Contents

1 Basic Test Results

```
Running...
Opening tar file
generalSwap.c
GenRangeTree.c
GenRangeTree.h
valdbg.out
Makefile
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Importing files
OK
Importing files
OK
Compilation check...
Compiling...
rm -f generalSwap debugTreeFile libGenRangeTree.a ex3 valdbg.out
rm -f *.o
OK
Compiling...
gcc -Wall generalSwap.c -o generalSwap
OK
Compiling...
gcc -Wall -D NDEBUG=1 -c GenRangeTree.c -o GenRangeTree.o
ar rcs libGenRangeTree.a GenRangeTree.o
OK
Compiling...
gcc -Wall GenRangeTree.c -o debugTreeFile
OK
Compilation went without errors, BUT you must check to see if you got warnings!!!
Check some inputs:
Running test...
OK
OK

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

2 GenRangeTree.h

```
1  /**
2   * A general binary search tree where the nodes are sorted according to their keys
3   * Each node have the functions for comparing, copy, printing and free
4   */
5  #ifndef GEN_RANGE_TREE_H
6  #define GEN_RANGE_TREE_H
7
8  /** Pointer to an element in the range tree */
9  typedef void* Element;
10 typedef const void* ConstElement;
11
12 /** The compare function of the range tree */
13 typedef int (*lmCmpFunc)(ConstElement, ConstElement);
14
15 /** The copy function of the range tree */
16 typedef Element (*lmCpyFunc)(ConstElement);
17
18 /** The print function of the range tree */
19 typedef char* (*lmLblFunc)(ConstElement);
20
21 /** The free function of the range tree */
22 typedef void (*lmFreFunc)(Element);
23
24 typedef enum
25 {
26     FALSE,
27     TRUE
28 } Boolean;
29
30 /** Pointer at a range tree */
31 typedef struct GenRangeTreeRec *RangeTreeP;
32 typedef const struct GenRangeTreeRec *ConstRangeTreeP;
33
34 /**
35  * create a new range tree, Returns a pointer to it.
36  * The nodes of the tree will contain the participateWorkers workers from the array.
37  * In addition, receive 4 pointers to functions:
38  * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
39  *   if the items are equal or positive number if the first element is larger than the second element.
40  * - cpy - duplicate an element. Return NULL in case of memory out.
41  * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
42  *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
43  * - fre - a function that free the memory allocate for the element.
44  * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
45  * from it.
46  * Same error handling as in the SimpleRangeTree.c file.
47 */
48 RangeTreeP createNewRangeTree(Element participateWorkers[], int arrsize,
49                               int cmp(ConstElement, ConstElement),
50                               Element cpy(ConstElement),
51                               char *lbl(ConstElement),
52                               void fre(Element));
53
54 /**
55  * Free the range tree from the memory (should be called when the user doesn't need the range tree anymore).
56  */
57 void destroyRangeTree(RangeTreeP tree);
58
59 /**
```

```
60   * Return the number of workers in the range tree.
61   */
62  int size(ConstRangeTreeP tree);
63
64  /**
65   * Print the tree according to a range quory - print all the workers that
66   * are paid at least as p1, and at most as p2.
67   */
68  void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2);
69
70  /**
71   * Debbuging function - you don't have to use it but you may find it helpful.
72   */
73  void debugStableCheck(ConstRangeTreeP tree);
74
75  #endif
```

3 GenRangeTree.c

```c
/**
 * The implementation of the binary tree for sorting workers according to their salary
 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "GenRangeTree.h"

/*********************************************************************************
 A tree node definitions and functions
 *********************************************************************************/

typedef struct Node *NodeP;
typedef const struct Node *ConstNodeP;
typedef enum
{
    LEFT,
    RIGHT
} Side;
typedef enum
{
    OUT_OF_MEMORY,
    NULL_INPUT,
    SET_A_ROOT_WHEN_EXISTS,
    GENERAL_ERROR,
    ELEMENT_ADD_TWICE,
    BAD_RANGE
} ErrorTypes;

