Output:

```
Tree #1 leaves: 3

Tree #1 non-leaves: 6

Tree #1 height: 4

Pre-order Tree #1: 1 2 4 7 3 5 6 8 9

In-order Tree #1: 7 4 2 1 5 3 6 8 9

Post-order Tree #1: 7 4 2 5 9 8 6 3 1

Tree #1 after removing leaves (Pre-order): 1 2 4 3 6 8

Tree #2 leaves: 4

Tree #2 non-leaves: 4

Tree #2 height: 3

Pre-order Tree #2: 6 4 2 1 3 8 5 7

In-order Tree #2: 1 2 3 4 6 5 8 7

Post-order Tree #2: 1 3 2 4 5 7 8 6

Tree #2 after removing leaves (Pre-order): 6 4 2 8
```

Bitree.h

```c
/*
 * bitree.h
 */
#ifndef BITREE_H
#define BITREE_H

#include <stdlib.h>

/* Define a structure for binary tree nodes. */
typedef struct BiTreeNode_ {

    void *data;
    struct BiTreeNode_ *left;
    struct BiTreeNode_ *right;

} BiTreeNode;

/* Define a structure for binary trees. */
typedef struct BiTree_ {
    int size;

    int (*compare)(const void *key1, const void *key2);
    void (*destroy)(void *data);

    BiTreeNode *root;

} BiTree;

/* Public Interface */
void bitree_init(BiTree *tree, void(*destroy)(void *data));

void bitree_destroy(BiTree *tree);

int bitree_ins_left(BiTree *tree, BiTreeNode *node, const void *data);

int bitree_ins_right(BiTree *tree, BiTreeNode *node, const void *data);

void bitree_rem_left(BiTree *tree, BiTreeNode *node);

void bitree_rem_right(BiTree *tree, BiTreeNode *node);

int bitree_merge(BiTree *merge, BiTree *left, BiTree *right, const void *data);

int count_leaves(BiTree *tree);//Returns the number of leaf nodes in the tree.

int count_non_leaves(BiTree *tree);//Returns the number of non-leaf nodes in the
tree.

int get_height(BiTree *tree);//Returns the height of the tree.

/*Prints the elements of the tree to stdout using a pre-order traversal. The
print
parameter should contain the logic to print the data held in each node in the
tree.*/
void print_pre_order(BiTree *tree, void (*print)(const void *data));

/*Prints the elements of the tree to stdout using an in-order traversal. The
print
parameter should contain the logic to print the data held in each node in the
tree.*/
void print_in_order(BiTree *tree, void (*print)(const void *data));

/*Prints the elements of the tree to stdout using a post-order traversal. The
print parameter should contain the logic to print the data held in each node in
```

```
the tree.*/
void print_post_order(BiTree *tree, void (*print)(const void *data));

/*Removes all leaf nodes from the tree. Use print_pre_order,
print_in_order, or print_post_order after calling remove_leaves
to show that remove_leaves successfully removed all leaves. */
void remove_leaves(BiTree *tree);

#define bitree_size(tree) ((tree)->size)

#define bitree_root(tree) ((tree)->root)

#define bitree_is_eob(node) ((node) == NULL)

#define bitree_is_leaf(node) ((node)->left == NULL && (node)->right == NULL)

#define bitree_data(node) ((node)->data)

#define bitree_left(node) ((node)->left)

#define bitree_right(node) ((node)->right)

#endif
```

Bitree.c

```
/*
 * bitree.c
 */
#include <stdlib.h>
#include <string.h>

#include "bitree.h"

void bitree_init(BiTree *tree, void(*destroy)(void *data)) {

    /* Initialize the binary tree. */
    tree->size = 0;
    tree->destroy = destroy;
    tree->root = NULL;
}

void bitree_destroy(BiTree *tree) {
```

```c
    /* Remove all the nodes from the tree. */
    bitree_rem_left(tree, NULL);

    /* No operations are allowed now, but clear the structure as a
     * precaution. */
    memset(tree, 0, sizeof(BiTree));
}

int bitree_ins_left(BiTree *tree, BiTreeNode *node, const void *data) {

    BiTreeNode *new_node, **position;

    /* Determine where to insert the node. */

    if (node == NULL) {

        /* Allow insertion at the root only in an empty tree. */
