

Contents

1	Basic Test Results	2
2	RBTree.c	4
3	Structs.c	17

1 Basic Test Results

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

```

60 *****
61
62 =====
63 = Checking coding style =
64 =====
65 RBTREE.c(589, 6):  deep_blocks {Do not make too deep block(6). It makes not readable code}
66 RBTREE.c(737, 5):  fname_case {Do not start function name(RBTREEContains) with uppercase}
67 RBTREE.c(737, 5):  fname_case {Do not start function name(RBTREEContains) with uppercase}
68 RBTREE.c(737, 5):  fname_case {Do not start function name(RBTREEContains) with uppercase}
69 ** Total Violated Rules      : 4
70 ** Total Errors Occurs      : 4
71 ** Total Violated Files Count: 1

```

2 RBTree.c

```
1  #ifndef RBTREE_C
2  #define RBTREE_C
3
4  /**
5   * @file RBTree.c
6   * @author Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
7   * @ID 300575297
8   * @date 23 May 2020
9   *
10  *
11  * @section DESCRIPTION
12  *      Program that implemented the Red-Black Tree.
13  */
14
15  // ----- includes -----
16
17  #include "RBTree.h"
18
19  #include <stdlib.h>
20  #include <string.h>
21  #include <stdio.h>
22  #include <stdbool.h>
23
24  // ----- macros & constants -----
25
26  #define LESS (-1)
27  #define EQUAL (0)
28  #define GREATER (1)
29
30  // ----- addition functions -----
31
32  Node *newRBNode(void *data);
33  void removeNode(RBTree *tree, Node **node);
34  void rotateLeft(RBTree *tree, Node *pivot);
35  void rotateRight(RBTree *tree, Node *pivot);
36  void helperInsertToRBTree(RBTree *tree, Node *new);
37  void insertionFixup(RBTree *tree, Node *node);
38  void helperDeleteFromRBTree(RBTree *tree, Node *node);
39  void deletionFixup(RBTree *tree, Node *node, Node * parent);
40  Node *findRBTree(const RBTree *tree, const void *data);
41  Node *getMin(Node *root);
42  int inOrderTraverse(const Node *root, forEachFunc func, void *args);
43
44  // -----
45
46  /**
47   * @brief: constructs a new empty RBTree with the given compFunc & freeFunc.
48   * @param compFunc: a function to compare two variables.
49   * @param freeFunc: a function to free a data item.
50   * @return: a pointer to RBTree.
51   */
52  RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)
53  {
54      RBTree *newEmptyRBTree = (RBTree *)malloc(sizeof(RBTree));
55      if (newEmptyRBTree == NULL)
56      {
57          return NULL;
58      }
59      newEmptyRBTree->root = NULL;
```

```

60     newEmptyRBTree->compFunc = compFunc;
61     newEmptyRBTree->freeFunc = freeFunc;
62     newEmptyRBTree->size = 0;
63
64     return newEmptyRBTree;
65 }
66
67 /**
68  * @brief: creates new RED-BLACK Node.
69  * @param data: the value of the new node.
70  * @return: a pointer to Node on success, otherwise NULL on failure.
71  */
72 Node *newRBNode(void *data)
73 {
74     if (data == NULL)
75     {
76         return NULL;
77     }
78     Node *RBNode = (Node *)malloc(sizeof(Node));
79     if (RBNode == NULL)
80     {
81         return NULL;
82     }
83     RBNode->data = data;
84     RBNode->color = RED;
85     RBNode->parent = RBNode->left = RBNode->right = NULL;
86
87     return RBNode;
88 }
89
90 /**
91  * @brief removes a node and the tree it induced.
92  * @param tree: RBTree to remove the node from.
93  * @param node: the node to remove.
94  */
95 void removeNode(RBTree *tree, Node **node)
96 {
97     if ((*node) == NULL)
98     {
99         return;
100     }
101     removeNode(tree, &((*node)->right));
102     removeNode(tree, &((*node)->left));
103
104     tree->freeFunc((*node)->data);
105     free(*node);
106     *node = NULL;
107 }
108
109 /**
110  * @brief get the minimum node of the RBTree spanned by the given root.
111  * @param root: root of RBTree.
112  * @return: the left most node in the RBTree.
113  */
114 Node *getMin(Node *root)
115 {
116     if (root == NULL)
117     {
118         return NULL;
119     }
120     while (root->left != NULL)
121     {
122         root = root->left;
123     }
124     return root;
125 }
126
127 /**

