Contents

1 Basic Test Results

```
Running...
Opening tar file
generalSwap.c
GenRangeTree.c
GenRangeTree.h
Makefile
valdbg.out
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 GenRangeTree GenRangeTree.o GenRangeTest.o generalSwap.o libGenRangeTree.a
OK
Compiling...
gcc -Wall -std=c99 -g -c ./generalSwap.c
gcc -Wall -std=c99 -g generalSwap.o -o ./generalSwap
OK
Compiling...
gcc -Wall -std=c99 -g -c -DNDEBUG ./GenRangeTree.c
ar rcs libGenRangeTree.a ./GenRangeTree.o
OK
Compiling...
gcc -Wall -std=c99 -g -c ./GenRangeTree.c -o ./GenRangeTest.o
gcc -Wall -std=c99 -g ./GenRangeTest.o ./Manager.o -o ./GenRangeTree
./GenRangeTree
Start testing:
check the root:
correct!
check left child:
correct!
check right child:
correct!
check get minimum:
correct!
check minimum successor:
correct!
test end!
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  typedef void* Element;
9  typedef const void* ConstElement;
10 typedef enum
11 {
12     FALSE,
13     TRUE
14 } Boolean;
15
16 /* Pointer at a range tree */
17 typedef struct GenRangeTreeRec *RangeTreeP;
18 typedef const struct GenRangeTreeRec *ConstRangeTreeP;
19
20 /**
21  * create a new range tree, Returns a pointer to it.
22  * The nodes of the tree will contain the participateWorkers workers from the array.
23  * In addition, receive 4 pointers to functions:
24  * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
25  *   if the items are equal or positive number if the first element is larger than the second element.
26  * - cpy - duplicate an element. Return NULL in case of memory out.
27  * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
28  *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
29  * - fre - a function that free the memory allocate for the element.
30  * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
31  * from it.
32  * Same error handling as in the SimpleRangeTree.c file.
33 */
34 RangeTreeP createNewRangeTree(Element participateWorkers[], int arrsize,
35                               int cmp(ConstElement, ConstElement),
36                               Element cpy(ConstElement),
37                               char *lbl(ConstElement),
38                               void fre(Element));
39
40 /**
41  * Free the range tree from the memory (should be called when the user doesn't need the range tree anymore).
42  */
43 void destroyRangeTree(RangeTreeP tree);
44
45 /**
46  * Return the number of workers in the range tree.
47  */
48 int size(ConstRangeTreeP tree);
49
50 /**
51  * Print the tree according to a range quory - print all the workers that
52  * are paid at least as p1, and at most as p2.
53  */
54 void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2);
55
56 /**
57  * Debbuging function - you don't have to use it but you may find it helpful.
58  */
59 void debugStableCheck(ConstRangeTreeP tree);
```

```
60
61  #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"

#ifndef NDEBUG
#include "Manager.h"
#endif

/*********************************************************************************
 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,
    WORNG_COPY_FUNC,
    WORNG_CMP_FUNC,
    WORNG_LBL_FUNC,
    WORNG_FREE_FUNC
} 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 workers with the same paycheck!!!\n");
```

```
60      }
61      else if (theErr == BAD_RANGE)
62      {
63          fprintf(stderr, "Bad input range for printRange: p1 is bigger than p2!!!\n");
64      }
65      else if (theErr == WORNG_COPY_FUNC)
66      {
67          fprintf(stderr, "Bad input for new tree: wrong copy function!\n");
68      }
69      else if (theErr == WORNG_CMP_FUNC)
70      {
71          fprintf(stderr, "Bad input for new tree: wrong compare function!\n");
72      }
73      else if (theErr == WORNG_LBL_FUNC)
74      {
75          fprintf(stderr, "Bad input for new tree: wrong label function!\n");
76      }