static void reporterrorMessage(ErrorTypes theErr, int currLineNumber)
{
    fprintf(stderr, "ERROR in line %d: ", currLineNumber);
    if (theErr == OUT_OF_MEMORY)
    {
        fprintf(stderr, "Out of memory!!!\n");
    }
    else if (theErr == NULL_INPUT)
    {
        fprintf(stderr, "Function received an illegal input (NULL Pointer)!!!\n");
    }
    else if (theErr == SET_A_ROOT_WHEN_EXISTS)
    {
        fprintf(stderr, "The root of the tree isn't empty, but you're trying to set it!!!\n");
    }
    else if (theErr == ELEMENT_ADD_TWICE)
    {
        fprintf(stderr, "The array contain two elements with the same data!!!\n");
    }
    else if (theErr == BAD_RANGE)
    {
        fprintf(stderr, "Bad input range for printRange: p1 is bigger than p2!!!\n");
    }
    else
    {
        fprintf(stderr, "General error.\n");
    }
    exit(1);
```

```
60  }
61
62  #define ERROR_MESSAGE(x) reporterrorMessage(x, __LINE__)
63
64  /**
65   * A node in the tree contains a pointer to the two sons, to the parent an to the key
66   */
67  struct Node
68  {
69      NodeP _left;
70      NodeP _right;
71      NodeP _parent;
72      Element _key;    // Points to data about the worker
73  };
74
75  static NodeP getNewNode(ConstElement key, NodeP left, NodeP right, NodeP parent, lmCpyFunc lmCpy)
76  {
77      assert(lmCpy != NULL);
78      NodeP retVal = (NodeP) malloc(sizeof(struct Node));
79      if (retVal == NULL)
80      {
81          ERROR_MESSAGE(OUT_OF_MEMORY);
82      }
83      if (key == NULL)
84      {
85          ERROR_MESSAGE(NULL_INPUT);
86      }
87      retVal->_left = left;
88      retVal->_right = right;
89      retVal->_parent = parent;
90      retVal->_key = lmCpy(key);
91      assert(retVal->_key != NULL);
92      return retVal;
93  }
94
95  static void freeNode(NodeP node, lmFreFunc lmFre)
96  {
97      assert(lmFre != NULL);
98      if (node == NULL)
99      {
100         ERROR_MESSAGE(NULL_INPUT);
101     }
102     lmFre(node->_key);
103     free(node);
104     node = NULL;
105 }
106
107 static NodeP getChildren(ConstNodeP node, Side side)
108 {
109     if (node == NULL)
110     {
111         ERROR_MESSAGE(NULL_INPUT);
112     }
113     return (side == LEFT) ? node->_left : node->_right;
114 }
115
116 static NodeP getParent(NodeP node)
117 {
118     if (node == NULL)
119     {
120         ERROR_MESSAGE(NULL_INPUT);
121     }
122     return node->_parent;
123 }
124
125 static Element getNodeKey(NodeP node)
126 {
127     return node->_key;
```

```
128 }
129
130 static void setChild(NodeP node, Side side, NodeP child)
131 {
132     if (node == NULL || child == NULL)
133     {
134         ERROR_MESSAGE(NULL_INPUT);
135     }
136     if (side == LEFT)
137     {
138         assert(node->_left == NULL);
139         node->_left = child;
140     }
141     else
142     {
143         assert(side == RIGHT);
144         assert(node->_right == NULL);
145         node->_right = child;
146     }
147 }
148
149 static Side whichChild(ConstNodeP node, ConstNodeP child)
150 {
151     if (node == NULL || child == NULL)
152     {
153         ERROR_MESSAGE(NULL_INPUT);
154     }
155     if (node->_right == child)
156     {
157         return RIGHT;
158     }
159     assert(node->_left == child);
160     return LEFT;
161 }
162
163
164 /*****************************************************************************
165  The range tree definitions and functions
166  *****************************************************************************/
167
168 /**
169  * A struct that contains the tree of Workers.
170  * Including the root, the maximal node and the number of leafs in the tree
171  */
172 struct GenRangeTreeRec
173 {
174     /* The tree root, contains NULL for an empty tree */
175     NodeP _root;
176
177     /* A pointer to the node with the maximum value in the tree (useful for the successor function).
178        We have to update this field in the Add/Remove element functions. */
179     NodeP _maxNode;
180
181     /* Number of nodes in the tree */
182     int _size;
183
184     /* Duplicate an element. Return NULL in case of memory out. */
185     lmCpyFunc lmCpy;
186
187     /* Compare between two elements, return negative number if the first is smaller than the
188      * second, zero if the items are equal or positive number if the first element is larger than
189      * the second element. */
190     lmCmpFunc lmCmp;
191
192     /* turn an element into a string (so we can print it). Allocate memory for the string - it's
193      * our responsibility to free the memory after using the string. In case of out-of-memory event,
194      * return NULL. */
195     lmLblFunc lmLbl;
```

