

Contents

1	Basic Test Results	2
2	RBTree.c	4
3	Structs.c	11

1 Basic Test Results

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

```
60  = Checking coding style =  
61  =====  
62  ** Total Violated Rules      : 0  
63  ** Total Errors Occurs       : 0  
64  ** Total Violated Files Count: 0
```

2 RBTREE.c

```
1  /**
2   * @file RBTREE.c
3   * @author Brahan Wassan <brahan>
4   * @version 1.0
5   * @date 12 Dec 2019
6   *
7   * @brief Program that build a RedBlack tree
8   *
9   * @section DESCRIPTION
10  * The program builds a generic RB tree
11  * Input : tree nodes
12  * Process: checks if the user input is valid, and then build the tree
13  * Output : a tree with the desired data types
14  */
15  #include <stdio.h>
16  #include "RBTREE.h"
17  #include <stdlib.h>
18
19  #define SUCCESS 1
20  #define FAIL 0
21  #define EQUAL 0
22
23  /**
24   * constructs a new RBTREE with the given CompareFunc.
25   * comp: a function two compare two variables.
26   */
27  RBTREE *newRBTREE(CompareFunc compFunc, FreeFunc freeFunc)
28  {
29      RBTREE *tree = NULL;
30      tree = (RBTREE *) malloc(sizeof(RBTREE));
31      if (tree == NULL)
32      {
33          return NULL;
34      }
35      tree->root = NULL;
36      tree->compFunc = compFunc;
37      tree->freeFunc = freeFunc;
38      tree->size = 0;
39      return tree;
40  }
41
42  /**
43   * created a new node
44   * @param data the data which the node holds
45   * @return a new node
46   */
47  Node *newNode(void *data)
48  {
49      Node *node = NULL;
50      node = (Node *) malloc(sizeof(Node));
51
52      if (node == NULL)
53      {
54          return NULL;
55      }
56      node->data = data;
57      node->color = RED;
58      node->left = NULL;
59      node->right = NULL;
```

```

60     node->parent = NULL;
61     return node;
62 }
63
64 /**
65  * rotates the tree nodes to the right as we saw in DAST
66  * @param node the node which we need to fix its position
67  */
68 void rotateRight(Node *node)
69 {
70     Node *left = node->left;
71     Node *parent = node->parent;
72     if (left->right != NULL)
73     {
74         left->right->parent = node;
75     }
76     node->left = left->right;
77     node->parent = left;
78     left->right = node;
79     left->parent = parent;
80     if (parent != NULL)
81     {
82         if (parent->right == node)
83         {
84             parent->right = left;
85         }
86         else
87         {
88             parent->left = left;
89         }
90     }
91 }
92
93 /**
94  * rotates the tree nodes to the left as we saw in DAST
95  * @param node the node which we need to fix its position
96  */
97 void rotateLeft(Node *node)
98 {
99     Node *right = node->right;
100    Node *parent = node->parent;
101    if (right->left != NULL)
102    {
103        right->left->parent = node;
104    }
105    node->right = right->left;
106    node->parent = right;
107    right->left = node;
108    right->parent = parent;
109
110    if (parent != NULL)
111    {
112        if (parent->left == node)
113        {
114            parent->left = right;
115        }
116        else
117        {
118            parent->right = right;
119        }
120    }
121 }
122
123 /**
124  * a getter
125  * @param node the node
126  * @return parent
127  */

```

```

128 Node *getParent(Node *node)
129 {
130     return node == NULL ? NULL : node->parent;
131 }
132
133 /**
134  * a getter
135  * @param node the node
136  * @return the grandparent
137  */
138 Node *getGrandParent(Node *node)
139 {
140     return getParent(getParent(node));
141 }
142
143 /**
144  * a getter
145  * @param node the node
146  * @return the node sibling
147  */
148 Node *getSibling(Node *node)
149 {
150     Node *parent = getParent(node);
151
152     if (parent == NULL)
153     {
154         return NULL;
155     }
156     if (node == parent->left)
157     {
158         return parent->right;
159     }
160     else
161     {
162         return parent->left;
163     }
164 }
165
166 /**
167  * a getter
168  * @param node the node
169  * @return the parent sibling
170  */
171 Node *getUncle(Node *n)
172 {
173     Node *p = getParent(n);
174     return getSibling(p);
175 }
176
177 /**
178  * case 4 second step defined by the pdf
179  * @param tree the tree
180  * @param node the node we added to the tree
181  */
182 void caseFourSecondStep(RBTree *tree, Node *node)
183 {
184     Node *grandP = getGrandParent(node);
185     Node *parent = node->parent;
186     parent->color = BLACK;
187     grandP->color = RED;
188     if (node == parent->left)
189     {
190         rotateRight(grandP);
191         if (getGrandParent(node) == NULL)
192         {
193             tree->root = node->parent;
194         }
195     }