```

```

128  * @brief rotates left over the pivot node
129  * @param tree: the tree to execute the rotation in.
130  * @param pivot: the node to rotate over.
131  */
132  void rotateLeft(RBTree *tree, Node *pivot)
133  {
134      // WE ASSUME THAT (pivot->right != NULL)
135      Node *ptrToRight = pivot->right; // pointer to right child of pivot
136      pivot->right = ptrToRight->left;
137      // updates the attributes of pivot new right child
138      if (ptrToRight->left != NULL)
139      {
140          ptrToRight->left->parent = pivot;
141      }
142      // updates ptrToRight's parent
143      ptrToRight->parent = pivot->parent;
144      if (pivot->parent == NULL) // pivot was the root of the tree
145      {
146          tree->root = ptrToRight;
147      }
148      else if (pivot == pivot->parent->left) // pivot was a left child
149      {
150          pivot->parent->left = ptrToRight;
151      }
152      else // pivot was a right child
153      {
154          pivot->parent->right = ptrToRight;
155      }
156      // updates connection between ptrToRight and pivot
157      ptrToRight->left = pivot;
158      pivot->parent = ptrToRight;
159  }
160
161  /**
162   * @brief rotates right over the pivot node
163   * @param tree: the tree to execute the rotation in.
164   * @param pivot: the node to rotate over.
165   */
166  void rotateRight(RBTree *tree, Node *pivot)
167  {
168      // WE ASSUME THAT (pivot->left != NULL)
169      Node *ptrToLeft = pivot->left; // pointer to left child of pivot
170      pivot->left = ptrToLeft->right;
171      // updates the attributes of pivot new right child
172      if (ptrToLeft->right != NULL)
173      {
174          ptrToLeft->right->parent = pivot;
175      }
176      // updates ptrToLeft's parent
177      ptrToLeft->parent = pivot->parent;
178      if (pivot->parent == NULL) // pivot was the root of the tree
179      {
180          tree->root = ptrToLeft;
181      }
182      else if (pivot == pivot->parent->left) // pivot was a left child
183      {
184          pivot->parent->left = ptrToLeft;
185      }
186      else // pivot was a right child
187      {
188          pivot->parent->right = ptrToLeft;
189      }
190      // updates connection between ptrToRight and pivot
191      ptrToLeft->right = pivot;
192      pivot->parent = ptrToLeft;
193  }
194
195  /**

```

```

196  * @brief: inserts an item to the tree.
197  * @param tree: the tree to add an item to.
198  * @param data: item to insert to the tree.
199  * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
200  */
201  int insertToRBTree(RBTree *tree, void *data)
202  {
203      if (tree == NULL || data == NULL || RBTreeContains(tree, data))
204      {
205          return false;
206      }
207
208      Node *new = newRBNode(data);
209      if (new == NULL) // create newRBNode fails
210      {
211          return false;
212      }
213
214      helperInsertToRBTree(tree, new);
215
216      tree->size++; // updates the tree size.
217
218      return true;
219  }
220
221  /**
222   * @brief: helper for insertToRBTree
223   * @param tree: the tree to add the given node to.
224   * @param new: the new node to be added.
225   */
226  void helperInsertToRBTree(RBTree *tree, Node *new)
227  {
228      Node *parent = NULL; // parent to the new node
229      Node *cur = tree->root;
230
231      while (cur != NULL)
232      {
233          parent = cur;
234          int res = tree->compFunc(cur->data, new->data);
235          if (res < 0) // cur->data is less than new->data
236          {
237              cur = cur->right;
238          }
239          else // cur->data is greater than new->data
240          {
241              cur = cur->left;
242          }
243      }
244      new->parent = parent;
245
246      /* determine if the newly add node is a root, right child, or left child. */
247      if (parent == NULL)
248      {
249          /* RBTree was empty, the newly added node is root */
250          tree->root = new;
251      }
252      else if ((tree->compFunc(parent->data, new->data)) < 0)
253      {
254          /* parent->data is less than new->data */
255          parent->right = new; // the newly added node is right child
256      }
257      else
258      {
259          /* parent->data is greater than new->data */
260          parent->left = new; // the newly added node is left child
261      }
262      /* fix up the violation of RBTree properties */
263      insertionFixup(tree, new);