        if (bitree_size(tree) > 0)
            return -1;

        position = &tree->root;
    }
    else {

        /* Normally allow insertion only at the end of a branch. */
        if (bitree_left(node) != NULL)
            return -1;

        position = &node->left;
    }

    /* Allocate storage for the node. */
    if ((new_node = (BiTreeNode *) malloc(sizeof(BiTreeNode))) == NULL)
        return -1;

    /* Insert the node into the tree. */
    new_node->data = (void *) data;
    new_node->left = NULL;
    new_node->right = NULL;
    *position = new_node;

    /* Adjust the size of the tree to account for the inserted node. */
    tree->size++;

    return 0;
```

```c
}

int bitree_ins_right(BiTree *tree, BiTreeNode *node, const void *data) {

    BiTreeNode *new_node, **position;

    /* Determine where to insert the node. */
    if (node == NULL) {

        /* Allow insertion at the root only in an empty tree. */
        if (bitree_size(tree) > 0)
            return -1;

        position = &tree->root;
    }
    else {

        /* Normally allow insertion only at the end of a branch. */
        if (bitree_right(node) != NULL)
            return -1;

        position = &node->right;
    }

    /* Allocate storage for the node. */
    if ((new_node = (BiTreeNode *) malloc(sizeof(BiTreeNode))) == NULL)
        return -1;

    /* Insert the node into the tree. */
    new_node->data = (void *) data;
    new_node->left = NULL;
    new_node->right = NULL;
    *position = new_node;

    /* Adjust the size of the tree to account for the inserted node. */
    tree->size++;

    return 0;
}

void bitree_rem_left(BiTree *tree, BiTreeNode *node) {

    BiTreeNode **position;

    /* Do not allow removal from an empty tree. */
    if (bitree_size(tree) == 0)
        return;

    /* Determine where to remove nodes. */
    if (node == NULL)
        position = &tree->root;
    else
        position = &node->left;

    /* Remove the nodes. */
    if (*position != NULL) {

        bitree_rem_left(tree, *position);
        bitree_rem_right(tree, *position);

        if (tree->destroy != NULL) {

            /* Call a user-defined function to free dynamically allocated
             * data. */
            tree->destroy((*position)->data);
        }

        free(*position);
        *position = NULL;

        /* Adjust the size of the tree to account for the removed node. */
        tree->size--;
    }
}

void bitree_rem_right(BiTree *tree, BiTreeNode *node) {

    BiTreeNode **position;

    /* Do not allow removal from an empty tree. */
    if (bitree_size(tree) == 0)
        return;

    /* Determine where to remove nodes. */
    if (node == NULL)
        position = &tree->root;
    else
        position = &node->right;

    /* Remove the nodes. */
```

```c
    if (*position != NULL) {

        bitree_rem_left(tree, *position);
        bitree_rem_right(tree, *position);

        if (tree->destroy != NULL) {

            /* Call a user-defined function to free dynamically allocated
             * data. */
            tree->destroy((*position)->data);
        }

        free(*position);
        *position = NULL;

        /* Adjust the size of the tree to account for the removed node. */
        tree->size--;
    }
}

int bitree_merge(BiTree *merge, BiTree *left, BiTree *right, const void *data) {

    /* Initialize the merged tree. */
    bitree_init(merge, left->destroy);

    /* Insert the data for the root node of the merged tree. */
    if (bitree_ins_left(merge, NULL, data) != 0) {

        bitree_destroy(merge);
        return -1;
    }

    /* Merge the two binary trees into a single binary tree. */
