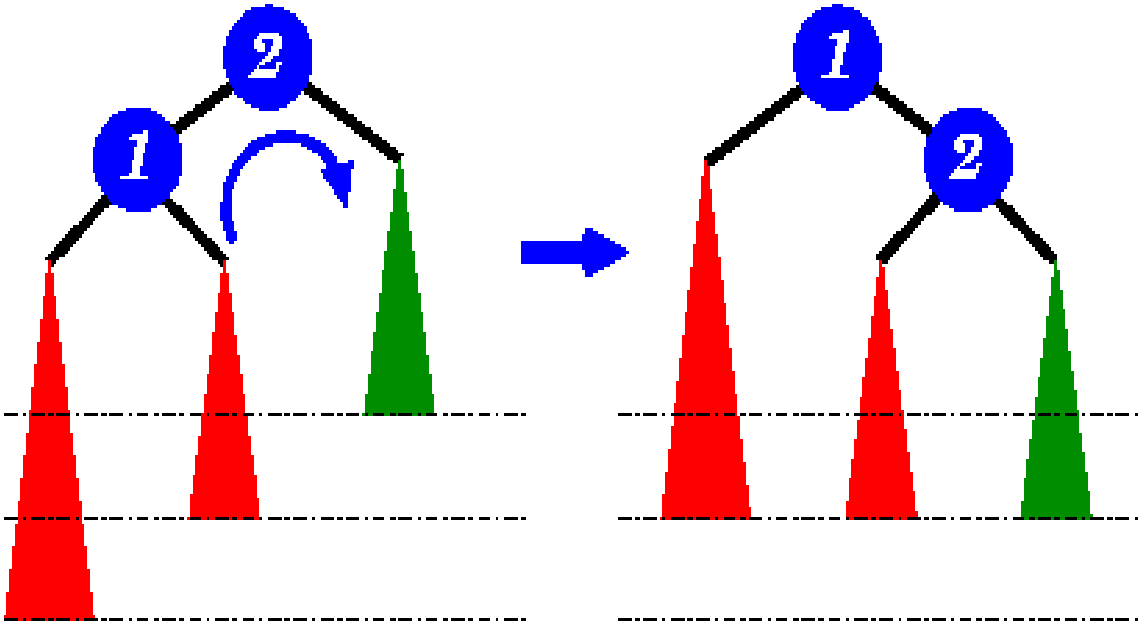
Avl Tree

कंप्यूटर विज्ञान में, एक AVL पेड़ (Adelson - VELSKII और अन्वेषकों के नाम पर रखा Landis ' पेड़ ,) एक आत्म संतुलन द्विआधारी खोज वृक्ष है . . यह आविष्कार होने वाले पहले ऐसे डेटा संरचना था [1] एक AVL ट्री में किसी भी नोड के दो बच्चे subtrees की ऊंचाइयों को अधिकतम एक से अलग ; वे एक से अधिक से अलग किसी भी समय अगर , पुनर्संतुलन इस संपत्ति को बहाल करने के लिए किया जाता है . लुक , सम्मिलन , और विलोपन सभी ताकेओ (लॉग एन) एन के पेड़ में नोड्स की संख्या पूर्व आपरेशन के लिए है , जहां औसत और सबसे खराब मामलों में दोनों समय . सम्मिलन और विलोपन पेड़ एक या एक से अधिक पेड़ घुमाव द्वारा पुनः संतुलित किया जा करना पड़ सकता है .

AVL पेड़ उनके 1962 कागज " सूचना के संगठन के लिए एक एल्गोरिथ्म " में इसे प्रकाशित किया अपने दो सोवियत आविष्कारक , जीएम Adelson - VELSKII और ईएम Landis , के नाम पर है . [2]

आपरेशन के एक ही सेट का समर्थन और हे (लॉग एन) बुनियादी कार्यों के लिए समय लेने के लिए दोनों क्योंकि AVL पेड़ अक्सर लाल काले पेड़ों के साथ तुलना की जाती है . वे और अधिक सख्ती से संतुलित कर रहे हैं क्योंकि देखने का गहन अनुप्रयोगों के लिए, AVL पेड़ लाल काले पेड़ों की तुलना में तेजी से कर रहे हैं . [3] लाल काले पेड़ों के लिए इसी प्रकार , AVL पेड़ ऊंचाई से संतुलित कर रहे हैं . दोनों को सामान्य रूप में नहीं कर रहे हैं वजन संतुलित और न ही μ संतुलित किसी के लिए [4] , सिबलिंग नोड्स वंश का बेहद संख्या भिन्न हो सकता है कि .



