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

```
/**
 * A general binary search tree where the nodes are sorted according to their keys
 * Each node have the functions for comparing, copy, printing and free
 */
#ifndef GEN_RANGE_TREE_H
#define GEN_RANGE_TREE_H

typedef void* Element;
typedef const void* ConstElement;
typedef enum
{
    FALSE,
    TRUE
} Boolean;

/* Pointer at a range tree */
typedef struct GenRangeTreeRec *RangeTreeP;
typedef const struct GenRangeTreeRec *ConstRangeTreeP;

/**
 * create a new range tree, Returns a pointer to it.
 * The nodes of the tree will contain the participateWorkers workers from the array.
 * In addition, receive 4 pointers to functions:
 * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
 *   if the items are equal or positive number if the first element is larger than the second element.
 * - cpy - duplicate an element. Return NULL in case of memory out.
 * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
 *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
 * - fre - a function that free the memory allocate for the element.
 * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
 * from it.
 * Same error handling as in the SimpleRangeTree.c file.
*/
RangeTreeP createNewRangeTree(Element participateWorkers[], int arrsize,
                              int cmp(ConstElement, ConstElement),
                              Element cpy(ConstElement),
                              char *lbl(ConstElement),
                              void fre(Element));

/**
 * Free the range tree from the memory (should be called when the user doesn't need the range tree anymore).
 */
void destroyRangeTree(RangeTreeP tree);

/**
 * Return the number of workers in the range tree.
 */
int size(ConstRangeTreeP tree);

/**
 * Print the tree according to a range quory - print all the workers that
 * are paid at least as p1, and at most as p2.
 */
void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2);

/**
 * Debbuging function - you don't have to use it but you may find it helpful.
 */
void debugStableCheck(ConstRangeTreeP tree);
```

```

#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");
    }
    else if (theErr == BAD_RANGE)
    {
        fprintf(stderr, "Bad input range for printRange: p1 is bigger than p2!!!\n");
    }
    else if (theErr == WORNG_COPY_FUNC)
    {
        fprintf(stderr, "Bad input for new tree: wrong copy function!\n");
    }
    else if (theErr == WORNG_CMP_FUNC)
    {
        fprintf(stderr, "Bad input for new tree: wrong compare function!\n");
    }
    else if (theErr == WORNG_LBL_FUNC)
    {
        fprintf(stderr, "Bad input for new tree: wrong label function!\n");
    }
    else if (theErr == WORNG_FREE_FUNC)
    {
        fprintf(stderr, "Bad input for new tree: wrong free function!\n");
    }
    else
    {
        fprintf(stderr, "General error.\n");
    }
    exit(1);
}

#define ERROR_MESSAGE(x) reporterrorMessage(x, __LINE__)

/**
 * A node in the tree contains a pointer to the two sons, to the parent an to the key
 */
struct Node
{
    NodeP _left;
    NodeP _right;
    NodeP _parent;
    Element _key;    // Points to data
};

/**
 * create new node
 * parameters:
 *     Element (*lmCpy)(ConstElement) - copy function
 *     ConstElement key
 *     NodeP left
 *     NodeP right
 *     NodeP parent
 * @return - the new node
 */
static NodeP getNewNode(Element (*lmCpy)(ConstElement), ConstElement key, NodeP left, NodeP right,
                        NodeP parent)
{
    assert(lmCpy != NULL);
    NodeP retVal = (NodeP) malloc(sizeof(struct Node));
    if (retVal == NULL)
    {
        ERROR_MESSAGE(OUT_OF_MEMORY);
    }
    if (key == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    retVal->_left = left;
    retVal->_right = right;
    retVal->_parent = parent;
    retVal->_key = (*lmCpy)(key);
```

```c
    assert(retVal->_key != NULL);
    return retVal;
}

/**
 * free the node memory
 * NodeP node
 * void (*lmFre)(Element) - free function
 */
static void freeNode(NodeP node, void (*lmFre)(Element))
{
    assert(lmFre != NULL);
    if (node == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    (*lmFre)(node->_key);
    free(node);
}

/**
 * @return node child
 * parameters:
 *     ConstNodeP node
 *     Side side - the child side
 */
static NodeP getChildren(ConstNodeP node, Side side)
{
    if (node == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    return (side == LEFT) ? node->_left : node->_right;
}

/**
 * @return node parent
 * parameters:
 *     NodeP node - child node
 */
static NodeP getParent(NodeP node)
{
    if (node == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    return node->_parent;
}

/**
 * @return node key
 * parameters:
 *     NodeP node
 */
static Element getNodeKey(NodeP node)
{
    return node->_key;
}

/**
 * set up a new child
 * parameters:
 *     NodeP node - parent node
 *     Side side - child side
 *     NodeP child - child node
 */
static void setChild(NodeP node, Side side, NodeP child)
{
    if (node == NULL || child == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    if (side == LEFT)
    {
        assert(node->_left == NULL);
        node->_left = child;
    }
    else
    {
        assert(side == RIGHT);
        assert(node->_right == NULL);
        node->_right = child;
    }
}

/**
 * @return child side
 * parameters:
 *     ConstNodeP node - parent node
 *  ConstNodeP child
 */
static Side whichChild(ConstNodeP node, ConstNodeP child)
{
    if (node == NULL || child == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    if (node->_right == child)
    {
        return RIGHT;
    }
    assert(node->_left == child);
    return LEFT;
}


/*********************************************************************************
 The range tree definitions and functions
 *********************************************************************************/

/**
 * A struct that contains the tree of Workers.
