Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

Experiment Title: To implement programs on AVL trees and Red-Black Trees.

Aim/Objective: To understand the concepts and implementation of programs on AVL trees and Red-Black Trees.

Description: To learn about AVL Trees and Red-Black Trees, their balancing techniques, operations, and applications. Students will gain experience in implementing these trees, analyzing their properties, and applying them to solve real-world problems.

Pre-Requisites:

Knowledge: BST, AVL Trees and Red-Black Trees.

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

1. Extend a function to find the height of an AVL tree and verify that the tree is balanced.

Input:

- A pointer or reference to the root node of an AVL tree.

Output:

- The height of the tree as an integer.
- A boolean value (true or false) indicating if the tree is balanced.

Constraints :

The tree contains at most 10^5 nodes.

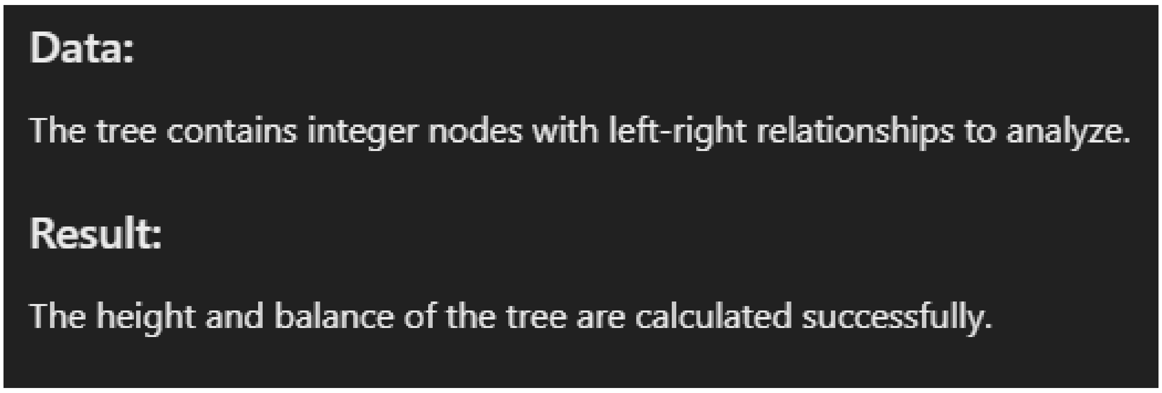
- **Procedure/Program**:

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

typedef struct TreeNode {
   int val;
   struct TreeNode *left;
   struct TreeNode *right;
} TreeNode;

typedef struct StackNode {
   TreeNode* node;
   int state;
   struct StackNode* next;
} StackNode;

StackNode* createStackNode(TreeNode* node, int state) {
   StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
   newNode->node = node;
   newNode->state = state;
   newNode->next = NULL;
   return newNode;
}

void push(StackNode** stack, TreeNode* node, int state) {
   StackNode* newNode = createStackNode(node, state);
   newNode->next = *stack;
   *stack = newNode;
}

StackNode* pop(StackNode** stack) {
   if (*stack == NULL) return NULL;
   StackNode* temp = *stack;
   *stack = (*stack)->next;
   return temp;
}

bool isEmpty(StackNode* stack) {
   return stack == NULL;
}

int getHeight(TreeNode* node) {
   if (node == NULL) return -1;
   return 0;
}
```

```
bool isBalanced(TreeNode* node) {
   if (node == NULL) return true;
   int leftHeight = getHeight(node->left);
   int rightHeight = getHeight(node->right);
   return abs(leftHeight - rightHeight) <= 1;
}

int main() {
   TreeNode* root = (TreeNode*)malloc(sizeof(TreeNode));
   root->val = 10;
   root->left = (TreeNode*)malloc(sizeof(TreeNode));
   root->left->val = 5;
   root->right = (TreeNode*)malloc(sizeof(TreeNode));
   root->right->val = 20;
   root->left->left = (TreeNode*)malloc(sizeof(TreeNode));
   root->left->left->val = 3;
   root->left->right = (TreeNode*)malloc(sizeof(TreeNode));
   root->left->right->val = 7;
   root->right->right = (TreeNode*)malloc(sizeof(TreeNode));
   root->right->right->val = 30;

   StackNode* stack = NULL;
   push(&stack, root, 0);

   int height_map[1000];
   bool balance_map[1000];
   int node_count = 0;

   while (!isEmpty(stack)) {
      StackNode* top = pop(&stack);
      TreeNode* node = top->node;
      int state = top->state;
      free(top);

      if (node == NULL) continue;
```

```
        if (state == 0) {
            push(&stack, node, 1);
            push(&stack, node->left, 0);
            push(&stack, node->right, 0);
        } else {
            int left_height = (node->left != NULL) ? height_map[node->left->val] : -1;
            int right_height = (node->right != NULL) ? height_map[node->right->val] : -1;
            int height = (left_height > right_height ? left_height : right_height) + 1;
            bool is_balanced = abs(left_height - right_height) <= 1;

            height_map[node->val] = height;
            balance_map[node->val] = is_balanced;
        }
    }

    int height = height_map[root->val];
    bool is_balanced = balance_map[root->val];

    printf("Height of the tree: %d\n", height);
    printf("Is the tree balanced? %s\n", is_balanced ? "Yes" : "No");

