שם: אביתר קורקוס תז: 207983198  
שם: נגה חסיד תז: 207486598

מטלה 5

קישור ל- GitHub Repository עבור המטלה: <https://github.com/EvyaKor/Adv_C_Exe6>

**1-6.**

**BST.C**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include "BST.h"

#include "TreePrintLibrary.h"

// Macro to check if memory allocation failed

#ifndef CHECK\_MEMORY\_ALLOCATION

#define CHECK\_MEMORY\_ALLOCATION(ptr) do { \

if(ptr == NULL) \

{ \

printf("Memory allocation failed\n"); \

exit(1); \

} \

} while (0)

#endif

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Private functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Creates new binary tree node

TreeNode\* createNode(int value)

{

TreeNode\* newNode = (TreeNode\*)calloc(1, sizeof(TreeNode)); // Memory allocation - using calloc to set pointers to null

CHECK\_MEMORY\_ALLOCATION(newNode); // Check memory allocation using macro

newNode->element = value;

return newNode;

}

// Inserts tree node to the correct place in tree using recursion

void recursiveInsertion(TreeNode\* root, TreeNode\* newNode)

{

if (newNode->element <= root->element) // In case value belongs to left sub-tree

{

if (root->left == NULL)

root->left = newNode;

else

recursiveInsertion(root->left, newNode); // Recursive call

}

else // In case value belongs to right sub-tree

{

if (root->right == NULL)

root->right = newNode;

else

recursiveInsertion(root->right, newNode); // Recursive call

}

}

// Prints binary tree in order using recursion

void recursivePrinting(TreeNode\* root)

{

if (root != NULL)

{

recursivePrinting(root->left); // Recursive call

printf("%d, ", root->element); // In-order

recursivePrinting(root->right); // Recursive call

}

}

// Frees allocated memory of binary tree using recursion

void recursiveFree(TreeNode\* root)

{

if (root == NULL) return; // Stop condition

recursiveFree(root->left); // Recursive call

recursiveFree(root->right); // Recursive call

// Free allocated memory

free(root);

}

// Counts how many elements the binary tree has using recursion

int recursiveCountTreeSize(TreeNode\* root)

{

if (root == NULL)

{

return 0; // Stop condition

}

return 1 + recursiveCountTreeSize(root->left) + recursiveCountTreeSize(root->right); // Recursive call

}

// Finds the index N from the end of tree using recursion

int recursiveFindIndexN(TreeNode\* root, int N)

{

if (root == NULL) // Stop condition

{

return;

}

int treeSize = 1 + recursiveCountTreeSize(root->right); // Get right sub-tree size by calling recursiveCountTreeSize + 1 for the root of the tree

if (treeSize == N) // Return the index N from the end of tree

{

return root->element;

}

if (treeSize <= N) // In case the index N is in right sub-tree

{

return recursiveFindIndexN(root->left, N - treeSize); // Recursive call

}

else // In case the index N is in left sub-tree

{

return recursiveFindIndexN(root->right, N); // Recursive call

}

}

// Checks if all the leaves in the binary tree are in the same height using recursion

void recursiveCheckIfSameHeight(TreeNode\* root, int level, int\* temp)

{

if (root->right == NULL && root->left == NULL) // Stop condition - reached a leaf

{

if (\*temp == 0)

{

\*temp = level;

}

else

{

if (\*temp != level)

{

\*temp = -1; // '-1' indicates leaves are not in the same height

}

}

return;

}

if (root->right != NULL) // Recursive call to right sub-tree

{

recursiveCheckIfSameHeight(root->right, level + 1, temp);

}

if (root->left != NULL) // Recursive call to left sub-tree

{

recursiveCheckIfSameHeight(root->left, level + 1, temp);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Shell/Interface functions of the administrative structure of the binary search tree \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Initiates binary search tree

void initBST(BST \* bst)

{

bst->root = NULL;

}

// Inserts new binary tree node with given value to tree by calling recursiveInsertion

void insertBST(BST\* bst, int value)

{

TreeNode\* newNode = createNode(value); // Create new tree node

if (bst->root == NULL) // In case tree is empty

{

bst->root = newNode;

return;

