Contents

1 Basic Test Results

```
Running...

Opening tar file
Structs.c
RBTree.c
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...
"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
OK

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


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


========================
```

```
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 RBTree.c

```c
/**
 * @file RBTree.c
 * @author  Brahan Wassan <brahan>
 * @version 1.0
 * @date 12 Dec 2019
 *
 * @brief Program that build a RedBlack tree
 *
 * @section DESCRIPTION
 * The program builds a generic RB tree
 * Input   : tree nodes
 * Process: checks if the user input is valid, and then build the tree
 * Output : a tree with the desired data types
 */
#include <stdio.h>
#include "RBTree.h"
#include <stdlib.h>

#define SUCCESS 1
#define FAIL 0
#define EQUAL 0

/**
 * constructs a new RBTree with the given CompareFunc.
 * comp: a function two compare two variables.
 */
RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
{
    RBTree *tree = NULL;
    tree = (RBTree *) malloc(sizeof(RBTree));
    if (tree == NULL)
    {
        return NULL;
    }
    tree->root = NULL;
    tree->compFunc = compFunc;
    tree->freeFunc = freeFunc;
    tree->size = 0;
    return tree;
}

/**
 * created a new node
 * @param data the data which the node holds
 * @return a new node
 */
Node *newNode(void *data)
{
    Node *node = NULL;
    node = (Node *) malloc(sizeof(Node));

    if (node == NULL)
    {
        return NULL;
    }
    node->data = data;
    node->color = RED;
    node->left = NULL;
    node->right = NULL;
```

```c
    node->parent = NULL;
    return node;
}

/**
 * rotates the tree nodes to the right as we saw in DAST
 * @param node the node which we need to fix its position
 */
void rotateRight(Node *node)
{
    Node *left = node->left;
    Node *parent = node->parent;
    if (left->right != NULL)
    {
        left->right->parent = node;
    }
    node->left = left->right;
    node->parent = left;
    left->right = node;
    left->parent = parent;
    if (parent != NULL)
    {
        if (parent->right == node)
        {
            parent->right = left;
        }
        else
        {
            parent->left = left;
        }
    }
}

/**
 * rotates the tree nodes to the left as we saw in DAST
 * @param node the node which we need to fix its position
 */
void rotateLeft(Node *node)
{
    Node *right = node->right;
    Node *parent = node->parent;
    if (right->left != NULL)
    {
        right->left->parent = node;
    }
    node->right = right->left;
    node->parent = right;
    right->left = node;
    right->parent = parent;

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

/**
 * a getter
 * @param node the node
 * @return parent
 */
```

```
Node *getParent(Node *node)
{
    return node == NULL ? NULL : node->parent;
}

/**
 * a getter
 * @param node the node
 * @return the grandparent
 */
Node *getGrandParent(Node *node)
{
    return getParent(getParent(node));
}

/**
 * a getter
 * @param node the node
 * @return the node sibling
 */
Node *getSibling(Node *node)
{
    Node *parent = getParent(node);

    if (parent == NULL)
    {
        return NULL;
    }
    if (node == parent->left)
    {
        return parent->right;
    }
    else
    {
        return parent->left;
    }
}

/**
 * a getter
 * @param node the node
 * @return the parent sibling
 */
Node *getUncle(Node *n)
{
    Node *p = getParent(n);
    return getSibling(p);
}

/**
 * case 4 second step defined by the pdf
 * @param tree the tree
 * @param node the node we added to the tree
 */
void caseFourSecondStep(RBTree *tree, Node *node)
{
    Node *grandP = getGrandParent(node);
    Node *parent = node->parent;
    parent->color = BLACK;
    grandP->color = RED;
    if (node == parent->left)
    {
        rotateRight(grandP);
        if (getGrandParent(node) == NULL)
        {
            tree->root = node->parent;
        }
    }
```