```

```

264 }
265
266 /**
267  * @brief: fixes up possible violations caused by insertion to RBTREE.
268  * @param tree: the tree to add an item to.
269  * @param node: the node caused the violation.
270  */
271 void insertionFixup(RBTree *tree, Node *node)
272 {
273     /* the inserted red leaf may be a child of a red node, so we have to
274      * fix the parent coloring recursively
275      */
276     Node *cur = node;
277     Node *grandparent = NULL;
278     Node *uncle = NULL;
279
280     /* the inserted leaf is not the root, and its parent is not black.
281     while (cur != tree->root && cur->parent->color == RED)
282     {
283         grandparent = cur->parent->parent;
284         /* parent is a left child, and uncle is a right child (CASE 3)
285         if (cur->parent == grandparent->left)
286         {
287             uncle = grandparent->right;
288             /* both parent and uncle are red
289             if (uncle != NULL && uncle->color == RED)
290             {
291                 /* color parent, uncle, and grandparent by complement
292                 cur->parent->color = BLACK;
293                 uncle->color = BLACK;
294                 grandparent->color = RED;
295
296                 cur = grandparent; // move the problem to the grandparent
297             }
298             else // uncle is black colored node, ordinary or RB leaf.
299             {
300                 /* if the node is an inner node: right child of left child (CASE 4.A),
301                 * then rotate the parent subtree to left, so the parent becomes an outer
302                 * leaf: left child of the current node (CASE 4.B)
303                 */
304
305                 if (cur == cur->parent->right)
306                 {
307                     cur = cur->parent;
308                     rotateLeft(tree, cur);
309                 }
310                 /* color the parent black and the grandparent red
311                 cur->parent->color = BLACK;
312                 grandparent->color = RED;
313                 /* rotate to right the grandparent's subtree
314                 rotateRight(tree, grandparent);
315             }
316         }
317     else
318     {
319         /* the symmetric case:
320         * the red parent is a right child, the uncle is the left child of
321         * the grandparent (CASE 3)
322         */
323         uncle = grandparent->left;
324
325         /* both parent and uncle are red
326         if (uncle != NULL && uncle->color == RED)
327         {
328             /* color parent, uncle, and grandparent by complement
329             cur->parent->color = BLACK;
330             uncle->color = BLACK;
331             grandparent->color = RED;

```



```

332         cur = grandparent; // move the problem to the grandparent.
333     }
334     else // uncle is black colored node, ordinary or RB leaf.
335     {
336         /* if the node is an inner node: left child of right child (CASE 4.A),
337          * then rotate the parent subtree to right, so the parent becomes an outer
338          * leaf: right child of the current node (CASE 4.B)
339          */
340         if (cur == cur->parent->left)
341         {
342             cur = cur->parent;
343             rotateRight(tree, cur);
344         }
345         // color the parent black and the grandparent red.
346         cur->parent->color = BLACK;
347         grandparent->color = RED;
348         // rotate to left the grandparent's subtree.
349         rotateLeft(tree, grandparent);
350     }
351 }
352 }
353 }
354 // Make sure that the root is black (CASE 1)
355 tree->root->color = BLACK;
356 }
357
358 /**
359  * @brief: deletes an item to the tree.
360  * @param tree: the tree to delete an item from.
361  * @param data: item to delete from the tree.
362  * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
363  */
364 int deleteFromRBTree(RBTree *tree, void *data)
365 {
366     if (tree == NULL || data == NULL)
367     {
368         return false;
369     }
370     Node *nodeToDel = findRBTree(tree, data);
371     if (nodeToDel == NULL)
372     {
373         return false;
374     }
375     helperDeleteFromRBTree(tree, nodeToDel);
376     tree->size--; // updates the tree size.
377     return true;
378 }
379
380 /**
381  * @brief swaps between two RB nodes
382  * @param tree: RBTree.
383  * @param node1: first RB node.
384  * @param node2: second RB node.
385  */
386 void swapValues(RBTree *tree, Node **node1, Node **node2)
387 {
388     /* check if second node is a right direct child of first node. */
389     int directChild = ((*node1)->right == (*node2));
390
391     Node *tempParent = (*node2)->parent;
392     Node *tempLeft = (*node2)->left;
393     Node *tempRight = (*node2)->right;
394     Color tempColor = (*node2)->color;
395
396     /* reset the pointers from second the node. */
397     (*node2)->parent = (*node1)->parent;