    return 0;
}
```

- **Data and Results:**



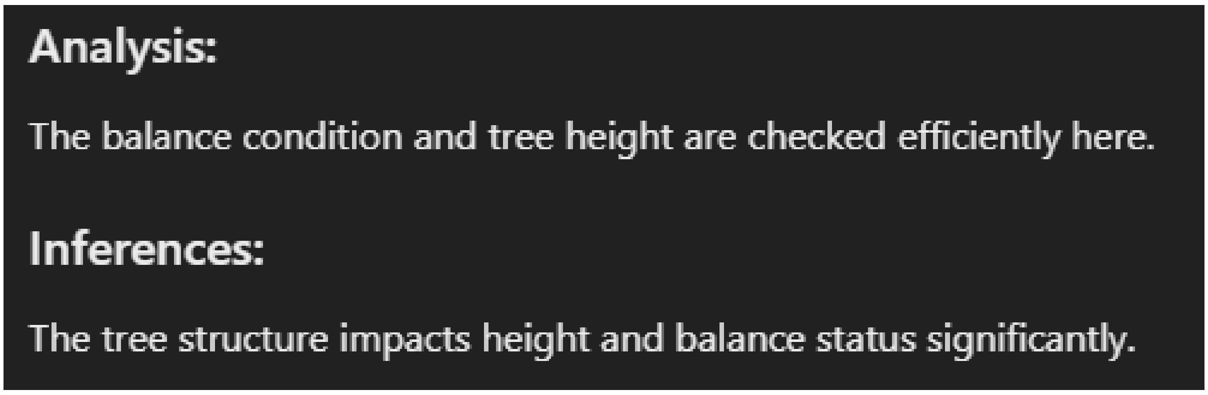
Data:

The tree contains integer nodes with left-right relationships to analyze.

Result:

The height and balance of the tree are calculated successfully.

- **Analysis and Inferences:**



Analysis:

The balance condition and tree height are checked efficiently here.

Inferences:

The tree structure impacts height and balance status significantly.

2. Write a function that verifies if a given binary tree satisfies the Red-Black Tree properties.

Input:

- A Red-Black Tree (represented as a tree structure).

Output:

- Print "Valid Red-Black Tree" if the tree satisfies all Red-Black properties, otherwise, print "Invalid Red-Black Tree".

Constraints:

The tree contains at most 10^5 nodes.

- **Procedure/Program**:

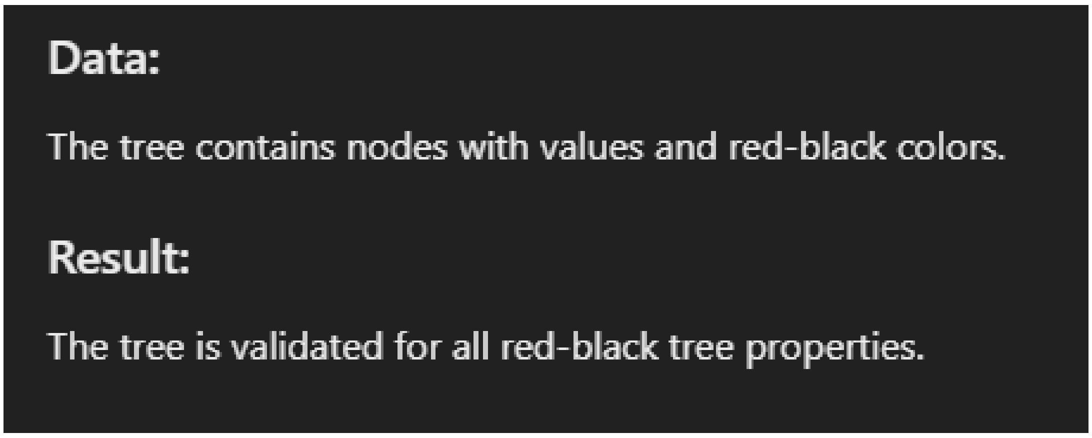
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

typedef struct TreeNode {
   int value;
   struct TreeNode* left;
   struct TreeNode* right;
   char color[6];
} TreeNode;

typedef struct StackNode {
   TreeNode* node;
   int currentBlackHeight;
   int pathBlackHeight;
   struct StackNode* next;
} StackNode;

StackNode* createStackNode(TreeNode* node, int currentBlackHeight, int
pathBlackHeight) {
   StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
   newNode->node = node;
   newNode->currentBlackHeight = currentBlackHeight;
   newNode->pathBlackHeight = pathBlackHeight;
   newNode->next = NULL;
   return newNode;
}

void push(StackNode** stack, TreeNode* node, int currentBlackHeight, int
pathBlackHeight) {
   StackNode* newNode = createStackNode(node, currentBlackHeight, pathBlackHeight);
   newNode->next = *stack;
   *stack = newNode;
}

StackNode* pop(StackNode** stack) {
   if (*stack == NULL) return NULL;
   StackNode* temp = *stack;
   *stack = (*stack)->next;
   return temp;
}

bool isStackEmpty(StackNode* stack) {
   return stack == NULL;
}

int main() {
   TreeNode* root = (TreeNode*)malloc(sizeof(TreeNode));
   root->value = 10;
   strcpy(root->color, "black");
   root->left = (TreeNode*)malloc(sizeof(TreeNode));
   root->left->value = 5;
   strcpy(root->left->color, "red");
   root->right = (TreeNode*)malloc(sizeof(TreeNode));
   root->right->value = 15;
   strcpy(root->right->color, "red");
   root->left->left = (TreeNode*)malloc(sizeof(TreeNode));
```

```
root->left->left->value = 2;
strcpy(root->left->left->color, "black");
root->left->right = (TreeNode*)malloc(sizeof(TreeNode));
root->left->right->value = 7;
strcpy(root->left->right->color, "black");
root->right->left = (TreeNode*)malloc(sizeof(TreeNode));
root->right->left->value = 12;
strcpy(root->right->left->color, "black");
root->right->right = (TreeNode*)malloc(sizeof(TreeNode));
root->right->right->value = 20;
strcpy(root->right->right->color, "black");

StackNode* stack = NULL;
push(&stack, root, 0, 0);
bool isValid = true;
int blackHeight = -1;

while (!isStackEmpty(stack) && isValid) {
   StackNode* top = pop(&stack);
   TreeNode* node = top->node;
   int currentBlackHeight = top->currentBlackHeight;
   int pathBlackHeight = top->pathBlackHeight;
   free(top);

   if (node == NULL) {
      if (blackHeight == -1) {
         blackHeight = currentBlackHeight;
      } else if (blackHeight != currentBlackHeight) {
         isValid = false;
      }
      continue;
   }

   if (strcmp(node->color, "red") == 0) {
      if ((node->left && strcmp(node->left->color, "red") == 0) ||
         (node->right && strcmp(node->right->color, "red") == 0)) {
         isValid = false;
```

```
            }
        }

        if (strcmp(node->color, "black") == 0) {
            currentBlackHeight++;
        }

        push(&stack, node->left, currentBlackHeight, pathBlackHeight);
        push(&stack, node->right, currentBlackHeight, pathBlackHeight);
    }

    if (strcmp(root->color, "black") != 0) {
        isValid = false;
    }

    printf("%s\n", isValid ? "Valid Red-Black Tree" : "Invalid Red-Black Tree");

    return 0;
}
```

Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

- **Data and Results:**



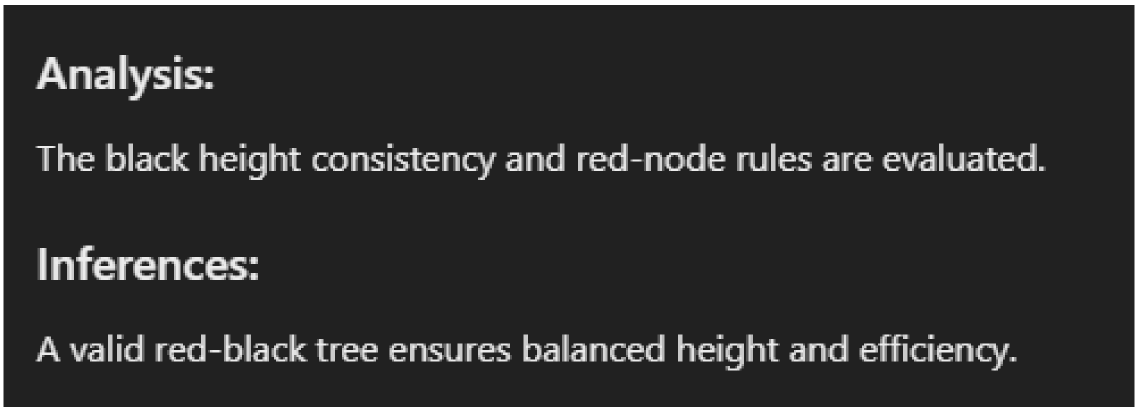
Data:

The tree contains nodes with values and red-black colors.

Result:

The tree is validated for all red-black tree properties.

- **Analysis and Inferences:**



Analysis:

The black height consistency and red-node rules are evaluated.

Inferences:

A valid red-black tree ensures balanced height and efficiency.

Course Title	Advanced Algorithms & Data Structures	ACADEMIC YEAR: 2024-25
Course Code	23CS03HF	

Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

In-Lab:

Problem :

1. Write a function to construct an AVL tree, where nodes are inserted while maintaining the balance property. After each insertion, the tree should self-balance to remain AVL. Print the inorder traversal of the tree after each insertion.

Input:

- An integer `n` ($1 \le n \le 10^4$) representing the number of nodes to insert.
- A list of `n` integers where each integer `x` ($1 \le x \le 10^6$) represents a value to be inserted into the AVL tree.

Output:

- After each insertion, print the inorder traversal of the tree as space-separated values.

Sample Input:

```
5
20 15 25 10 5
```

Sample Output:

```
20
15 20
15 20 25
10 15 20 25
5 10 15 20 25
```

Constraints:

- Each insertion should maintain the AVL property, with rotations performed as needed to keep the tree balanced.

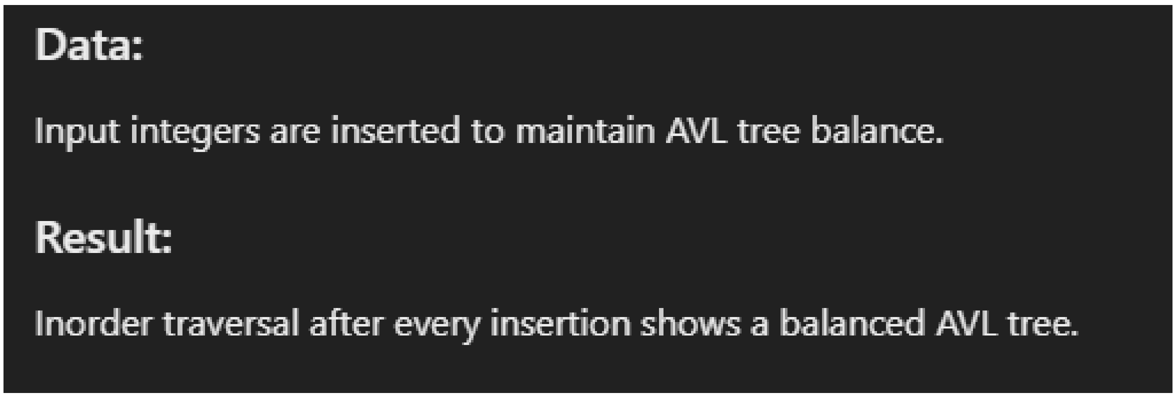
- **Procedure/Program**:

