

# Contents

<b>1</b>	<b>Basic Test Results</b>	<b>2</b>
<b>2</b>	<b>RBTree.c</b>	<b>4</b>
<b>3</b>	<b>Structs.c</b>	<b>18</b>

# 1 Basic Test Results

```
1 Structs.c: In function 'stringCompare':
2 Structs.c:48:12: warning: unused variable 'min' [-Wunused-variable]
3     size_t min = lenA > lenB ? lenB : lenA;
4         ^~~
5 "MacBook Pro" is in the tree.
6 "iPod" is not in the tree.
7 "iPhone" is in the tree.
8 "iPad" is in the tree.
9 "Apple Watch" is in the tree.
10 "Apple TV" is not in the tree.
11
12 The number of products in the tree is 4.
13
14 Name: Apple Watch.      Price: 299.00
15 Name: MacBook Pro.     Price: 1499.00
16 Name: iPad.            Price: 499.00
17 Name: iPhone.          Price: 599.00
18 test passed
19 Running...
20
21 Opening tar file
22 OK
23 Tar extracted O.K.
24
25 Checking files...
26 OK
27 Making sure files are not empty...
28 OK
29 Compilation check...
30 Compiling...
31 OK
32 Compiling...
33 OK
34 Compiling...
35 OK
36 Compiling...
37 OK
38 Compiling...
39 OK
40 Compilation seems OK! Check if you got warnings!
41
42
43 =====
44     Public test cases
45 =====
46
47 ~~~~~
48 ~   ProductExample output:   ~
49
50 Running test...
51 OK
52
53 ~ End of ProductExample output ~
54 ~~~~~
55
56
57 Test Succeeded.
58 =====
59
```

```

60 *****
61 *                                     *
62 *   presubmission script passed   *
63 *                                     *
64 *****
65
66 =====
67 = Checking coding style =
68 =====
69 RBTtree.c(373, 5):  fname_case {Do not start function name(RBTtreeContains) with uppercase}
70 RBTtree.c(373, 5):  fname_case {Do not start function name(RBTtreeContains) with uppercase}
71 RBTtree.c(373, 5):  fname_case {Do not start function name(RBTtreeContains) with uppercase}
72 ** Total Violated Rules      : 3
73 ** Total Errors Occurs       : 3
74 ** Total Violated Files Count: 1

```

## 2 RBTree.c

```
1  // ----- includes -----
2  #include "RBTree.h"
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdio.h>
6  #include <stdbool.h>
7
8  // ----- const definitions -----
9  #define IN_TREE 1
10 #define NOT_IN_TREE 0
11
12
13 /**
14  * //the family surround myNode
15  */
16 typedef struct NodeFamily
17 {
18     struct Node *myNode, *father, *brother, *uncle, *grandfather;
19     int myNodeIsleft, fatherIsLeft;
20 } NodeFamily;
21
22 /**
23  * this function create new node from the given data, and returns it
24  * @param parent
25  * @param left
26  * @param right
27  * @param color
28  * @param data
29  * @return NULL if process fails, valid Node else
30  */
31 Node *createNode(Node *parent, Node *left, Node *right, Color color, void *data)
32 {
33     Node *myNode = (Node *)malloc(sizeof(Node));
34     if (myNode == NULL)
35     {
36         return NULL;
37     }
38     myNode->parent = parent;
39     myNode->left = left;
40     myNode->right = right;
41     myNode->color = color;
42     myNode->data = data;
43     return myNode;
44 }
45
46 /**
47  * constructs a new RBTree with the given CompareFunc.
48  * comp: a function two compare two variables.
49  */
50 RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
51 {
52
53     RBTree *MyTree;
54     MyTree = (RBTree *)malloc(sizeof(RBTree));
55     if (MyTree == NULL)
56     {
57         return NULL;
58     }
59     MyTree->root = NULL;
```

