Contents

1 Basic Test Results 2

2 RBTree.c 4

3 Structs.c 17

1 Basic Test Results

```
"MacBook Pro" is in the tree.
"iPod" is not in the tree.
"iPhone" is in the tree.
"iPad" is in the tree.
"Apple Watch" is in the tree.
"Apple TV" is not in the tree.

The number of products in the tree is 4.

Name: Apple Watch.        Price: 299.00
Name: MacBook Pro.        Price: 1499.00
Name: iPad.        Price: 499.00
Name: iPhone.        Price: 599.00
test passed
Running...

Opening tar file
OK
Tar extracted O.K.

Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
Compiling...
OK
Compiling...
OK
Compiling...
OK
Compiling...
OK
Compiling...
OK
Compilation seems OK! Check if you got warnings!


=====================
 Public test cases
=====================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~    ProductExample output:    ~

Running test...
OK

~ End of ProductExample output ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Test Succeeded.
=====================

***********************************
*                                 *
*   presubmission script passed   *
*                                 *
```

```
***********************************

========================
= Checking coding style =
========================
RBTree.c(589, 6):  deep_blocks {Do not make too deep block(6). It makes not readable code}
RBTree.c(737, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
RBTree.c(737, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
RBTree.c(737, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
 ** Total Violated Rules      : 4
 ** Total Errors Occurs       : 4
 ** Total Violated Files Count: 1
```

2 RBTree.c

```c
#ifndef RBTREE_C
#define RBTREE_C

/**
 * @file RBTree.c
 * @author  Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
 * @ID 300575297
 * @date 23 May 2020
 *
 *
 * @section DESCRIPTION
 *              Program that implemented the Red-Black Tree.
 */

// ----------------------------- includes -------------------------------

#include "RBTree.h"

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

// ----------------------------- macros & constants -------------------------------

#define LESS (-1)
#define EQUAL (0)
#define GREATER (1)

// ----------------------------- addition functions -------------------------------

Node *newRBNode(void *data);
void removeNode(RBTree *tree, Node **node);
void rotateLeft(RBTree *tree, Node *pivot);
void rotateRight(RBTree *tree, Node *pivot);
void helperInsertToRBTree(RBTree *tree, Node *new);
void insertionFixup(RBTree *tree, Node *node);
void helperDeleteFromRBTree(RBTree *tree, Node *node);
void deletionFixup(RBTree *tree, Node *node, Node * parent);
Node *findRBTree(const RBTree *tree, const void *data);
Node *getMin(Node *root);
int inOrderTraverse(const Node *root, forEachFunc func, void *args);

// ------------------------------------------------------------------------------

/**
 * @brief: constructs a new empty RBTree with the given compFunc & freeFunc.
 * @param compFunc: a function to compare two variables.
 * @param freeFunc: a function to free a data item.
 * @return: a pointer to RBTree.
 */
RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
{
    RBTree *newEmptyRBTree = (RBTree *)malloc(sizeof(RBTree));
    if (newEmptyRBTree == NULL)
    {
        return NULL;
    }
    newEmptyRBTree->root = NULL;
```

```c
    newEmptyRBTree->compFunc = compFunc;
    newEmptyRBTree->freeFunc = freeFunc;
    newEmptyRBTree->size = 0;

    return newEmptyRBTree;
}

/**
 * @brief: creates new RED-BLACK Node.
 * @param data: the value of the new node.
 * @return: a pointer to Node on success, otherwise NULL on failure.
 */
Node *newRBNode(void *data)
{
    if (data == NULL)
    {
        return NULL;
    }
    Node *RBNode = (Node *)malloc(sizeof(Node));
    if (RBNode == NULL)
    {
        return NULL;
    }
    RBNode->data = data;
    RBNode->color = RED;
    RBNode->parent = RBNode->left = RBNode->right = NULL;

    return RBNode;
}

/**
 * @brief removes a node and the tree it induced.
 * @param tree: RBTree to remove the node from.
 * @param node: the node to remove.
 */
void removeNode(RBTree *tree, Node **node)
{
    if ((*node) == NULL)
    {
        return;
    }
    removeNode(tree, &((*node)->right));
    removeNode(tree, &((*node)->left));

    tree->freeFunc((*node)->data);
    free(*node);
    *node = NULL;
}

/**
 * @brief get the minimum node of the RBTree spanned by the given root.