```

```

400     (*node2)->left = (*node1)->left;
401     (*node2)->right = (directChild) ? (*node1) : (*node1)->right;
402     (*node2)->color = (*node1)->color;
403
404     /* reset the pointers from first the node. */
405     (*node1)->parent = (directChild) ? (*node2) : tempParent;
406     (*node1)->left = tempLeft;
407     (*node1)->right = tempRight;
408     (*node1)->color = tempColor;
409
410     /* reset the pointers to first the node. */
411     if (! directChild)
412     {
413         if ((*node2) == (*node1)->parent->left)
414         {
415             (*node1)->parent->left = (*node1);
416         }
417         else
418         {
419             (*node1)->parent->right = (*node1);
420         }
421     }
422
423     if ((*node1)->left != NULL)
424     {
425         (*node1)->left->parent = (*node1);
426     }
427     if ((*node1)->right != NULL)
428     {
429         (*node1)->right->parent = (*node1);
430     }
431
432     /* reset the pointers to second the node. */
433     if ((*node2)->parent != NULL)
434     {
435         if ((*node1) == (*node2)->parent->left)
436         {
437             (*node2)->parent->left = (*node2);
438         }
439         else
440         {
441             (*node2)->parent->right = (*node2);
442         }
443     }
444     else
445     {
446         tree->root = (*node2);
447     }
448
449     if ((*node2)->left != NULL)
450     {
451         (*node2)->left->parent = (*node2);
452     }
453     if ((*node2)->right != NULL)
454     {
455         (*node2)->right->parent = (*node2);
456     }
457 }
458
459 /**
460  * @brief helper for deleteFromRBTree.
461  * @param tree: the tree to delete an item from
462  * @param node: the node to be deleted from the tree.
463  */
464 void helperDeleteFromRBTree(RBTree *tree, Node *node)
465 {
466     if (tree->size == 1)
467     {

```

```

468     removeNode(tree, &node);
469     tree->root = NULL;
470     return;
471 }
472
473 if ((node->right != NULL) && (node->left != NULL))
474 {
475     /* the node to delete has two children (that are NOT NULL),
476      * we swap the node with its successor.
477      */
478     Node *successor = getMin(node->right);
479     swapValues(tree, &node, &successor);
480 }
481
482 /* The node to delete is now has at most one child, because
483  * in the case of having two children we swap it with its
484  * successor which has at most one child (the right one).
485  */
486 Node *child = node->left ? node->left : node->right;
487 // get a pointer to the node's parent or use it in fix up the violation.
488 Node *parent = node->parent;
489
490 if (child != NULL) // NOT both of children are NULL
491 {
492     child->parent = node->parent;
493 }
494
495 if (node->parent == NULL) // the node is the root
496 {
497     tree->root = child;
498 }
499 else
500 {
501     if (node == node->parent->left)
502     {
503         node->parent->left = child;
504     }
505     else
506     {
507         node->parent->right = child;
508     }
509 }
510
511 /* RBTREE properties could be violated only if the color of the
512  * deleted node is black */
513 if (node->color == BLACK)
514 {
515     deletionFixup(tree, child, parent);
516 }
517
518 // delete the node, it is not connected to the tree anymore.
519 node->left = NULL;
520 node->right = NULL;
521 removeNode(tree, &node);
522 }
523
524 /**
525  * @brief fixes up possible violations caused by deletion from RBTREE.
526  * @param tree: the tree to add an item to.
527  * @param node: the node caused the violation.
528  */
529 void deletionFixup(RBTree *tree, Node *node, Node * parent)
530 {
531     /* Get a pointer to the current node and determine its color */
532     Node *curr = node;
533     Color currColor = curr ? curr->color : BLACK;
534
535     while ((curr != tree->root) && (currColor == BLACK))