```
196
197         /* a function that free the memory allocate for the element. */
198         lmFreFunc lmFre;
199     };
200
201     static NodeP getRoot(ConstRangeTreeP tree)
202     {
203         if (tree == NULL)
204         {
205             ERROR_MESSAGE(NULL_INPUT);
206         }
207         return tree->_root;
208     }
209
210     /* For save setRoot, the root must be NULL in order to set it */
211     static void setRoot(RangeTreeP tree, NodeP node, Boolean safe)
212     {
213         if (tree == NULL || node == NULL)
214         {
215             ERROR_MESSAGE(NULL_INPUT);
216         }
217         if (getRoot(tree) != NULL && safe)
218         {
219             ERROR_MESSAGE(SET_A_ROOT_WHEN_EXISTS);
220         }
221         tree->_root = node;
222     }
223
224     /*
225         Search for keyToSearchFor in the SubTree. Helper function of subTreeSearch (see below).
226     */
227     static NodeP subTreeSearchRec(NodeP root, ConstElement keyToSearchFor, lmCmpFunc lmCmp)
228     {
229         assert(lmCmp != NULL);
230         int cmpRetVal;
231         assert(keyToSearchFor != NULL);
232         if (root == NULL)
233         {
234             return NULL;
235         }
236         cmpRetVal = lmCmp(root->_key, keyToSearchFor);
237         if (cmpRetVal == 0)
238         {
239             return root;
240         }
241         if (cmpRetVal > 0)
242         {
243             if (getChildren(root, LEFT) == NULL)
244             {
245                 return root;
246             }
247             return subTreeSearchRec(getChildren(root, LEFT), keyToSearchFor, lmCmp);
248         }
249         if (getChildren(root, RIGHT) == NULL)
250         {
251             return root;
252         }
253         return subTreeSearchRec(getChildren(root, RIGHT), keyToSearchFor, lmCmp);
254     }
255
256     /* Search for keyToSearchFor in the range tree. Will return NULL for an empty range tree,
257        a pointer to the node if the node exists in the tree or a pointer to the last
258        node in the search path otherwise.                                    */
259     static NodeP subTreeSearch(ConstRangeTreeP tree, ConstElement keyToSearchFor)
260     {
261         if (tree == NULL || keyToSearchFor == NULL)
262         {
263             ERROR_MESSAGE(NULL_INPUT);
```

```
264     }
265     return subTreeSearchRec(tree->_root, keyToSearchFor, tree->lmCmp);
266 }
267 
268 static void addElement(RangeTreeP tree, ConstElement keyToSearchFor)
269 {
270     int direct;
271     NodeP parent;
272     debugStableCheck(tree);
273     if (tree == NULL || keyToSearchFor == NULL)
274     {
275         ERROR_MESSAGE(NULL_INPUT);
276     }
277     parent = subTreeSearch(tree, keyToSearchFor);
278     if (parent == NULL)
279     {
280         /* An empty tree - the new node will be the root (special case) */
281         NodeP newRoot = getNewNode(keyToSearchFor, NULL, NULL, NULL, tree->lmCpy);
282         assert(tree->_size == 0);
283         assert(newRoot != NULL);
284         setRoot(tree, newRoot, TRUE);
285         tree->_maxNode = newRoot;
286         ++tree->_size;
287         return;
288     }
289     direct = tree->lmCmp(getNodeKey(parent), keyToSearchFor);
290     if (direct == 0)
291     {
292         /* The element is already in the tree */
293         ERROR_MESSAGE(ELEMENT_ADD_TWICE);
294     }
295     ++tree->_size;
296     if (direct > 0)
297     {
298         NodeP newNode = getNewNode(keyToSearchFor, NULL, NULL, parent, tree->lmCpy);
299         assert(newNode != NULL);
300         setChild(parent, LEFT, newNode);
301     }
302     else
303     {
304         NodeP newNode = getNewNode(keyToSearchFor, NULL, NULL, parent, tree->lmCpy);
305         assert(newNode != NULL);
306         if (tree->lmCmp(getNodeKey(tree->_maxNode), getNodeKey(newNode)) < 0)
307         {
308             tree->_maxNode = newNode;
309         }
310         setChild(parent, RIGHT, newNode);
311     }
312 }
313 
314 /*
315  * Initializes the random number seed.
316  *
317  * The seed is initialized from the environment variable SRAND_SEED, or,
318  * if SRAND_SEED is undefined, uses the system time as the seed.
319  */
320 static void initializeSeed()
321 {
322     char *seedStr = getenv("SRAND_SEED");
323     unsigned int seed;
324 
325     if (seedStr != NULL)
326     {
327         /* read seed from the environment variable and convert to an integer */
328         seed = atoi(seedStr);
329     }
330     else
331     {
```