77      else if (theErr == WORNG_FREE_FUNC)
78      {
79          fprintf(stderr, "Bad input for new tree: wrong free function!\n");
80      }
81      else
82      {
83          fprintf(stderr, "General error.\n");
84      }
85      exit(1);
86  }
87
88  #define ERROR_MESSAGE(x) reporterrorMessage(x, __LINE__)
89
90  /**
91   * A node in the tree contains a pointer to the two sons, to the parent an to the key
92   */
93  struct Node
94  {
95      NodeP _left;
96      NodeP _right;
97      NodeP _parent;
98      Element _key;    // Points to data
99  };
100
101 /**
102  * create new node
103  * parameters:
104  *     Element (*lmCpy)(ConstElement) - copy function
105  *     ConstElement key
106  *     NodeP left
107  *     NodeP right
108  *     NodeP parent
109  * @return - the new node
110  */
111 static NodeP getNewNode(Element (*lmCpy)(ConstElement), ConstElement key, NodeP left, NodeP right,
112                         NodeP parent)
113 {
114     assert(lmCpy != NULL);
115     NodeP retVal = (NodeP) malloc(sizeof(struct Node));
116     if (retVal == NULL)
117     {
118         ERROR_MESSAGE(OUT_OF_MEMORY);
119     }
120     if (key == NULL)
121     {
122         ERROR_MESSAGE(NULL_INPUT);
123     }
124     retVal->_left = left;
125     retVal->_right = right;
126     retVal->_parent = parent;
127     retVal->_key = (*lmCpy)(key);
```

```
128     assert(retVal->_key != NULL);
129     return retVal;
130 }
131 
132 /**
133  * free the node memory
134  * NodeP node
135  * void (*lmFre)(Element) - free function
136  */
137 static void freeNode(NodeP node, void (*lmFre)(Element))
138 {
139     assert(lmFre != NULL);
140     if (node == NULL)
141     {
142         ERROR_MESSAGE(NULL_INPUT);
143     }
144     (*lmFre)(node->_key);
145     free(node);
146 }
147 
148 /**
149  * @return node child
150  * parameters:
151  *     ConstNodeP node
152  *     Side side - the child side
153  */
154 static NodeP getChildren(ConstNodeP node, Side side)
155 {
156     if (node == NULL)
157     {
158         ERROR_MESSAGE(NULL_INPUT);
159     }
160     return (side == LEFT) ? node->_left : node->_right;
161 }
162 
163 /**
164  * @return node parent
165  * parameters:
166  *     NodeP node - child node
167  */
168 static NodeP getParent(NodeP node)
169 {
170     if (node == NULL)
171     {
172         ERROR_MESSAGE(NULL_INPUT);
173     }
174     return node->_parent;
175 }
176 
177 /**
178  * @return node key
179  * parameters:
180  *     NodeP node
181  */
182 static Element getNodeKey(NodeP node)
183 {
184     return node->_key;
185 }
186 
187 /**
188  * set up a new child
189  * parameters:
190  *     NodeP node - parent node
191  *     Side side - child side
192  *     NodeP child - child node
193  */
194 static void setChild(NodeP node, Side side, NodeP child)
195 {
```

```
196     if (node == NULL || child == NULL)
197     {
198         ERROR_MESSAGE(NULL_INPUT);
199     }
200     if (side == LEFT)
201     {
202         assert(node->_left == NULL);
203         node->_left = child;
204     }
205     else
206     {
207         assert(side == RIGHT);
208         assert(node->_right == NULL);
209         node->_right = child;
210     }
211 }
212
213 /**
214  * @return child side
215  * parameters:
216  *     ConstNodeP node - parent node
217  *  ConstNodeP child
218  */
219 static Side whichChild(ConstNodeP node, ConstNodeP child)
220 {
221     if (node == NULL || child == NULL)
222     {
223         ERROR_MESSAGE(NULL_INPUT);
224     }
225     if (node->_right == child)
226     {
227         return RIGHT;
228     }
229     assert(node->_left == child);
230     return LEFT;
231 }
232
233
234 /*****************************************************************************
235  The range tree definitions and functions
236  *****************************************************************************/
237
238 /**
239  * A struct that contains the tree of Workers.