 * Including the root, the maximal node and the number of leafs in the tree
 */
struct GenRangeTreeRec
{

    int (*lmCmp)(ConstElement, ConstElement);
    Element (*lmCpy)(ConstElement);
    char *(*lmLbl)(ConstElement);
    void (*lmFre)(Element);

    /* The tree root, contains NULL for an empty tree */
    NodeP _root;

    /* A pointer to the node with the maximum value in the tree (usefull for the successor function).
       We have to update this field in the Add/Remove element fuctions. */
    NodeP _maxNode;

    /* Number of nodes in the tree */
    int _size;
};

/**
 * @return tree root
 * 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);
    }
```

```c
    return subTreeSearchRec(tree->_root, keyToSearchFor, tree->lmCmp);
}

/**
 * add new element to tree
 * parameters:
 *     RangeTreeP tree
 *  ConstElement keyToSearchFor - new element
 */
static void addElement(RangeTreeP tree, ConstElement keyToSearchFor)
{
    int direct;
    NodeP parent;
    debugStableCheck(tree);
    if (tree == NULL || keyToSearchFor == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    parent = subTreeSearch(tree, keyToSearchFor);
    if (parent == NULL)
    {
        /* An empty tree - the new node will be the root (special case) */
        NodeP newRoot = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, NULL);
        assert(tree->_size == 0);
        assert(newRoot != NULL);
        setRoot(tree, newRoot, TRUE);
        tree->_maxNode = newRoot;
        ++tree->_size;
        return;
    }
    direct = (*tree->lmCmp)(getNodeKey(parent), keyToSearchFor);
    if (direct == 0)
    {
        /* The element is already in the tree */
        ERROR_MESSAGE(ELEMENT_ADD_TWICE);
    }
    ++tree->_size;
    if (direct > 0)
    {
        NodeP newNode = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, parent);
        assert(newNode != NULL);
        setChild(parent, LEFT, newNode);
    }
    else
    {
        NodeP newNode = getNewNode(tree->lmCpy, keyToSearchFor, NULL, NULL, parent);
        assert(newNode != NULL);
        if ((*tree->lmCmp)(getNodeKey(tree->_maxNode), getNodeKey(newNode)) < 0)
        {
            tree->_maxNode = newNode;
        }
        setChild(parent, RIGHT, newNode);
    }
}

/*
 * Initializes the random number seed.
 *
 * The seed is initialized from the environment variable SRAND_SEED, or,
 * if SRAND_SEED is undefined, uses the system time as the seed.
 */
static void initializeSeed()
{
    char *seedStr = getenv("SRAND_SEED");
    unsigned int seed;

    if (seedStr != NULL)
    {
        /* read seed from the environment variable and convert to an integer */
        seed = atoi(seedStr);
    }
    else
    {
        /* use the system time as a seed. it changes every second and never repeats. */
        seed = time(NULL);
    }

    srand(seed);
}

/*
 * Returns a random integer from the range [low,high].
 */
static int chooseRandomNumber(int low, int high)
{
    /* In Numerical Recipes in C: The Art of Scientific Computing
       (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New  York:  Cambridge
       University Press, 1992 (2nd ed., p. 277)), the following comments are made:
              "If you want to generate a random integer between 1 and 10, you should always do it
               by using high-order bits, as in

                     j = 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));

       (cited by rand(3) man page) */
    int num = low + (int) ( ((double)(high - low + 1)) * (rand() / (RAND_MAX + 1.0)));

    return num;
}


/* Get the inserted order entered by the user and "mix" the array to create a "random" insertion order.