```

```

536 {
537     /* Get pointers to the current node's parent and sibling */
538     Node *currParent = curr ? curr->parent : parent;
539     Node *sibling = NULL;
540
541     if (curr == currParent->left)
542     {
543         /* If the current node is a left child, then its sibling is the right
544         * child of the parent.
545         */
546         sibling = currParent->right;
547
548         /* Check the sibling's color. (NULL nodes are colored black) */
549         if ((sibling != NULL) && (sibling->color == RED))
550         {
551             /* In case the sibling is red, color it black and rotate.
552             * Then color the parent red (and the grandparent is now black).
553             */
554             sibling->color = BLACK;
555             currParent->color = RED;
556             rotateLeft(tree, currParent);
557             sibling = currParent->right;
558         }
559
560         if ((sibling != NULL) &&
561             (!sibling->left) || sibling->left->color == BLACK) &&
562             (!sibling->right) || sibling->right->color == BLACK))
563         {
564             /* If the sibling has two black children, color it red */
565             sibling->color = RED;
566             if (currParent->color == RED)
567             {
568                 /* If the parent is red, color it black and terminate
569                 * the fix-up process.
570                 */
571                 currParent->color = BLACK;
572                 curr = tree->root; /* In order to stop the while loop */
573             }
574             else
575             {
576                 /* The black depth of the entire sub-tree rooted at the parent is
577                 * now too small - fix it up recursively.
578                 */
579                 curr = currParent;
580             }
581         }
582     }
583     else
584     {
585         if (sibling == NULL)
586         {
587             /* The case of a NULL sibling */
588             if (currParent->color == RED)
589             {
590                 currParent->color = BLACK;
591                 curr = tree->root; /* In order to stop the while loop */
592             }
593             else
594             {
595                 curr = currParent;
596             }
597         }
598         else
599         {
600             /* In this case, at least one of the sibling's children is red.
601             * It is therefore obvious that the sibling itself is black.
602             */
603             if ((sibling->left != NULL) && (sibling->left->color == RED))

```

```

604     {
605         /* If the left child of the sibling is red, color it black,
606         * then color the sibling itself red, and rotate right around
607         * the sibling.
608         * Notice that the left child is the closest to the current node.
609         */
610         sibling->left->color = BLACK;
611         sibling->color = RED;
612         rotateRight(tree, sibling);
613         sibling = currParent->right;
614     }
615
616
617     /* If the right child of the sibling is red, swap the colors of the
618     * sibling and its parent, then color the child itself black and
619     * rotate around the current parent.
620     * Notice that the right child is the farthest from the current node.
621     */
622     sibling->color = currParent->color;
623     currParent->color = BLACK;
624
625     sibling->right->color = BLACK;
626     rotateLeft(tree, currParent);
627
628     curr = tree->root;      /* In order to stop the while loop */
629 }
630
631 }
632
633 else
634 {
635     /* If the current node is a right child, then its sibling is the left
636     * child of the parent.
637     */
638     sibling = currParent->left;
639
640     /* Check the sibling's color. (NULL nodes are colored black) */
641     if (sibling && sibling->color == RED)
642     {
643         /* In case the sibling is red, color it black and rotate.
644         * Then color the parent red (and the grandparent is now black).
645         */
646         sibling->color = BLACK;
647         currParent->color = RED;
648         rotateRight(tree, currParent);
649         sibling = currParent->left;
650     }
651
652     if ((sibling != NULL) &&
653         (!sibling->left) || sibling->left->color == BLACK) &&
654         (!sibling->right) || sibling->right->color == BLACK))
655     {
656         /* If the sibling has two black children, color it red */
657         sibling->color = RED;
658         if (currParent->color == RED)
659         {
660             /* If the parent is red, color it black and terminate
661             * the fix-up process.
662             */
663             currParent->color = BLACK;
664             curr = tree->root;      /* In order to stop the while loop */
665         }
666         else
667         {
668             /* The black depth of the entire sub-tree rooted at the parent is
669             * now too small - fix it up recursively.
670             */
671             curr = currParent;