4.1

```

60     MyTree->compFunc = compFunc;
61     MyTree->freeFunc = freeFunc;
62     MyTree->size = 0;
63     return MyTree;
64 }
65
66 /**
67  * check if the color of given node is black
68  * @param node
69  * @return true if the node is black, false else
70  */
71 int isNodeBlack(Node *node)
72 {
73     if (node == NULL) //leaf is black
74     {
75         return true;
76     }
77     if (node->color == BLACK)
78     {
79         return true;
80     }
81     return false;
82 }
83
84 /**
85  * get node and return its whole family in special struct
86  * @param node
87  * @return
88  */
89 void findMyFamily(Node *node, NodeFamily *family)
90 {
91     //first init to prevent bugs
92     family->brother = NULL;
93     family->uncle = NULL;
94     family->grandfather = NULL;
95     family->myNode = node;
96     family->father = node->parent;
97
98     if (family->father == NULL)
99     {
100         return;
101     }
102     //init brother
103     if (family->father->right == node)
104     {
105         family->brother = family->father->left;
106         family->myNodeIsleft = false;
107     }
108     else if (family->father->left == node)
109     {
110         family->brother = family->father->right;
111         family->myNodeIsleft = true;
112     }
113
114     //init grandpa
115     if (node->parent->parent != NULL)
116     {
117         family->grandfather = node->parent->parent;
118     }
119
120     if (family->grandfather != NULL)
121     {
122         //init uncle
123         if (family->father == family->grandfather->right)
124         {
125             family->uncle = family->grandfather->left;
126             family->fatherIsLeft = false;
127         }

```

```

128         else if (family->father == family->grandfather->left)
129         {
130             family->uncle = family->grandfather->right;
131             family->fatherIsLeft = true;
132         }
133     }
134 }
135
136 /**
137  * left rotation on RBTREE
138  * @param tree
139  * @param node
140  */
141 void rotateLeft(RBTree *tree, Node *node)
142 {
143     Node *rightPointer = node->right;
144     node->right = rightPointer->left;
145
146     if (node->right != NULL)
147     {
148         node->right->parent = node;
149     }
150
151     rightPointer->parent = node->parent;
152
153     if (node->parent == NULL)
154     {
155         tree->root = rightPointer;
156     }
157     else if (node == node->parent->left)
158     {
159         node->parent->left = rightPointer;
160     }
161     else
162     {
163         node->parent->right = rightPointer;
164     }
165
166     rightPointer->left = node;
167     node->parent = rightPointer;
168 }
169
170 /**
171  * right rotation on RBTREE
172  * @param tree
173  * @param node
174  */
175 void rotateRight(RBTree *tree, Node *node)
176 {
177     Node *leftPointer = node->left;
178
179     node->left = leftPointer->right;
180
181     if (node->left != NULL)
182     {
183         node->left->parent = node;
184     }
185
186     leftPointer->parent = node->parent;
187
188     if (node->parent == NULL)
189     {
190         tree->root = leftPointer;
191     }
192     else if (node == node->parent->right)
193     {
194         node->parent->right = leftPointer;
195     }

```

```

196     else
197     {
198         node->parent->right = leftPointer;
199     }
200
201     leftPointer->right = node;
202     node->parent = leftPointer;
203 }
204
205 /**
206  * fixes violations caused by BST insertion
207  * @param tree
208  * @param nodeWeAdded
209  * @return
210  */
211 int fixViolationInRBTree(RBTree *tree, Node *nodeWeAdded)
212 {
213     //1. the nodeWeAdded is the root
214     //replace its color to black
215     if (tree->root == nodeWeAdded)
216     {
217         nodeWeAdded->color = BLACK;
218         return EXIT_SUCCESS;
219     }
220
221     //2. nodeWeAdded parent is black
222     //return
223     if (nodeWeAdded->parent->color == BLACK)
224     {
225         return EXIT_SUCCESS;
226     }
227
228     NodeFamily family;
229     findMyFamily(nodeWeAdded, &family);
230
231     //3. nodeWeAdded parent(P) is red and nodeWeAdded uncle(U) is red
232     //transform P and U color to black
233     //transform parent of P (G) to red
234     //recursive run the algorithm on G
235     if (isNodeBlack(family.father) == false && isNodeBlack(family.uncle) == false)
236     {
237         family.father->color = BLACK;
238         family.uncle->color = BLACK;
239         family.grandfather->color = RED;
240         fixViolationInRBTree(tree, family.grandfather);
241     }
242
243     //4. nodeWeAdded parent(P) is red and nodeWeAdded uncle(U) is black
244     //a. if(nodeWeAdded is right child of left child ||
245     //    nodeWeAdded is left child of right child)
246     //{ do rotation so nodeWeAdded will be right-right son or left-left son}
247     //b1. if(nodeWeAdded is left-left son){ do right rotation on G}
248     //b2. if(nodeWeAdded is right-right son){ do left rotation on G}
249     //c. transform P to red, and G to black
250     if (isNodeBlack(family.father) == false && isNodeBlack(family.uncle) == true)
251     {
252         if (family.myNodeIsleft == true && family.fatherIsLeft == false)
253         {
254             rotateRight(tree, family.father);
255             findMyFamily(family.father, &family);
256         }
257         else if (family.myNodeIsleft == false && family.fatherIsLeft == true)
258         {
259             rotateLeft(tree, family.father);
260             findMyFamily(family.father, &family);
261         }
262     }
263