    bitree_root(merge)->left = bitree_root(left);
    bitree_root(merge)->right = bitree_root(right);

    /* Adjust the size of the new binary tree. */
    merge->size = merge->size + bitree_size(left) + bitree_size(right);

    /* Do not let the original trees access the merged nodes. */
    left->root = NULL;
    left->size = 0;
    right->root = NULL;
    right->size = 0;
    return 0;
}
static int count_leaves_node(BiTreeNode *node) {
    if (node == NULL) {
        return 0;
    }

    if (node->left == NULL && node->right == NULL) {
        return 1;
    }

    return count_leaves_node(node->left) + count_leaves_node(node->right);
}

int count_leaves(BiTree *tree) {
    return count_leaves_node(tree->root);
}

static int count_non_leaves_node(BiTreeNode *node) {
    if (node == NULL || (node->left == NULL && node->right == NULL)) {
        return 0;
    }

    return 1 + count_non_leaves_node(node->left) +
count_non_leaves_node(node->right);
}

int count_non_leaves(BiTree *tree) {
    return count_non_leaves_node(tree->root);
}

static int get_height_node(BiTreeNode *node) {
    if (node == NULL) {
        return -1; // height of empty tree is -1
    }

    int left_height = get_height_node(node->left);
    int right_height = get_height_node(node->right);

    return (left_height > right_height ? left_height : right_height) + 1;
}

int get_height(BiTree *tree) {
    return get_height_node(tree->root);
}
```

```c
static void print_pre_order_node(BiTreeNode *node, void (*print)(const void
*data)) {
    if (node != NULL) {
        print(node->data);
        print_pre_order_node(node->left, print);
        print_pre_order_node(node->right, print);
    }
}

void print_pre_order(BiTree *tree, void (*print)(const void *data)) {
    print_pre_order_node(tree->root, print);
}

static void print_in_order_node(BiTreeNode *node, void (*print)(const void
*data)) {
    if (node != NULL) {
        print_in_order_node(node->left, print);
        print(node->data);
        print_in_order_node(node->right, print);
    }
}

void print_in_order(BiTree *tree, void (*print)(const void *data)) {
    print_in_order_node(tree->root, print);
}

static void print_post_order_node(BiTreeNode *node, void (*print)(const void
*data)) {
    if (node != NULL) {
        print_post_order_node(node->left, print);
        print_post_order_node(node->right, print);
        print(node->data);
    }
}

void print_post_order(BiTree *tree, void (*print)(const void *data)) {
    print_post_order_node(tree->root, print);
}

static void remove_leaves_node(BiTreeNode **node, BiTree *tree) {
    if (*node != NULL) {
        if ((*node)->left == NULL && (*node)->right == NULL) {
            if (tree->destroy != NULL) {
                tree->destroy((*node)->data);
            }
            free(*node);
            *node = NULL;
            tree->size--;
        } else {
            remove_leaves_node(&((*node)->left), tree);
            remove_leaves_node(&((*node)->right), tree);
        }
    }
}

void remove_leaves(BiTree *tree) {
    remove_leaves_node(&(tree->root), tree);
}
```

treeTest.c

```c
#include "bitree.h"
#include <stdio.h>

static int insert_int(BiTree *tree, BiTreeNode *node, int value, int is_left) {
    int *data = (int*)malloc(sizeof(int));
    if (data == NULL) {
        return -1; // Return an error if memory allocation fails
    }

    *data = value;

    // If the node is NULL, we are inserting the root node
    if (node == NULL) {
        if (bitree_size(tree) > 0) {
            free(data); // Avoid inserting root if the tree is not empty
            return -1;
        }