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

typedef struct TreeNode {
   int value;
   struct TreeNode* left;
   struct TreeNode* right;
   int height;
} TreeNode;

TreeNode* create_node(int key) {
   TreeNode* new_node = (TreeNode*)malloc(sizeof(TreeNode));
   new_node->value = key;
   new_node->left = new_node->right = NULL;
   new_node->height = 1;
   return new_node;
}

int get_height(TreeNode* node) {
   if (node == NULL)
      return 0;
   return node->height;
}
```

```
int get_balance(TreeNode* node) {
   if (node == NULL)
      return 0;
   return get_height(node->left) - get_height(node->right);
}

TreeNode* left_rotate(TreeNode* z) {
   TreeNode* y = z->right;
   TreeNode* T2 = y->left;

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

   z->height = 1 + (get_height(z->left) > get_height(z->right) ? get_height(z->left) :
get_height(z->right));
   y->height = 1 + (get_height(y->left) > get_height(y->right) ? get_height(y->left) :
get_height(y->right));

   return y;
}

TreeNode* right_rotate(TreeNode* z) {
   TreeNode* y = z->left;
   TreeNode* T3 = y->right;

   y->right = z;
   z->left = T3;
```

```
    z->height = 1 + (get_height(z->left) > get_height(z->right) ? get_height(z->left) : get_height(z->right));
    y->height = 1 + (get_height(y->left) > get_height(y->right) ? get_height(y->left) : get_height(y->right));

    return y;
}

TreeNode* insert(TreeNode* node, int key) {
    if (node == NULL)
        return create_node(key);

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

    node->height = 1 + (get_height(node->left) > get_height(node->right) ? get_height(node->left) : get_height(node->right));

    int balance = get_balance(node);

    if (balance > 1 && key < node->left->value)
        return right_rotate(node);
    if (balance < -1 && key > node->right->value)
        return left_rotate(node);

    if (balance > 1 && key > node->left->value) {
        node->left = left_rotate(node->left);
        return right_rotate(node);
    }

    if (balance < -1 && key < node->right->value) {
        node->right = right_rotate(node->right);
        return left_rotate(node);
    }

    return node;
}

void inorder(TreeNode* node) {
    if (node != NULL) {
        inorder(node->left);
        printf("%d ", node->value);
        inorder(node->right);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    int values[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    TreeNode* root = NULL;
    for (int i = 0; i < n; i++) {
        root = insert(root, values[i]);
        inorder(root);
        printf("\n");
    }