240  * Including the root, the maximal node and the number of leafs in the tree
241  */
242 struct GenRangeTreeRec
243 {
244
245     int (*lmCmp)(ConstElement, ConstElement);
246     Element (*lmCpy)(ConstElement);
247     char *(*lmLbl)(ConstElement);
248     void (*lmFre)(Element);
249
250     /* The tree root, contains NULL for an empty tree */
251     NodeP _root;
252
253     /* A pointer to the node with the maximum value in the tree (usefull for the successor function).
254        We have to update this field in the Add/Remove element fuctions. */
255     NodeP _maxNode;
256
257     /* Number of nodes in the tree */
258     int _size;
259 };
260
261 /**
262  * @return tree root
263  * parameters:
```

```
 *     ConstRangeTreeP tree
 */
static NodeP getRoot(ConstRangeTreeP tree)
{
    if (tree == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    return tree->_root;
}

/* For save setRoot, the root must be NULL in order to set it */
static void setRoot(RangeTreeP tree, NodeP node, Boolean safe)
{
    if (tree == NULL || node == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    if (getRoot(tree) != NULL && safe)
    {
        ERROR_MESSAGE(SET_A_ROOT_WHEN_EXISTS);
    }
    tree->_root = node;
}

/*
    Search for keyToSearchFor in the SubTree. Helper function of subTreeSearch (see below).
*/
static NodeP subTreeSearchRec(NodeP root, ConstElement keyToSearchFor,
                              int (*lmCmp)(ConstElement, ConstElement))
{
    assert(lmCmp != NULL);
    int cmpRetVal;
    assert(keyToSearchFor != NULL);
    if (root == NULL)
    {
        return NULL;
    }
    cmpRetVal = lmCmp(root->_key, keyToSearchFor);

    if (cmpRetVal == 0)
    {
        return root;
    }
    if (cmpRetVal > 0)
    {
        if (getChildren(root, LEFT) == NULL)
        {
            return root;
        }
        return subTreeSearchRec(getChildren(root, LEFT), keyToSearchFor, lmCmp);
    }
    if (getChildren(root, RIGHT) == NULL)
    {
        return root;
    }
    return subTreeSearchRec(getChildren(root, RIGHT), keyToSearchFor, lmCmp);
}

/* Search for keyToSearchFor in the range tree. Will return NULL for an empty range tree,
   a pointer to the node if the node exists in the tree or a pointer to the last
   node in the search path otherwise.                                          */
static NodeP subTreeSearch(ConstRangeTreeP tree, ConstElement keyToSearchFor)
{
    if (tree == NULL || keyToSearchFor == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
```

```
332     return subTreeSearchRec(tree->_root, keyToSearchFor, tree->lmCmp);
333 }
334 
335 /**
336  * add new element to tree
337  * parameters:
338  *     RangeTreeP tree
339  *  ConstElement keyToSearchFor - new element
340  */
341 static void addElement(RangeTreeP tree, ConstElement keyToSearchFor)
342 {
343     int direct;
344     NodeP parent;
345     debugStableCheck(tree);
346     if (tree == NULL || keyToSearchFor == NULL)
347     {
348         ERROR_MESSAGE(NULL_INPUT);
349     }
350     parent = subTreeSearch(tree, keyToSearchFor);
351     if (parent == NULL)
352     {
353         /* An empty tree - the new node will be the root (special case) */
354         NodeP newRoot = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, NULL);
355         assert(tree->_size == 0);
356         assert(newRoot != NULL);
357         setRoot(tree, newRoot, TRUE);
358         tree->_maxNode = newRoot;
359         ++tree->_size;
360         return;
361     }
362     direct = (*tree->lmCmp)(getNodeKey(parent), keyToSearchFor);
363     if (direct == 0)
364     {
365         /* The element is already in the tree */
366         ERROR_MESSAGE(ELEMENT_ADD_TWICE);
367     }
368     ++tree->_size;
369     if (direct > 0)
370     {
371         NodeP newNode = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, parent);
372         assert(newNode != NULL);
373         setChild(parent, LEFT, newNode);
374     }
375     else
376     {
377         NodeP newNode = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, parent);
378         assert(newNode != NULL);
379         if ((*tree->lmCmp)(getNodeKey(tree->_maxNode), getNodeKey(newNode)) < 0)
380         {
381             tree->_maxNode = newNode;
382         }
383         setChild(parent, RIGHT, newNode);
384     }
385 }
386 
387 /*
388  * Initializes the random number seed.