 * @param root: root of RBTree.
 * @return: the left most node in the RBTree.
 */
Node *getMin(Node *root)
{
    if (root == NULL)
    {
        return NULL;
    }
    while (root->left != NULL)
    {
        root = root->left;
    }
    return root;
}

/**
```

```c
 * @brief rotates left over the pivot node
 * @param tree: the tree to execute the rotation in.
 * @param pivot: the node to rotate over.
 */
void rotateLeft(RBTree *tree, Node *pivot)
{
    // WE ASSUME THAT (pivot->right != NULL)
    Node *ptrToRight = pivot->right;  // pointer to right child of pivot
    pivot->right = ptrToRight->left;
    // updates the attributes of pivot new right child
    if (ptrToRight->left != NULL)
    {
        ptrToRight->left->parent = pivot;
    }
    // updates ptrToRight's parent
    ptrToRight->parent = pivot->parent;
    if (pivot->parent == NULL)  // pivot was the root of the tree
    {
        tree->root = ptrToRight;
    }
    else if (pivot == pivot->parent->left)  // pivot was a left child
    {
        pivot->parent->left = ptrToRight;
    }
    else  // pivot was a right child
    {
        pivot->parent->right = ptrToRight;
    }
    // updates connection between ptrToRight and pivot
    ptrToRight->left = pivot;
    pivot->parent = ptrToRight;
}

/**
 * @brief rotates right over the pivot node
 * @param tree: the tree to execute the rotation in.
 * @param pivot: the node to rotate over.
 */
void rotateRight(RBTree *tree, Node *pivot)
{
    // WE ASSUME THAT (pivot->left != NULL)
    Node *ptrToLeft = pivot->left;  // pointer to left child of pivot
    pivot->left = ptrToLeft->right;
    // updates the attributes of pivot new right child
    if (ptrToLeft->right != NULL)
    {
        ptrToLeft->right->parent = pivot;
    }
    // updates ptrToLeft's parent
    ptrToLeft->parent = pivot->parent;
    if (pivot->parent == NULL)  // pivot was the root of the tree
    {
        tree->root = ptrToLeft;
    }
    else if (pivot == pivot->parent->left)  // pivot was a left child
    {
        pivot->parent->left = ptrToLeft;
    }
    else  // pivot was a right child
    {
        pivot->parent->right = ptrToLeft;
    }
    // updates connection between ptrToRight and pivot
    ptrToLeft->right = pivot;
    pivot->parent = ptrToLeft;
}

/**
```

```c
 * @brief: inserts an item to the tree.
 * @param tree: the tree to add an item to.
 * @param data: item to insert to the tree.
 * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
 */
int insertToRBTree(RBTree *tree, void *data)
{
    if (tree == NULL || data == NULL || RBTreeContains(tree, data))
    {
        return false;
    }

    Node *new = newRBNode(data);
    if (new == NULL)  // create newRBNode fails
    {
        return false;
    }

    helperInsertToRBTree(tree, new);

    tree->size++;  // updates the tree size.

    return true;
}

/**
 * @brief: helper for insertToRBTree
 * @param tree: the tree to add the given node to.
 * @param new: the new node to be added.
 */
void helperInsertToRBTree(RBTree *tree, Node *new)
{
    Node *parent = NULL;  // parent to the new node
    Node *cur = tree->root;

    while (cur != NULL)
    {
        parent = cur;
        int res = tree->compFunc(cur->data, new->data);
        if (res < 0)  // cur->data is less than new->data
        {
            cur = cur->right;
        }
        else  // cur->data is greater than new->data
        {
            cur = cur->left;
        }
    }
    new->parent = parent;

    /* determine if the newly add node is a root, right child, or left child. */
    if (parent == NULL)
    {
        /* RBTree was empty, the newly added node is root */
        tree->root = new;
    }
    else if ((tree->compFunc(parent->data, new->data)) < 0)
    {
        /* parent->data is less than new->data */
        parent->right = new;   // the newly added node is right child
    }
    else
    {
        /* parent->data is greater than new->data */
        parent->left = new;  // the newly added node is left child
    }
    /* fix up the violation of RBTree properties */
    insertionFixup(tree, new);
```

```c
}

/**
 * @brief: fixes up possible violations caused by insertion to RBTree.