```

7.1

```

264         if (family.myNodeIsleft == true && family.fatherIsLeft == true) //left-left
265         {
266             rotateRight(tree, family.grandfather);
267             // findMyFamily(nodeWeAdded,&family);
268             family.father->color = BLACK;
269             family.grandfather->color = RED;
270         }
271         if (family.myNodeIsleft == false && family.fatherIsLeft == false) //right-right
272         {
273             rotateLeft(tree, family.grandfather);
274             // findMyFamily(nodeWeAdded,&family);
275             family.father->color = BLACK;
276             family.grandfather->color = RED;
277         }
278     }
279     return EXIT_SUCCESS;
280 }
281
282 /**
283  * recursive function to insert node into tree as we do in bst
284  * @param compFunc
285  * @param currentNode
286  * @param nodeToAdd
287  * @param addMe
288  * @param parent
289  * @return
290  */
291 int insertNodeBstHelper(RBTree *tree, CompareFunc compFunc, Node *currentNode, Node *nodeToAdd,
292                        Node **addMe, Node *parent)
293 {
294     if (currentNode == NULL)
295     {
296         //add that node
297         *addMe = nodeToAdd;
298         tree->size += 1;
299         nodeToAdd->parent = parent;
300         return true;
301     }
302
303     int ans = compFunc(currentNode->data, nodeToAdd->data);
304     //ans equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
305     if (ans < 0) // node.data < data.data
306     {
307         //search in the right sub-tree
308         return insertNodeBstHelper(tree, compFunc, currentNode->right, nodeToAdd,
309                                   &currentNode->right, currentNode);
310     }
311     else //root > data
312     {
313         //search in the left sub-tree
314         return insertNodeBstHelper(tree, compFunc, currentNode->left, nodeToAdd, &currentNode->left,
315                                   currentNode);
316     }
317 }
318
319 /**
320  * insert new node to tree, as it done in BST
321  * @return true in success or false in failure
322  */
323 int insertNodeBst(RBTree *tree, Node *nodeToAdd)
324 {
325     //if the tree is empty, update the root to be the node
326     if (tree->root == NULL)
327     {
328         tree->root = nodeToAdd;
329         tree->size += 1;
330         return true;
331     }

```



```

332     //else, compare the items to left\right sons, and add in the right places.
333     insertNodeBstHelper(tree, tree->compFunc, tree->root, nodeToAdd, &tree->root, NULL);
334     return true;
335 }
336
337 /**
338  *check whether the sub-tree of given node Contains this item
339  * @param node
340  * @param data
341  * @return
342  */
343 int rBTreeContainsHelper(CompareFunc compFunc, const Node *currentNode, const void *data)
344 {
345     if (currentNode == NULL)
346     {
347         return NOT_IN_TREE;
348     }
349     int ans = compFunc(currentNode->data, data);
350     //equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
351     if (ans == 0) //they are equal
352     {
353         return IN_TREE;
354     }
355     if (ans < 0) // node.data < data.data
356     {
357         //search in the right sub-tree
358         return rBTreeContainsHelper(compFunc, currentNode->right, data);
359     }
360     else //root > data
361     {
362         //search in the left sub-tree
363         return rBTreeContainsHelper(compFunc, currentNode->left, data);
364     }
365 }
366
367 /**
368  * check whether the tree RBTree Contains this item.
369  * @param tree: the tree to check an item in.
370  * @param data: item to check.
371  * @return: 0 if the item is not in the tree, other if it is.
372  */
373 int RBTreeContains(const RBTree *tree, const void *data)
374 {
375     if (tree->root == NULL)
376     {
377         return NOT_IN_TREE;
378     }
379     return rBTreeContainsHelper(tree->compFunc, tree->root, data);
380 }
381
382 /**
383  * add an item to the tree
384  * @param tree: the tree to add an item to.
385  * @param data: item to add to the tree.
386  * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
387  */
388 int insertToRBTree(RBTree *tree, void *data)
389 {
390     //check if data is valid
391     if (data == NULL)
392     {
393         return false;
394     }
395
396     //check if item already in tree
397     if (RBTreeContains(tree, data) == IN_TREE)
398     {
399         return false;