   There exists better algorithm for randomness, but the following algorithm is good enough
   for our purpose. */
static void generateRandomPermutation(Element participateWorkers[], int arrsize)
{
    int it;
    if (arrsize < 2)
    {
        return;
    }
    initializeSeed();
    for (it = 0 ; it < arrsize ; ++it)
    {
        Element tempWork;
        int f1 = chooseRandomNumber(0, arrsize-1);
        int f2 = chooseRandomNumber(0, arrsize-1);
        if (f1 == f2)
        {
            continue;
        }
        tempWork = participateWorkers[f1];
        participateWorkers[f1] = participateWorkers[f2];
        participateWorkers[f2] = tempWork;
    }
}

/**
 * create a new range tree, Returns a pointer to it.
 * The nodes of the tree will contain the participateWorkers workers from the array.
 * In addition, receive 4 pointers to functions:
 * - cmp - compare between two elements, return negative number if the first is smaller than the second, zero
 *   if the items are equal or positive number if the first element is larger than the second element.
 * - cpy - duplicate an element. Return NULL in case of memory out.
 * - lbl - turn an element into a string (so we can print it). Allocate memory for the string - it's our
 *   responsibility to free the memory after using the string. In case of out-of-memory event, return NULL.
 * - fre - a function that free the memory allocate for the element.
 * Note that the tree is a static tree - once the tree was created, we can't add / remove elements
 * from it.
 * Same error handling as in the SimpleRangeTree.c file.
*/
RangeTreeP createNewRangeTree(Element participateWorkers[], int arrsize,
                              int cmp(ConstElement, ConstElement), Element cpy(ConstElement),
                              char *lbl(ConstElement), void fre(Element))
{
    if (cpy == NULL)
    {
        ERROR_MESSAGE(WORNG_COPY_FUNC);
    }
    if (cmp == NULL)
    {
        ERROR_MESSAGE(WORNG_CMP_FUNC);
    }
    if (lbl == NULL)
    {
        ERROR_MESSAGE(WORNG_LBL_FUNC);
    }
    if (fre == NULL)
    {
        ERROR_MESSAGE(WORNG_FREE_FUNC);
    }

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

/**
 * helper function to clearTree
 */
static void freeNodeRec(NodeP node, void (*lmFre)(Element))
{
    assert(lmFre != NULL);
    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;
}

/**
 * Call this function when you don’t want to use the tree anymore (a moment before you exit the program)
 */
void destroyRangeTree(RangeTreeP tree)
{
    if (tree == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    clearTree(tree);
    free(tree);
}

/**
 * Retrun the size of the tree
 * Report error in case the pointer is NULL
 */
int size(ConstRangeTreeP tree)
{
    if (tree == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    return tree->_size;
}

/**
 * @return sub-tree minimum
 * parameter:
 *     NodeP n - sub-tree root
 */
static NodeP getMinimum(NodeP n)
{
    while(1)
    {
        NodeP tempN;
        assert(n != NULL);
        tempN = getChildren(n, LEFT);
        if (tempN == NULL)
        {
            return n;
        }
        n = tempN;
    }
    return 0;
}

/* Return the successor of the node ’n’ in the range tree, or NULL if ’n’ is already the maximum */
static NodeP successor(NodeP n, NodeP maximumNode)
{
    NodeP tempN;
    assert(n != NULL);

    /* Check if ’n’ is the maximum */
    if (n == maximumNode)
    {
        return NULL;
    }

    /* if ’n’ has a right child go visit its minimum */
    tempN = getChildren(n, RIGHT);
    if (tempN != NULL)
    {
        return getMinimum(tempN);
    }

    /* Get ’n’ node first father such that ’n’ it’s his left son */
    while (1)
    {
        NodeP oldN = n;
        n = getParent(n);
        assert(n != NULL);
        if (LEFT == whichChild(n, oldN))
        {
            break;
        }
    }