 * @param tree: the tree to add an item to.
 * @param node: the node caused the violation.
 */
void insertionFixup(RBTree *tree, Node *node)
{
    /* the inserted red leaf may be a child of a red node, so we have to
     * fix the parent coloring recursively
     */
    Node *cur = node;
    Node *grandparent = NULL;
    Node *uncle = NULL;

    // the inserted leaf is not the root, and its parent is not black.
    while (cur != tree->root && cur->parent->color == RED)
    {
        grandparent = cur->parent->parent;
        // parent is a left child, and uncle is a right child  (CASE 3)
        if (cur->parent == grandparent->left)
        {
            uncle = grandparent->right;
            // both parent and uncle are red
            if (uncle != NULL && uncle->color == RED)
            {
                // color parent, uncle, and grandparent by complement
                cur->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;

                cur = grandparent;  // move the problem to the grandparent
            }
            else  // uncle is black colored node, ordinary or RB leaf.
            {
                /* if the node is an inner node: right child of left child (CASE 4.A),
                 * then rotate the parent subtree to left, so the parent becomes an outer
                 * leaf: left child of the current node (CASE 4.B)
                 */

                if (cur == cur->parent->right)
                {
                    cur = cur->parent;
                    rotateLeft(tree, cur);
                }
                // color the parent black and the grandparent red
                cur->parent->color = BLACK;
                grandparent->color = RED;
                // rotate to right the grandparent's subtree
                rotateRight(tree, grandparent);
            }
        }
        else
        {
            /* the symmetric case:
             * the red parent is a right child, the uncle is the left child of
             * the grandparent  (CASE 3)
             */
            uncle = grandparent->left;

            // both parent and uncle are red
            if (uncle != NULL && uncle->color == RED)
            {
                // color parent, uncle, and grandparent by complement
                cur->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;
```

```c

                cur = grandparent;  // move the problem to the grandparent.
            }
            else  // uncle is black colored node, ordinary or RB leaf.
            {
                /* if the node is an inner node: left child of right child (CASE 4.A),
                 * then rotate the parent subtree to right, so the parent becomes an outer
                 * leaf: right child of the current node (CASE 4.B)
                 */
                if (cur == cur->parent->left)
                {
                    cur = cur->parent;
                    rotateRight(tree, cur);
                }
                // color the parent black and the grandparent red.
                cur->parent->color = BLACK;
                grandparent->color = RED;
                // rotate to left the grandparent's subtree.
                rotateLeft(tree, grandparent);
            }
        }
    }
    // Make sure that the root is black   (CASE 1)
    tree->root->color = BLACK;
}

/**
 * @brief: deletes an item to the tree.
 * @param tree: the tree to delete an item from.
 * @param data: item to delete from the tree.
 * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
 */
int deleteFromRBTree(RBTree *tree, void *data)
{
    if (tree == NULL || data == NULL)
    {
        return false;
    }
    Node *nodeToDel = findRBTree(tree, data);
    if (nodeToDel == NULL)
    {
        return false;
    }

    helperDeleteFromRBTree(tree, nodeToDel);

    tree->size--;  // updates the tree size.
    return true;
}

/**
 * @brief swaps between two RB nodes
 * @param tree: RBTree.
 * @param node1: first RB node.
 * @param node2: second RB node.
