SCHOOL OF COMPUTING

B.S.Chenthil Hari

CH.SC.U4CSE24103

Week – 4

Date - 08/01/2026

Design and Analysis of Algorithm(23CSE211)

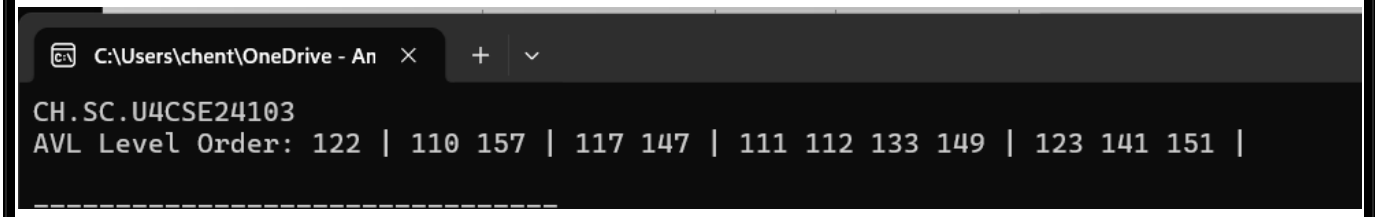
1. BST Balancing using Rotation Method

Code:

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

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

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

int getHeight(struct Node *n) {
    return (n == NULL) ? 0 : n->height;
}

struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = 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(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(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(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}

struct Node* balanceTree(struct Node* node) {
    if (node == NULL)
        return NULL;

    node->left = balanceTree(node->left);
    node->right = balanceTree(node->right);

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

    // Left Left Case
    if (balance > 1 && getHeight(node->left->left) >= getHeight(node->left->right))
        return rightRotate(node);

    // Left Right Case
    if (balance > 1 && getHeight(node->left->left) < getHeight(node->left->right)) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```c
    // Right Right Case
    if (balance < -1 && getHeight(node->right->right) >= getHeight(node->right->left))
        return leftRotate(node);

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

    return node;
}

void printGivenLevel(struct Node* root, int level) {
    if (root == NULL)
        return;

    if (level == 1)
        printf("%d ", root->key);
    else if (level > 1) {
        printGivenLevel(root->left, level - 1);
        printGivenLevel(root->right, level - 1);
    }
}

void printLevelOrder(struct Node* root) {
    int h = getHeight(root);
    for (int i = 1; i <= h; i++) {
        printGivenLevel(root, i);
        printf("| ");
    }
}

int main() {
    printf("CH.SC.U4CSE24103\n");

    struct Node *root = newNode(157);
    root->left = newNode(110);
    root->left->right = newNode(147);
    root->left->right->left = newNode(122);
    root->left->right->right = newNode(149);
    root->left->right->right->right = newNode(151);
    root->left->right->left->left = newNode(111);
    root->left->right->left->right = newNode(141);
    root->left->right->left->left->right = newNode(112);
    root->left->right->left->right->left = newNode(123);
    root->left->right->left->right->left->right = newNode(133);
    root->left->right->left->left->right->left = newNode(117);

    root = balanceTree(root);

    printf("AVL Level Order: ");
    printLevelOrder(root);
    printf("\n");

    return 0;
}
```

Output:



Time Complexity: O(n)
Justification: The balanceTree function is called once on the root, but it recursively visits every node in the tree exactly once to calculate heights and perform rotations. Since the code manually builds a tree with n nodes and then runs a single post-order traversal to balance it, the time taken is proportional to the number of nodes.

Note: While standard AVL insertions are O(log n), your specific implementation balances an existing unbalanced tree in a single pass, making it linear for the total number of nodes.

Space Complexity: O(log n)
Justification: The space complexity is determined by the maximum depth of the function call stack during recursion. Since an AVL tree is strictly balanced, its height h is guaranteed to be O(log n). Even though you start with an unbalanced tree, the recursive depth of balanceTree or printLevelOrder will not exceed the initial height of the tree. No significant auxiliary data structures (like arrays or queues) are used, only the memory for the nodes themselves and the recursion stack.

2. BST Balancing using Red-Black Method

Code:

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

enum Color { RED, BLACK };

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

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

void rotateLeft(struct Node** root, struct Node* x) {
    struct Node* 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;
}
```