    return n;
}

/* Search the tree, find the node that contains the worker with the smallest
   paycheck that is bigger than p1 paycheck */
static NodeP findMinAboveWorker(ConstRangeTreeP tree, ConstElement p)
{
    NodeP retVal = NULL;
    NodeP curr;
    assert(tree != NULL);
    assert(p != NULL);
    curr = getRoot(tree);
    while (curr != NULL)
    {
        ConstElement currElement = getNodeKey(curr);
        if (tree->lmCmp(currElement, p) >= 0)
        {
            if (retVal == NULL)
            {
                retVal = curr;
            }
            if ((*tree->lmCmp)(getNodeKey(curr), getNodeKey(retVal)) < 0)
            {
                retVal = curr;
            }
            curr = getChildren(curr, LEFT);
        }
        else
        {
            curr = getChildren(curr, RIGHT);
        }
    }
    return retVal;
}

/**
 * Print all the nodes in the given range in the tree
 * Report NULL_INPUT in case of a NULL pointer
 */
void printRange(ConstRangeTreeP tree, ConstElement p1, ConstElement p2)
{
    NodeP opt;
    char* detail;
    if (tree == NULL || p1 == NULL || p2 == NULL)
    {
        ERROR_MESSAGE(NULL_INPUT);
    }
    if (tree->lmCmp(p1, p2) > 0)
    {
        ERROR_MESSAGE(BAD_RANGE);
    }
```

```c
    opt = findMinAboveWorker(tree, p1);
    if (opt == NULL)
    {
        return;
    }

    while ((*tree->lmCmp)(getNodeKey(opt), p2) <= 0)
    {
        detail = (*tree->lmLbl)(getNodeKey(opt));
        printf("%s\n", detail);
        free(detail);
        opt = successor(opt, tree->_maxNode);
        if (opt == NULL)
        {
            return;
        }
    }
}
```

missing_check_if_null{lmLbl might return NULL}

```c
/**
 * Used for debbugin
 * Verify that the Node is leagal (as a node in a binary search tree)
 * Then verify all it successors recursively
 */
static void debugCheckNode(NodeP n, ConstRangeTreeP tree)
{
    if (n == NULL)
    {
        return;
    }
    if (n->_left != NULL)
    {
        assert(n->_left->_parent == n);
        assert((*tree->lmCmp)(getNodeKey(n->_left), getNodeKey(n)) < 0);
        debugCheckNode(n->_left, tree);
    }
    if (n->_right != NULL)
    {
        assert(n->_right->_parent == n);
        assert((*tree->lmCmp)(getNodeKey(n->_right), getNodeKey(n)) > 0);
        debugCheckNode(n->_right, tree);
    }
}

/**
 * Used for debugging
 * Verify that the tree is legal
 */
void debugStableCheck(ConstRangeTreeP tree)
{
    assert(tree != NULL);
    debugCheckNode(tree->_root, tree);
    assert(tree->_root == NULL || (tree->_maxNode != NULL && tree->_maxNode->_right == NULL));
}

#ifndef NDEBUG

#define NUM_PAR 10

/**
 * Compare Manager Salaries
 */
int managerCmpSalary(ConstElement c1, ConstElement c2)
{
    ManagerP m1 = (ManagerP) c1;
    ManagerP m2 = (ManagerP) c2;
    assert(m1 != NULL && m2 != NULL);
    return compareManagers(m1, m2);
```

```
}

/**
 * Compare Manager Attractivity
 */
int managerCmpAttract(ConstElement c1, ConstElement c2)
{
    ManagerP m1 = (ManagerP) c1;
    ManagerP m2 = (ManagerP) c2;
    assert(m1 != NULL && m2 != NULL);
    return compareManagersAttract(m1, m2);
}

/**
 *  Copy Manager function
 */
Element cpyManager(ConstElement c)
{
    ConstManagerP m = (ConstManagerP) c;
    ManagerP mc = copyManager(m);
    return ((Element) mc);
}


/**
 * lbl Manager function
 */
char *lblManager(ConstElement c)
{
    ConstManagerP m = (ConstManagerP) c;
    assert(m != NULL);
    return getManagerInfo(m);
}

/**
 * Free Manager function
*/
void freManager(Element c)
{
    ManagerP m = (ManagerP) c;
    freeManager(m);
}

void printError(char *(*lmLbl)(ConstElement), ConstElement key1, ConstElement ke2)
{
    char* detailKey1 = (lmLbl)(key1);
    char* detailKey2 = (lmLbl)(ke2);
    printf("current output: %s\n", detailKey1);
    printf("expected output: %s\n", detailKey2);
    free(detailKey1);
    free(detailKey2);