```

```

400     }
401
402
403     //create node -red and with data inside
404     Node *newNode = createNode(NULL, NULL, NULL, RED, data);
405     if (newNode == NULL)
406     {
407         return false;
408     }
409
410     //do a normal Bst Insert
411     if (insertNodeBst(tree, newNode) == false)
412     {
413         return false;
414     }
415
416     //fix Red Black Tree violations
417     fixViolationInRBTtree(tree, newNode);
418     return true;
419 }
420
421
422 /**
423 * find the successor of node, at specific case the the given node do not have left child
424 * @param GivenNode
425 * @return the sucessor
426 */
427 Node *semiSuccessor(Node *GivenNode)
428 {
429     Node *tempPointer = GivenNode;
430     tempPointer = tempPointer->right;
431
432     while ( tempPointer->left != NULL)
433     {
434         tempPointer = tempPointer->left;
435     }
436
437     return tempPointer;
438 }
439
440 /**
441 * prepare the RBTtree to delete, while making sure that the node we want's to delete
442 * has at least one leaf. if needed we will find the successor, and replace the node with it
443 * @param tree
444 * @param nodeToDelete
445 * @return
446 */
447 void prepareToDeleteRBTtree(Node **nodeToDelete)
448 {
449     //check if nodeToDelete(M) has one or two child that a leaf?
450     if ((*nodeToDelete)->left == NULL || (*nodeToDelete)->right == NULL)
451     {
452         //yes? good. just return
453         return;
454     }
455     //no? find the successor
456     Node *successor = semiSuccessor(*nodeToDelete);
457     //switch the data of M and the value of its successor (not their colors!!)
458     void *tempData = successor->data;
459     successor->data = (*nodeToDelete)->data;
460     (*nodeToDelete)->data = tempData;
461     //mark the successor vertex as M
462     (*nodeToDelete) = successor;
463 }
464
465 /**
466 * we assume that only one(!) of the nodes is null
467 * @param currentNode

```

10.1

```

468  * @return
469  */
470  Node *findSonWhoIsNotNull(Node *currentNode)
471  {
472      if (currentNode == NULL)
473      {
474          return NULL;
475      }
476      if (currentNode->left != NULL)
477      {
478          return currentNode->left;
479      }
480      if (currentNode->right != NULL)
481      {
482          return currentNode->right;
483      }
484      return NULL;
485  }
486
487  /**
488   * in this function we get node to delete from tree
489   * (!!!) we assume that nodeToDelete has at most one child who is not a leaf (!!!)
490   * @param tree
491   * @param nodeToDelete
492   * @return
493   */
494  void deleteNodeAction(RBTree *tree, Node **inputNode)
495  {
496
497      Node *nodeToDelete = *inputNode;
498      //find family
499      NodeFamily family;
500      findMyFamily(nodeToDelete, &family);
501
502      //if nodeToDelete does't have any childrens:
503      if (nodeToDelete->right == NULL && nodeToDelete->left == NULL)
504      {
505          if (tree->root == nodeToDelete) // input node is the root
506          {
507              //free the root pointer to null
508              tree->root = NULL;
509          }
510          else //node have parent
511          {
512              //go to his father and free the pointer to the node to NULL
513              if (family.myNodeIsleft == true)
514              {
515                  nodeToDelete->parent->left = NULL;
516              }
517              else
518              {
519                  nodeToDelete->parent->right = NULL;
520              }
521          }
522          //free the data allocation
523          tree->freeFunc(nodeToDelete->data);
524          free(nodeToDelete);
525          *inputNode = NULL;
526          tree->size = tree->size - 1;
527          return;
528      }
529
530      // the left child of M is not a leaf
531      if (nodeToDelete->left != NULL)
532      {
533          if (tree->root == nodeToDelete) // input node is the root
534          {
535              //root point to left son