```c
    else
    {
        rotateLeft(grandP);
        if (getGrandParent(node) == NULL)
        {
            tree->root = node->parent;
        }
    }
}// need to add it
/**
 * the 4 cases which we fix the tree accordingly
 * @param tree the tree which need to be fixed
 * @param node the node which we added
 */
void treeFix(RBTree *tree, Node *node)
{
    Node *uncle = getUncle(node);
    Node *parent = getParent(node);
    Node *grandP = getGrandParent(node);
    if (parent == NULL)
    {
        node->color = BLACK;
    }
    else if (parent->color == BLACK)
    {
        return;
    }
    else if (uncle != NULL && uncle->color == RED)
    {
        parent->color = BLACK;
        uncle->color = BLACK;
        grandP->color = RED;
        treeFix(tree, grandP);
    }
    else
    {
        if (node == parent->right && parent == grandP->left)
        {
            rotateLeft(parent);
            if (getGrandParent(node) == NULL)
            {
                tree->root = node->parent;
            }
            node = node->left;
        }
        else if (node == parent->left && parent == grandP->right)
        {
            rotateRight(parent);
            if (getGrandParent(node) == NULL)
            {
                tree->root = node->parent;
            }
            node = node->right;
        }
        caseFourSecondStep(tree, node);
    }
}

/**
 * inserting a node to the first null place its finds recurse
 * @param cur current node
 * @param node the node we want to insert into the tree
 * @param compareFunc the tree comp func
 */
void regularBSTInsert(RBTree *tree, Node *cur, Node *node, CompareFunc compareFunc)
{
    if (cur != NULL && cur->data != NULL)
    {
```

```c
        if (compareFunc(cur->data, node->data) > EQUAL)
        {
            if (cur->left == NULL)
            {
                cur->left = node;
            }
            else
            {
                regularBSTInsert(tree, cur->left, node, compareFunc);
                return;
            }
        }
        else
        {
            if (cur->right == NULL)
            {
                cur->right = node;
            }
            else
            {
                regularBSTInsert(tree, cur->right, node, compareFunc);
                return;
            }
        }
        node->parent = cur;
        node->color = RED;
        ++tree->size;
    }
}

/**
 * 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 addToRBTree(RBTree *tree, void *data)
{
    if (tree == NULL || data == NULL)
    {
        return FAIL;
    }
    else
    {
        Node *node = newNode(data);
        if (tree->root == NULL) // case 1 new node is root
        {
            node->color = BLACK;
            tree->root = node;
            ++tree->size;
            return SUCCESS;
        }
        else if (containsRBTree(tree, data) == SUCCESS)
        {
            free(node);
            return FAIL;
        }
        else
        {
            regularBSTInsert(tree, tree->root, node, tree->compFunc);
            treeFix(tree, node);
            return SUCCESS;
        }
    }
}

/**
 * a helper function which search inorder for the node
```

```
 * @param root the current node
 * @param data the data which we looking for in the tree
 * @param compFunc the compare function which we can check if the nodes hold the identical data
 * @return 1 if found 0 if not
 */
int recursiveContains(Node *root, void *data, CompareFunc compFunc)
{
    if (root == NULL)
    {
        return FAIL;
    }
    else
    {
        if (compFunc(root->data, data) == EQUAL)
        {
            return SUCCESS;
        }
        else if (compFunc(root->data, data) < EQUAL)
        {
            return recursiveContains(root->right, data, compFunc);
        }
        return recursiveContains(root->left, data, compFunc);
    }
}

/**
 * check whether the tree contains 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 containsRBTree(RBTree *tree, void *data)
{
    if (tree->root == NULL || data == NULL)
    {
        return FAIL;
    }
    return recursiveContains(tree->root, data, tree->compFunc) ? SUCCESS : FAIL;
}

int inOrder(Node *root, forEachFunc func, void *args)
{
    int isOk = SUCCESS;
    if (root != NULL)
    {
        if (root->left != NULL)
        {
            isOk = inOrder(root->left, func, args);
            if (!isOk)
            {
                return isOk;
            }
        }
        isOk = func(root->data, args);
        if (!isOk)
        {
            return isOk;
        }
        if (root->right != NULL)
        {
            isOk = inOrder(root->right, func, args);
            if (!isOk)
            {
                return isOk;
            }
        }
    }
    return isOk;
```

```c
}

/**
 * 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(RBTree *tree, forEachFunc func, void *args)
{
    int ret = FAIL;
    if (tree != NULL)
    {
        ret = inOrder(tree->root, func, args);
    }
    return ret;
}

/**
 * helper function which traverse thru the tree and free all the nodes
 * @param root the current node
 * @param freeFunc the tree free function
 */
void freeAll(Node *root, FreeFunc freeFunc)
{
    if (root == NULL)
    {
        return;
    }
    freeAll(root->left, freeFunc);
    freeAll(root->right, freeFunc);
    freeFunc(root->data);
    free(root);
}