}

int main()
{
    printf("Start testing:\n");

    RangeTreeP rt1, rt2;

    //create empty tree and create nodes manually
    ManagerP man1 = getManager(00, "avi", 100, 1., 10);
    ManagerP man2 = getManager(01, "beni", 200, 1.1, 20);
    ManagerP man3 = getManager(02, "gabi", 300, 1.2, 30);
    NodeP node1 = getNewNode(&cpyManager, (ConstElement)man1, NULL, NULL, NULL);
    NodeP node2 = getNewNode(&cpyManager, (ConstElement)man2, NULL, NULL, NULL);
    NodeP node3 = getNewNode(&cpyManager, (ConstElement)man3, NULL, NULL, NULL);
    rt1 = createNewRangeTree(NULL, 0, &managerCmpSalary, &cpyManager,
                               &lblManager, &freManager);
    //create tree with managers
    int it;
    ManagerP mana[NUM_PAR];
    Element tempArr[NUM_PAR];

    mana[0] = getManager(10, "Kaz", 1150, 1., 700);
    mana[1] = getManager(11, "Levi", 1050, 1.1, 650);
    mana[2] = getManager(12, "Mor", 2657, 1.5, 2000);
    mana[3] = getManager(13, "Netanel", 677, 2.3, 350);
    mana[4] = getManager(14, "Orit", 1399, 9., 786);
    mana[5] = getManager(15, "PLAB", 1900, 8.1, 1453);
    mana[6] = getManager(16, "Sitvanit", 890, 1.8, 389);
    mana[7] = getManager(17, "UV", 1555, 2.6, 1197);
    mana[8] = getManager(18, "Vera", 1466, 5.5, 1155);
    mana[9] = getManager(19, "WallE", 999, 3.3, 600);

    for (it = 0 ; it < NUM_PAR ; ++it)
    {
        assert(mana[it] != NULL);
        tempArr[it] = (Element) mana[it];
    }

    rt2 = createNewRangeTree(tempArr, NUM_PAR, &managerCmpSalary, &cpyManager,
                            &lblManager, &freManager);

    //setting root with children
    setRoot(rt1, node1, TRUE);
    setChild(node1, LEFT, node2);
    setChild(node1, RIGHT, node3);

    //test the setting
    printf("check the root:\n");
    if (rt1->_root == node1)
    {
        printf("correct!\n");
    }
    else
    {
        printf("fail function: ’setRoot’\n");
        printf("parameters: ’RangeTreeP tree, NodeP node, Boolean safe’\n");
        printError(*rt2->lmLbl, getNodeKey(rt1->_root), node1->_key);
    }

    printf("check left child:\n");
    if (node1->_left == node2)
    {
        printf("correct!\n");
    }
    else
    {
        printf("fail function: ’setChild’\n");
        printf("parameters: ’NodeP node, Side side, NodeP child’\n");
        printError(*rt2->lmLbl, getNodeKey(node1->_left), node2->_key);
    }

    printf("check right child:\n");
    if (node1->_right == node3)
    {
        printf("correct!\n");
    }
    else
    {
        printf("fail function: ’setChild’\n");
        printf("parameters: ’NodeP node, Side side, NodeP child’\n");
        printError(*rt2->lmLbl, getNodeKey(node1->_right), node3->_key);
    }
    //test second tree
    NodeP treeRoot = rt2->_root;
    NodeP minTree = getMinimum(treeRoot);
    printf("check get minimum:\n");
    if ((*rt2->lmCmp)(minTree->_key, (Element)mana[3]) == 0)
    {
        printf("correct!\n");
    }
    else
    {
        printf("fail function: ’getMinimum’\n");
        printf("Parameters: ’NodeP n’\n");
        printError(*rt2->lmLbl, getNodeKey(minTree), (Element)mana[3]);
    }

    NodeP minSucc = successor(minTree, treeRoot);
    printf("check minimum successor:\n");
    if ((*rt2->lmCmp)(minSucc->_key, (Element)mana[6]) == 0)
    {
        printf("correct!\n");
    }
    else
    {
        printf("fail function: ’successor’\n");
        printf("Parameters: ’NodeP n, NodeP maximumNode’\n");
        printError(*rt2->lmLbl, getNodeKey(minSucc), (Element)mana[6]);
    }

    (*rt1->lmFre)(man1);
    (*rt1->lmFre)(man2);
    (*rt1->lmFre)(man3);

    for (it = 0 ; it < NUM_PAR ; ++it)
    {
        (*rt2->lmFre)(mana[it]);
    }
    destroyRangeTree(rt1);
    destroyRangeTree(rt2);

    printf("test end!\n");
    return 0;

}

#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

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