```

```

196     else
197     {
198         rotateLeft(grandP);
199         if (getGrandParent(node) == NULL)
200         {
201             tree->root = node->parent;
202         }
203     }
204 } // need to add it
205 /**
206  * the 4 cases which we fix the tree accordingly
207  * @param tree the tree which need to be fixed
208  * @param node the node which we added
209  */
210 void treeFix(RBTree *tree, Node *node)
211 {
212     Node *uncle = getUncle(node);
213     Node *parent = getParent(node);
214     Node *grandP = getGrandParent(node);
215     if (parent == NULL)
216     {
217         node->color = BLACK;
218     }
219     else if (parent->color == BLACK)
220     {
221         return;
222     }
223     else if (uncle != NULL && uncle->color == RED)
224     {
225         parent->color = BLACK;
226         uncle->color = BLACK;
227         grandP->color = RED;
228         treeFix(tree, grandP);
229     }
230     else
231     {
232         if (node == parent->right && parent == grandP->left)
233         {
234             rotateLeft(parent);
235             if (getGrandParent(node) == NULL)
236             {
237                 tree->root = node->parent;
238             }
239             node = node->left;
240         }
241         else if (node == parent->left && parent == grandP->right)
242         {
243             rotateRight(parent);
244             if (getGrandParent(node) == NULL)
245             {
246                 tree->root = node->parent;
247             }
248             node = node->right;
249         }
250         caseFourSecondStep(tree, node);
251     }
252 }
253 /**
254  * inserting a node to the first null place its finds recurse
255  * @param cur current node
256  * @param node the node we want to insert into the tree
257  * @param compareFunc the tree comp func
258  */
259 void regularBSTInsert(RBTree *tree, Node *cur, Node *node, CompareFunc compareFunc)
260 {
261     if (cur != NULL && cur->data != NULL)
262     {
263

```

```

264         if (compareFunc(cur->data, node->data) > EQUAL)
265         {
266             if (cur->left == NULL)
267             {
268                 cur->left = node;
269             }
270             else
271             {
272                 regularBSTInsert(tree, cur->left, node, compareFunc);
273                 return;
274             }
275         }
276         else
277         {
278             if (cur->right == NULL)
279             {
280                 cur->right = node;
281             }
282             else
283             {
284                 regularBSTInsert(tree, cur->right, node, compareFunc);
285                 return;
286             }
287         }
288         node->parent = cur;
289         node->color = RED;
290         ++tree->size;
291     }
292 }
293
294 /**
295  * add an item to the tree
296  * @param tree: the tree to add an item to.
297  * @param data: item to add to the tree.
298  * @return: 0 on failure, other on success. (if the item is already in the tree - failure).
299  */
300 int addToRBTREE(RBTree *tree, void *data)
301 {
302     if (tree == NULL || data == NULL)
303     {
304         return FAIL;
305     }
306     else
307     {
308         Node *node = newNode(data);
309         if (tree->root == NULL) // case 1 new node is root
310         {
311             node->color = BLACK;
312             tree->root = node;
313             ++tree->size;
314             return SUCCESS;
315         }
316         else if (containsRBTREE(tree, data) == SUCCESS)
317         {
318             free(node);
319             return FAIL;
320         }
321         else
322         {
323             regularBSTInsert(tree, tree->root, node, tree->compFunc);
324             treeFix(tree, node);
325             return SUCCESS;
326         }
327     }
328 }
329
330 /**
331  * a helper function which search inorder for the node