```

```

536         tree->root = nodeToDelete->left;
537         nodeToDelete->left->parent = NULL;
538     }
539     else //node has parent
540     {
541         if (family.myNodeIsleft == true) //M is left child
542         {
543             nodeToDelete->parent->left = nodeToDelete->left; //connect the child to the father
544         }
545         else //M is right child
546         {
547             nodeToDelete->parent->right = nodeToDelete->left; //connect the child to the father
548         }
549         nodeToDelete->left->parent = nodeToDelete->parent; //connect the father to the child
550     }
551     //free the node allocations
552     tree->freeFunc(nodeToDelete->data);
553     Node *temp = nodeToDelete->left;
554     free(nodeToDelete);
555     *inputNode = temp;
556     tree->size = tree->size - 1;
557     return;
558 }
559
560 // the right child of M is not a leaf
561 if (nodeToDelete->right != NULL)
562 {
563     if (tree->root == nodeToDelete) // input node is the root
564     {
565         //root points to right son
566         tree->root = nodeToDelete->right;
567         nodeToDelete->right->parent = NULL;
568     }
569     else
570     {
571         if (family.myNodeIsleft == true) //M is left child
572         {
573             nodeToDelete->parent->left = nodeToDelete->right; //connect the child to the father
574         }
575         else //M is right child
576         {
577             nodeToDelete->parent->right = nodeToDelete->right; //connect the child to the father
578         }
579         nodeToDelete->right->parent = nodeToDelete->parent; //connect the father to the child
580     }
581     //free the node allocations
582     tree->freeFunc(nodeToDelete->data);
583     Node *temp = nodeToDelete->right;
584     free(nodeToDelete);
585     *inputNode = temp;
586     tree->size = tree->size - 1;
587     return;
588 }
589 }
590
591 /**
592  * get node and return its brother
593  * @param myNode
594  * @return
595  */
596 Node *findBrother(Node *myNode, Node *father)
597 {
598
599     if (myNode == NULL)
600     {
601         if (father->left == NULL)
602         {
603             return father->right;

```

```

604     }
605     return father->left;
606 }
607
608 //init brother
609 if (father->right == myNode)
610 {
611     return myNode->parent->left;
612 }
613 else if (father->left == myNode)
614 {
615     return myNode->parent->right;
616 }
617 return NULL; //we do not suppose arrive this case
618 }
619
620 /**
621  *
622  * @param nodeToCheck
623  * @return true if the node is left child, false else
624  */
625 int isLeftChild(Node *nodeToCheck, Node *father)
626 {
627     if (father->left == nodeToCheck)
628     {
629         return true;
630     }
631     else
632     {
633         return false;
634     }
635 }
636
637 /**
638  * find the close S son to C
639  * @param nodeC
640  * @param nodeS
641  * @return the close SC
642  */
643
644 Node *findNodeCloseToC(Node *nodeC, Node *nodeS, Node *father)
645 {
646     if (isLeftChild(nodeC, father) == true)
647     {
648         return nodeS->left;
649     }
650     else // nodeC is right child
651     {
652         return nodeS->right;
653     }
654 }
655
656 /**
657  * find the fur S son to C
658  * @param nodeC
659  * @param nodeS
660  * @return the far SF
661  */
662
663 Node *findNodeFarToC(Node *nodeC, Node *nodeS, Node *father)
664 {
665     if (isLeftChild(nodeC, father) == true)
666     {
667         return nodeS->right;
668     }
669     else // nodeC is right child
670     {
671         return nodeS->left;

```

13.1

13.2

```

672     }
673
674 }
675
676 /**
677  * helper function to do the third case deletion
678  * @param tree
679  * @param nodeToDelete
680  */
681 void deleteThirdCase(RBTree *tree, Node *nodeToDelete, Node *father)
682 {
683     ///a. C is the root
684     if (father == NULL)
685     {
686         return; ///its ok. just return
687     }
688
689     Node *brother = findBrother(nodeToDelete, father);
690
691     ///b. S is black and he has two black sons
692     if (isNodeBlack(brother) == true && isNodeBlack(brother->right) == true &&
693         isNodeBlack(brother->left) == true)
694     {
695         ///i. if P is red
696         if (isNodeBlack(father) == false)
697         {
698             // make P black, and S to red
699             father->color = BLACK;
700             brother->color = RED;
701             return;
702         }
703         ///ii. if P is black
704         else if (isNodeBlack(father) == true)
705         {
706             //make S red ,and recursive call (from 3.a) on P
707             brother->color = RED;
708             deleteThirdCase(tree, father, father->parent);
709             return;
710         }
711     }
712
713     ///c. if S is red
714     if (isNodeBlack(brother) == false)
715     {
716         //make S black and P red
717         brother->color = BLACK;
718         father->color = RED;
719         //do rotation on P to side of C
720         if (isLeftChild(nodeToDelete, father) == true)
721         {
722             rotateLeft(tree, father);
723         }
724         else
725         {
726             rotateRight(tree, father);
727         }
728
729         //recursive call on on C from 3.a
730         deleteThirdCase(tree, nodeToDelete, father);
731         return;
732     }
733
734     ///d. if S is black and (son close to C) SC is red, and (son fur from C) SF is black
735     Node *sC = findNodeCloseToC(nodeToDelete, brother, father);
736     Node *sF = findNodeFarToC(nodeToDelete, brother, father);
737     if (isNodeBlack(brother) == true && isNodeBlack(sC) == false
738         && isNodeBlack(sF) == true)
739     {

