Contents

1 Basic Test Results 2

2 RBTree.c 4

3 Structs.c 18

1 Basic Test Results

```
Structs.c: In function ‘stringCompare’:
Structs.c:48:12: warning: unused variable ‘min’ [-Wunused-variable]
     size_t min = lenA > lenB ? lenB : lenA;
            ^~~
"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(373, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
RBTree.c(373, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
RBTree.c(373, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
 ** Total Violated Rules      : 3
 ** Total Errors Occurs       : 3
 ** Total Violated Files Count: 1
```

2 RBTree.c

```c
// ------------------------------ includes ------------------------------
#include "RBTree.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdbool.h>

// -------------------------- const definitions -------------------------
#define IN_TREE 1
#define NOT_IN_TREE 0


/**
 * //the family surround myNode
 */
typedef struct NodeFamily
{
    struct Node *myNode, *father, *brother, *uncle, *grandfather;
    int myNodeIsleft, fatherIsLeft;
} NodeFamily;

/**
 * this function create new node from the given data, and returns it
 * @param parent
 * @param left
 * @param right
 * @param color
 * @param data
 * @return NULL if process fails, valid Node else
 */
Node *createNode(Node *parent, Node *left, Node *right, Color color, void *data)
{
    Node *myNode = (Node *)malloc(sizeof(Node));
    if (myNode == NULL)
    {
        return NULL;
    }
    myNode->parent = parent;
    myNode->left = left;
    myNode->right = right;
    myNode->color = color;
    myNode->data = data;
    return myNode;
}

/**
 * constructs a new RBTree with the given CompareFunc.
 * comp: a function two compare two variables.
 */
RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
{

    RBTree *MyTree;
    MyTree = (RBTree *)malloc(sizeof(RBTree));
    if (MyTree == NULL)
    {
        return NULL;
    }
    MyTree->root = NULL;
```

```
    MyTree->compFunc = compFunc;
    MyTree->freeFunc = freeFunc;
    MyTree->size = 0;
    return MyTree;
}

/**
 * check if the color of given node is black
 * @param node
 * @return true if the node is black, false else
 */
int isNodeBlack(Node *node)
{
    if (node == NULL) //leaf is black
    {
        return true;
    }
    if (node->color == BLACK)
    {
        return true;
    }
    return false;
}

/**
 * get node and return its whole family in special struct
 * @param node
 * @return
 */
void findMyFamily(Node *node, NodeFamily *family)
{
    //first init to prevent bugs
    family->brother = NULL;
    family->uncle = NULL;
    family->grandfather = NULL;
    family->myNode = node;
    family->father = node->parent;

    if (family->father == NULL)
    {
        return;
    }
    //init brother
    if (family->father->right == node)
    {
        family->brother = family->father->left;
        family->myNodeIsleft = false;
    }
    else if (family->father->left == node)
    {
        family->brother = family->father->right;
        family->myNodeIsleft = true;
    }

    //init grandpa
    if (node->parent->parent != NULL)
    {
        family->grandfather = node->parent->parent;
    }

    if (family->grandfather != NULL)
    {
        //init uncle
        if (family->father == family->grandfather->right)
        {
            family->uncle = family->grandfather->left;
            family->fatherIsLeft = false;
        }
```

```
        else if (family->father == family->grandfather->left)
        {
            family->uncle = family->grandfather->right;
            family->fatherIsLeft = true;
        }
    }
}

/**
 * left rotation on RBTree
 * @param tree
 * @param node
 */
void rotateLeft(RBTree *tree, Node *node)
{
    Node *rightPointer = node->right;
    node->right = rightPointer->left;

    if (node->right != NULL)
    {
        node->right->parent = node;
    }

    rightPointer->parent = node->parent;

    if (node->parent == NULL)
    {
        tree->root = rightPointer;
    }
    else if (node == node->parent->left)
    {
        node->parent->left = rightPointer;
    }
    else
    {
        node->parent->right = rightPointer;
    }

    rightPointer->left = node;
    node->parent = rightPointer;
}

/**
 * right rotation on RBTree
 * @param tree
 * @param node
 */
void rotateRight(RBTree *tree, Node *node)
{
    Node *leftPointer = node->left;

    node->left = leftPointer->right;

    if (node->left != NULL)
    {
        node->left->parent = node;
    }

    leftPointer->parent = node->parent;

    if (node->parent == NULL)
    {
        tree->root = leftPointer;
    }
    else if (node == node->parent->left)
    {
        node->parent->left = leftPointer;
    }
```

```
    else
    {
        node->parent->right = leftPointer;
    }

    leftPointer->right = node;
    node->parent = leftPointer;
}

/**
 * fixes violations caused by BST insertion
 * @param tree
 * @param nodeWeAdded
 * @return
 */
int fixViolationInRBTree(RBTree *tree, Node *nodeWeAdded)
{
    //1. the nodeWeAddedis the root
    //replace its color to black
    if (tree->root == nodeWeAdded)
    {
        nodeWeAdded->color = BLACK;
        return EXIT_SUCCESS;
    }

    //2.nodeWeAdded parent is black
    //return
    if (nodeWeAdded->parent->color == BLACK)
    {
        return EXIT_SUCCESS;
    }

    NodeFamily family;
    findMyFamily(nodeWeAdded, &family);

    //3. nodeWeAdded parent(P) is red and nodeWeAdded uncle(U) is red
    //transform P and U color to black
    //transform parent of P  (G) to red
    //recursive run the algorithm on G
    if (isNodeBlack(family.father) == false && isNodeBlack(family.uncle) == false)
    {
        family.father->color = BLACK;
        family.uncle->color = BLACK;
        family.grandfather->color = RED;
        fixViolationInRBTree(tree, family.grandfather);
    }

    //4. nodeWeAdded parent(P) is red and nodeWeAdded uncle(U) is black
    //a. if(nodeWeAdded is right child of left child ||
    //              nodeWeAdded is left child of right child)
    //{ do rotation so nodeWeAdded will be right-right son or left-left son}
    //b1. if(nodeWeAdded is left-left son){ do right rotation on G}
    //b2. if(nodeWeAdded is right-right son){ do left rotation on G}
    //c. transform P to red, ang G to black
    if (isNodeBlack(family.father) == false && isNodeBlack(family.uncle) == true)
    {
        if (family.myNodeIsleft == true && family.fatherIsLeft == false)
        {
            rotateRight(tree, family.father);
            findMyFamily(family.father, &family);

        }
        else if (family.myNodeIsleft == false && family.fatherIsLeft == true)
        {
            rotateLeft(tree, family.father);
            findMyFamily(family.father, &family);
        }