389  *
390  * The seed is initialized from the environment variable SRAND_SEED, or,
391  * if SRAND_SEED is undefined, uses the system time as the seed.
392  */
393 static void initializeSeed()
394 {
395     char *seedStr = getenv("SRAND_SEED");
396     unsigned int seed;
397 
398     if (seedStr != NULL)
399     {
```

```
400         /* read seed from the environment variable and convert to an integer */
401         seed = atoi(seedStr);
402     }
403     else
404     {
405         /* use the system time as a seed. it changes every second and never repeats. */
406         seed = time(NULL);
407     }
408 
409     srand(seed);
410 }
411 
412 /*
413  * Returns a random integer from the range [low,high].
414  */
415 static int chooseRandomNumber(int low, int high)
416 {
417     /* In Numerical Recipes in C: The Art of Scientific Computing
418        (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New  York:  Cambridge
419        University Press, 1992 (2nd ed., p. 277)), the following comments are made:
420               "If you want to generate a random integer between 1 and 10, you should always do it
421                by using high-order bits, as in
422 
423                       j = 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));
424 
425        (cited by rand(3) man page) */
426     int num = low + (int) ( ((double)(high - low + 1)) * (rand() / (RAND_MAX + 1.0)));
427 
428     return num;
429 }
430 
431 
432 /* Get the inserted order entered by the user and "mix" the array to create a "random" insertion order.
433    There exists better algorithm for randomness, but the following algorithm is good enough
434    for our purpose. */
435 static void generateRandomPermutation(Element participateWorkers[], int arrsize)
436 {
437     int it;
438     if (arrsize < 2)
439     {
440         return;
441     }
442     initializeSeed();
443     for (it = 0 ; it < arrsize ; ++it)
444     {
445         Element tempWork;
446         int f1 = chooseRandomNumber(0, arrsize-1);
447         int f2 = chooseRandomNumber(0, arrsize-1);
448         if (f1 == f2)
449         {
450             continue;
451         }
452         tempWork = participateWorkers[f1];
453         participateWorkers[f1] = participateWorkers[f2];
454         participateWorkers[f2] = tempWork;
455     }
456 }
457 
458 /**
459  * create a new range tree, Returns a pointer to it.
460  * The nodes of the tree will contain the participateWorkers workers from the array.
461  * In addition, receive 4 pointers to functions:
462  * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
463  *   if the items are equal or positive number if the first element is larger than the second element.
464  * - cpy - duplicate an element. Return NULL in case of memory out.
465  * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
466  *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
467  * - fre - a function that free the memory allocate for the element.
```

```
468  * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
469  * from it.
470  * Same error handling as in the SimpleRangeTree.c file.