    return 0;
}
```

- **Data and Results:**



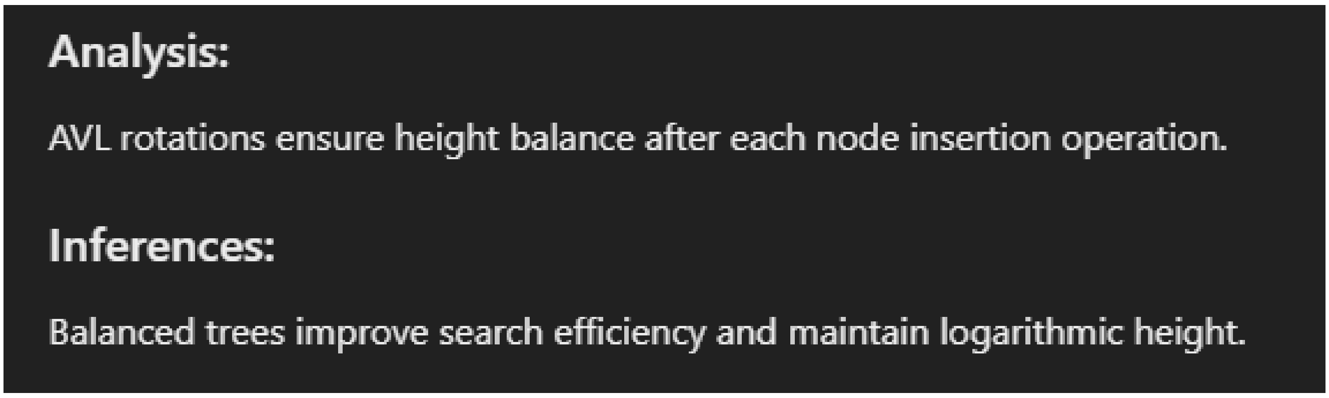
Data:

Input integers are inserted to maintain AVL tree balance.

Result:

Inorder traversal after every insertion shows a balanced AVL tree.

- **Analysis and Inferences:**



Analysis:

AVL rotations ensure height balance after each node insertion operation.

Inferences:

Balanced trees improve search efficiency and maintain logarithmic height.

2. You are given an unbalanced binary search tree, transform it into a Red-Black Tree. The program should read the values, build the binary search tree, and then balance it into a Red-Black Tree while maintaining its properties.

Input:

- A series of integers representing the values to be inserted into the binary search tree.

Output:

- Print the in-order traversal of the tree after balancing.

Sample Input:

- 50 30 70 20 40 60 80

Sample Output:

- 20 30 40 50 60 70 80

Constraints:

- The input tree must be transformed in O(n log n) time, and the Red-Black Tree properties should be verified after the transformation.

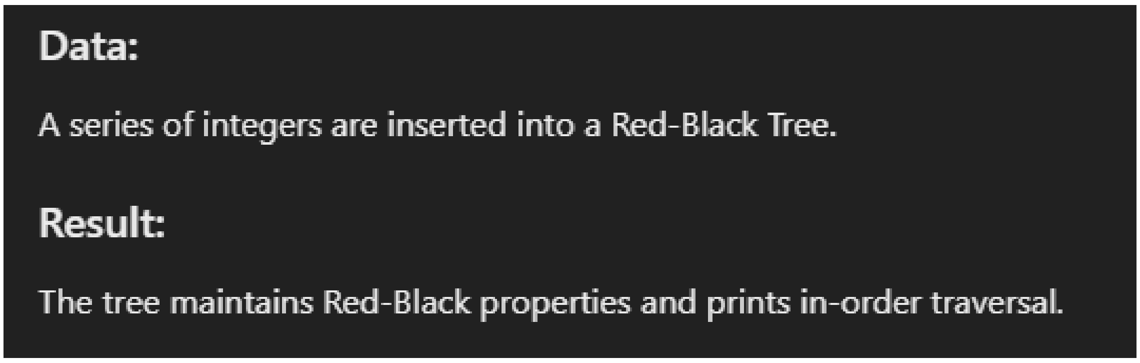
- **Procedure/Program**:

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

typedef enum { RED, BLACK } Color;

typedef struct TreeNode {
   int value;
   Color color;
   struct TreeNode *left, *right, *parent;
} TreeNode;
```

```
TreeNode *TNULL;

TreeNode* newNode(int value) {
   TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
   node->value = value;
   node->color = RED;
   node->left = node->right = node->parent = TNULL;
   return node;
}

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

void fixInsert(TreeNode** root, TreeNode* k) {
    TreeNode* u;
    while (k->parent->color == RED) {
        if (k->parent == k->parent->parent->left) {
            u = k->parent->parent->right;
            if (u->color == RED) {
                k->parent->color = BLACK;
                u->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->right) {
                    k = k->parent;
                    leftRotate(root, k);
                }
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                rightRotate(root, k->parent->parent);
            }
        } else {
            u = k->parent->parent->left;
            if (u->color == RED) {
                k->parent->color = BLACK;
                u->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->left) {
                    k = k->parent;
                    rightRotate(root, k);
                }
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                leftRotate(root, k->parent->parent);
            }
        }
        if (k == *root) {
            break;
        }
    }
    (*root)->color = BLACK;
}

void insert(TreeNode** root, int value) {
    TreeNode* node = newNode(value);
    TreeNode* y = TNULL;
    TreeNode* x = *root;

    while (x != TNULL) {
        y = x;
        if (value < x->value) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    node->parent = y;
    if (y == TNULL) {
        *root = node;
    } else if (value < y->value) {
        y->left = node;
    } else {
        y->right = node;
    }
    fixInsert(root, node);
}

void inorder(TreeNode* node) {
    if (node != TNULL) {
        inorder(node->left);
        printf("%d ", node->value);
        inorder(node->right);
    }
}

int main() {
    int n;
    scanf("%d", &n);

    // Initialize TNULL
    TNULL = (TreeNode*)malloc(sizeof(TreeNode));
    TNULL->color = BLACK;
    TNULL->left = TNULL->right = TNULL->parent = NULL;

    TreeNode* root = TNULL;
    for (int i = 0; i < n; i++) {
        int value;
        scanf("%d", &value);
        insert(&root, value);
    }

    inorder(root);
    printf("\n");

    return 0;
}
```

Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

- **Data and Results:**



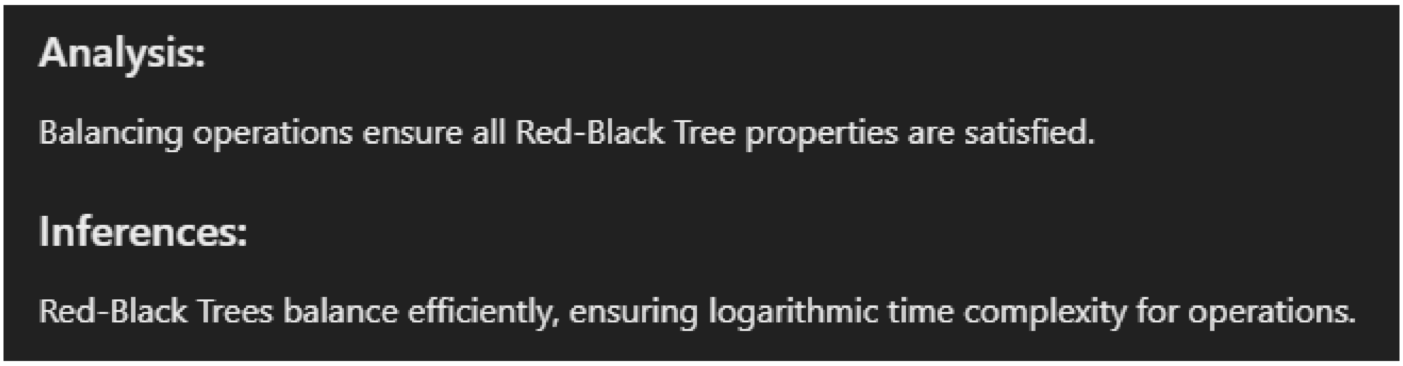
Data:

A series of integers are inserted into a Red-Black Tree.

