Contents

1 Basic Test Results 2

2 RBTree.c 4

3 Structs.c 16

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   *
*                                *
```

```
60  **********************************
61
62  ========================
63  = Checking coding style =
64  ========================
65  RBTree.c(472, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
66  RBTree.c(472, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
67  RBTree.c(472, 5):  fname_case {Do not start function name(RBTreeContains) with uppercase}
68   ** Total Violated Rules      : 3
69   ** Total Errors Occurs       : 3
70   ** Total Violated Files Count: 1
```

2 RBTree.c

```c
#include <stdio.h>
#include "stdlib.h"
#include "RBTree.h"
#include "string.h"

#define RIGHT 0
#define LEFT 1
#define SUCCESS 1
#define FAILURE 0

//EX3 - RED BLACK TREES by yoav eshed 305384869

int deleteFromTree(Node *M, RBTree *tree);

/**
 * A function that checks if a nodes color is black, null nodes counts as black
 * @param node : the node which is being tested
 * @return : 1 if the node is black, 0 if not
 */
int isBlack(Node *node)
{
    if ((node == NULL) || (node->color == BLACK))
    {
        return SUCCESS;
    }
    return FAILURE;
}


/**
 * A function that checks if a nodes color is red
 * @param node : the node which is being tested
 * @return : 1 if the node is red, 0 if not
 */
int isRed(Node *node)
{
    if (node == NULL)
    {
        return FAILURE;
    }
    return node->color == RED;
}

/**
 * A function that retrieves the Parent node  of a node
 * @param node : the node whose Parent will be retrieved
 * @return: node if found , else returns NULL
 */
Node *getParent(Node *node)
{
    if (node == NULL)
    {
        return NULL;
    }
    return node->parent;
}

/**
 * A function that retrieves the sibling of a node
```

```
 * @param node : the node whose sibling will be returned
 * @return: node if found , else returns NULL
 */
Node *getSibling(Node *node)
{
    Node *parent = getParent(node);

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

/**
 * A function that retrieves the grandparent of the node
 * @param node :node
 * @return :the grand parent node
 */
Node *getUncle(Node *node)
{
    if (node != NULL)
    {
        return getSibling(getParent(node));
    }
    return NULL;
}

/**
 * A function that retrieves the grandparent of the node
 * @param node :node
 * @return :the grand parent node
 */
Node *getGrandParent(Node *node)
{
    if (node != NULL)
    {
        return getParent(getParent(node));
    }
    return NULL;
}

/**
 * A function that gets a successor for a node in an bst
 * @param node : the node that will get a successor
 * @return the successor node if found , null if not
 */
Node *getSuccessor(Node *node)
{
    if (node != NULL)
    {
        Node *successor = node;
        {
            while (successor->left != NULL)
            {
                successor = successor->left;
            }
            return successor;
        }
    }
    return NULL;
}

/**
```

```
128  * A function that retrieves a node as a successor to a node in a bst
129  * @param node: the node to find a successor to
130  * @return: A node - if the node has two leaves the function returns the most left node
131  * At the right sub tree, ,if the node has two leaves it returns null;
132  */
133 Node *bstSuccessor(Node *node)
134 {
135     if (node->left != NULL && node->right != NULL)
136     {
137         return getSuccessor(node->right);
138     }
139     if (node->left == NULL && node->right == NULL)
140     {
141         return NULL;
142     }
143     if (node->left != NULL)
144     {
145         return node->left;
146     }
147     return node->right;
148 }
149
150
151 /**
152  * A function the rotates a node inside a tree by moving the node and its parent and grand parent
153  * nodes to the right while conserving the tree balancing attributes of the tree
154  * @param node: the node to be rotated
155  */
156 void rotateRight(Node *node)
157 {
158     Node *newNode = node->left;
159     Node *P = node->parent;
160     if (newNode->right != NULL)
161     {
162         newNode->right->parent = node;
163     }
164     node->left = newNode->right;
165     node->parent = newNode;
166     newNode->right = node;
167     newNode->parent = P;
168     if (P != NULL)
169     {
170         if (P->right == node)
171         {
172             P->right = newNode;
173         }
174         else
175         {
176             P->left = newNode;
177         }
178     }
179 }
180
181 /**
182  * A function the rotates a node inside a tree by moving the node and its parent and grand parent
183  * nodes to the left while conserving the tree balancing attributes of the tree
184  * @param node: the node to be rotated
185  */
186 void rotateLeft(Node *node)
187 {
188     Node *newNode = node->right;
189     Node *P = node->parent;
190     if (newNode->left != NULL)
191     {
192         newNode->left->parent = node;
193     }
194     node->right = newNode->left;
195     node->parent = newNode;
```