```

```

740         //SC to black, S to red
741         sC->color = BLACK;
742         brother->color = RED;
743         //do rotation to S from opposite direction of C
744         if (isLeftChild(nodeToDelete, father) == true)
745         {
746             rotateRight(tree, brother);
747         }
748         else
749         {
750             rotateLeft(tree, brother);
751         }
752         //recursive call from 3.a on C
753         deleteThirdCase(tree, nodeToDelete, father);
754         return;
755     }
756
757     ///e. if S is black and SF is red
758     if (isNodeBlack(brother) == true && isNodeBlack(sF) == false)
759     {
760         //replace colors of S and P (swap)
761         Color tmp = brother->color;
762         brother->color = father->color;
763         father->color = tmp;
764         //do rotation on P to direction of C
765         if (isLeftChild(nodeToDelete, father) == true)
766         {
767             rotateLeft(tree, father);
768         }
769         else
770         {
771             rotateRight(tree, father);
772         }
773         //SF to black
774         sF->color = BLACK;
775         return;
776     }
777 }
778
779 /**
780  * part B of deleting
781  * in this function we assume that nodeToDelete has at most one child who is not a leaf
782  * @param tree
783  * @param nodeToDelete
784  */
785 void deleteSavingRBtreeProperties(RBTree *tree, Node *nodeToDelete)
786 {
787     //nodeToDelete (M) , child (C), father (F), brother (S)
788
789     //1. if M is red - just delete M
790     if (isNodeBlack(nodeToDelete) == false)
791     {
792         deleteNodeAction(tree, &nodeToDelete);
793         // tree->size = tree->size -1;
794         return;
795     }
796
797     //find son who is not null
798     Node *child = findSonWhoIsNotNull(nodeToDelete);
799     //2. if M is black and C is red
800     if (isNodeBlack(nodeToDelete) == true && child != NULL && isNodeBlack(child) == false)
801     {
802         deleteNodeAction(tree, &nodeToDelete); //delete M (wile connecting its son to his father)
803         // tree->size = tree->size -1;
804         nodeToDelete->color = BLACK;
805         return;
806     }
807

```

```

808 //3. if M is black and C is black
809 if (isNodeBlack(nodeToDelete) == true && isNodeBlack(child) == true)
810 {
811     Node *father = nodeToDelete->parent;
812     //delete M, and make the father points to (C)
813     deleteNodeAction(tree, &nodeToDelete);
814 //     tree->size = tree->size -1;
815     deleteThirdCase(tree, nodeToDelete, father); //problem is here
816 }
817 }
818
819 /**
820  * find the node to delete by the given data :)
821  * @param compFunc
822  * @param currentNode
823  * @param nodeWithData
824  * @param data
825  * @return
826  */
827 Node *findNode(CompareFunc compFunc, Node *currentNode, const void *data)
828 {
829     if (currentNode == NULL)
830     {
831         return NULL;
832     }
833     int ans = compFunc(currentNode->data, data);
834     //equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
835     if (ans == 0) //they are equal
836     {
837         return currentNode;
838     }
839     if (ans < 0) // node.data < data.data
840     {
841         //search in the right sub-tree
842         return findNode(compFunc, currentNode->right, data);
843     }
844     else //root > data
845     {
846         //search in the left sub-tree
847         return findNode(compFunc, currentNode->left, data);
848     }
849 }
850
851 /**
852  * remove an item from the tree
853  * @param tree: the tree to remove an item from.
854  * @param data: item to remove from the tree.
855  * @return: 0 on failure, other on success. (if data is not in the tree - failure).
856  */
857 int deleteFromRBTree(RBTree *tree, void *data)
858 {
859
860     //get the node with the data
861     Node *nodeToDelete = NULL;
862     nodeToDelete = findNode(tree->compFunc, tree->root, data);
863     if (nodeToDelete == NULL)
864     {
865         return false;
866     }
867     //check that the node we want to delete has at least one leaf
868     prepareToDeleteRBTree(&nodeToDelete);
869
870     //do the delete action while saving the RBTree properties
871     deleteSavingRBtreeProperties(tree, nodeToDelete);
872
873     return true;
874 }
875

