AMRITA VISHWA VIDYAPEETHAM

Amrita School of Computing

23CSE211

Design and Analysis of Algorithms

WEEK – 4 (08 January 2026)

Susendran M

CH.SC.U4CSE24154

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF COMPUTING

CHENNAI

1. AVL TREE

An AVL tree is a self-balancing Binary Search Tree in which the balance factor of every node is maintained between –1 and +1, where
Balance Factor = Height(left subtree) − Height(right subtree).
The elements
157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133
are inserted one by one following BST insertion rules. After each insertion, the balance factor of every ancestor node is checked.
Whenever a node becomes unbalanced, appropriate rotations are applied:

- LL rotation for left–left imbalance
- RR rotation for right–right imbalance
- LR rotation for left–right imbalance
- RL rotation for right–left imbalance

During the construction:

- An LR rotation occurs when inserting 147
- An RL rotation occurs when inserting 111
- Another LR rotation occurs when inserting 151
- A final LR rotation occurs when inserting 112, making 122 the root

After all insertions and rotations, the AVL tree remains height-balanced.

Code :

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

struct Node {
   int key;
   struct Node *left;
   struct Node *right;
   int height;
};

int max(int a, int b) {
   return (a > b) ? a : b;
}

int height(struct Node *n) {
   if (n == NULL)
      return 0;
   return n->height;
}
```

```
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

int getBalance(struct Node* n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

struct Node* insert(struct Node* node, int key) {
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
```

```
    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

int main() {
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = 12, i;

    struct Node* root = NULL;

    for (i = 0; i < n; i++)
        root = insert(root, arr[i]);

    preorder(root);

    return 0;
}
```

Output :

```
122 111 110 112 117 147 133 123 141 151 149 157
--------------------------------
Process exited after 0.1128 seconds with return value 0
Press any key to continue . . .
```

2. RED – BLACK TREE

A Red-Black Tree is a self-balancing Binary Search Tree in which each node is colored RED or BLACK and satisfies the following properties:

1. The root is always BLACK
2. No two consecutive RED nodes are allowed
3. Every path from root to NULL has the same number of BLACK nodes

The elements
157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133
are inserted one by one following BST insertion rules.
Each newly inserted node is initially colored RED.
After insertion, Red-Black properties are restored using:

- Recoloring when the uncle node is RED
- Left and Right rotations when the uncle node is BLACK

During construction:

- Color conflicts (RED-RED) are resolved using recoloring or rotations
- Rotations occur to maintain tree balance
- The root is always recolored to BLACK after adjustments

After all insertions, the Red-Black Tree remains balanced and satisfies all Red-Black properties.

Code :

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

#define RED 1
#define BLACK 0
```

```
struct Node {
    int data;
    int color;
    struct Node *left, *right, *parent;
};

struct Node* root = NULL;

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->color = RED;
    node->left = node->right = node->parent = NULL;
    return node;
}

void leftRotate(struct Node* x) {
    struct Node* y = x->right;
    x->right = y->left;
    if (y->left)
        y->left->parent = x;
    y->parent = x->parent;
    if (!x->parent)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

void rightRotate(struct Node* y) {
    struct Node* x = y->left;
    y->left = x->right;
    if (x->right)
        x->right->parent = y;
    x->parent = y->parent;
    if (!y->parent)
        root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}

void fixInsert(struct Node* z) {
    while (z->parent && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            struct Node* y = z->parent->parent->right;
```

```
            if (y && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        } else {
            struct Node* y = z->parent->parent->left;
            if (y && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}

void insert(int data) {
    struct Node* z = newNode(data);
    struct Node* y = NULL;
    struct Node* x = root;

    while (x) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }

    z->parent = y;
    if (!y)
        root = z;
    else if (z->data < y->data)
```

```
        y->left = z;
    else
        y->right = z;

    fixInsert(z);
}

void inorder(struct Node* node) {
    if (node) {
        inorder(node->left);
        printf("%d ", node->data);
        inorder(node->right);
    }
}

int main() {
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = 12, i;

    for (i = 0; i < n; i++)
        insert(arr[i]);

    inorder(root);
    return 0;
}
```

Output :