```
196         newNode->left = node;
197         newNode->parent = P;
198         if (P != NULL)
199         {
200             if (P->left == node)
201             {
202                 P->left = newNode;
203             }
204             else
205             {
206                 P->right = newNode;
207             }
208         }
209     }
210
211     /**
212      * A an aiding function for case 4 of insertion
213      * @param tree:
214      * @param node: the node to be inserted
215      */
216     void caseFour(RBTree *tree, Node *node)
217     {
218         Node *G = getGrandParent(node);
219         Node *P = node->parent;
220         //phase A
221         if (node == P->left)
222         {
223             rotateRight(G);
224             if (getGrandParent(node) == NULL)
225             {
226                 tree->root = node->parent;
227             }
228         }
229         else
230             //phase B
231         {
232             rotateLeft(G);
233             if (getGrandParent(node) == NULL)
234             {
235                 tree->root = node->parent;
236             }
237         }
238         //phase C
239         P->color = BLACK;
240         G->color = RED;
241     }
242
243     /**
244      * A function the re-aligns a red black tree and repairs any violations
245      * @param root : the tree root
246      * @param curNode: the node that needs to be checked for violations
247      */
248     void fixTree(RBTree *tree, Node *node)
249     {
250         Node *U = getUncle(node);
251         Node *P = getParent(node);
252         Node *G = getGrandParent(node);
253         //case 1
254         if (tree->compFunc(tree->root->data, node->data) == 0)
255         {
256             node->color = BLACK;
257         }
258         //case 2
259         else if (isBlack(P))
260         {
261             return;
262         }
263         //case 3
```

```
264         else if (isRed(P) && isRed(U))//alarm
265         {
266             P->color = BLACK;
267             U->color = BLACK;
268             G->color = RED;
269             fixTree(tree, G);
270         }
271         else
272         {
273             if (node == P->right && P == G->left)
274             {
275                 rotateLeft(P);
276                 if (getGrandParent(node) == NULL)
277                 {
278                     tree->root = node->parent;
279                 }
280                 node = node->left;
281             }
282             else if (node == P->left && P == G->right)
283             {
284                 rotateRight(P);
285                 if (getGrandParent(node) == NULL)
286                 {
287                     tree->root = node->parent;
288                 }
289                 node = node->right;
290             }
291             caseFour(tree, node);
292         }
293     }
294 
295     /**
296      * An Aiding  function for inserting nodes into an bst search tree
297      * @param root : root of the three
298      * @param node: the node which is going to be inserted
299      * @param tree : Red black tree object
300      * @return: the node if the insertion is successful , NULL if not
301      */
302 
303     Node *insertNode(Node *root, Node *node, RBTree *tree)
304     {
305         if (root == NULL)
306         {
307             return node;
308         }
309         int res = tree->compFunc(root->data, node->data);
310         if (res > 0)
311         {
312             root->left = insertNode(root->left, node, tree);
313             root->left->parent = root;
314         }
315         else if (res < 0)
316         {
317             root->right = insertNode(root->right, node, tree);
318             root->right->parent = root;
319         }
320         return root;
321     }
322 
323     /**
324      * A function for removing a node entirely from a red black tree
325      * @param root :three root
326      * @param tree : Red black tree object
327      */
328     void deleteNode(Node **root, RBTree *tree)
329     {
330         if (*root == NULL)
331         {
```