```

```c
        if (family.myNodeIsleft == true && family.fatherIsLeft == true) //left-left
        {
            rotateRight(tree, family.grandfather);
//              findMyFamily(nodeWeAdded,&family);
            family.father->color = BLACK;
            family.grandfather->color = RED;
        }
        if (family.myNodeIsleft == false && family.fatherIsLeft == false) //right-right
        {
            rotateLeft(tree, family.grandfather);
//              findMyFamily(nodeWeAdded,&family);
            family.father->color = BLACK;
            family.grandfather->color = RED;
        }
    }
    return EXIT_SUCCESS;
}

/**
 * recursive function to insert node into tree as we do in bst
 * @param compFunc
 * @param currentNode
 * @param nodeToAdd
 * @param addMe
 * @param parent
 * @return
 */
int insertNodeBstHelper(RBTree *tree, CompareFunc compFunc, Node *currentNode, Node *nodeToAdd,
                        Node **addMe, Node *parent)
{
    if (currentNode == NULL)
    {
        //add that node
        *addMe = nodeToAdd;
        tree->size += 1;
        nodeToAdd->parent = parent;
        return true;
    }

    int ans = compFunc(currentNode->data, nodeToAdd->data);
    //ans equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
    if (ans < 0) // node.data < data.data
    {
        //search in the right sub-tree
        return insertNodeBstHelper(tree, compFunc, currentNode->right, nodeToAdd,
                                   &currentNode->right, currentNode);
    }
    else //root > data
    {
        //search in the left sub-tree
        return insertNodeBstHelper(tree, compFunc, currentNode->left, nodeToAdd, &currentNode->left,
                                   currentNode);
    }
}

/**
 * insert new node to tree, as it done in BST
 * @return true in success or false in failure
 */
int insertNodeBst(RBTree *tree, Node *nodeToAdd)
{
    //if the tree is empty, update the root to be the node
    if (tree->root == NULL)
    {
        tree->root = nodeToAdd;
        tree->size += 1;
        return true;
    }
```

```c
    //else, compare the items to left\right sons, and add in the right places.
    insertNodeBstHelper(tree, tree->compFunc, tree->root, nodeToAdd, &tree->root, NULL);
    return true;
}

/**
 *check whether the sub-tree of given node Contains this item
 * @param node
 * @param data
 * @return
 */
int rBTreeContainsHelper(CompareFunc compFunc, const Node *currentNode, const void *data)
{
    if (currentNode == NULL)
    {
        return NOT_IN_TREE;
    }
    int ans = compFunc(currentNode->data, data);
    //equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
    if (ans == 0) //they are equal
    {
        return IN_TREE;
    }
    if (ans < 0) // node.data < data.data
    {
        //search in the right sub-tree
        return rBTreeContainsHelper(compFunc, currentNode->right, data);
    }
    else //root > data
    {
        //search in the left sub-tree
        return rBTreeContainsHelper(compFunc, currentNode->left, data);
    }
}

/**
 * check whether the tree RBTree Contains this item.
 * @param tree: the tree to check an item in.
 * @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)
{
    if (tree->root == NULL)
    {
        return NOT_IN_TREE;
    }
    return rBTreeContainsHelper(tree->compFunc, tree->root, data);
}

/**
 * add an item to the tree
 * @param tree: the tree to add an item to.
 * @param data: item to add 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)
{
    //check if data is valid
    if (data == NULL)
    {
        return false;
    }

    //check if item already in tree
    if (RBTreeContains(tree, data) == IN_TREE)
    {
        return false;
```

```c
    }


    //create node -red and with data inside
    Node *newNode = createNode(NULL, NULL, NULL, RED, data);
    if (newNode == NULL)
    {
        return false;
    }

    //do a normal Bst Insert
    if (insertNodeBst(tree, newNode) == false)
    {
        return false;
    }

    //fix Red Black Tree violations
    fixViolationInRBTree(tree, newNode);
    return true;

}

/**
 *find the successor of node, at specific case the the given node do not have left child
 * @param GivenNode
 * @return the sucessor
 */
Node *semiSuccessor(Node *GivenNode)
{
    Node *tempPointer = GivenNode;
    tempPointer = tempPointer->right;

    while ( tempPointer->left != NULL)
    {
        tempPointer = tempPointer->left;
    }

    return tempPointer;
}

/**
 * prepare the RBTree to delete, while making sure that the node we want's to delete
 * has at least one leaf. if needed we will find the successor, and replace the node with it
 * @param tree
 * @param nodeToDelete
 * @return
 */