CODES(C)

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

 struct AVLTree_Node {
     int data, bfactor;
     struct AVLTree_Node *link[2];
 };

 struct AVLTree_Node *root = NULL;

 struct AVLTree_Node * createNode(int data) {
     struct AVLTree_Node *newnode;
     newnode = (struct AVLTree_Node *)malloc(sizeof (struct AVLTree_Node));
     newnode->data    = data;
     newnode->bfactor = 0;
     newnode->link[0] = newnode->link[1] = NULL;
     return newnode;
 }


 void insertion (int data) {
     struct AVLTree_Node *bf, *parent_bf, *subtree, *temp;
     struct AVLTree_Node *current, *parent, *newnode, *ptr;
     int res = 0, link_dir[32], i = 0;

     if (!root) {
         root = createNode(data);
         return;
     }

     bf = parent_bf = root;
     /* find the location for inserting the new node*/
     for (current = root; current != NULL; ptr = current, current = current->link[res]) {
         if (data == current->data) {
             printf("Cannot insert duplicates!!\n");
             return;
         }
         res = (data > current->data) ? 1 : 0;
         parent = current;

         if (current->bfactor != 0) {
             bf = current;
             parent_bf = ptr;
             i = 0;
         }
         link_dir[i++] = res;
     }
     /* create the new node */
```

```
newnode = createNode(data);
parent->link[res] = newnode;
res = link_dir[i = 0];
/* updating the height balance after insertion */
for (current = bf; current != newnode; res = link_dir[++i]) {
        if (res == 0)
                current->bfactor--;
        else
                current->bfactor++;
        current = current->link[res];
}

/* right sub-tree */
if (bf->bfactor == 2) {
        printf("bfactor = 2\n");
        temp = bf->link[1];
        if (temp->bfactor == 1) {
                /*
                 * single rotation(SR) left
                 *        x             y
                 *         \          /   \
                 *          y    =>   x     z
                 *           \
                 *            z
                 */
                subtree = temp;
                bf->link[1] = temp->link[0];
                temp->link[0] = bf;
                temp->bfactor = bf->bfactor = 0;
        } else {
                /*
                 * double rotation (SR right + SR left)
                 *       x        x           z
                 *        \        \        /   \
                 *         y   =>   z   =>  x     y
                 *        /          \      ///
                 *       z            y
                 */
                subtree = temp->link[0];
                temp->link[0] = subtree->link[1];
                subtree->link[1] = temp;
                bf->link[1] = subtree->link[0];
                subtree->link[0] = bf;
                /* update balance factors */
                if (subtree->bfactor == -1) {
                        bf->bfactor = 0;
```

```
                temp->bfactor = 1;
            } else if (subtree->bfactor == 0) {
                bf->bfactor = 0;
                temp->bfactor = 0;
            } else if (subtree->bfactor == 1) {
                bf->bfactor = -1;
                temp->bfactor = 0;
            }
            subtree->bfactor = 0;
        }
    /* left sub-tree */
    } else if (bf->bfactor == -2) {
        temp = bf->link[0];
        if (temp->bfactor == -1) {
            /*
             * single rotation(SR) right
             *         x          y
             *        /         /   \
             *       y     =>  z     x
             *      /
             *     z
             */
            subtree = temp;
            bf->link[0] = temp->link[1];
            temp->link[1] = bf;
            temp->bfactor = bf->bfactor = 0;
        } else {
            /*
             * double rotation - (SR left + SR right)
             *        x         x         z
             *       /         /        /   \
             *      y    =>   z    =>  y     x
             *       \       /
             *        z     y
             */
            subtree = temp->link[1];
            temp->link[1] = subtree->link[0];
            subtree->link[0] = temp;
            bf->link[0] = subtree->link[1];
            subtree->link[1] = bf;
            /* update balance factors */
            if (subtree->bfactor == -1) {
                bf->bfactor = 1;
                temp->bfactor = 0;
            } else if (subtree->bfactor == 0) {
                bf->bfactor = 0;
```