        return bitree_ins_left(tree, NULL, data);
    }

    // Insert the data to the left or right as needed
    return is_left ? bitree_ins_left(tree, node, data) : bitree_ins_right(tree,
node, data);
}

// Helper function to print integers
void print_int(const void *data) {
    printf("%d ", *(int *)data);
}


int main(){
    BiTree tree1, tree2;
    bitree_init(&tree1, free);
    bitree_init(&tree2, free);

    // Build Tree #1
    insert_int(&tree1, NULL, 1, 1); // Insert root
    BiTreeNode *node;

    node = bitree_root(&tree1);
    insert_int(&tree1, node, 2, 1); // Insert left child of root
    insert_int(&tree1, node, 3, 0); // Insert right child of root

    node = bitree_left(node);
    insert_int(&tree1, node, 4, 1); // Continue inserting for left subtree

    node = bitree_left(node);
    insert_int(&tree1, node, 7, 1); // Continue inserting for left subtree

    node = bitree_root(&tree1); // Reset to root to insert right subtree
    node = bitree_right(node);
    insert_int(&tree1, node, 5, 1); // Insert left child of right subtree of root
    insert_int(&tree1, node, 6, 0); // Insert right child of right subtree of
root

    node = bitree_right(node);
    insert_int(&tree1, node, 8, 0); // Continue inserting for right subtree

    node = bitree_right(node);
    insert_int(&tree1, node, 9, 0); // Continue inserting for right subtree

    insert_int(&tree2, NULL, 6, 1); // Insert root for Tree #2

    //Build tree2
    BiTreeNode *node2;
    node2 = bitree_root(&tree2);

    // Insert left subtree
    insert_int(&tree2, node2, 4, 1);

    // Work on left child of the left subtree
    node2 = bitree_left(node2);
    insert_int(&tree2, node2, 2, 1); // Insert left child

    node2 = bitree_left(node2); // Go to left child
    insert_int(&tree2, node2, 1, 1); // Insert left child
    insert_int(&tree2, node2, 3, 0); // Insert right child

    // Move back to root to work on the right subtree
    node2 = bitree_root(&tree2);

    // Insert right subtree
    insert_int(&tree2, node2, 8, 0);

    // Work on the right child of the right subtree
    node2 = bitree_right(node2);
    insert_int(&tree2, node2, 5, 1); // Insert left child
    insert_int(&tree2, node2, 7, 0); // Go to right child
    insert_int(&tree2, node2, 9, 0); // Insert right child of right subtree


    /* Test functions */
    //tree1
    printf("Tree #1 leaves: %d\n", count_leaves(&tree1));
    printf("Tree #1 non-leaves: %d\n", count_non_leaves(&tree1));
    printf("Tree #1 height: %d\n", get_height(&tree1));

    printf("Pre-order Tree #1: ");
    print_pre_order(&tree1, print_int);
    printf("\n");

    printf("In-order Tree #1: ");
    print_in_order(&tree1, print_int);
    printf("\n");

    printf("Post-order Tree #1: ");
    print_post_order(&tree1, print_int);
    printf("\n");

    /* Remove leaves and print again */
    remove_leaves(&tree1);
    printf("Tree #1 after removing leaves (Pre-order): ");
    print_pre_order(&tree1, print_int);
    printf("\n");

    //tree2
```

```c
    printf("Tree #2 leaves: %d\n", count_leaves(&tree2));
    printf("Tree #2 non-leaves: %d\n", count_non_leaves(&tree2));
    printf("Tree #2 height: %d\n", get_height(&tree2));

    printf("Pre-order Tree #2: ");
    print_pre_order(&tree2, print_int);
    printf("\n");

    printf("In-order Tree #2: ");
    print_in_order(&tree2, print_int);
    printf("\n");

    printf("Post-order Tree #2: ");
    print_post_order(&tree2, print_int);
    printf("\n");

    /* Remove leaves and print again */
    remove_leaves(&tree2);
    printf("Tree #2 after removing leaves (Pre-order): ");
    print_pre_order(&tree2, print_int);
    printf("\n");

    return 0;
}
```