 */
void swapValues(RBTree *tree, Node **node1, Node **node2)
{
    /* check if second node is a right direct child of first node. */
    int directChild = ((*node1)->right == (*node2));

    Node *tempParent = (*node2)->parent;
    Node *tempLeft = (*node2)->left;
    Node *tempRight = (*node2)->right;
    Color tempColor = (*node2)->color;

    /* reset the pointers from second the node. */
    (*node2)->parent = (*node1)->parent;
```

```
    (*node2)->left = (*node1)->left;
    (*node2)->right = (directChild) ? (*node1) : (*node1)->right;
    (*node2)->color = (*node1)->color;

    /* reset the pointers from first the node. */
    (*node1)->parent = (directChild) ? (*node2) : tempParent;
    (*node1)->left = tempLeft;
    (*node1)->right = tempRight;
    (*node1)->color = tempColor;

    /* reset the pointers to first the node. */
    if (! directChild)
    {
        if ((*node2) == (*node1)->parent->left)
        {
            (*node1)->parent->left = (*node1);
        }
        else
        {
            (*node1)->parent->right = (*node1);
        }
    }

    if ((*node1)->left != NULL)
    {
        (*node1)->left->parent = (*node1);
    }
    if ((*node1)->right != NULL)
    {
        (*node1)->right->parent = (*node1);
    }

    /* reset the pointers to second the node. */
    if ((*node2)->parent != NULL)
    {
        if ((*node1) == (*node2)->parent->left)
        {
            (*node2)->parent->left = (*node2);
        }
        else
        {
            (*node2)->parent->right = (*node2);
        }
    }
    else
    {
        tree->root = (*node2);
    }

    if ((*node2)->left != NULL)
    {
        (*node2)->left->parent = (*node2);
    }
    if ((*node2)->right != NULL)
    {
        (*node2)->right->parent = (*node2);
    }
}

/**
 * @brief helper for deleteFromRBTree.
 * @param tree: the tree to delete an item from
 * @param node: the node to be deleted from the tree.
 */
void helperDeleteFromRBTree(RBTree *tree, Node *node)
{
    if (tree->size == 1)
    {
```

```c
        removeNode(tree, &node);
        tree->root = NULL;
        return;
    }

    if ((node->right != NULL) && (node->left != NULL))
    {
        /* the node to delete has two children (that are NOT NULL),
         * we swap the node with its successor.
         */
        Node *successor = getMin(node->right);
        swapValues(tree, &node, &successor);
    }

    /* The node to delete is now has at most one child, because
     * in the case of having two children we swap it with its
     * successor which has at most one child (the right one).
     */
    Node *child = node->left ? node->left : node->right;
    // get a pointer to the node's parent ot use it in fix up the violation.
    Node *parent = node->parent;

    if (child != NULL)  // NOT both of children are NULL
    {
        child->parent = node->parent;
    }

    if (node->parent == NULL)  // the node is the root
    {
        tree->root = child;
    }
    else
    {
        if (node == node->parent->left)
        {
            node->parent->left = child;
        }
        else
        {
            node->parent->right = child;
        }
    }

    /* RBTree properties could be violated only if the color of the
     * deleted node is black */
    if (node->color == BLACK)
    {
        deletionFixup(tree, child, parent);
    }

    // delete the node, it is not connected to the tree anymore.
    node->left = NULL;
    node->right = NULL;
    removeNode(tree, &node);
}

/**
 * @brief fixes up possible violations caused by deletion from RBTree.
 * @param tree: the tree to add an item to.
 * @param node: the node caused the violation.
 */
void deletionFixup(RBTree *tree, Node *node, Node * parent)
{
    /* Get a pointer to the current node and determine its color */
    Node *curr = node;
    Color currColor = curr ? curr->color : BLACK;

    while ((curr != tree->root) && (currColor == BLACK))
```

```c
{
    /* Get pointers to the current node's parent and sibling */
    Node *currParent = curr ? curr->parent : parent;
    Node *sibling = NULL;

    if (curr == currParent->left)
    {
        /* If the current node is a left child, then its sibling is the right
         * child of the parent.
         */
        sibling = currParent->right;

        /* Check the sibling's color. (NULL nodes are colored black) */
        if ((sibling != NULL) && (sibling->color == RED))
        {
            /* In case the sibling is red, color it black and rotate.
            * Then color the parent red (and the grandparent is now black).
            */
            sibling->color = BLACK;
            currParent->color = RED;
            rotateLeft(tree, currParent);
            sibling = currParent->right;
        }

        if ((sibling != NULL) &&
            (!(sibling->left) || sibling->left->color == BLACK) &&
            (!(sibling->right) || sibling->right->color == BLACK))
        {
            /* If the sibling has two black children, color it red */
            sibling->color = RED;
            if (currParent->color == RED)
            {
                /* If the parent is red, color it black and terminate
                * the fix-up process.
                */
                currParent->color = BLACK;
                curr = tree->root;      /* In order to stop the while loop */
            }
            else
            {
                /* The black depth of the entire sub-tree rooted at the parent is
                 * now too small - fix it up recursively.