Result:

The tree maintains Red-Black properties and prints in-order traversal.

- **Analysis and Inferences:**



Analysis:

Balancing operations ensure all Red-Black Tree properties are satisfied.

Inferences:

Red-Black Trees balance efficiently, ensuring logarithmic time complexity for operations.

Post-Lab:

1. Given the root of an AVL tree and a level k, count the number of nodes at that level.

Input:

- The root node of the AVL tree.
- An integer k ($1 \leq k \leq 10^4$) representing the level in the tree to count nodes.

Output:

- Print the number of nodes at the level k.

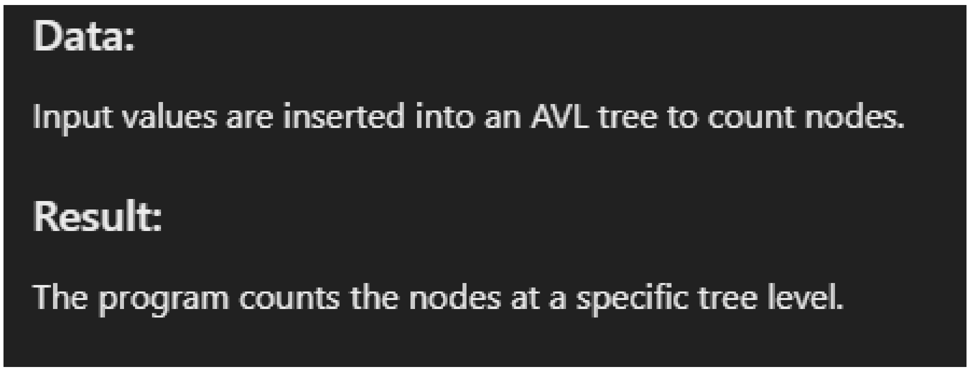
- **Procedure/Program**:

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

typedef struct TreeNode {
   int value;
   struct TreeNode *left;
   struct TreeNode *right;
} TreeNode;

TreeNode* createNode(int value) {
   TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
   newNode->value = value;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

int main() {
   int n, k;
   scanf("%d", &n);
   int values[n];
   for (int i = 0; i < n; i++) {
      scanf("%d", &values[i]);
   }
```

```
scanf("%d", &k);

TreeNode* root = NULL;
for (int i = 0; i < n; i++) {
   TreeNode* node = createNode(values[i]);
   if (!root) {
      root = node;
   } else {
      TreeNode* temp = root;
      while (1) {
         if (values[i] < temp->value) {
            if (temp->left == NULL) {
               temp->left = node;
               break;
            }
            temp = temp->left;
         } else {
            if (temp->right == NULL) {
               temp->right = node;
               break;
            }
            temp = temp->right;
         }
      }
   }
}

int count = 0;
TreeNode* queue[10000];
int level[10000];
int front = 0, rear = 0;

queue[rear] = root;
level[rear++] = 1;

while (front < rear) {
   TreeNode* node = queue[front];
```

```
        int levelNode = level[front++];
        if (levelNode == k) {
            count++;
        }

        if (node->left) {
            queue[rear] = node->left;
            level[rear++] = levelNode + 1;
        }
        if (node->right) {
            queue[rear] = node->right;
            level[rear++] = levelNode + 1;
        }
    }

    printf("%d\n", count);