```
                temp->bfactor = 0;
            } else if (subtree->bfactor == 1) {
                bf->bfactor = 0;
                temp->bfactor = -1;
            }
            subtree->bfactor = 0;
        }
    } else {
        return;
    }

    if (bf == root) {
        root = subtree;
        return;
    }
    if (bf != parent_bf->link[0]) {
        parent_bf->link[1] = subtree;
    } else {
        parent_bf->link[0] = subtree;
    }
    return;
}

void deletion(int data) {
    int link_dir[32], res = 0, i = 0, j = 0, index = 0;
    struct AVLTree_Node *ptr[32], *current, *temp, *x, *y, *z;

    current = root;
    if (!root) {
        printf("Tree not present\n");
        return;
    }

    if ((root->data == data) && (root->link[0] == NULL)
         && (root->link[1] == NULL)) {
        free(root);
        root = NULL;
        return;
    }
    /* search the node to delete */
    while (current != NULL) {
        if (current->data == data)
            break;
        res = data > current->data ? 1 : 0;
        link_dir[i] = res;
        ptr[i++] = current;
```

```
        current = current->link[res];
    }

    if (!current) {
        printf("Given data is not present!!\n");
        return;
    }
    index = link_dir[i - 1];
    temp = current->link[1];
    /* delete the node from the AVL tree - similar to BST deletion */
    if (current->link[1] == NULL) {
        if (i == 0) {
            temp = current->link[0];
            free(current);
            root = temp;
            return;
        } else {
            ptr[i - 1]->link[index] = current->link[0];
        }
    } else if (temp->link[0] == NULL) {
        temp->link[0] = current->link[0];
        temp->bfactor = current->bfactor;
        if (i > 0) {
            ptr[i-1]->link[index] = temp;
        } else {
            root = temp;
        }
        link_dir[i] = 1;
        ptr[i++] = temp;
    } else {
        /* delete node with two children */
        j = i++;
        while (1) {
            link_dir[i] = 0;
            ptr[i++] = temp;
            x = temp->link[0];
            if (x->link[0] == NULL)
                break;
            temp = x;
        }
        x->link[0] = current->link[0];
        temp->link[0] = x->link[1];
        x->link[1] = current->link[1];
        x->bfactor = current->bfactor;
        if (j > 0) {
            ptr[j - 1]->link[index] = x;
```

```
        } else {
                root = x;
        }
        link_dir[j] = 1;
        ptr[j] = x;
}
free(current);
for (i = i - 1; i >= 0; i = i--) {
        x = ptr[i];
        if (link_dir[i] == 0) {
                x->bfactor++;
                if (x->bfactor == 1) {
                        break;
                } else if (x->bfactor == 2) {
                        y = x->link[1];
                        if (y->bfactor == -1) {
                                /* double rotation - (SR right + SR left) */
                                z = y->link[0];
                                y->link[0] = z->link[1];
                                z->link[1] = y;
                                x->link[1] = z->link[0];
                                z->link[0] = x;
                                /* update balance factors */
                                if (z->bfactor == -1) {
                                        x->bfactor = 0;
                                        y->bfactor = 1;
                                } else if (z->bfactor == 0) {
                                        x->bfactor = 0;
                                        y->bfactor = 0;
                                } else if (z->bfactor == 1) {
                                        x->bfactor = -1;
                                        y->bfactor = 0;
                                }
                                z->bfactor = 0;
                                if (i > 0) {
                                        index = link_dir[i - 1];
                                        ptr[i - 1]->link[index] = z;
                                } else {
                                        root = z;
                                }
                        } else {
                                /* single rotation left */
                                x->link[1] = y->link[0];
                                y->link[0] = x;
                                if (i > 0) {
                                        index = link_dir[i - 1];
```