void prepareToDeleteRBTree(Node **nodeToDelete)
{
    //check if nodeToDelete(M) has one or two child that a leaf?
    if ((*nodeToDelete)->left == NULL || (*nodeToDelete)->right == NULL)
    {
        //yes? good. just return
        return;
    }
    //no? find the successor
    Node *successor = semiSuccessor(*nodeToDelete);
    //switch the data of M and the value of its successor (not their colors!!)
    void *tempData = successor->data;
    successor->data = (*nodeToDelete)->data;
    (*nodeToDelete)->data = tempData;
    //mark the successor vertex as M
    (*nodeToDelete) = successor;
}

/**
 * we assume that only one(!) of the nodes is null
 * @param currentNode
```

```c
 * @return
 */
Node *findSonWhoIsNotNull(Node *currentNode)
{
    if (currentNode == NULL)
    {
        return NULL;
    }
    if (currentNode->left != NULL)
    {
        return currentNode->left;
    }
    if (currentNode->right != NULL)
    {
        return currentNode->right;
    }
    return NULL;
}

/**
 * in this function we get node to delete from tree
 * (!!!) we assume that nodeToDelete has at most one child who is not a leaf (!!!)
 * @param tree
 * @param nodeToDelete
 * @return
 */
void deleteNodeAction(RBTree *tree, Node **inputNode)
{

    Node *nodeToDelete = *inputNode;
    //find family
    NodeFamily family;
    findMyFamily(nodeToDelete, &family);

    //if nodeToDelete does't have any childrens:
    if (nodeToDelete->right == NULL && nodeToDelete->left == NULL)
    {
        if (tree->root == nodeToDelete) // input node is the root
        {
            //free the root pointer to null
            tree->root = NULL;
        }
        else //node have parent
        {
            //go to his father and free the pointer to the node to NULL
            if (family.myNodeIsleft == true)
            {
                nodeToDelete->parent->left = NULL;
            }
            else
            {
                nodeToDelete->parent->right = NULL;
            }
        }
        //free the data allocation
        tree->freeFunc(nodeToDelete->data);
        free(nodeToDelete);
        *inputNode = NULL;
        tree->size = tree->size - 1;
        return;
    }

    // the left child of M is not a leaf
    if (nodeToDelete->left != NULL)
    {
        if (tree->root == nodeToDelete) // input node is the root
        {
            //root point to left son
```

```
            tree->root = nodeToDelete->left;
            nodeToDelete->left->parent = NULL;
        }
        else //node has parent
        {
            if (family.myNodeIsleft == true) //M is left child
            {
                nodeToDelete->parent->left = nodeToDelete->left; //connect the child to the father
            }
            else //M is right child
            {
                nodeToDelete->parent->right = nodeToDelete->left; //connect the child to the father
            }
            nodeToDelete->left->parent = nodeToDelete->parent;  //connect the father to the child
        }
        //free the node allocations
        tree->freeFunc(nodeToDelete->data);
        Node *temp = nodeToDelete->left;
        free(nodeToDelete);
        *inputNode = temp;
        tree->size = tree->size - 1;
        return;
    }

    // the right child of M is not a leaf
    if (nodeToDelete->right != NULL)
    {
        if (tree->root == nodeToDelete) // input node is the root
        {
            //root points to right son
            tree->root = nodeToDelete->right;
            nodeToDelete->right->parent = NULL;
        }
        else
        {
            if (family.myNodeIsleft == true) //M is left child
            {
                nodeToDelete->parent->left = nodeToDelete->right; //connect the child to the father
            }
            else //M is right child
            {
                nodeToDelete->parent->right = nodeToDelete->right; //connect the child to the father
            }
            nodeToDelete->right->parent = nodeToDelete->parent;  //connect the father to the child
        }
        //free the node allocations
        tree->freeFunc(nodeToDelete->data);
        Node *temp = nodeToDelete->right;
        free(nodeToDelete);
        *inputNode = temp;
        tree->size = tree->size - 1;
        return;
    }
}

/**
 * get node and return its brother
 * @param myNode
 * @return
 */
Node *findBrother(Node *myNode, Node *father)
{

    if (myNode == NULL)
    {
        if (father->left == NULL)
        {
            return father->right;
```

```c
        }
        return father->left;
    }

    //init brother
    if (father->right == myNode)
    {
        return myNode->parent->left;
    }
    else if (father->left == myNode)
    {
        return myNode->parent->right;