    return 0;
}
```

- **Data and Results:**



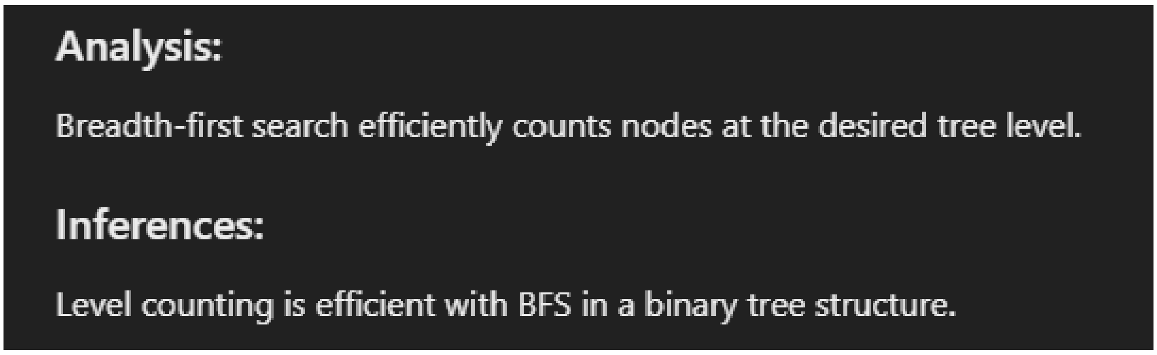
Data:

Input values are inserted into an AVL tree to count nodes.

Result:

The program counts the nodes at a specific tree level.

- **Analysis and Inferences:**



Analysis:

Breadth-first search efficiently counts nodes at the desired tree level.

Inferences:

Level counting is efficient with BFS in a binary tree structure.

2. Calculate the maximum depth of a given Red-Black Tree.

Input:

- The root node of a Red-Black tree, represented as a list of values that would form the tree structure.

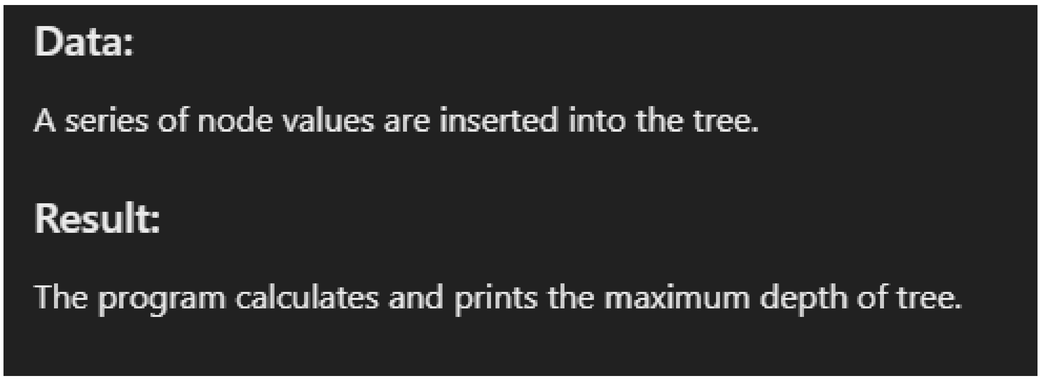
Output:

- Print the maximum depth as an integer.

- **Procedure/Program**:

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

typedef struct TreeNode {
    int value;
    char color[5];
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;

TreeNode* create_node(int value) {
    TreeNode* new_node = (TreeNode*)malloc(sizeof(TreeNode));
    new_node->value = value;
    new_node->left = new_node->right = NULL;
    snprintf(new_node->color, sizeof(new_node->color), "red");
    return new_node;
}

int max_depth(TreeNode* node) {
    if (node == NULL) {
        return 0;
    }
    int left_depth = max_depth(node->left);
    int right_depth = max_depth(node->right);
    return (left_depth > right_depth ? left_depth : right_depth) + 1;
}

int main() {
    int n;
    scanf("%d", &n);

    int values[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    TreeNode* root = NULL;

    for (int i = 0; i < n; i++) {
        TreeNode* node = create_node(values[i]);
        if (root == NULL) {
            root = node;
        } else {
            TreeNode* temp = root;
            while (1) {
                if (values[i] < temp->value) {
                    if (temp->left == NULL) {
                        temp->left = node;
                        break;
                    }
                    temp = temp->left;
                } else {
                    if (temp->right == NULL) {
                        temp->right = node;
                        break;
                    }
                    temp = temp->right;
                }
            }
        }
    }

    printf("%d\n", max_depth(root));

    return 0;
}
```

- **Data and Results:**



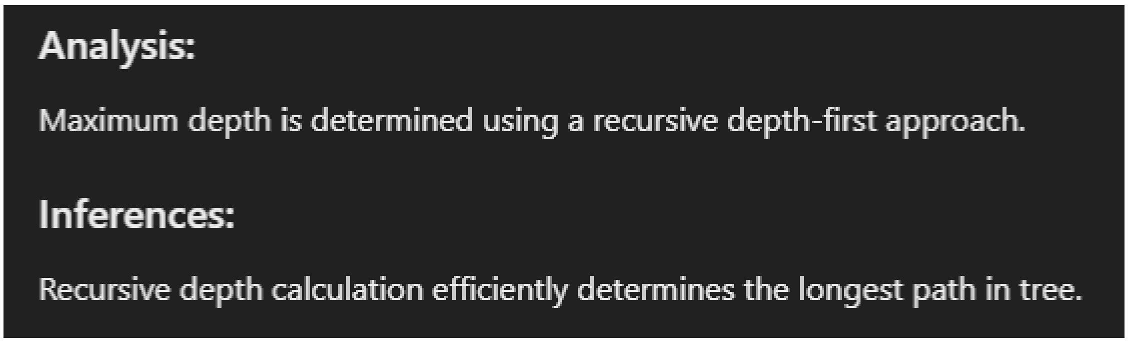
Data:

A series of node values are inserted into the tree.

Result:

The program calculates and prints the maximum depth of tree.

- **Analysis and Inferences:**



Analysis:

Maximum depth is determined using a recursive depth-first approach.

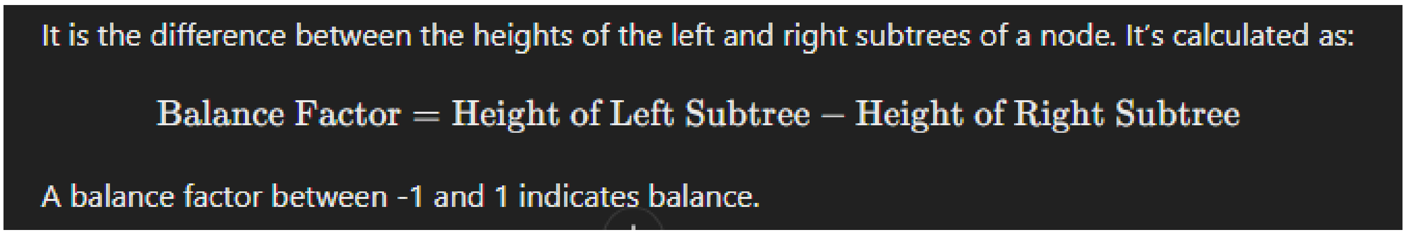
Inferences:

Recursive depth calculation efficiently determines the longest path in tree.

Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

- **Sample VIVA-VOCE Questions (In-Lab):**

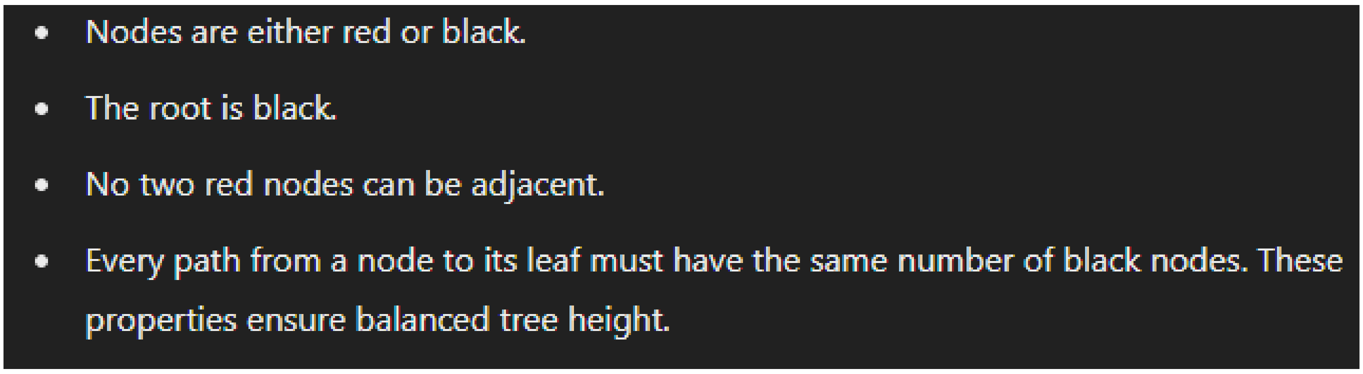
1. What is the balance factor in an AVL Tree? How do you calculate it?


It is the difference between the heights of the left and right subtrees of a node. It's calculated as:

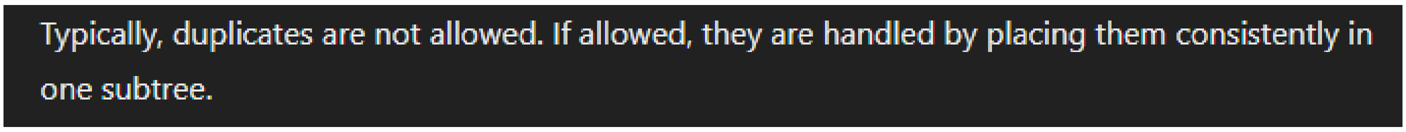
$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

A balance factor between -1 and 1 indicates balance.

2. What is the significance of the red and black properties in Red-Black Trees?

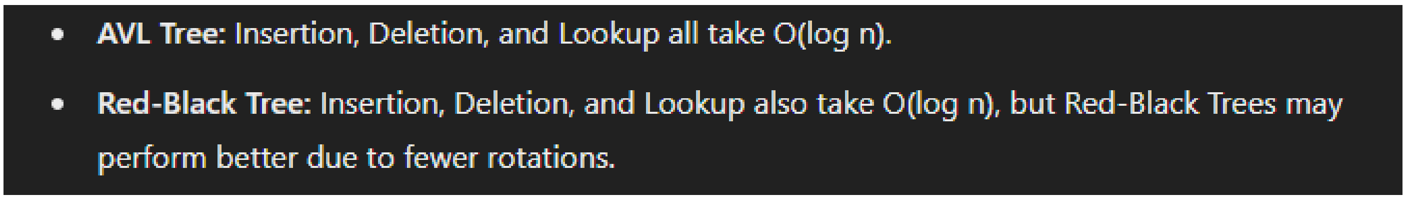

- Nodes are either red or black.
- The root is black.
- No two red nodes can be adjacent.
- Every path from a node to its leaf must have the same number of black nodes. These properties ensure balanced tree height.

3. Can an AVL Tree have more than one node with the same value? How does it handle duplicates?


Typically, duplicates are not allowed. If allowed, they are handled by placing them consistently in one subtree.

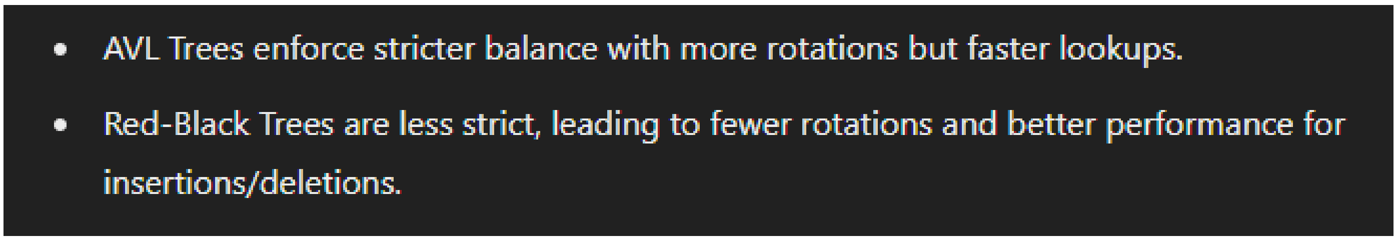
\

4. Compare the time complexity of insertion, deletion, and lookup in AVL Trees and Red-Black Trees.


- **AVL Tree:** Insertion, Deletion, and Lookup all take O(log n).
- **Red-Black Tree:** Insertion, Deletion, and Lookup also take O(log n), but Red-Black Trees may perform better due to fewer rotations.

Course Title	Advanced Algorithms & Data Structures	ACADEMIC YEAR: 2024-25
Course Code	23CS03HF	31 | P a g e

5. What is the difference between an AVL Tree and a Red-Black Tree in terms of balancing and performance?



- AVL Trees enforce stricter balance with more rotations but faster lookups.
- Red-Black Trees are less strict, leading to fewer rotations and better performance for insertions/deletions.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.