**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**

/\*

 \* 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) {

    /\* 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;

}

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. \*/

    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);

}

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**

#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

    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;

}