```
332             return;
333         }
334         deleteNode(&(*root)->left, tree);
335         deleteNode(&(*root)->right, tree);
336         tree->freeFunc((*root)->data);
337         free(*root);
338     }
339 
340     /**
341      * A function that swaps data between two nodes
342      * @param a : first data object
343      * @param b : second data object
344      */
345     void swapNodes(Node *a, Node *b)
346     {
347         if (a == NULL || b == NULL)
348         {
349             return;
350         }
351         void *tmp = a->data;
352         a->data = b->data;
353         b->data = tmp;
354     }
355 
356     /**
357      * A function that traverses the tree recursively and checks if there value exists in it.
358      * @param root : the root of the tree been searched on.
359      * @param data : that data that is being searched
360      * @param compFunc: a comparison function
361      * @return: 1 if the value exists in the tree, 0 if not.
362      */
363     Node *searchTree(Node *root, const void *data, CompareFunc compFunc)
364     {
365         if (root == NULL)
366         {
367             return NULL;
368         }
369         int res = compFunc(root->data, data);
370         if (res == 0)
371         {
372             return root;
373         }
374         if (res > 0)
375         {
376             return searchTree(root->left, data, compFunc);
377         }
378         if (res < 0)
379         {
380             return searchTree(root->right, data, compFunc);
381         }
382         return EXIT_SUCCESS;
383     }
384 
385     /**
386      * A function that preforms in order traverse on the tree and executes a function for each node
387      * @param root : A node of the tree
388      * @param func : A function to be applied
389      * @param args : Arguments to the function if it needs them
390      */
391     void inOrderPass(Node *root, forEachFunc func, void *args)
392     {
393         if (root == NULL)
394         {
395             return;
396         }
397         inOrderPass(root->left, func, args);
398         func(root->data, args);
399         inOrderPass(root->right, func, args);
```

```
400 }
401
402 /**
403  * constructs a new RBTree with the given CompareFunc.
404  * compFunc: a function two compare two variables.
405  * freeFunc: a function that frees allocated memory
406  */
407 RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
408 {
409     if (compFunc == NULL || freeFunc == NULL)
410     {
411         return NULL;
412     }
413     RBTree *tree = (RBTree *) malloc(sizeof(RBTree));
414     if (tree != NULL)
415     {
416         tree->root = NULL;
417         tree->compFunc = compFunc;
418         tree->freeFunc = freeFunc;
419         tree->size = 0;
420         return tree;
421     }
422     return NULL;
423 }
424
425 /**
426  * add an item to the tree
427  * @param tree: the tree to add an item to.
428  * @param data: item to add to the tree.
429  * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
430  */
431 int insertToRBTree(RBTree *tree, void *data)
432 {
433     if (tree == NULL || data == NULL)
434     {
435         return FAILURE;
436     }
437     Node *new = (Node *) malloc(sizeof(Node));
438     Node *tmp = tree->root;
439     if (new == NULL)
440     {
441         return FAILURE;
442     }
443     if (RBTreeContains(tree, data))
444     {
445         free(new);
446         return FAILURE;
447     }
448     new->data = data, new->color = RED, new->parent = NULL, new->left = NULL, new->right = NULL;
449     if (tmp == NULL)
450     {
451         new->color = BLACK, tree->root = new;
452         fixTree(tree, new);
453         tree->size++;
454         return SUCCESS;
455     }
456     if (insertNode(tree->root, new, tree))
457     {
458         fixTree(tree, new);
459         tree->size++;
460         return SUCCESS;
461     }
462     free(new);
463     return FAILURE;
464 }
465
466 /**
467  * 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)
{
    if (tree == NULL || data == NULL)
    {
        return FAILURE;
    }
    if (searchTree(tree->root, data, tree->compFunc) == NULL)
    {
        return FAILURE;
    }
    return SUCCESS;
}


/**
 * A function that disconnects A node from a treee
 * @param tree : Red black tree object
 * @param node :The node to be removed
 */
void disconnect(RBTree *tree, Node *node)
{
    Node *Parent = getParent(node);
    if (Parent->right == node)
    {
        Parent->right = NULL;
    }
    else if (Parent->left == node)
    {
        Parent->left = NULL;
    }
    tree->freeFunc(node->data);
}

/**
 * A function that handles the double black cases.
 * @param M: current node - the one marked double black
 * @param P: the node's parents node
 * @param S: the node's sibling node
 * @param C: the node's descendant node
 * @param tree : Red black tree object
 * @return: deleted node if deletion was successful , null if not
 */
Node *doubleBlack(Node *M, Node *P, Node *S, Node *C, RBTree *tree)
{
    //case A
    if (M->parent == NULL)
    {
        if (C != NULL)
        {
            return C;
        }
        return M;
    }
    //case B
    if ((S == NULL) || ((isBlack(S)) && (isBlack(S->left)) && (isBlack(S->right))))
    {
        if (isRed(P))
        {
            // case B.I
            P->color = BLACK;
            if (S != NULL)
            {
                S->color = RED;
            }
```