```



```

332  * @param root the current node
333  * @param data the data which we looking for in the tree
334  * @param compFunc the compare function which we can check if the nodes hold the identical data
335  * @return 1 if found 0 if not
336  */
337  int recursiveContains(Node *root, void *data, CompareFunc compFunc)
338  {
339      if (root == NULL)
340      {
341          return FAIL;
342      }
343      else
344      {
345          if (compFunc(root->data, data) == EQUAL)
346          {
347              return SUCCESS;
348          }
349          else if (compFunc(root->data, data) < EQUAL)
350          {
351              return recursiveContains(root->right, data, compFunc);
352          }
353          return recursiveContains(root->left, data, compFunc);
354      }
355  }
356
357  /**
358   * check whether the tree contains this item.
359   * @param tree: the tree to add an item to.
360   * @param data: item to check.
361   * @return: 0 if the item is not in the tree, other if it is.
362   */
363  int containsRBTree(RBTree *tree, void *data)
364  {
365      if (tree->root == NULL || data == NULL)
366      {
367          return FAIL;
368      }
369      return recursiveContains(tree->root, data, tree->compFunc) ? SUCCESS : FAIL;
370  }
371
372  int inOrder(Node *root, forEachFunc func, void *args)
373  {
374      int isOk = SUCCESS;
375      if (root != NULL)
376      {
377          if (root->left != NULL)
378          {
379              isOk = inOrder(root->left, func, args);
380              if (!isOk)
381              {
382                  return isOk;
383              }
384          }
385          isOk = func(root->data, args);
386          if (!isOk)
387          {
388              return isOk;
389          }
390          if (root->right != NULL)
391          {
392              isOk = inOrder(root->right, func, args);
393              if (!isOk)
394              {
395                  return isOk;
396              }
397          }
398      }
399      return isOk;

```

```

400 }
401
402 /**
403  * Activate a function on each item of the tree. the order is an ascending order. if one of the activations of the
404  * function returns 0, the process stops.
405  * @param tree: the tree with all the items.
406  * @param func: the function to activate on all items.
407  * @param args: more optional arguments to the function (may be null if the given function support it).
408  * @return: 0 on failure, other on success.
409  */
410 int foreachRBTREE(RBTree *tree, foreachFunc func, void *args)
411 {
412     int ret = FAIL;
413     if (tree != NULL)
414     {
415         ret = inOrder(tree->root, func, args);
416     }
417     return ret;
418 }
419
420 /**
421  * helper function which traverse thru the tree and free all the nodes
422  * @param root the current node
423  * @param freeFunc the tree free function
424  */
425 void freeAll(Node *root, FreeFunc freeFunc)
426 {
427     if (root == NULL)
428     {
429         return;
430     }
431     freeAll(root->left, freeFunc);
432     freeAll(root->right, freeFunc);
433     freeFunc(root->data);
434     free(root);
435 }
436
437 /**
438  * free all memory of the data structure.
439  * @param tree: the tree to free.
440  */
441 void freeRBTREE(RBTree *tree)
442 {
443     freeAll(tree->root, tree->freeFunc);
444     free(tree);
445 }

```

3 Structs.c

```
1  /**
2   * @file Structs.c
3   * @author Brahan Wassan <brahan>
4   * @version 1.0
5   * @date 12 Dec 2019
6   *
7   * @brief Program that define Vectors which can be nodes in the RBTREE
8   *
9   * @section DESCRIPTION
10  * The program builds a generic Vector
11  * Input : Vector data
12  * Process: checks if the user input is valid, and then define a node
13  * Output : a valid node
14  */
15 #include <string.h>
16 #include <malloc.h>
17 #include "Structs.h"
18
19 #define LESS (-1)
20 #define EQUAL (0)
21 #define GREATER (1)
22 #define SUCCESS (1)
23 #define TRUE (1)
24 #define FAIL (0)
25 #define SQUARE(a) (a)*(a)
26 #define UNDEFINED_SIZE (-1)
27
28 /**
29  * CompFunc for strings (assumes strings end with "\0")
30  * @param a - char* pointer
31  * @param b - char* pointer
32  * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a. (lexicographic
33  * order)
34  */
35 int stringCompare(const void *a, const void *b)
36 {
37     char *v = (char *) a;
38     char *u = (char *) b;
39     return strcmp(v, u);
40 }
41
42 /**
43  * ForEach function that concatenates the given word to pConcatenated. pConcatenated is already allocated with
44  * enough space.
45  * @param word - char* to add to pConcatenated
46  * @param pConcatenated - char*
47  * @return 0 on failure, other on success
48  */
49 int concatenate(const void *word, void *pConcatenated)
50 {
51     const char *cWord = (char *) word;
52     char *cP = (char *) pConcatenated;
53     size_t firstLen = strlen(cWord);
54     size_t secLen = strlen(cP);
55     if (strcat(cP, cWord))
56     {
57         strcat(cP, "\n");
58     }
59     if (strlen(cP) < firstLen + secLen)
```

```

60     {
61         return FAIL;
62     }
63     return SUCCESS;
64
65 }
66
67 /**
68  * CompFunc for Vectors, compares element by element, the vector that has the first larger
69  * element is considered larger. If vectors are of different lengths and identify for the length
70  * of the shorter vector, the shorter vector is considered smaller.
71  * @param a - first vector
72  * @param b - second vector
73  * @return equal to 0 iff a == b. lower than 0 if a < b. Greater than 0 iff b < a.
74  */
75 int vectorCompare1By1(const void *a, const void *b)
76 {
77     Vector *v = (Vector *) a;
78     Vector *u = (Vector *) b;
79     int isEq = 0;
80     if (v->len == u->len)
81     {
82         isEq = TRUE;
83     }
84     int minLen = v->len;
85     if (minLen > u->len)
86     {
87         minLen = u->len;
88     }
89     int i = 0;
90     while (i < minLen)
91     {
92         if (v->vector[i] != u->vector[i])
93         {
94             if (v->vector[i] > u->vector[i])
95             {
96                 return GREATER;
97             }
98             return LESS;
99         }
100         ++i;
101     }
102     if (isEq == TRUE)
103     {
104         return EQUAL;
105     }
106     else
107     {
108         if (minLen == v->len)
109         {
110             return LESS;
111         }
112         return GREATER;
113     }
114 }
115 /**
116  * calculate vector norm
117  * @param v the vector
118  * @return the vector norm
119  */
120 double calcNorm(const Vector *v)
121 {
122     if (v == NULL || v->vector == NULL)
123     {
124         return UNDEFINED_SIZE;
125     }
126     int len = v->len;
127     double norm = 0;