```



```

876 /**
877  * helper function for the foreach action on the data
878  * @param currentNode
879  * @param func
880  * @param args
881  * @return status of activating the function
882  */
883 int foreachHelper(const Node *currentNode, foreachFunc func, void *args)
884 {
885     if (currentNode == NULL) //leaf
886     {
887         return true;
888     }
889     foreachHelper(currentNode->left, func, args);
890     if (func(currentNode->data, args) == false)
891     {
892         return false;
893     }
894     foreachHelper(currentNode->right, func, args);
895     return true;
896 }
897
898 /**
899  * Activate a function on each item of the tree. the order is an ascending order. if one of the activations of the
900  * function returns 0, the process stops.
901  * @param tree: the tree with all the items.
902  * @param func: the function to activate on all items.
903  * @param args: more optional arguments to the function (may be null if the given function support it).
904  * @return: 0 on failure, other on success.
905  */
906 int foreachRBTREE(const RBTREE *tree, foreachFunc func, void *args)
907 {
908     return (foreachHelper(tree->root, func, args));
909 }
910
911 int freeRBTREEHelper(RBTREE *tree, Node *currentNode)
912 {
913     if (currentNode == NULL) //leaf
914     {
915         return EXIT_SUCCESS;
916     }
917     freeRBTREEHelper(tree, currentNode->left);
918     tree->freeFunc(currentNode->data); //free the data
919     freeRBTREEHelper(tree, currentNode->right);
920     //we free the left and right child's both. now free the node itself
921     free(currentNode);
922     return EXIT_SUCCESS;
923 }
924
925 /**
926  * free all memory of the data structure.
927  * @param tree: pointer to the tree to free.
928  */
929 void freeRBTREE(RBTREE **tree)
930 {
931     //recursive :) , active freeFunc on data.
932     freeRBTREEHelper(*tree, (*tree)->root);
933     free(*tree);
934     *tree = NULL;
935 }

```

## 3 Structs.c

```
1 // implementation of Structs.h
2 //we will do this part on vectors
3 // ----- includes -----
4 #include "RBTREE.h"
5 #include "Structs.h"
6 #include <stdlib.h>
7 #include <string.h>
8
9 // ----- const definitions -----
10 #define LESS (-1)
11 #define EQUAL (0)
12 #define GREATER (1)
13 #define BACKSLASH_N "\n"
14
15 // ----- functions implementation -----
16
17 /**
18  * return the minimal int from the given input
19  * @param a
20  * @param b
21  * @return
22  */
23 int min(int a, int b)
24 {
25     if (a >= b)
26     {
27         return b;
28     }
29     else
30     {
31         return a;
32     }
33 }
34
35 /**
36  * CompFunc for strings (assumes strings end with "\0")
37  * @param a - char* pointer
38  * @param b - char* pointer
39  * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a. (lexicographic
40  * order)
41  */
42 int stringCompare(const void *a, const void *b)
43 {
44     char *stringA = (char *)a;
45     char *stringB = (char *)b;
46     size_t lenA = strlen(stringA);
47     size_t lenB = strlen(stringB);
48     size_t min = lenA > lenB ? lenB : lenA;
49     return strcmp(a, b);
50 }
51
52 /**
53  * ForEach function that concatenates the given word and \n to pConcatenated. pConcatenated is
54  * already allocated with enough space.
55  * @param word - char* to add to pConcatenated
56  * @param pConcatenated - char*
57  * @return 0 on failure, other on success
58  */
59 int concatenate(const void *word, void *pConcatenated)
```