```
332         /* use the system time as a seed. it changes every second and never repeats. */
333         seed = time(NULL);
334     }
335 
336     srand(seed);
337 }
338 
339 /*
340  * Returns a random integer from the range [low,high].
341  */
342 static int chooseRandomNumber(int low, int high)
343 {
344     /* In Numerical Recipes in C: The Art of Scientific Computing
345        (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New  York:  Cambridge
346        University Press, 1992 (2nd ed., p. 277)), the following comments are made:
347               "If you want to generate a random integer between 1 and 10, you should always do it
348                by using high-order bits, as in
349 
350                       j = 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));
351 
352        (cited by rand(3) man page) */
353     int num = low + (int) ( ((double)(high - low + 1)) * (rand() / (RAND_MAX + 1.0)));
354 
355     return num;
356 }
357 
358 
359 /* Get the inserted order entered by the user and "mix" the array to create a "random" insertion order.
360    There exists better algorithm for randomness, but the following algorithm is good enough
361    for our purpose. */
362 static void generateRandomPermutation(Element participateElements[], int arrsize)
363 {
364     int it;
365     if (arrsize < 2)
366     {
367         return;
368     }
369     initializeSeed();
370     for (it = 0 ; it < arrsize ; ++it)
371     {
372         Element tempElement;
373         int f1 = chooseRandomNumber(0, arrsize-1);
374         int f2 = chooseRandomNumber(0, arrsize-1);
375         if (f1 == f2)
376         {
377             continue;
378         }
379         tempElement = participateElements[f1];
380         participateElements[f1] = participateElements[f2];
381         participateElements[f2] = tempElement;
382     }
383 }
384 
385 /**
386  * create a new range tree, Returns a pointer to it.
387  * The nodes of the tree will contain the participateWorkers workers from the array.
388  * In addition, receive 4 pointers to functions:
389  * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
390  *   if the items are equal or positive number if the first element is larger than the second element.
391  * - cpy - duplicate an element. Return NULL in case of memory out.
392  * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
393  *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
394  * - fre - a function that free the memory allocate for the element.
395  * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
396  * from it.
397  * Same error handling as in the SimpleRangeTree.c file.
398 */
399 RangeTreeP createNewRangeTree(Element lmArray[], int arrSize, lmCmpFunc cmp,
```

```
                                lmCpyFunc cpy, lmLblFunc lbl, lmFreFunc fre)
{
    if (cmp == NULL || cpy == NULL || lbl == NULL || fre == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    int it;
    RangeTreeP retVal = (RangeTreeP) malloc(sizeof(struct GenRangeTreeRec));
    if (retVal == NULL)
    {
        ERROR_MESSAGE(OUT_OF_MEMORY);
    }
    generateRandomPermutation(lmArray, arrSize);
    retVal->_root = NULL;
    retVal->_maxNode = NULL;
    retVal->_size = 0;
    retVal->lmCpy = cpy;
    retVal->lmCmp = cmp;
    retVal->lmLbl = lbl;
    retVal->lmFre = fre;
    for (it = 0 ; it < arrSize ; ++it)
    {
        addElement(retVal, lmArray[it]);
    }
    return retVal;
}

static void freeNodeRec(NodeP node, lmFreFunc lmFre)
{
    if (node == NULL)
    {
        return;
    }
    freeNodeRec(getChildren(node, RIGHT), lmFre);
    freeNodeRec(getChildren(node, LEFT), lmFre);
    freeNode(node, lmFre);
}

/**
 * Call this function if you want to clear all the elements in the node.
*/
static void clearTree(RangeTreeP tree)
{
    if (tree == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    freeNodeRec(getRoot(tree), tree->lmFre);
    tree->_size = 0;
    tree->_maxNode = NULL;
}