471  */
472 RangeTreeP createNewRangeTree(Element participateWorkers[], int arrsize,
473                               int cmp(ConstElement, ConstElement), Element cpy(ConstElement),
474                               char *lbl(ConstElement), void fre(Element))
475 {
476     if (cpy == NULL)
477     {
478         ERROR_MESSAGE(WORNG_COPY_FUNC);
479     }
480     if (cmp == NULL)
481     {
482         ERROR_MESSAGE(WORNG_CMP_FUNC);
483     }
484     if (lbl == NULL)
485     {
486         ERROR_MESSAGE(WORNG_LBL_FUNC);
487     }
488     if (fre == NULL)
489     {
490         ERROR_MESSAGE(WORNG_FREE_FUNC);
491     }
492
493     int it;
494     RangeTreeP retVal = (RangeTreeP) malloc(sizeof(struct GenRangeTreeRec));
495     if (retVal == NULL)
496     {
497         ERROR_MESSAGE(OUT_OF_MEMORY);
498     }
499     generateRandomPermutation(participateWorkers, arrsize);
500     retVal->lmCmp = cmp;
501     retVal->lmCpy = cpy;
502     retVal->lmFre = fre;
503     retVal->lmLbl = lbl;
504     retVal->_root = NULL;
505     retVal->_maxNode = NULL;
506     retVal->_size = 0;
507     for (it = 0 ; it < arrsize ; ++it)
508     {
509         addElement(retVal, participateWorkers[it]);
510     }
511     return retVal;
512 }
513
514 /**
515  * helper function to clearTree
516  */
517 static void freeNodeRec(NodeP node, void (*lmFre)(Element))
518 {
519     assert(lmFre != NULL);
520     if (node == NULL)
521     {
522         return;
523     }
524     freeNodeRec(getChildren(node, RIGHT), lmFre);
525     freeNodeRec(getChildren(node, LEFT), lmFre);
526     freeNode(node, lmFre);
527 }
528
529 /**
530  * Call this function if you want to clear all the elements in the node.
531 */
532 static void clearTree(RangeTreeP tree)
533 {
534     if (tree == NULL)
535     {
```

```
536             ERROR_MESSAGE(NULL_INPUT);
537         }
538         freeNodeRec(getRoot(tree), tree->lmFre);
539         tree->_size = 0;
540         tree->_maxNode = NULL;
541     }
542 
543     /**
544      * Call this function when you don't want to use the tree anymore (a moment before you exit the program)
545      */
546     void destroyRangeTree(RangeTreeP tree)
547     {
548         if (tree == NULL)
549         {
550             ERROR_MESSAGE(NULL_INPUT);
551         }
552         clearTree(tree);
553         free(tree);
554     }
555 
556     /**
557      * Retrun the size of the tree
558      * Report error in case the pointer is NULL
559      */
560     int size(ConstRangeTreeP tree)
561     {
562         if (tree == NULL)
563         {
564             ERROR_MESSAGE(NULL_INPUT);
565         }
566         return tree->_size;
567     }
568 
569     /**
570      * @return sub-tree minimum
571      * parameter:
572      *     NodeP n - sub-tree root
573      */
574     static NodeP getMinimum(NodeP n)
575     {
576         while(1)
577         {
578             NodeP tempN;
579             assert(n != NULL);
580             tempN = getChildren(n, LEFT);
581             if (tempN == NULL)
582             {
583                 return n;
584             }
585             n = tempN;
586         }
587         return 0;
588     }
589 
590     /* Return the successor of the node 'n' in the range tree, or NULL if 'n' is already the maximum */
591     static NodeP successor(NodeP n, NodeP maximumNode)
592     {
593         NodeP tempN;
594         assert(n != NULL);
595 
596         /* Check if 'n' is the maximum */
597         if (n == maximumNode)
598         {
599             return NULL;
600         }
601 
602         /* if 'n' has a right child go visit its minimum */
603         tempN = getChildren(n, RIGHT);
```

```
604     if (tempN != NULL)
605     {
606         return getMinimum(tempN);