                 */
                curr = currParent;
            }

        }
        else
        {
            if (sibling == NULL)
            {
                /* The case of a NULL sibling */
                if (currParent->color == RED)
                {
                    currParent->color = BLACK;
                    curr = tree->root;    /* In order to stop the while loop */
                }
                else
                {
                    curr = currParent;
                }
            }
            else
            {
                /* In this case, at least one of the sibling's children is red.
                 * It is therefore obvious that the sibling itself is black.
                 */
                if ((sibling->left != NULL) && (sibling->left->color == RED))
```

```
                    {
                        /* If the left child of the sibling is red, color it black,
                         * then color the sibling itself red, and rotate right around
                         * the sibling.
                         * Notice that the left child is the closest to the current node.
                         */
                        sibling->left->color = BLACK;
                        sibling->color = RED;
                        rotateRight(tree, sibling);
                        sibling = currParent->right;
                    }


                    /* If the right child of the sibling is red, swap the colors of the
                     * sibling and its parent, then color the child itself black and
                     * rotate around the current parent.
                     * Notice that the right child is the farthest from the current node.
                     */
                    sibling->color = currParent->color;
                    currParent->color = BLACK;

                    sibling->right->color = BLACK;
                    rotateLeft(tree, currParent);

                    curr = tree->root;      /* In order to stop the while loop */
                }

            }
        }
        else
        {
            /* If the current node is a right child, then its sibling is the left
             * child of the parent.
             */
            sibling = currParent->left;

            /* Check the sibling's color. (NULL nodes are colored black) */
            if (sibling && sibling->color == RED)
            {
                /* In case the sibling is red, color it black and rotate.
                 * Then color the parent red (and the grandparent is now black).
                 */
                sibling->color = BLACK;
                currParent->color = RED;
                rotateRight(tree, currParent);
                sibling = currParent->left;
            }

            if ((sibling != NULL) &&
                (!(sibling->left) || sibling->left->color == BLACK) &&
                (!(sibling->right) || sibling->right->color == BLACK))
            {
                /* If the sibling has two black children, color it red */
                sibling->color = RED;
                if (currParent->color == RED)
                {
                    /* If the parent is red, color it black and terminate
                     * the fix-up process.
                     */
                    currParent->color = BLACK;
                    curr = tree->root;      /* In order to stop the while loop */
                }
                else
                {
                    /* The black depth of the entire sub-tree rooted at the parent is
                     * now too small - fix it up recursively.
                     */
                    curr = currParent;
```

```
                }
            }
            else
            {
                if (sibling == NULL)
                {
                    /* Take care of a NULL sibling */
                    if (currParent->color == RED)
                    {
                        currParent->color = BLACK;
                        curr = tree->root;    /* In order to stop the while loop */
                    }
                    else
                    {
                        curr = currParent;
                    }
                }
                else
                {
                    /* In this case, at least one of the sibling's children is red.
                     * It is therefore obvious that the sibling itself is black.
                     */
                    if ((sibling->right != NULL) && (sibling->right->color == RED))
                    {
                        /* If the right child of the sibling is red, color it black,
                         * then color the sibling itself red, and rotate left around
                         * the sibling.
                         * Notice that the left right is the closest to the current node.
                         */
                        sibling->right->color = BLACK;
                        sibling->color = RED;
                        rotateLeft(tree, sibling);
                        sibling = currParent->left;
                    }

                    /* If the left child of the sibling is red, swap the colors of the
                     * sibling and its parent, then color the child itself black and
                     * rotate around the current parent.
                     * Notice that the left child is the farthest from the current node.
                     */
                    sibling->color = currParent->color;
                    currParent->color = BLACK;

                    sibling->left->color = BLACK;
                    rotateRight(tree, currParent);

                    curr = tree->root;       /* In order to stop the while loop */
                }
            }
        }
    }

    /* The root can always be colored black */
    if (curr != NULL)
    {
        curr->color = BLACK;
    }
}

/**
 * @brief: check whether the tree RBTreeContains this item.
 * @param tree: the tree to add an item to.
 * @param data: item to check.
 * @return: 0 if the item is not in the tree, other if it is.