/**
 * Free the range tree from the memory (should be called when the user doesn't need the range tree anymore).
 */
void destroyRangeTree(RangeTreeP tree)
{
    if (tree == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    clearTree(tree);
    free(tree);
    tree = NULL;
}

/**
 * Return the number of workers in the range tree.
```

```
468  */
469 int size(ConstRangeTreeP tree)
470 {
471     if (tree == NULL)
472     {
473         ERROR_MESSAGE(NULL_INPUT);
474     }
475     return tree->_size;
476 }
477
478 static NodeP getMinimum(NodeP n)
479 {
480     while(1)
481     {
482         NodeP tempN;
483         assert(n != NULL);
484         tempN = getChildren(n, LEFT);
485         if (tempN == NULL)
486         {
487             return n;
488         }
489         n = tempN;
490     }
491     return NULL;
492 }
493
494 /* Return the successor of the node 'n' in the range tree, or NULL if 'n' is already the maximum */
495 static NodeP successor(NodeP n, NodeP maximumNode)
496 {
497     NodeP tempN;
498     assert(n != NULL);
499
500     /* Check if 'n' is the maximum */
501     if (n == maximumNode)
502     {
503         return NULL;
504     }
505
506     /* if 'n' has a right child go visit its minimum */
507     tempN = getChildren(n, RIGHT);
508     if (tempN != NULL)
509     {
510         return getMinimum(tempN);
511     }
512
513     /* Get 'n' node first father such that 'n' it's his left son */
514     while (1)
515     {
516         NodeP oldN = n;
517         n = getParent(n);
518         assert(n != NULL);
519         if (LEFT == whichChild(n, oldN))
520         {
521             break;
522         }
523     }
524
525     return n;
526 }
527
528 /* Search the tree, find the node that contains the worker with the smallest
529    paycheck that is bigger than p1 paycheck */
530 static NodeP findMinAboveElement(ConstRangeTreeP tree, ConstElement p)
531 {
532     NodeP retVal = NULL;
533     NodeP curr;
534     assert(tree != NULL);
535     assert(p != NULL);
```

```
536     curr = getRoot(tree);
537     while (curr != NULL)
538     {
539         ConstElement currElement = getNodeKey(curr);
540         if (tree->lmCmp(currElement, p) >= 0)
541         {
542             if (retVal == NULL)
543             {
544                 retVal = curr;
545             }
546             if (tree->lmCmp(getNodeKey(curr), getNodeKey(retVal)) < 0)
547             {
548                 retVal = curr;
549             }
550             curr = getChildren(curr, LEFT);
551         }
552         else
553         {
554             curr = getChildren(curr, RIGHT);
555         }
556     }
557     return retVal;
558 }
559
560 /**
561  * Print the tree according to a range quory - print all the workers that
562  * are paid at least as p1, and at most as p2.
563  */
564 void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2)
565 {
566     NodeP opt;
567     if (tree == NULL || p1 == NULL || p2 == NULL)
568     {
569         ERROR_MESSAGE(NULL_INPUT);
570     }
571     if (tree->lmCmp(p1, p2) > 0)
572     {
573         ERROR_MESSAGE(BAD_RANGE);
574     }
575     opt = findMinAboveElement(tree, p1);
576     if (opt == NULL)
577     {
578         return;
579     }
580     while (tree->lmCmp(getNodeKey(opt), p2) <= 0)
581     {
582         char* elementString = tree->lmLbl(getNodeKey(opt));
583         printf("%s\n", elementString);
584         free(elementString);
585         elementString = NULL;
586         opt = successor(opt, tree->_maxNode);
587         if (opt == NULL)
588         {
589             return;
590         }
591     }
592 }
593
594 /**
595  * Used for debbugin
596  * Verify that the Node is leagal (as a node in a binary search tree)
597  * Then verify all it successors recursively
598  */
599 static void debugCheckNode(NodeP n, int (*lmCmp)(ConstElement, ConstElement))
600 {
601     if (n == NULL)
602     {
603         return;
```