607     }
608 
609     /* Get 'n' node first father such that 'n' it's his left son */
610     while (1)
611     {
612         NodeP oldN = n;
613         n = getParent(n);
614         assert(n != NULL);
615         if (LEFT == whichChild(n, oldN))
616         {
617             break;
618         }
619     }
620 
621     return n;
622 }
623 
624 /* Search the tree, find the node that contains the worker with the smallest
625    paycheck that is bigger than p1 paycheck */
626 static NodeP findMinAboveWorker(ConstRangeTreeP tree, ConstElement p)
627 {
628     NodeP retVal = NULL;
629     NodeP curr;
630     assert(tree != NULL);
631     assert(p != NULL);
632     curr = getRoot(tree);
633     while (curr != NULL)
634     {
635         ConstElement currElement = getNodeKey(curr);
636         if (tree->lmCmp(currElement, p) >= 0)
637         {
638             if (retVal == NULL)
639             {
640                 retVal = curr;
641             }
642             if ((*tree->lmCmp)(getNodeKey(curr), getNodeKey(retVal)) < 0)
643             {
644                 retVal = curr;
645             }
646             curr = getChildren(curr, LEFT);
647         }
648         else
649         {
650             curr = getChildren(curr, RIGHT);
651         }
652     }
653     return retVal;
654 }
655 
656 /**
657  * Print all the nodes in the given range in the tree
658  * Report NULL_INPUT in case of a NULL pointer
659  */
660 void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2)
661 {
662     NodeP opt;
663     char* detail;
664     if (tree == NULL || p1 == NULL || p2 == NULL)
665     {
666         ERROR_MESSAGE(NULL_INPUT);
667     }
668     if (tree->lmCmp(p1, p2) > 0)
669     {
670         ERROR_MESSAGE(BAD_RANGE);
671     }
```

```
672     opt = findMinAboveWorker(tree, p1);
673     if (opt == NULL)
674     {
675         return;
676     }
677
678     while ((*tree->lmCmp)(getNodeKey(opt), p2) <= 0)
679     {
680         detail = (*tree->lmLbl)(getNodeKey(opt));
681         printf("%s\n", detail);
682         free(detail);
683         opt = successor(opt, tree->_maxNode);
684         if (opt == NULL)
685         {
686             return;
687         }
688     }
689 }
690
691 /**
692  * Used for debbugin
693  * Verify that the Node is leagal (as a node in a binary search tree)
694  * Then verify all it successors recursively
695  */
696 static void debugCheckNode(NodeP n, ConstRangeTreeP tree)
697 {
698     if (n == NULL)
699     {
700         return;
701     }
702     if (n->_left != NULL)
703     {
704         assert(n->_left->_parent == n);
705         assert((*tree->lmCmp)(getNodeKey(n->_left), getNodeKey(n)) < 0);
706         debugCheckNode(n->_left, tree);
707     }
708     if (n->_right != NULL)
709     {
710         assert(n->_right->_parent == n);
711         assert((*tree->lmCmp)(getNodeKey(n->_right), getNodeKey(n)) > 0);
712         debugCheckNode(n->_right, tree);
713     }
714 }
715
716 /**
717  * Used for debugging
718  * Verify that the tree is legal
719  */
720 void debugStableCheck(ConstRangeTreeP tree)
721 {
722     assert(tree != NULL);
723     debugCheckNode(tree->_root, tree);
724     assert(tree->_root == NULL || (tree->_maxNode != NULL && tree->_maxNode->_right == NULL));
725 }
726
727 #ifndef NDEBUG
728
729 #define NUM_PAR 10
730
731 /**
732  * Compare Manager Salaries
733  */
734 int managerCmpSalary(ConstElement c1, ConstElement c2)
735 {
736     ManagerP m1 = (ManagerP) c1;
737     ManagerP m2 = (ManagerP) c2;
738     assert(m1 != NULL && m2 != NULL);
739     return compareManagers(m1, m2);
```

missing_check_if_null{lmLbl might return NULL}

```
740 }
741
742 /**
743  * Compare Manager Attractivity
744  */