 */
int RBTreeContains(const RBTree *tree, const void *data)
{
    return ((findRBTree(tree, data)) != NULL);
```

```
}

/**
 * @brief: helper for RBTreeContains
 * @param root: root of a RBTree.
 * @param data: item to check.
 * @return: pointer to the node contains the data, else NULL.
 */
Node *findRBTree(const RBTree *tree, const void *data)
{
    if (tree == NULL || data == NULL)
    {
        return NULL;
    }

    Node *cur = tree->root;
    int result;

    while (cur != NULL)
    {
        result = tree->compFunc(cur->data, data);
        if (result == 0)
        {
            return cur;
        }
        cur = (result < 0) ? cur->right : cur->left;
    }
    return NULL;
}

/**
 * @brief Activate a function on each item of the tree. the order is an ascending order.
 *          if one of the activations of the function returns 0, the process stops.
 * @param tree: the tree with all the items.
 * @param func: the function to activate on all items.
 * @param args: more optional arguments to the function.
 * @return: 0 on failure, other on success.
 */
int forEachRBTree(const RBTree *tree, forEachFunc func, void *args)
{
    if (tree == NULL || func == NULL)
    {
        return false;
    }
    if (! inOrderTraverse(tree->root, func, args))
    {
        return false;
    }
    return true;
}

/**
 * @brief Traverses on tree in order and activates the func on the node data.
 * @param root: root of a RBTree.
 * @param func: the function to activate on all items.
 * @param args: more optional arguments to the function.
 * @return: 0 on failure, other on success.
 */
int inOrderTraverse(const Node *root, forEachFunc func, void *args)
{
    if (root == NULL)  // the tree is empty
    {
        return true;
    }
    inOrderTraverse(root->left, func, args);
    if (! func(root->data, args))
    {
        return false;
```

```c
    }
    inOrderTraverse(root->right, func, args);
    return true;
}

/**
 * @brief free all memory of the data structure.
 * @param tree: pointer to the tree to free.
 */
void freeRBTree(RBTree **tree)
{
    if (tree == NULL || *tree == NULL)
    {
        return;
    }
    if ((*tree)->root != NULL)
    {
        removeNode((*tree), &((*tree)->root));
    }
    free(*tree);
    *tree = NULL;
}


#endif //RBTREE_C

```

3 Structs.c

```c
#ifndef STRUCTS_C
#define STRUCTS_C

/**
 * @file Structs.c
 * @author  Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
 * @ID 300575297
 * @date 26 May 2020
 *
 *
 * @section DESCRIPTION
 *              Two concrete examples on implementation of RBTree.h library:
 *              (1) In the first the data of the node is of type C string.
 *              (2) In the second the data of the node is of type Vector.
 */

// ------------------------------ includes --------------------------------

#include "RBTree.h"
#include "Structs.h"

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

// ------------------------------ macros & constants --------------------------------

#define LESS (-1)
#define EQUAL (0)
#define GREATER (1)

// ------------------------------ addition functions --------------------------------

double vecNorm(const Vector *pVector);
int deepCopy(const Vector *source, Vector *target);

// -------------------------------------------------------------------------------------

/**
 * CompFunc for strings (assumes strings end with "\0")
 * @param a - char* pointer
 * @param b - char* pointer
 * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a. (lexicographic
 * order)
 */
int stringCompare(const void *a, const void *b)
{
    if (a == NULL || b == NULL)
    {
        return false;
    }
    char *firstString = (char *) a;
    char *secondString = (char *) b;

    return strcmp(firstString, secondString);
}
```

```c
/**
 * ForEach function that concatenates the given word and \n to pConcatenated. pConcatenated is
 * already allocated with enough space.
 * @param word - char* to add to pConcatenated
 * @param pConcatenated - char*
 * @return 0 on failure, other on success
 */
int concatenate(const void *word, void *pConcatenated)
{
    // CHECK IN CASE OF OVERLAP STRINGS.
    if (word == NULL || pConcatenated == NULL)
    {
        return false;
    }
    char *firstString = (char *) pConcatenated;
    char *secondString = (char *) word;

    strcat(strcat(firstString, secondString), "\n");

    return true;
}

/**
 * FreeFunc for strings
 */
void freeString(void *s)
{
    char *string = (char *)s;
    free(string);
}

/**
 * CompFunc for Vectors, compares element by element, the vector that has the first larger
 * element is considered larger. If vectors are of different lengths and identify for the length
 * of the shorter vector, the shorter vector is considered smaller.