```
604         }
605         if (n->_left != NULL)
606         {
607             assert(n->_left->_parent == n);
608             assert(lmCmp(getNodeKey(n->_left), getNodeKey(n)) < 0);
609             debugCheckNode(n->_left, lmCmp);
610         }
611         if (n->_right != NULL)
612         {
613             assert(n->_right->_parent == n);
614             assert(lmCmp(getNodeKey(n->_right), getNodeKey(n)) > 0);
615             debugCheckNode(n->_right, lmCmp);
616         }
617     }
618
619     /**
620      * Debbuging function - you don't have to use it but you may find it helpful.
621      */
622     void debugStableCheck(ConstRangeTreeP tree)
623     {
624         assert(tree != NULL);
625         debugCheckNode(tree->_root, tree->lmCmp);
626         assert(tree->_root == NULL || (tree->_maxNode != NULL && tree->_maxNode->_right == NULL));
627     }
628
629     #ifndef NDEBUG
630     #define NUM_PAR 10
631
632     /**
633      * If the condition false, prints an error
634      */
635     static void checkForError(Boolean condition, char *message)
636     {
637         if(condition)
638         {
639             return;
640         }
641         fprintf(stderr, "ERROR: %s\n", message);
642         exit(1);
643     }
644
645     /**
646      * Creating a new integer with a given value
647      */
648     int* getInt(int value)
649     {
650         int* retVal = (int*) malloc(sizeof(int));
651         if (retVal == NULL)
652         {
653             ERROR_MESSAGE(OUT_OF_MEMORY);
654         }
655         *retVal = value;
656         return retVal;
657     }
658
659     /**
660      * Compare Manager Salaries
661      */
662     static int intCmp(ConstElement c1, ConstElement c2)
663     {
664         return *(int*)c1 - *(int*)c2;
665     }
666
667     /**
668      * Copy Integer function
669      */
670     static Element cpyInt(ConstElement c)
671     {
```

```
672     int* temp = (int*)malloc(sizeof(int));
673     if (temp == NULL)
674     {
675         ERROR_MESSAGE(OUT_OF_MEMORY);
676     }
677     *temp = *(int*)c;
678     return ((Element) temp);
679 }
680 
681 /**
682  * lbl Manager function
683  */
684 static char* lblInt(ConstElement c)
685 {
686     char* numToStr = (char*)malloc(sizeof(char)*10);
687     if (numToStr == NULL)
688     {
689         ERROR_MESSAGE(OUT_OF_MEMORY);
690     }
691     sprintf(numToStr, "%d", *(int*)c);
692     return numToStr;
693 }
694 
695 /**
696  * Free Manager function
697 */
698 static void freInt(Element c)
699 {
700     free((int*)c);
701     c = NULL;
702 }
703 
704 /**
705  * Create a tree and check if its creation went well. Return the tree.
706  */
707 static RangeTreeP createTreeCheck(int treeSize, Element lmArray[treeSize], lmCmpFunc cmp,
708                                   lmCpyFunc cpy, lmLblFunc lbl, lmFreFunc fre, Element maxKey,
709                                   Element minKey)
710 {
711     // Check if empty tree being created correctly
712     Element emptyArray[0];
713     RangeTreeP emptyTree = createNewRangeTree(emptyArray, 0, *cmp, *cpy, *lbl, *fre);
714     checkForError(emptyTree != NULL, "Tree can't be NULL after being created");
715     checkForError(emptyTree->_root == NULL, "Root must be NULL in an empty tree");
716     checkForError(emptyTree->_maxNode == NULL, "Tree max element must be NULL in an empty tree");
717     checkForError(emptyTree->_size == 0, "Tree size must be 0 in an empty tree");
718     checkForError(emptyTree->lmCmp != NULL, "Compare function must not be NULL after empty tree "
719                   "creation");
720     checkForError(emptyTree->lmCpy != NULL, "Copy function must not be NULL after empty tree "
721                   "creation");
722     checkForError(emptyTree->lmFre != NULL, "Free function must not be NULL after empty tree "
723                   "creation");
724     checkForError(emptyTree->lmLbl != NULL, "Lbl function must not be NULL after empty tree "
725                   "creation");
726     destroyRangeTree(emptyTree);
727 
728     // Check if not an empty tree being created correctly
729     RangeTreeP tree = createNewRangeTree(lmArray, treeSize, *cmp, *cpy, *lbl, *fre);
730     checkForError(emptyTree != NULL, "Tree can't be NULL after being created");
731     checkForError(tree->_root != NULL, "Tree root must not be NULL in a tree that ain't empty");
732     checkForError(tree->lmCmp(tree->_maxNode->_key, maxKey) == 0,
733              "Tree max element is incorrect after creation");
734     ConstElement minTreeKey = getMinimum(tree->_root)->_key;
735     checkForError(tree->lmCmp(minTreeKey, minKey) == 0, "Tree minimum element is incorrect after "
736             "creation");
737     checkForError(tree->_size == treeSize, "Tree size is incorrect after creation");
738     return tree;
739 }
```