```

```

672     }
673 }
674 else
675 {
676     if (sibling == NULL)
677     {
678         /* Take care of a NULL sibling */
679         if (currParent->color == RED)
680         {
681             currParent->color = BLACK;
682             curr = tree->root;    /* In order to stop the while loop */
683         }
684         else
685         {
686             curr = currParent;
687         }
688     }
689     else
690     {
691         /* In this case, at least one of the sibling's children is red.
692          * It is therefore obvious that the sibling itself is black.
693          */
694         if ((sibling->right != NULL) && (sibling->right->color == RED))
695         {
696             /* If the right child of the sibling is red, color it black,
697              * then color the sibling itself red, and rotate left around
698              * the sibling.
699              * Notice that the left right is the closest to the current node.
700              */
701             sibling->right->color = BLACK;
702             sibling->color = RED;
703             rotateLeft(tree, sibling);
704             sibling = currParent->left;
705         }
706
707         /* If the left child of the sibling is red, swap the colors of the
708          * sibling and its parent, then color the child itself black and
709          * rotate around the current parent.
710          * Notice that the left child is the farthest from the current node.
711          */
712         sibling->color = currParent->color;
713         currParent->color = BLACK;
714
715         sibling->left->color = BLACK;
716         rotateRight(tree, currParent);
717
718         curr = tree->root;    /* In order to stop the while loop */
719     }
720 }
721 }
722 }
723
724 /* The root can always be colored black */
725 if (curr != NULL)
726 {
727     curr->color = BLACK;
728 }
729 }
730
731 /**
732  * @brief: check whether the tree RBTreContains this item.
733  * @param tree: the tree to add an item to.
734  * @param data: item to check.
735  * @return: 0 if the item is not in the tree, other if it is.
736  */
737 int RBTreContains(const RBTre *tree, const void *data)
738 {
739     return ((findRBTre(tree, data)) != NULL);

```

```

740 }
741
742 /**
743  * @brief: helper for RBTREEContains
744  * @param root: root of a RBTREE.
745  * @param data: item to check.
746  * @return: pointer to the node contains the data, else NULL.
747  */
748 Node *findRBTREE(const RBTREE *tree, const void *data)
749 {
750     if (tree == NULL || data == NULL)
751     {
752         return NULL;
753     }
754
755     Node *cur = tree->root;
756     int result;
757
758     while (cur != NULL)
759     {
760         result = tree->compFunc(cur->data, data);
761         if (result == 0)
762         {
763             return cur;
764         }
765         cur = (result < 0) ? cur->right : cur->left;
766     }
767     return NULL;
768 }
769
770 /**
771  * @brief Activate a function on each item of the tree. the order is an ascending order.
772  *         if one of the activations of the function returns 0, the process stops.
773  * @param tree: the tree with all the items.
774  * @param func: the function to activate on all items.
775  * @param args: more optional arguments to the function.
776  * @return: 0 on failure, other on success.
777  */
778 int forEachRBTREE(const RBTREE *tree, forEachFunc func, void *args)
779 {
780     if (tree == NULL || func == NULL)
781     {
782         return false;
783     }
784     if (!inOrderTraverse(tree->root, func, args))
785     {
786         return false;
787     }
788     return true;
789 }
790
791 /**
792  * @brief Traverses on tree in order and activates the func on the node data.
793  * @param root: root of a RBTREE.
794  * @param func: the function to activate on all items.
795  * @param args: more optional arguments to the function.
796  * @return: 0 on failure, other on success.
797  */
798 int inOrderTraverse(const Node *root, forEachFunc func, void *args)
799 {
800     if (root == NULL) // the tree is empty
801     {
802         return true;
803     }
804     inOrderTraverse(root->left, func, args);
805     if (!func(root->data, args))
806     {
807         return false;