/**
 * free all memory of the data structure.
 * @param tree: the tree to free.
 */
void freeRBTree(RBTree *tree)
{
    freeAll(tree->root, tree->freeFunc);
    free(tree);
}
```

3 Structs.c

```c
/**
 * @file Structs.c
 * @author  Brahan Wassan <brahan>
 * @version 1.0
 * @date 12 Dec 2019
 *
 * @brief Program that define Vectors which can be nodes in the RBTree
 *
 * @section DESCRIPTION
 * The program builds a generic Vector
 * Input   : Vector data
 * Process: checks if the user input is valid, and then define a node
 * Output : a valid node
 */
#include <string.h>
#include <malloc.h>
#include "Structs.h"

#define LESS (-1)
#define EQUAL (0)
#define GREATER (1)
#define SUCCESS (1)
#define TRUE (1)
#define FAIL (0)
#define SQUARE(a) (a)*(a)
#define UNDEFINED_SIZE (-1)

/**
 * 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 *v = (char *) a;
    char *u = (char *) b;
    return strcmp(v, u);
}

/**
 * ForEach function that concatenates the given word 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)
{
    const char *cWord = (char *) word;
    char *cP = (char *) pConcatenated;
    size_t firstLen = strlen(cWord);
    size_t secLen = strlen(cP);
    if (strcat(cP, cWord))
    {
        strcat(cP, "\n");
    }
    if (strlen(cP) < firstLen + secLen)
```

```c
    {
        return FAIL;
    }
    return SUCCESS;

}

/**
 * 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 *v = (Vector *) a;
    Vector *u = (Vector *) b;
    int isEq = 0;
    if (v->len == u->len)
    {
        isEq = TRUE;
    }
    int minLen = v->len;
    if (minLen > u->len)
    {
        minLen = u->len;
    }
    int i = 0;
    while (i < minLen)
    {
        if (v->vector[i] != u->vector[i])
        {
            if (v->vector[i] > u->vector[i])
            {
                return GREATER;
            }
            return LESS;
        }
        ++i;
    }
    if (isEq == TRUE)
    {
        return EQUAL;
    }
    else
    {
        if (minLen == v->len)
        {
            return LESS;
        }
        return GREATER;
    }
}
/**
 * calculate vector norm
 * @param v the vector
 * @return the vector norm
 */
double calcNorm(const Vector *v)
{
    if (v == NULL || v->vector == NULL)
    {
        return UNDEFINED_SIZE;
    }
    int len = v->len;
    double norm = 0;
```

```c
    double cur = 0;
    for (int i = 0; i < len; ++i)
    {
        cur = SQUARE(v->vector[i]);
        norm += cur;
    }
    return norm;
}

/**
 * copy pVector to pMaxVector if : 1. The norm of pVector is greater then the norm of pMaxVector.
 *                                 2. pMaxVector == 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)
    {
        return FAIL;
    }
    const Vector *toCopy = (Vector *) pVector;
    if (toCopy->vector == NULL)
    {
        return FAIL;
    }
    Vector *maxVec = (Vector *) pMaxVector;
    if (maxVec->vector == NULL || calcNorm(maxVec) < calcNorm(toCopy))
    {
        if (maxVec->vector != NULL)
        {
            free(maxVec->vector);
        }
        maxVec->vector = (double *) malloc(sizeof(double) * toCopy->len);
        if (maxVec->vector == NULL)
        {
            free(maxVec);
            return FAIL;
        }
        for (int i = 0; i < toCopy->len; ++i)
        {
            maxVec->vector[i] = toCopy->vector[i];
        }
        maxVec->len = toCopy->len;
    }
    return SUCCESS;
}

/**
 * @param tree a pointer to a tree of Vectors
 * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm).
 * // implement it in Structs.c You must use copyIfNormIsLarger in the implementation!
 */
Vector *findMaxNormVectorInTree(RBTree *tree)
{
    Vector *cur = (Vector *) malloc(sizeof(Vector));
    if (cur == NULL)
    {
        return NULL;
    }
    cur->len = 0;
    cur->vector = NULL;
    Vector *pMaxNorm = (Vector *) cur;
    int flag = forEachRBTree(tree, copyIfNormIsLarger, pMaxNorm);
    if (flag == FAIL)
    {
        return NULL;
```

```c
    }
    return pMaxNorm;
}

/**
 * FreeFunc for strings
 */
void freeString(void *s)
{
    if (s == NULL)
    {
        return;
    }
    free(s);
}

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