```

```

128     double cur = 0;
129     for (int i = 0; i < len; ++i)
130     {
131         cur = SQUARE(v->vector[i]);
132         norm += cur;
133     }
134     return norm;
135 }
136
137 /**
138  * copy pVector to pMaxVector if : 1. The norm of pVector is greater then the norm of pMaxVector.
139  *                                2. pMaxVector == NULL.
140  * @param pVector pointer to Vector
141  * @param pMaxVector pointer to Vector
142  * @return 1 on success, 0 on failure (if pVector == NULL: failure).
143  */
144 int copyIfNormIsLarger(const void *pVector, void *pMaxVector)
145 {
146     if (pVector == NULL)
147     {
148         return FAIL;
149     }
150     const Vector *toCopy = (Vector *) pVector;
151     if (toCopy->vector == NULL)
152     {
153         return FAIL;
154     }
155     Vector *maxVec = (Vector *) pMaxVector;
156     if (maxVec->vector == NULL || calcNorm(maxVec) < calcNorm(toCopy))
157     {
158         if (maxVec->vector != NULL)
159         {
160             free(maxVec->vector);
161         }
162         maxVec->vector = (double *) malloc(sizeof(double) * toCopy->len);
163         if (maxVec->vector == NULL)
164         {
165             free(maxVec);
166             return FAIL;
167         }
168         for (int i = 0; i < toCopy->len; ++i)
169         {
170             maxVec->vector[i] = toCopy->vector[i];
171         }
172         maxVec->len = toCopy->len;
173     }
174     return SUCCESS;
175 }
176
177 /**
178  * @param tree a pointer to a tree of Vectors
179  * @return pointer to a *copy* of the vector that has the largest norm (L2 Norm).
180  * // implement it in Structs.c You must use copyIfNormIsLarger in the implementation!
181  */
182 Vector *findMaxNormVectorInTree(RBTree *tree)
183 {
184     Vector *cur = (Vector *) malloc(sizeof(Vector));
185     if (cur == NULL)
186     {
187         return NULL;
188     }
189     cur->len = 0;
190     cur->vector = NULL;
191     Vector *pMaxNorm = (Vector *) cur;
192     int flag = forEachRBTree(tree, copyIfNormIsLarger, pMaxNorm);
193     if (flag == FAIL)
194     {
195         return NULL;

```

```

196     }
197     return pMaxNorm;
198 }
199
200 /**
201  * FreeFunc for strings
202  */
203 void freeString(void *s)
204 {
205     if (s == NULL)
206     {
207         return;
208     }
209     free(s);
210 }
211
212 /**
213  * FreeFunc for vectors
214  */
215 void freeVector(void *pVector)
216 {
217     Vector *node = (Vector *) pVector;
218     if (node != NULL)
219     {
220         if (node->vector != NULL)
221         {
222             free(node->vector);
223         }
224         free(pVector);
225     }
226 }

```