```
536             return M;
537         }
538             // case B.II
539         else if (isBlack(P))
540         {
541             S->color = RED;
542             return doubleBlack(P, getParent(P), getSibling(P), bstSuccessor(P), tree);
543         }
544     }
545 
546         //case C
547     else if (isRed(S))
548     {
549         Color temp = P->color;
550         P->color = S->color;
551         S->color = temp;
552         if (M == P->left)
553         {
554             rotateLeft(P);
555             rotateRight(P);
556             return doubleBlack(M, P, S, C, tree);
557         }
558         else if (M == P->right)
559         {
560             rotateRight(P);
561             return doubleBlack(M, P, S, C, tree);
562         }
563     }
564     else
565         //S is black
566     {
567         int direction = RIGHT;
568         Node *SF = NULL;
569         Node *SC = NULL;
570         if (S == P->right)
571         {
572             SF = S->right;
573             SC = S->left;
574         }
575         else if (S == P->left)
576         {
577             SF = S->left;
578             SC = S->right;
579             direction = LEFT;
580         }
581         //case 3.4
582         if (isBlack(SF) == 1 && isRed(SC) == 1)
583         {
584             SC->color = BLACK;
585             S->color = RED;
586 
587             if (S != NULL && S == P->right)
588             {
589                 SF = S->right;
590                 SC = S->left;
591             }
592             else if (S != NULL && S == P->right)
593             {
594                 SF = S->left;
595                 SC = S->right;
596                 direction = LEFT;
597             }
598             if (direction == RIGHT)
599             {
600                 rotateRight(S);
601             }
602             else
603             {
```

```
                rotateLeft(S);
            }
            S = getSibling(M);
            if (S != NULL && S == P->right)
            {
                SF = S->right;
                SC = S->left;
            }
            else if (S != NULL && S == P->right)
            {
                SF = S->left;
                SC = S->right;
                direction = LEFT;
            }
        }
        //case 3.5
        if (isRed(SF))
        {
            Color temp = P->color;
            P->color = S->color;
            S->color = temp;
            SF->color = BLACK;
            if (direction == RIGHT)
            {
                rotateLeft(P);
            }
            else
            {
                rotateRight(P);
            }
            return M;
        }
    }
    return NULL;
}


/**
 * A function that deletes a node from a tree
 * @param M : node to be replaced
 * @param tree : Red black tree object
 * @return
 */
int deleteFromTree(Node *M, RBTree *tree)
{
    Node *C = bstSuccessor(M); // descendant
    Node *P = M->parent;
    Node *S = getSibling(M);
    //Case A
    swapNodes(M, C);
    if (isRed(M))
    {
        if (C == NULL)
        {
            disconnect(tree, M);
            fixTree(tree, P);
            return SUCCESS;
        }
        disconnect(tree, C);
        fixTree(tree, M);
        return SUCCESS;

    }
    //Case B
    if ((isBlack(M) == 1) && (isRed(C) == 1))
    {
        disconnect(tree, C);
        return SUCCESS;
```

```
672         }
673         //Case C
674         if ((isBlack(M) == 1) && (isBlack(C) == 1))
675         {
676             disconnect(tree, doubleBlack(M, P, S, C, tree));
677             return SUCCESS;
678         }
679         return FAILURE;
680     }
681 
682     /**
683      * remove an item from the tree
684      * @param tree: the tree to remove an item from.
685      * @param data: item to remove from the tree.
686      * @return: 0 on failure, other on success. (if data is not in the tree - failure).
687      */
688     int deleteFromRBTree(RBTree *tree, void *data)
689     {
690         Node *SUCCESSOR = searchTree(tree->root, data, tree->compFunc);
691         if (SUCCESSOR == NULL || tree == NULL || data == NULL)
692         {
693             return FAILURE;
694         }
695         if (!RBTreeContains(tree, data))
696         {
697             return FAILURE;
698         }
699         if (!deleteFromTree(SUCCESSOR, tree))
700         {
701             return FAILURE;
702         }
703         tree->size--;
704         return EXIT_SUCCESS;
705     }
706 
707     /**
708      * Activate a function on each item of the tree. the order is an ascending order. if one of the
709      * activations of the
710      * function returns 0, the process stops.
711      * @param tree: the tree with all the items.
712      * @param func: the function to activate on all items.
713      * @param args: more optional arguments to the function (may be null if the given function
714      * support it).
715      * @return: 0 on failure, other on success.
716      */
717     int forEachRBTree(const RBTree *tree, forEachFunc func, void *args)
718     {
719         if (tree->root == NULL)
720         {
721             return FAILURE;
722         }
723         inOrderPass(tree->root, func, args);
724         return SUCCESS;
725     }
726 
727     /**
728      * free all memory of the data structure.
729      * @param tree: pointer to the tree to free.
730      */
731     void freeRBTree(RBTree **tree)
732     {
733         if (tree != NULL)
734         {
735             deleteNode(&(*tree)->root, *tree);
736             free(*tree);
737         }
738     }
739 
```

740
741
742

3 Structs.c

