

# Balancing BST

## 1. Balancing BST (AVL Rotations)

Code :

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

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

/* Utility Functions */
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* createNode(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;
}

/* Rotations */
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);
}

/* AVL Insert */
struct Node* insertAVL(struct Node *node, int key) {
    int balance;

    if (node == NULL)
        return createNode(key);

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

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

    balance = getBalance(node);

    /* 4 Cases */

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

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

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

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

    return node;
}

```

```

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

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

int main() {
    struct Node *root = NULL;

    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr) / sizeof(arr[0]);
    int i;

    printf("CH.SC.U4CSE24108\n");
    printf("CH.SC.U4CSE24108\n");

    for (i = 0; i < n; i++) {
        root = insertAVL(root, arr[i]);
    }

    printf("\nAVL Tree Inorder: ");
    inorder(root);

    printf("\nAVL Tree Preorder: ");
    preorder(root);

    printf("\n");
}

return 0;
}

```

## Output :

```
CH.SC.U4CSE24108> gcc "AVL Tree Implementation.c" -o "AVL Tree Implementation"
CH.SC.U4CSE24108> ./"AVL Tree Implementation"
CH.SC.U4CSE24108
CH.SC.U4CSE24108

AVL Tree Inorder: 110 111 112 117 122 123 133 141 147 149 151 157
AVL Tree Preorder: 122 111 110 112 117 147 133 123 141 151 149 157
CH.SC.U4CSE24108>
```

<b>Operation</b>	<b>Time Complexity</b>	<b>Justification</b>
Search	$O(\log n)$	Height is logarithmic
Insert	$O(\log n)$	BST insert + constant rotations
Delete	$O(\log n)$	Rebalancing bounded by height
Traversal	$O(n)$	Visit every node
Rotation	$O(1)$	Constant pointer updates
Space	$O(n)$	Store all nodes

## 2. Balancing BST (Red Black Tree)

### Code :

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

#define RED 1
#define BLACK 0

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

struct Node *root = NULL;

/* Create a new node */
struct Node* createNode(int data) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    newNode->data = data;
    newNode->color = RED;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->parent = NULL;

    return newNode;
}

/* Left Rotation */
void leftRotate(struct Node *x) {
    struct Node *y;

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

    if (y->left != NULL)
        y->left->parent = x;

    y->parent = x->parent;

    if (x->parent == NULL)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
```

```

    else
        x->parent->right = y;

    y->left = x;
    x->parent = y;
}

/* Right Rotation */
void rightRotate(struct Node *y) {
    struct Node *x;

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

    if (x->right != NULL)
        x->right->parent = y;

    x->parent = y->parent;

    if (y->parent == NULL)
        root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    x->right = y;
    y->parent = x;
}

/* Fix violations after insertion */
void fixInsert(struct Node *z) {
    struct Node *parent;
    struct Node *grandparent;
    struct Node *uncle;

    while (z != root && z->parent != NULL && z->parent->color == RED) {

        parent = z->parent;
        grandparent = parent->parent;

        if (grandparent == NULL)
            break;

        /* Parent is left child */
        if (parent == grandparent->left) {
            uncle = grandparent->right;

            /* Case 1: Uncle is RED -> recolor */
            if (uncle != NULL && uncle->color == RED) {
                grandparent->color = RED;
                parent->color = BLACK;
                uncle->color = BLACK;
                z = grandparent;
            }
            else {
                /* Case 2: z is right child -> Left Rotate */

```

```

        if (z == parent->right) {
            z = parent;
            leftRotate(z);
            parent = z->parent;
            grandparent = parent->parent;
        }

        /* Case 3: z is left child -> Right Rotate */
        if (parent != NULL && grandparent != NULL) {
            parent->color = BLACK;
            grandparent->color = RED;
            rightRotate(grandparent);
        }
    }
}

else {
    /* Parent is right child */
    uncle = grandparent->left;

    /* Case 1: Uncle is RED -> recolor */
    if (uncle != NULL && uncle->color == RED) {
        grandparent->color = RED;
        parent->color = BLACK;
        uncle->color = BLACK;
        z = grandparent;
    }
    else {
        /* Case 2: z is left child -> Right Rotate */
        if (z == parent->left) {
            z = parent;
            rightRotate(z);
            parent = z->parent;
            grandparent = parent->parent;
        }

        /* Case 3: z is right child -> Left Rotate */
        if (parent != NULL && grandparent != NULL) {
            parent->color = BLACK;
            grandparent->color = RED;
            leftRotate(grandparent);
        }
    }
}

if (root != NULL)
    root->color = BLACK;
}

/* Insert node like BST */
void insertRB(int data) {
    struct Node *z;
    struct Node *y;
    struct Node *x;

    z = createNode(data);
    y = NULL;
}

```

```

x = root;

/* Normal BST insertion */
while (x != NULL) {
    y = x;
    if (z->data < x->data)
        x = x->left;
    else
        x = x->right;
}

z->parent = y;

if (y == NULL)
    root = z;
else if (z->data < y->data)
    y->left = z;
else
    y->right = z;

fixInsert(z);
}

/* Inorder traversal */
void inorder(struct Node *r) {
    if (r != NULL) {
        inorder(r->left);
        printf("%d(%c) ", r->data, (r->color == RED) ? 'R' : 'B');
        inorder(r->right);
    }
}

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

    printf("CH.SC.U4CSE24108\n");
    printf("CH.SC.U4CSE24108\n");

    for (i = 0; i < n; i++) {
        insertRB(arr[i]);
    }

    printf("\nRed-Black Tree Inorder (with colors):\n");
    inorder(root);

    printf("\n");
    return 0;
}

```

## Output :

```
CH.SC.U4CSE24108>
CH.SC.U4CSE24108> gcc "Red Black Tree Implementation.c" -o "Red Black Tree Implementation"
CH.SC.U4CSE24108> ./"Red Black Tree Implementation"
CH.SC.U4CSE24108
CH.SC.U4CSE24108

Red-Black Tree Inorder (with colors):
110(B) 111(R) 112(R) 117(B) 122(R) 123(B) 133(R) 141(B) 147(R) 149(R) 151(B) 157(R)
CH.SC.U4CSE24108>
```

<b>Operation</b>	<b>Time Complexity</b>	<b>Justification</b>
Search	$O(\log n)$	Height is logarithmic
Insert	$O(\log n)$	BST insert + constant rotations
Delete	$O(\log n)$	Rebalancing bounded by height
Traversal	$O(n)$	Visit every node
Rotation	$O(1)$	Constant pointer updates
Space	$O(n)$	Store all nodes