18.1

```

60 {
61     if (strcat(pConcatenated, word) == NULL)
62     {
63         return EXIT_FAILURE;
64     }
65     if (strcat(pConcatenated, BACKSLASH_N) == NULL)
66     {
67         return EXIT_FAILURE;
68     }
69     return EXIT_SUCCESS;
70 }
71
72 /**
73  * FreeFunc for strings
74  */
75 void freeString(void *s)
76 {
77     free(s);
78 }
79
80 /**
81  * CompFunc for Vectors, compares element by element, the vector that has the first larger
82  * element is considered larger. If vectors are of different lengths and identify for the length
83  * of the shorter vector, the shorter vector is considered smaller.
84  * @param a - first vector
85  * @param b - second vector
86  * @return equal to 0 iff a == b. Lower than 0 if a < b. Greater than 0 iff b < a.
87  */
88 int vectorCompare1By1(const void *a, const void *b)
89 {
90
91
92     Vector *aVec = (Vector *)a;
93     Vector *bVec = (Vector *)b;
94     int minLen = min(aVec->len, bVec->len);
95     for (int i = 0; i < minLen; i++)
96     {
97         double currentA = aVec->vector[i];
98         double currentB = bVec->vector[i];
99         if (currentA > currentB)
100         {
101             return GREATER;
102         }
103         else if (currentA < currentB)
104         {
105             return LESS;
106         }
107         else if (currentA == currentB)
108         {
109             return EQUAL;
110         }
111     }
112     //they equal until one is ends-the shorter is smaller
113     int lenA = aVec->len;
114     int lenB = bVec->len;
115
116     if (lenA > lenB)
117     {
118         return GREATER;
119     }
120     if (lenA < lenB)
121     {
122         return LESS;
123     }
124     return EQUAL; //the last option- they are totally equal
125 }
126
127 /**

```

```

128  * FreeFunc for vectors
129  */
130  void freeVector(void *pVector)
131  {
132      Vector *myVector = (Vector *)pVector;
133      free(myVector->vector);
134      free(myVector);
135  }
136
137  /**
138   * helper function- calculate norm of given vector pointer
139   * @param pVector
140   * @return
141   */
142  double calcNorm(const void *pVector)
143  {
144      Vector *myVec = (Vector *)pVector;
145      double myNorm = 0;
146      for (int i = 0; i < myVec->len; i++)
147      {
148          double elemToAdd = myVec->vector[i] * myVec->vector[i];
149          myNorm += elemToAdd;
150      }
151      return myNorm;
152  }
153
154  /**
155   * copy pVector to pMaxVector if : 1. The norm of pVector is greater then the norm of pMaxVector.
156   *                               2. pMaxVector->vector == NULL.
157   * @param pVector pointer to Vector
158   * @param pMaxVector pointer to Vector that will hold a copy of the data of pVector.
159   * @return 1 on success, 0 on failure (if pVector == NULL || pMaxVector==NULL: failure).
160   */
161  int copyIfNormIsLarger(const void *pVector, void *pMaxVector)
162  {
163
164      if (pVector == NULL || pMaxVector == NULL)
165      {
166          return EXIT_FAILURE;
167      }
168      double norm1 = calcNorm(pVector);
169      double norm2 = calcNorm(pMaxVector);
170      Vector *myVec1 = (Vector *)pVector;
171      Vector *myVec2 = (Vector *)pMaxVector;
172
173      if (norm1 > norm2)
174      {
175          //deep copy of 1st vector to 2nd
176          myVec2->len = myVec1->len;
177          myVec2->vector = realloc(myVec2->vector, sizeof(double) * myVec2->len);
178          if (myVec2->vector == NULL)
179          {
180              return EXIT_FAILURE;
181          }
182          for (int i = 0; i < myVec2->len; i++)
183          {
184              myVec2->vector[i] = myVec1->vector[i];
185          }
186          return EXIT_SUCCESS;
187      }
188      return EXIT_SUCCESS;
189  }
190
191  /**
192   * This function allocates memory it does not free.
193   * @param tree a pointer to a tree of Vectors
194   * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm), NULL on failure.
195   */

```

```

196 Vector *findMaxNormVectorInTree(RBTree *tree)
197 {
198     Vector *myVec = (Vector *)malloc(sizeof(Vector));
199     if (myVec == NULL)
200     {
201         return NULL;
202     }
203     myVec->len = 0;
204     myVec->vector = NULL;
205     forEachRBTree(tree, copyIfNormIsLarger, myVec);
206     return myVec;
207 }

```

# 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)