```
                        ptr[i - 1]->link[index] = y;
                    } else  {
                        root = y;
                    }
                    /* update balance factors */
                    if (y->bfactor == 0) {
                        x->bfactor = 1;
                        y->bfactor = -1;
                        break;
                    } else {
                        x->bfactor = 0;
                        y->bfactor = 0;
                    }
                }
            }
        } else {
            x->bfactor--;
            if (x->bfactor == -1) {
                break;
            } else if (x->bfactor == -2) {
                y = x->link[0];
                if  (y->bfactor == 1) {
                    /* double rotation - (SR right + SR left) */
                    z = y->link[1];
                    y->link[1] = z->link[0];
                    z->link[0] = y;
                    x->link[0] = z->link[1];
                    z->link[1] = x;
                    /* update balance factors */
                    if (z->bfactor == -1) {
                        x->bfactor = 1;
                        y->bfactor = 0;
                    } else if (z->bfactor == 0) {
                        x->bfactor = 0;
                        y->bfactor = 0;
                    } else if (z->bfactor == 1) {
                        x->bfactor = 0;
                        y->bfactor = -1;
                    }
                    z->bfactor = 0;
                    if (i > 0) {
                        index = link_dir[i - 1];
                        ptr[i - 1]->link[index] = z;
                    } else {
                        root = z;
                    }
```

```
                } else {
                    /* single rotation right */
                    x->link[0] = y->link[1];
                    y->link[1] = x;
                    if (i <= 0) {
                        root = y;
                    } else {
                        index = link_dir[i - 1];
                        ptr[i - 1]->link[index] = y;
                    }
                    /* update balance factors */
                    if (y->bfactor == 0) {
                        x->bfactor = -1;
                        y->bfactor = 1;
                        break;
                    } else {
                        x->bfactor = 0;
                        y->bfactor = 0;
                    }
                }
            }
        }
    }
}

void searchElement(int data) {
    int flag = 0, res = 0;
    struct AVLTree_Node *node = root;
    if (!node) {
        printf("AVL tree unavailable!!\n");
        return;
    }
    while (node != NULL) {
        if (data == node->data) {
            printf("%d is present in AVL Tree\n", data);
            flag = 1;
            break;
        }
        res = data > node->data ? 1 : 0;
        node = node->link[res];
    }
    if (!flag)
        printf("Search Element not found in AVL tree\n");
    return;
}
```

```
void inorderTraversal(struct AVLTree_Node *myNode) {
    if (myNode) {
        inorderTraversal(myNode->link[0]);
        printf("%d  ", myNode->data);
        inorderTraversal(myNode->link[1]);
    }
    return;
}

int main() {
    int key, ch;
    while (1) {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Traversal\n");
        printf("5. Exit\nEnter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter the key value:");
                scanf("%d", &key);
                insertion(key);
                break;
            case 2:
                printf("Enter the key value to delete:");
                scanf("%d", &key);
                deletion(key);
                break;
            case 3:
                printf("Enter the search key:");
                scanf("%d", &key);
                searchElement(key);
                break;
            case 4:
                inorderTraversal(root);
                printf("\n");
                break;
            case 5:
                exit(0);
            default:
                printf("Wrong Option!!\n");
                break;
        }
        printf("\n");
    }
```

```
    return 0;
}
```

CODES(JAVA)

```
import java.util.Scanner;

 /* Class AVLNode */
 class AVLNode
 {
    AVLNode left, right;
    int data;
    int height;

    /* Constructor */
    public AVLNode()
    {
        left = null;
        right = null;
        data = 0;
        height = 0;
    }
    /* Constructor */
    public AVLNode(int n)
    {
        left = null;
        right = null;
        data = n;
        height = 0;
    }
 }

 /* Class AVLTree */
 class AVLTree
```

```
{
    private AVLNode root;

    /* Constructor */
    public AVLTree()
    {
        root = null;
    }
    /* Function to check if tree is empty */
    public boolean isEmpty()
    {
        return root == null;
    }
    /* Make the tree logically empty */
    public void makeEmpty()
    {
        root = null;
    }
    /* Function to insert data */
    public void insert(int data)
    {
        root = insert(data, root);
    }
    /* Function to get height of node */
    private int height(AVLNode t )
    {
        return t == null ? -1 : t.height;
    }
    /* Function to max of left/right node */
    private int max(int lhs, int rhs)
```