```
740
741 /**
742  * Add an element to the tree and check if it was added correctly.
743  */
744 static void addElementCheck(RangeTreeP tree, int treeSize, Element keyToAdd)
745 {
746     addElement(tree, keyToAdd);
747     treeSize++;
748     checkForError(tree->_size == treeSize, "Tree size is incorrect after addition");
749     checkForError(tree->lmCmp(tree->_maxNode->_key, keyToAdd) >= 0,
750             "Tree max element is incorrect after addition");
751     NodeP nodeFound = subTreeSearch(tree, keyToAdd);
752     checkForError(tree->lmCmp(nodeFound->_key, keyToAdd) == 0,
753             "The element isn't found in the tree after addition");
754 }
755
756 /**
757  * The main method for the debug process
758  */
759 int main()
760 {
761     RangeTreeP tree;
762     Element tempArr[NUM_PAR];
763     int* numbers[NUM_PAR];
764
765     // Initializing the numbers array
766     numbers[0] = getInt(5);
767     numbers[1] = getInt(2);
768     numbers[2] = getInt(3);
769     numbers[3] = getInt(4);
770     numbers[4] = getInt(1);
771     numbers[5] = getInt(6);
772     numbers[6] = getInt(7);
773     numbers[7] = getInt(10);
774     numbers[8] = getInt(8);
775     numbers[9] = getInt(9);
776     int* maxKey = numbers[7];
777     int* minKey = numbers[4];
778
779     int currNum;
780     for (currNum = 0 ; currNum < NUM_PAR ; ++currNum)
781     {
782         tempArr[currNum] = (Element) numbers[currNum];
783     }
784
785     // Checking the creation of the tree
786     tree = createTreeCheck(NUM_PAR, tempArr, &intCmp, &cpyInt, &lblInt,
787                            &freInt, maxKey, minKey);
788
789     // Checking the element adding
790     int* newInt = getInt(11);
791     addElementCheck(tree, NUM_PAR, newInt);
792
793     // Free all the variables
794     for (currNum = 0 ; currNum < NUM_PAR ; ++currNum)
795     {
796         freInt(numbers[currNum]);
797     }
798     freInt(newInt);
799     destroyRangeTree(tree);
800     return EXIT_SUCCESS;
801 }
802 #endif
```

4 generalSwap.c

```c
/**
 * A swap functions that works with everything
 * Need to get also the size of the elements that are being swapped
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef void* Element;

void generalSwap(Element a, Element b, int sizeOfElement);

int main()
{
    double a = 4., b = 5.;
    int c = 44, d = 55;
    generalSwap(&a, &b, sizeof(double));
    generalSwap(&c, &d, sizeof(int));
    printf("a=%f b=%f\n", a, b); /* should print a=5 b=4 */
    printf("a=%d b=%d\n", c, d); /* should print a=55 b=44 */
    return 0;
}

/**
 * The swap function
 * Swap the two given elements of the given size
 */
void generalSwap(Element a, Element b, int sizeOfElements)
{
    Element temp = malloc(sizeOfElements);
    if (temp == NULL)
    {
        return;
    }
    else
    {
        memcpy(temp, a, sizeOfElements);
        memcpy(a, b, sizeOfElements);
        memcpy(b, temp, sizeOfElements);
        free(temp);
        temp = NULL;
    }
}
```

5 valdbg.out

```
==12945== Memcheck, a memory error detector
==12945== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==12945== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==12945== Command: GenRangeTree
==12945== Parent PID: 12943
==12945==
==12945==
==12945== HEAP SUMMARY:
==12945==     in use at exit: 0 bytes in 0 blocks
==12945==   total heap usage: 35 allocs, 35 frees, 552 bytes allocated
==12945==
==12945== All heap blocks were freed -- no leaks are possible
==12945==
==12945== For counts of detected and suppressed errors, rerun with: -v
==12945== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```