    }
    return NULL; //we do no suppose arrive this case
}

/**
 *
 * @param nodeToCheck
 * @return true if the node is left child, false else
 */
int isLeftChild(Node *nodeToCheck, Node *father)
{
    if (father->left == nodeToCheck)
    {
        return true;
    }
    else
    {
        return false;
    }

}

/**
 * find the close S son to C
 * @param nodeC
 * @param nodeS
 * @return the close SC
 */
Node *findNodeCloseToC(Node *nodeC, Node *nodeS, Node *father)
{
    if (isLeftChild(nodeC, father) == true)
    {
        return nodeS->left;
    }
    else // nodeC is right child
    {
        return nodeS->right;
    }

}

/**
 * find the fur S son to C
 * @param nodeC
 * @param nodeS
 * @return the far SF
 */
Node *findNodeFarToC(Node *nodeC, Node *nodeS, Node *father)
{
    if (isLeftChild(nodeC, father) == true)
    {
        return nodeS->right;
    }
    else // nodeC is right child
    {
        return nodeS->left;
```

```c
    }

}

/**
 * helper function to do the third case deletion
 * @param tree
 * @param nodeToDelete
 */
void deleteThirdCase(RBTree *tree, Node *nodeToDelete, Node *father)
{
    ///a. C is the root
    if (father == NULL)
    {
        return; //its ok. just return
    }

    Node *brother = findBrother(nodeToDelete, father);

    ///b. S is black and he has two black sons
    if (isNodeBlack(brother) == true && isNodeBlack(brother->right) == true &&
        isNodeBlack(brother->left) == true)
    {
        //i. if P is red
        if (isNodeBlack(father) == false)
        {
            // make P black, and S to red
            father->color = BLACK;
            brother->color = RED;
            return;
        }
            //ii. if P is black
        else if (isNodeBlack(father) == true)
        {
            //make S red ,and recursive call (from 3.a) on P
            brother->color = RED;
            deleteThirdCase(tree, father, father->parent);
            return;
        }
    }

    ///c.if S is red
    if (isNodeBlack(brother) == false)
    {
        //make S black and P red
        brother->color = BLACK;
        father->color = RED;
        //do rotation on P to side of C
        if (isLeftChild(nodeToDelete, father) == true)
        {
            rotateLeft(tree, father);
        }
        else
        {
            rotateRight(tree, father);
        }

        //recursive call on on C from 3.a
        deleteThirdCase(tree, nodeToDelete, father);
        return;
    }

    ///d. if S is black and (son close to C) SC is red, and (son fur from C) SF is black
    Node *sC = findNodeCloseToC(nodeToDelete, brother, father);
    Node *sF = findNodeFarToC(nodeToDelete, brother, father);
    if (isNodeBlack(brother) == true && isNodeBlack(sC) == false
        && isNodeBlack(sF) == true)
    {
```

```
        //SC to black, S to red
        sC->color = BLACK;
        brother->color = RED;
        //do rotation to S from opposite direction of C
        if (isLeftChild(nodeToDelete, father) == true)
        {
            rotateRight(tree, brother);
        }
        else
        {
            rotateLeft(tree, brother);
        }
        //recursive call from 3.a on C
        deleteThirdCase(tree, nodeToDelete, father);
        return;
    }

    ///e. if S is black and SF is red
    if (isNodeBlack(brother) == true && isNodeBlack(sF) == false)
    {
        //replace colors of S and P (swap)
        Color tmp = brother->color;
        brother->color = father->color;
        father->color = tmp;
        //do rotation on P to direction of C
        if (isLeftChild(nodeToDelete, father) == true)
        {
            rotateLeft(tree, father);
        }
        else
        {
            rotateRight(tree, father);
        }
        //SF to black
        sF->color = BLACK;
        return;
    }
}

/**
 * part B of deleting
 * in this function we assume that nodeToDelete has at most one child who is not a leaf
 * @param tree
 * @param nodeToDelete
 */
void deleteSavingRBtreeProperties(RBTree *tree, Node *nodeToDelete)
{
    //nodeToDelete (M) , child (C), father (F), brother (S)

    //1. if M is red - just delete M
    if (isNodeBlack(nodeToDelete) == false)
    {
        deleteNodeAction(tree, &nodeToDelete);
//          tree->size = tree->size -1;
        return;
    }

    //find son who is not null
    Node *child = findSonWhoIsNotNull(nodeToDelete);
    //2. if M is black and C is red
    if (isNodeBlack(nodeToDelete) == true && child != NULL && isNodeBlack(child) == false)
    {
        deleteNodeAction(tree, &nodeToDelete); //delete M (wile connecting its son to his father)
//          tree->size = tree->size -1;
        nodeToDelete->color = BLACK;
        return;
    }