745 int managerCmpAttract(ConstElement c1, ConstElement c2)
746 {
747     ManagerP m1 = (ManagerP) c1;
748     ManagerP m2 = (ManagerP) c2;
749     assert(m1 != NULL && m2 != NULL);
750     return compareManagersAttract(m1, m2);
751 }
752
753 /**
754  *  Copy Manager function
755  */
756 Element cpyManager(ConstElement c)
757 {
758     ConstManagerP m = (ConstManagerP) c;
759     ManagerP mc = copyManager(m);
760     return ((Element) mc);
761 }
762
763
764 /**
765  * lbl Manager function
766  */
767 char *lblManager(ConstElement c)
768 {
769     ConstManagerP m = (ConstManagerP) c;
770     assert(m != NULL);
771     return getManagerInfo(m);
772 }
773
774 /**
775  * Free Manager function
776 */
777 void freManager(Element c)
778 {
779     ManagerP m = (ManagerP) c;
780     freeManager(m);
781 }
782
783 void printError(char *(*lmLbl)(ConstElement), ConstElement key1, ConstElement ke2)
784 {
785     char* detailKey1 = (lmLbl)(key1);
786     char* detailKey2 = (lmLbl)(ke2);
787     printf("current output: %s\n", detailKey1);
788     printf("expected output: %s\n", detailKey2);
789     free(detailKey1);
790     free(detailKey2);
791 }
792
793 int main()
794 {
795     printf("Start testing:\n");
796
797     RangeTreeP rt1, rt2;
798
799     //create empty tree and create nodes manually
800     ManagerP man1 = getManager(00, "avi", 100, 1., 10);
801     ManagerP man2 = getManager(01, "beni", 200, 1.1, 20);
802     ManagerP man3 = getManager(02, "gabi", 300, 1.2, 30);
803     NodeP node1 = getNewNode(&cpyManager, (ConstElement)man1, NULL, NULL, NULL);
804     NodeP node2 = getNewNode(&cpyManager, (ConstElement)man2, NULL, NULL, NULL);
805     NodeP node3 = getNewNode(&cpyManager, (ConstElement)man3, NULL, NULL, NULL);
806     rt1 = createNewRangeTree(NULL, 0, &managerCmpSalary, &cpyManager,
807                                  &lblManager, &freManager);
```

```
808
809     //create tree with managers
810     int it;
811     ManagerP mana[NUM_PAR];
812     Element tempArr[NUM_PAR];
813
814     mana[0] = getManager(10, "Kaz", 1150, 1., 700);
815     mana[1] = getManager(11, "Levi", 1050, 1.1, 650);
816     mana[2] = getManager(12, "Mor", 2657, 1.5, 2000);
817     mana[3] = getManager(13, "Netanel", 677, 2.3, 350);
818     mana[4] = getManager(14, "Orit", 1399, 9., 786);
819     mana[5] = getManager(15, "PLAB", 1900, 8.1, 1453);
820     mana[6] = getManager(16, "Sitvanit", 890, 1.8, 389);
821     mana[7] = getManager(17, "UV", 1555, 2.6, 1197);
822     mana[8] = getManager(18, "Vera", 1466, 5.5, 1155);
823     mana[9] = getManager(19, "WallE", 999, 3.3, 600);
824
825     for (it = 0 ; it < NUM_PAR ; ++it)
826     {
827         assert(mana[it] != NULL);
828         tempArr[it] = (Element) mana[it];
829     }
830
831     rt2 = createNewRangeTree(tempArr, NUM_PAR, &managerCmpSalary, &cpyManager,
832                              &lblManager, &freManager);
833
834     //setting root with children
835     setRoot(rt1, node1, TRUE);
836     setChild(node1, LEFT, node2);
837     setChild(node1, RIGHT, node3);
838
839     //test the setting
840     printf("check the root:\n");
841     if (rt1->_root == node1)
842     {
843         printf("correct!\n");
844     }
845     else
846     {
847         printf("fail function: 'setRoot'\n");
848         printf("parameters: 'RangeTreeP tree, NodeP node, Boolean safe'\n");
849         printError(*rt2->lmLbl, getNodeKey(rt1->_root), node1->_key);
850     }
851
852     printf("check left child:\n");
853     if (node1->_left == node2)
854     {
855         printf("correct!\n");
856     }
857     else
858     {
859         printf("fail function: 'setChild'\n");
860         printf("parameters: 'NodeP node, Side side, NodeP child'\n");