```
{
    return lhs > rhs ? lhs : rhs;
}
/* Function to insert data recursively */
private AVLNode insert(int x, AVLNode t)
{
    if (t == null)
        t = new AVLNode(x);
    else if (x < t.data)
    {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x < t.left.data )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x > t.data )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x > t.right.data)
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
    else
      ;  // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
```

```
}
/* Rotate binary tree node with left child */
private AVLNode rotateWithLeftChild(AVLNode k2)
{
    AVLNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = max( height( k1.left ), k2.height ) + 1;
    return k1;
}

/* Rotate binary tree node with right child */
private AVLNode rotateWithRightChild(AVLNode k1)
{
    AVLNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child */
private AVLNode doubleWithLeftChild(AVLNode k3)
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

```
/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child */
private AVLNode doubleWithRightChild(AVLNode k1)
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}
/* Functions to count number of nodes */
public int countNodes()
{
    return countNodes(root);
}
private int countNodes(AVLNode r)
{
    if (r == null)
        return 0;
    else
    {
        int l = 1;
        l += countNodes(r.left);
        l += countNodes(r.right);
        return l;
    }
}
/* Functions to search for an element */
public boolean search(int val)
{
    return search(root, val);
}
```

```
private boolean search(AVLNode r, int val)
{
    boolean found = false;
    while ((r != null) && !found)
    {
        int rval = r.data;
        if (val < rval)
            r = r.left;
        else if (val > rval)
            r = r.right;
        else
        {
            found = true;
            break;
        }
        found = search(r, val);
    }
    return found;
}
/* Function for inorder traversal */
public void inorder()
{
    inorder(root);
}
private void inorder(AVLNode r)
{
    if (r != null)
    {
        inorder(r.left);
        System.out.print(r.data +" ");
```

```
        inorder(r.right);
    }
}
/* Function for preorder traversal */
public void preorder()
{
    preorder(root);
}
private void preorder(AVLNode r)
{
    if (r != null)
    {
        System.out.print(r.data +" ");
        preorder(r.left);
        preorder(r.right);
    }
}
/* Function for postorder traversal */
public void postorder()
{
    postorder(root);
}
private void postorder(AVLNode r)
{
    if (r != null)
    {
        postorder(r.left);
        postorder(r.right);
        System.out.print(r.data +" ");
    }
```

```
    }
}

/* Class AVL Tree Test */
public class AVLTreeTest
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        /* Creating object of AVLTree */
        AVLTree avlt = new AVLTree();

        System.out.println("AVLTree Tree Test\n");
        char ch;
        /*  Perform tree operations  */
        do
        {
            System.out.println("\nAVLTree Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. search");
            System.out.println("3. count nodes");
            System.out.println("4. check empty");
            System.out.println("5. clear tree");

            int choice = scan.nextInt();
            switch (choice)
            {
            case 1 :
                System.out.println("Enter integer element to insert");
                avlt.insert( scan.nextInt() );
```

```
            break;
        case 2 :
            System.out.println("Enter integer element to search");
            System.out.println("Search result : "+ avlt.search( scan.nextInt() ));
            break;
        case 3 :
            System.out.println("Nodes = "+ avlt.countNodes());
            break;
        case 4 :
            System.out.println("Empty status = "+ avlt.isEmpty());
            break;
        case 5 :
            System.out.println("\nTree Cleared");
            avlt.makeEmpty();
            break;
        default :
            System.out.println("Wrong Entry \n ");
            break;
        }
        /*  Display tree  */
        System.out.print("\nPost order : ");
        avlt.postorder();
        System.out.print("\nPre order : ");
        avlt.preorder();
        System.out.print("\nIn order : ");
        avlt.inorder();

        System.out.println("\nDo you want to continue (Type y or n) \n");
        ch = scan.next().charAt(0);
    } while (ch == 'Y'|| ch == 'y');
```

```
    }
}
```