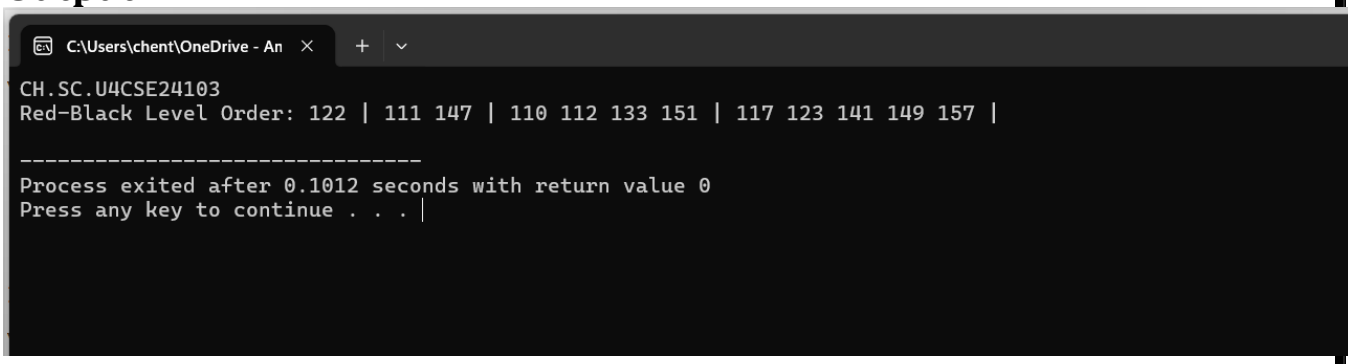
```c
void rotateRight(struct Node** root, struct Node* y) {
    struct Node* x = y->left;
    y->left = x->right;

    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;
}

void fixViolation(struct Node** root, struct Node* z) {
    while (z != *root && z->parent->color == RED) {

        if (z->parent == z->parent->parent->left) {

            struct Node* y = z->parent->parent->right;

            if (y != NULL && 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;
                    rotateLeft(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rotateRight(root, z->parent->parent);
            }

        } else {

            struct Node* y = z->parent->parent->left;

            if (y != NULL && 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;
                    rotateRight(root, z);
                }

                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rotateLeft(root, z->parent->parent);
            }
        }
    }

    (*root)->color = BLACK;
}

int height(struct Node* node) {
    if (node == NULL)
        return 0;

    int l = height(node->left);
    int r = height(node->right);
```

```
115     return (l > r ? l : r) + 1;
116 }
117
118 void printGivenLevel(struct Node* root, int level) {
119     if (root == NULL)
120         return;
121
122     if (level == 1)
123         printf("%d ", root->data);
124     else {
125         printGivenLevel(root->left, level - 1);
126         printGivenLevel(root->right, level - 1);
127     }
128 }
129
130 void printLevelOrder(struct Node* root) {
131     int h = height(root);
132
133     for (int i = 1; i <= h; i++) {
134         printGivenLevel(root, i);
135         printf("| ");
136     }
137 }
138
139 int main() {
140
141     printf("CH.SC.U4CSE24103\n");
142     public int __cdecl printf (const char * __restrict__ _Format, ...)
143     struct Node* root = newNode(157);
144     root->color = BLACK;
145
146     struct Node* n110 = newNode(110);
147     root->left = n110; n110->parent = root;
148
149     struct Node* n147 = newNode(147);
150     n110->right = n147; n147->parent = n110;
151
152     struct Node* n122 = newNode(122);
153     n147->left = n122; n122->parent = n147;
154
155     struct Node* n149 = newNode(149);
156     n147->right = n149; n149->parent = n147;
157
158     struct Node* n111 = newNode(111);
159     n122->left = n111; n111->parent = n122;
160
161     struct Node* n141 = newNode(141);
162     n122->right = n141; n141->parent = n122;
163
164     struct Node* n151 = newNode(151);
165     n149->right = n151; n151->parent = n149;
166
167     struct Node* n112 = newNode(112);
168     n111->right = n112; n112->parent = n111;
169
170     struct Node* n123 = newNode(123);
171     n141->left = n123; n123->parent = n141;
172
173     struct Node* n117 = newNode(117);
174     n112->left = n117; n117->parent = n112;
175
176     struct Node* n133 = newNode(133);
177     n123->right = n133; n133->parent = n123;
178
179     fixViolation(&root, n147);
180     fixViolation(&root, n111);
181     fixViolation(&root, n141);
182     fixViolation(&root, n151);
183     fixViolation(&root, n117);
```

```
    fixViolation(&root, n117);
    fixViolation(&root, n133);

    printf("Red-Black Level Order: ");
    printLevelOrder(root);
    printf("\n");

    return 0;
}
```


```
    fixViolation(&root, n117);
    fixViolation(&root, n133);

    printf("Red-Black Level Order: ");
    printLevelOrder(root);
    printf("\n");

    return 0;
}
```

Output:



Time Complexity: O(n log n)
Justification: While a single insertion and balance (fixViolation) in a Red-Black Tree takes O(log n) time, this code manually links nodes and then calls fixViolation multiple times in a sequence.

- **Balancing:** Each call to fixViolation takes O(log n) as it may traverse up to the root.
- **Printing:** The printLevelOrder function uses a nested loop and recursion that visits nodes level-by-level. For a balanced tree, this specific level-order implementation (without a queue) takes O(n) time.
- **Overall:** For n elements, if you were inserting them normally, it would be O(n log n).

Space Complexity: O(n)
Justification: The program requires memory to store each element as a struct Node, which includes pointers for left, right, parent, and the color attribute. This auxiliary memory is directly proportional to the number of nodes n in the tree. Furthermore, the recursive functions for height and level-printing use the function call stack, which requires O(log n) space for a balanced Red-Black Tree.