```

```c
    //3. if M is black and C is black
    if (isNodeBlack(nodeToDelete) == true && isNodeBlack(child) == true)
    {
        Node *father = nodeToDelete->parent;
        //delete M, and make the father points to (C)
        deleteNodeAction(tree, &nodeToDelete);
//        tree->size = tree->size -1;
        deleteThirdCase(tree, nodeToDelete, father); //problem is here
    }
}

/**
 * find the node to delete by the given data :)
 * @param compFunc
 * @param currentNode
 * @param nodeWithData
 * @param data
 * @return
 */
Node *findNode(CompareFunc compFunc, Node *currentNode, const void *data)
{
    if (currentNode == NULL)
    {
        return NULL;
    }
    int ans = compFunc(currentNode->data, data);
    //equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
    if (ans == 0) //they are equal
    {
        return currentNode;
    }
    if (ans < 0) // node.data < data.data
    {
        //search in the right sub-tree
        return findNode(compFunc, currentNode->right, data);
    }
    else //root > data
    {
        //search in the left sub-tree
        return findNode(compFunc, currentNode->left, data);
    }
}

/**
 * remove an item from the tree
 * @param tree: the tree to remove an item from.
 * @param data: item to remove from the tree.
 * @return: 0 on failure, other on success. (if data is not in the tree - failure).
 */
int deleteFromRBTree(RBTree *tree, void *data)
{

    //get the node with the data
    Node *nodeToDelete = NULL;
    nodeToDelete = findNode(tree->compFunc, tree->root, data);
    if (nodeToDelete == NULL)
    {
        return false;
    }
    //check that the node we want to delete has at least one leaf
    prepareToDeleteRBTree(&nodeToDelete);

    //do the delete action while saving the RBTree properties
    deleteSavingRBtreeProperties(tree, nodeToDelete);

    return true;
}
```

```c
/**
 * helper function for the foreach action on the data
 * @param currentNode
 * @param func
 * @param args
 * @return status of activating the function
 */
int forEachHelper(const Node *currentNode, forEachFunc func, void *args)
{
    if (currentNode == NULL) //leaf
    {
        return true;
    }
    forEachHelper(currentNode->left, func, args);
    if (func(currentNode->data, args) == false)
    {
        return false;
    }
    forEachHelper(currentNode->right, func, args);
    return true;
}

/**
 * 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 (may be null if the given function support it).
 * @return: 0 on failure, other on success.
 */
int forEachRBTree(const RBTree *tree, forEachFunc func, void *args)
{
    return (forEachHelper(tree->root, func, args));
}

int freeRBTreeHelper(RBTree *tree, Node *currentNode)
{
    if (currentNode == NULL) //leaf
    {
        return EXIT_SUCCESS;
    }
    freeRBTreeHelper(tree, currentNode->left);
    tree->freeFunc(currentNode->data); //free the data
    freeRBTreeHelper(tree, currentNode->right);
    //we free the left and right child's both. now free the node itself
    free(currentNode);
    return EXIT_SUCCESS;
}

/**
 * free all memory of the data structure.
 * @param tree: pointer to the tree to free.
 */
void freeRBTree(RBTree **tree)
{
    //recursive :) , active freeFunc on data.
    freeRBTreeHelper(*tree, (*tree)->root);
    free(*tree);
    *tree = NULL;
}
```