 * @param a - first vector
 * @param b - second vector
 * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
 */
int vectorCompare1By1(const void *a, const void *b)
{
    Vector *firstVec = (Vector *)a;
    Vector *secondVec = (Vector *)b;

    int minLen = (firstVec->len < secondVec->len) ? firstVec->len : secondVec->len;

    for (int i = 0; i < minLen; ++i)
    {
        double max = fmax(*((firstVec->vector) + i), *((secondVec->vector) + i));

        if (*((firstVec->vector) + i) < max)
        {
            return LESS;
        }
        else if (*((secondVec->vector) + i) < max)
        {
            return GREATER;
        }
    }

    if ((firstVec->len != secondVec->len)) // CHECK (&& len > 0)
    {
        return (firstVec->len < secondVec->len) ? LESS : GREATER;
    }

    return EQUAL;
}
```

```c
/**
 * FreeFunc for vectors
 */
void freeVector(void *pVector)
{
    Vector *vec = (Vector *)pVector;
    free(vec->vector);
    free(vec);  // free(pVector);
}

/**
 * copy pVector to pMaxVector if : 1. The norm of pVector is greater than the norm of pMaxVector.
 *                                 2. pMaxVector->vector == NULL.
 * @param pVector pointer to Vector
 * @param pMaxVector pointer to Vector
 * @return 1 on success, 0 on failure (if pVector == NULL: failure).
 */
int copyIfNormIsLarger(const void *pVector, void *pMaxVector)
{
    if (pVector == NULL || pMaxVector == NULL)
    {
        return false;
    }

    Vector *vec = (Vector *)pVector;
    Vector *maxVec = (Vector *)pMaxVector;

    /* if there is no coordinates in pVector so it is not greater */
    if (vec->vector == NULL)
    {
        return true;
    }

//    if (maxVec->vector == NULL)
//    {
//        return deepCopy(vec, maxVec);
//    }

    double vectorNorm = vecNorm(pVector);
    double maxVectorNorm = vecNorm(pMaxVector);

    /* norm of pMaxVector is greater or equal to pVector */
    if (maxVectorNorm == fmax(vectorNorm, maxVectorNorm))
    {
        return true;
    }

    return deepCopy(vec, maxVec);
}

/**
 * @brief make a deep copy of a vector.
 * @param source: the vector to be copied.
 * @param target: the vector to copy to it.
 * @return a deep copy of a given vector
 */
int deepCopy(const Vector *source, Vector *target)
{
    /* (source != NULL && *target != NULL && source->vector != NULL) */

//    double *newVec = (double *)calloc(source->len, sizeof(double));
//    if (newVec == NULL)
//    {
//        return false;
//    }
//
//    free(target->vector);
//    target->vector = newVec;
```

```c

    target->vector = realloc(target->vector, source->len * sizeof(double));

    target->len = source->len;
    for (int i = 0; i < source->len; ++i)
    {
        target->vector[i] = source->vector[i];
    }

    return true;
}

/**
 * @brief calculates the norm of a given vector.
 * @param pVector: the vector to calculate its norm.
 * @return the norm of the vector.
 */
double vecNorm(const Vector *pVector)
{
    if (pVector->vector == NULL)
    {
        /* if there is no coordinates in pVector then the nor, is ZERO */
        return 0;
    }
    double coordsSquaresSum = 0;
    for (int i = 0; i < pVector->len; ++i)
    {
        coordsSquaresSum += pow((*((pVector->vector) + i)) , 2);
    }
    return sqrt(coordsSquaresSum);
}

/**
 * @param tree a pointer to a tree of Vectors
 *            You must use copyIfNormIsLarger in the implementation!
 * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm).
 */
Vector *findMaxNormVectorInTree(RBTree *tree)
{
    if (tree == NULL || tree->root == NULL)
    {
        return NULL;
    }
    Vector *maxVector = (Vector *)malloc(sizeof(Vector));
    forEachRBTree(tree, copyIfNormIsLarger, maxVector);

    return maxVector;
}

#endif  // STRUCTS_C
```