```c
#include "Structs.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"

#define SUCCESS 1
#define FAILURE 0
#define  B -1
#define  A 1
#define EQUAL 0
#define SUCCESS 1
#define FAILURE 0

/**
 * 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)
{
    if (a == NULL || b == NULL)
    {
        return FAILURE;
    }
    Vector *vecA = (Vector *) a;
    Vector *vecB = (Vector *) b;
    int longerVector = B;
    int len = vecA->len;
    if (vecA->len > vecB->len)
    {
        len = vecB->len;
        longerVector = A;
    }
    for (int i = 0; i < len; i++)
    {
        if (vecA->vector[i] != vecB->vector[i])
        {
            if (vecA->vector[i] > vecB->vector[i])
            {
                longerVector = A;
            }
            longerVector = B;
        }
        if (vecA->len == vecB->len)
        {
            longerVector = 0;
        }
        return longerVector;
    }
    return FAILURE;
}

/**
 * A function that gets the norm of a given vector
 * @param vector :the vector that norm is going to be calculated
 * @return :the vectors norm
```

```c
 */
double getNorm(Vector *vector)
{
    double sum = 0;
    for (int i = 0; i < vector->len; i++)
    {
        sum += pow(vector->vector[i], 2);
    }
    return sum;
}

/**
 * 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
 * @return 1 on success, 0 on failure (if pVector == NULL: failure).
 */
int copyIfNormIsLarger(const void *pVector, void *pMaxVector)
{
    if (pVector == NULL || pMaxVector == NULL)
    {
        return FAILURE;
    }
    Vector *vecP = (Vector *) pVector;
    Vector *vecMax = (Vector *) pMaxVector;
    if (vecP == NULL || vecMax == NULL)
    {
        return FAILURE;
    }
    if (getNorm(vecP) > getNorm(pMaxVector))
    {
        vecMax->len = vecP->len;
        vecMax->vector = realloc(vecMax->vector, vecMax->len * sizeof(double));
        if (vecMax->vector == NULL)
        {
            return FAILURE;
        }
        memcpy(&vecMax->vector, &vecP->vector, vecP->len * sizeof(double));
    }
    return SUCCESS;
}

/**
 * A function that finds the vector with the maximal norm in an RBtree
 * @param tree a pointer to a tree of Vectors
 * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm).
 */
Vector *findMaxNormVectorInTree(RBTree *tree)
{
    if (tree == NULL)
    {
        return NULL;
    }
    Vector *maxVec = (Vector *) malloc(sizeof(Vector));
    maxVec->vector = malloc(maxVec->len * sizeof(double *));
    maxVec->vector = NULL, maxVec->len = 0;
    if (maxVec == NULL || maxVec->vector == NULL)
    {
        return NULL;
    }
    if (!forEachRBTree(tree, copyIfNormIsLarger,  maxVec))
    {
        freeVector(maxVec);
        return NULL;
    }
    return maxVec;
}
```

```c


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


/**
 * 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 *string1 = (char *) a;
    char *string2 = (char *) b;
    return strcmp(string1, string2);
}

/**
 * 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)
{
    char *word1 = (char *) word;
    if (word1 == NULL || pConcatenated == NULL)
    {
        return FAILURE;
    }
    if (strcat(pConcatenated, word1) == NULL)
    {
        return FAILURE;
    }
    strcat(pConcatenated, "\n");
    return SUCCESS;
}

/**
 * A function to free string allocated memory
 * @param s:string to be freed
 */

void freeString(void *s)
{
    char *string = (char *) s;
    if (s != NULL)
    {
        free(string);
    }
}

```