```

```

808     }
809     inOrderTraverse(root->right, func, args);
810     return true;
811 }
812
813 /**
814  * @brief free all memory of the data structure.
815  * @param tree: pointer to the tree to free.
816  */
817 void freeRBTREE(RBTree **tree)
818 {
819     if (tree == NULL || *tree == NULL)
820     {
821         return;
822     }
823     if ((*tree)->root != NULL)
824     {
825         removeNode((*tree), &((*tree)->root));
826     }
827     free(*tree);
828     *tree = NULL;
829 }
830
831
832 #endif //RBTREE_C
833

```


3 Structs.c

```
1  #ifndef STRUCTS_C
2  #define STRUCTS_C
3
4  /**
5   * @file Structs.c
6   * @author Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
7   * @ID 300575297
8   * @date 26 May 2020
9   *
10  *
11  * @section DESCRIPTION
12  *      Two concrete examples on implementation of RBTREE.h library:
13  *      (1) In the first the data of the node is of type C string.
14  *      (2) In the second the data of the node is of type Vector.
15  */
16
17  // ----- includes -----
18
19  #include "RBTREE.h"
20  #include "Structs.h"
21
22  #include <stdlib.h>
23  #include <string.h>
24  #include <stdio.h>
25  #include <math.h>
26  #include <stdbool.h>
27
28  // ----- macros & constants -----
29
30  #define LESS (-1)
31  #define EQUAL (0)
32  #define GREATER (1)
33
34  // ----- addition functions -----
35
36  double vecNorm(const Vector *pVector);
37  int deepCopy(const Vector *source, Vector *target);
38
39  // -----
40
41  /**
42   * CompFunc for strings (assumes strings end with "\0")
43   * @param a - char* pointer
44   * @param b - char* pointer
45   * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a. (lexicographic
46   * order)
47   */
48  int stringCompare(const void *a, const void *b)
49  {
50      if (a == NULL || b == NULL)
51      {
52          return false;
53      }
54      char *firstString = (char *) a;
55      char *secondString = (char *) b;
56
57      return strcmp(firstString, secondString);
58  }
59
```

```

60  /**
61   * ForEach function that concatenates the given word and \n to pConcatenated. pConcatenated is
62   * already allocated with enough space.
63   * @param word - char* to add to pConcatenated
64   * @param pConcatenated - char*
65   * @return 0 on failure, other on success
66   */
67  int concatenate(const void *word, void *pConcatenated)
68  {
69      // CHECK IN CASE OF OVERLAP STRINGS.
70      if (word == NULL || pConcatenated == NULL)
71      {
72          return false;
73      }
74      char *firstString = (char *) pConcatenated;
75      char *secondString = (char *) word;
76
77      strcat(strcat(firstString, secondString), "\n");
78
79      return true;
80  }
81
82  /**
83   * FreeFunc for strings
84   */
85  void freeString(void *s)
86  {
87      char *string = (char *)s;
88      free(string);
89  }
90
91  /**
92   * CompFunc for Vectors, compares element by element, the vector that has the first larger
93   * element is considered larger. If vectors are of different lengths and identify for the length
94   * of the shorter vector, the shorter vector is considered smaller.
95   * @param a - first vector
96   * @param b - second vector
97   * @return equal to 0 iff a == b. Lower than 0 if a < b. Greater than 0 iff b < a.
98   */
99  int vectorCompare1By1(const void *a, const void *b)
100  {
101      Vector *firstVec = (Vector *)a;
102      Vector *secondVec = (Vector *)b;
103
104      int minLen = (firstVec->len < secondVec->len) ? firstVec->len : secondVec->len;
105
106      for (int i = 0; i < minLen; ++i)
107      {
108          double max = fmax(*((firstVec->vector) + i), *((secondVec->vector) + i));
109
110          if (*((firstVec->vector) + i) < max)
111          {
112              return LESS;
113          }
114          else if (*((secondVec->vector) + i) < max)
115          {
116              return GREATER;
117          }
118      }
119
120      if ((firstVec->len != secondVec->len)) // CHECK (len > 0)
121      {
122          return (firstVec->len < secondVec->len) ? LESS : GREATER;
123      }
124
125      return EQUAL;
126  }
127