}

// Call the recursive function to insert new node to the correct place in tree

recursiveInsertion(bst->root, newNode);

}

// Prints binary tree in order by calling recursivePrinting

void printTreeInorder(BST\* bst)

{

if (bst->root == NULL) // In case tree is empty

{ // case empty tree

printf("Binary tree is empty\n");

return;

}

// Call the recursive function for printing

recursivePrinting(bst->root);

printf("\n");

}

// Destroys binary tree by calling recursiveFree

void destroyBST(BST\* bst)

{

recursiveFree(bst->root); // Call the recursive function for freeing allocated memory

bst->root = NULL; // Set root to null - empty tree

}

// Finds the N index from the biggest element in tree (in reverse) by calling recursiveFindIndexN

int findIndexNFromLast(BST\* bst, int N)

{

if (bst->root == NULL) //In case tree is empty

{

printf("Binary tree is empty. Default value returned 0\n");

return 0;

}

if (N < 1) // In case user entered N that is smaller than 1

{

printf("You enterted an invalid index! Please enter N that is bigger than 1. Default value returned 0\n");

return 0;

}

int treeSize = recursiveCountTreeSize(bst->root); // Get the binary tree size by calling recursiveCountTreeSize

if (N > treeSize) // In case user entered N that is bigger than the binary tree size

{

printf("You enterted an invalid index! Please enter N that is smaller than or equal to %d. Default value returned 0\n", treeSize);

return NULL;

}

//find index from largest number in recursive function and return

return recursiveFindIndexN(bst->root, N);

}

// Checks if all the leaves in the binary tree are in the same height by calling recursiveCheckIfSameHeight

int sameHeightLeaves(BST\* bst)

{

if (bst->root == NULL) //In case tree is empty

{

printf("Binary tree is empty. Default value returned 0\n");

return 0;

}

int var = 0;

recursiveCheckIfSameHeight(bst->root, 0, &var); // Check if all the leaves in the tree are in the same height by calling recursiveCheckIfSameHeight and update var accordingly

if (var == -1) // In case not all leaves are in the same height

{

return 0;

}

else // In case all leaves are in the same height

{

return 1;

}

}

**Main .c**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <crtdbg.h>

#include "BST.h"

#include "TreePrintLibrary.h"

void main()

{

BST Test;

initBST(&Test); // Test initBST

// Test functions with empty tree

print\_ascii\_tree(Test.root);

printf("\n");

printTreeInorder(&Test); // Test printTreeInorder + recursivePrinting

printf("\n");

printf("Tree size is: %d\n", recursiveCountTreeSize(Test.root)); // Test recursiveCountTreeSize

findIndexNFromLast(&Test, 0); // Test findIndexNFromLast

if (sameHeightLeaves(&Test))

{

printf("All leaves in the binary tree are in the same height\n");

}

else

{

printf("Not all leaves in the binary tree are in the same height\n");

}

// Test functions with full tree

printf("\n");

printf("After inserting elements to binary tree:\n");

insertBST(&Test, 5); // Test insertBST + recursiveInsertion

insertBST(&Test, 2);

insertBST(&Test, 1);

insertBST(&Test, 7);

insertBST(&Test, 3);

insertBST(&Test, 9);

insertBST(&Test, 8);

insertBST(&Test, 1);

insertBST(&Test, 4);

insertBST(&Test, 12);

print\_ascii\_tree(Test.root);

printf("\n");

printf("Print Tree Inorder:\n");

printTreeInorder(&Test); // Test printTreeInorder + recursivePrinting

printf("\n");

printf("Tree size is: %d\n", recursiveCountTreeSize(Test.root)); // Test recursiveCountTreeSize

int N = 8;

printf("The element in the N=%d place (from the end) is: %d\n", N, findIndexNFromLast(&Test, N)); // Test findIndexNFromLast

if (sameHeightLeaves(&Test))

{

printf("All leaves in the binary tree are in the same height\n");

}

else

{

printf("Not all leaves in the binary tree are in the same height\n");

}

// Test destroyBST + recursiveFree

printf("\n");

destroyBST(&Test);

if (Test.root == NULL)

{

printf("Binary tree was destroyed\n");

}

\_CrtDumpMemoryLeaks(); // Check for memory leaks

}