861         printError(*rt2->lmLbl, getNodeKey(node1->_left), node2->_key);
862     }
863
864     printf("check right child:\n");
865     if (node1->_right == node3)
866     {
867         printf("correct!\n");
868     }
869     else
870     {
871         printf("fail function: 'setChild'\n");
872         printf("parameters: 'NodeP node, Side side, NodeP child'\n");
873         printError(*rt2->lmLbl, getNodeKey(node1->_right), node3->_key);
874     }
875
```

```
876         //test second tree
877         NodeP treeRoot = rt2->_root;
878         NodeP minTree = getMinimum(treeRoot);
879         printf("check get minimum:\n");
880         if ((*rt2->lmCmp)(minTree->_key, (Element)mana[3]) == 0)
881         {
882             printf("correct!\n");
883         }
884         else
885         {
886             printf("fail function: 'getMinimum'\n");
887             printf("Parameters: 'NodeP n'\n");
888             printError(*rt2->lmLbl, getNodeKey(minTree), (Element)mana[3]);
889         }
890 
891         NodeP minSucc = successor(minTree, treeRoot);
892         printf("check minimum successor:\n");
893         if ((*rt2->lmCmp)(minSucc->_key, (Element)mana[6]) == 0)
894         {
895             printf("correct!\n");
896         }
897         else
898         {
899             printf("fail function: 'successor'\n");
900             printf("Parameters: 'NodeP n, NodeP maximumNode'\n");
901             printError(*rt2->lmLbl, getNodeKey(minSucc), (Element)mana[6]);
902         }
903 
904         (*rt1->lmFre)(man1);
905         (*rt1->lmFre)(man2);
906         (*rt1->lmFre)(man3);
907 
908         for (it = 0 ; it < NUM_PAR ; ++it)
909         {
910             (*rt2->lmFre)(mana[it]);
911         }
912         destroyRangeTree(rt1);
913         destroyRangeTree(rt2);
914 
915         printf("test end!\n");
916         return 0;
917 
918     }
919 
920     #endif
```

4 Makefile

```
cc = gcc -Wall -std=c99 -g

#targets
generalSwap: generalSwap.o
    $(cc) generalSwap.o -o ./generalSwap

generalSwap.o: generalSwap.c
    $(cc) -c ./generalSwap.c

GenRangeTree.o: GenRangeTree.c GenRangeTree.h Manager.h
    $(cc) -c -DNDEBUG ./GenRangeTree.c

GenRangeTree: GenRangeTree.o
    ar rcs libGenRangeTree.a ./GenRangeTree.o

tests: GenRangeTree.c
    $(cc) -c ./GenRangeTree.c -o ./GenRangeTest.o
    $(cc) ./GenRangeTest.o ./Manager.o -o ./GenRangeTree
    ./GenRangeTree

clean:
    rm -f generalSwap GenRangeTree GenRangeTree.o GenRangeTest.o generalSwap.o libGenRangeTree.a

tar: generalSwap.c GenRangeTree.c GenRangeTree.h Makefile valdbg.out
    tar cfv ex3.tar generalSwap.c GenRangeTree.c GenRangeTree.h Makefile valdbg.out


.PHONY: generalSwap GenRangeTree tests tar clean
```

5 generalSwap.c

```c
/*
 * generalSwap.c
 *
 *  Created on: Aug 11, 2014
 *      Author: roigreenberg
 */

/**
 * 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("c=%d d=%d\n", c, d); /* should print c=55 d=44 */
    return 0;
}

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

q1aMC

6 valdbg.out

```
1  ==21192== Memcheck, a memory error detector
2  ==21192== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
3  ==21192== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
4  ==21192== Command: ./GenRangeTree
5  ==21192== Parent PID: 20979
6  ==21192==
7  ==21192==
8  ==21192== HEAP SUMMARY:
9  ==21192==     in use at exit: 0 bytes in 0 blocks
10 ==21192==   total heap usage: 41 allocs, 41 frees, 1,984 bytes allocated
11 ==21192==
12 ==21192== All heap blocks were freed -- no leaks are possible
13 ==21192==
14 ==21192== For counts of detected and suppressed errors, rerun with: -v
15 ==21192== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```