```

```

128  /**
129   * FreeFunc for vectors
130   */
131  void freeVector(void *pVector)
132  {
133      Vector *vec = (Vector *)pVector;
134      free(vec->vector);
135      free(vec); // free(pVector);
136  }
137
138  /**
139   * copy pVector to pMaxVector if : 1. The norm of pVector is greater than the norm of pMaxVector.
140   *                                     2. pMaxVector->vector == NULL.
141   * @param pVector pointer to Vector
142   * @param pMaxVector pointer to Vector
143   * @return 1 on success, 0 on failure (if pVector == NULL: failure).
144   */
145  int copyIfNormIsLarger(const void *pVector, void *pMaxVector)
146  {
147      if (pVector == NULL || pMaxVector == NULL)
148      {
149          return false;
150      }
151
152      Vector *vec = (Vector *)pVector;
153      Vector *maxVec = (Vector *)pMaxVector;
154
155      /* if there is no coordinates in pVector so it is not greater */
156      if (vec->vector == NULL)
157      {
158          return true;
159      }
160
161      // if (maxVec->vector == NULL)
162      // {
163      //     return deepCopy(vec, maxVec);
164      // }
165
166      double vectorNorm = vecNorm(pVector);
167      double maxVectorNorm = vecNorm(pMaxVector);
168
169      /* norm of pMaxVector is greater or equal to pVector */
170      if (maxVectorNorm == fmax(vectorNorm, maxVectorNorm))
171      {
172          return true;
173      }
174
175      return deepCopy(vec, maxVec);
176  }
177
178  /**
179   * @brief make a deep copy of a vector.
180   * @param source: the vector to be copied.
181   * @param target: the vector to copy to it.
182   * @return a deep copy of a given vector
183   */
184  int deepCopy(const Vector *source, Vector *target)
185  {
186      /* (source != NULL && *target != NULL && source->vector != NULL) */
187
188      // double *newVec = (double *)calloc(source->len, sizeof(double));
189      // if (newVec == NULL)
190      // {
191      //     return false;
192      // }
193      //
194      // free(target->vector);
195      // target->vector = newVec;

```

```

196
197     target->vector = realloc(target->vector, source->len * sizeof(double));
198
199     target->len = source->len;
200     for (int i = 0; i < source->len; ++i)
201     {
202         target->vector[i] = source->vector[i];
203     }
204
205     return true;
206 }
207
208 /**
209  * @brief calculates the norm of a given vector.
210  * @param pVector: the vector to calculate its norm.
211  * @return the norm of the vector.
212  */
213 double vecNorm(const Vector *pVector)
214 {
215     if (pVector->vector == NULL)
216     {
217         /* if there is no coordinates in pVector then the nor, is ZERO */
218         return 0;
219     }
220     double coordsSquaresSum = 0;
221     for (int i = 0; i < pVector->len; ++i)
222     {
223         coordsSquaresSum += pow(((pVector->vector) + i)) , 2);
224     }
225     return sqrt(coordsSquaresSum);
226 }
227
228 /**
229  * @param tree a pointer to a tree of Vectors
230  *         You must use copyIfNormIsLarger in the implementation!
231  * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm).
232  */
233 Vector *findMaxNormVectorInTree(RBTree *tree)
234 {
235     if (tree == NULL || tree->root == NULL)
236     {
237         return NULL;
238     }
239     Vector *maxVector = (Vector *)malloc(sizeof(Vector));
240     forEachRBTree(tree, copyIfNormIsLarger, maxVector);
241
242     return maxVector;
243 }
244
245 #endif // STRUCTS_C

```