3 Structs.c

```c
// implementation of Structs.h
//we will do this part on vectors
// ------------------------------ includes ------------------------------
#include "RBTree.h"
#include "Structs.h"
#include <stdlib.h>
#include <string.h>

// -------------------------- const definitions -------------------------
#define LESS (-1)
#define EQUAL (0)
#define GREATER (1)
#define BACKSLASH_N "\n"

// ----------------------- functions implementation -----------------------

/**
 * return the minimal int from the given input
 * @param a
 * @param b
 * @return
 */
int min(int a, int b)
{
    if (a >= b)
    {
        return b;
    }
    else
    {
        return a;
    }
}

/**
 * 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)
{
    char *stringA = (char *)a;
    char *stringB = (char *)b;
    size_t lenA = strlen(stringA);
    size_t lenB = strlen(stringB);
    size_t min = lenA > lenB ? lenB : lenA;
    return strcmp(a, b);
}

/**
 * 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)
```

```c
{
    if (strcat(pConcatenated, word) == NULL)
    {
        return EXIT_FAILURE;
    }
    if (strcat(pConcatenated, BACKSLASH_N) == NULL)
    {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

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

/**
 * 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 *aVec = (Vector *)a;
    Vector *bVec = (Vector *)b;
    int minLen = min(aVec->len, bVec->len);
    for (int i = 0; i < minLen; i++)
    {
        double currentA = aVec->vector[i];
        double currentB = bVec->vector[i];
        if (currentA > currentB)
        {
            return GREATER;
        }
        else if (currentA < currentB)
        {
            return LESS;
        }
        else if (currentA == currentB)
        {
            return EQUAL;
        }
    }
    //they equal until one is ends-the shorter is smaller
    int lenA = aVec->len;
    int lenB = bVec->len;

    if (lenA > lenB)
    {
        return GREATER;
    }
    if (lenA < lenB)
    {
        return LESS;
    }
    return EQUAL; //the last option- they are totally equal
}

/**
```

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

/**
 * helper function- calculate norm of given vector pointer
 * @param pVector
 * @return
 */
double calcNorm(const void *pVector)
{
    Vector *myVec = (Vector *)pVector;
    double myNorm = 0;
    for (int i = 0; i < myVec->len; i++)
    {
        double elemToAdd = myVec->vector[i] * myVec->vector[i];
        myNorm += elemToAdd;
    }
    return myNorm;
}

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

    if (pVector == NULL || pMaxVector == NULL)
    {
        return EXIT_FAILURE;
    }
    double norm1 = calcNorm(pVector);
    double norm2 = calcNorm(pMaxVector);
    Vector *myVec1 = (Vector *)pVector;
    Vector *myVec2 = (Vector *)pMaxVector;

    if (norm1 > norm2)
    {
        //deep copy of 1st vector to 2nd
        myVec2->len = myVec1->len;
        myVec2->vector = realloc(myVec2->vector, sizeof(double) * myVec2->len);
        if (myVec2->vector == NULL)
        {
            return EXIT_FAILURE;
        }
        for (int i = 0; i < myVec2->len; i++)
        {
            myVec2->vector[i] = myVec1->vector[i];
        }
        return EXIT_SUCCESS;
    }
    return EXIT_SUCCESS;
}

/**
 * This function allocates memory it does not free.
 * @param tree a pointer to a tree of Vectors
 * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm), NULL on failure.
 */
```

```c
Vector *findMaxNormVectorInTree(RBTree *tree)
{
    Vector *myVec = (Vector *)malloc(sizeof(Vector));
    if (myVec == NULL)
    {
        return NULL;
    }
    myVec->len = 0;
    myVec->vector = NULL;
    forEachRBTree(tree, copyIfNormIsLarger, myVec);
    return myVec;
}
```

Index of comments

4.1 creative, nicely done

7.1 there is such a thing as over commenting - your code is clean and clear, no need to clutter it with so much documentation. Stating the case (parent is red & uncle is red, etc.) is good - exactly what the fix is, is overkill

8.1 no need for else after return. Also - if documenting, do it right - the else is root >= data.

Also, avoiding code duplication was using the trenary - insertNodeBstHelper(tree,compFunc, (ans<0) ? currentNode->right : currentNode->right ,...) etc.

10.1 again - too much documentation. Amusing as it is - overkill and clutters your readable code

13.1 no need for else after return

13.2 no need for else after return

18.1 min is unused - redundant computation. Also, you defined min function - either use that or the trenary (personally I prefer the trenary)