INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
RENNES

C language 2

Labs 8 and 9
S5 EII

2020-2021

Contents

LAB 8

Doubly linked list : a basic leak memory detector

Objectives : (*1h30*)
— implement a doubly linked list with sentinel nodes ;
— manipulate addresses and more particularly *void* pointers ;
— overload functions using macros ;
— use your doubly linked list implementation to implement a basic leak memory detector.

Requirement :
— read the course document on preprocessor directives ;
— read the course document on linked lists ;
— read the whole practical subject ;
— write the Makefile for the project before the lab.

8.1 Positioning of this practical work

Some tools can help you to improve your code, like PurifyPlus, which is a real-time analysis tool for application reliability and performance. For example, it can detect memory leaks, *ie.* if some dynamically allocated memory was not released and what variable is concerned. It can profile your code : how often are used each code line, what is time consuming, ... You will have to use such tools to prove the robustness and efficiency of the software you will deliver to a customer.

In this practical lesson on linked lists, you will adapt your list implementation to this specific issue : simulate a leak memory detector.

8.2 Work for this session

The work for this session is split in 3 steps :
— provide a sorted list implementation to handle the information of the allocated memory block
— study the code of the provided leak memory detector that only counts the allocated memory block ;
— modify the leak memory detector functionalities to handle the memory block information using the list you have coded.

8.2.A List of allocated blocks

The objective of this exercise is to maintain a list of allocated blocks. An allocated block will be represented by its memory address, its size and the file, function and line where it has been allocated. Strings will be allocated dynamically.

There are some changes to be performed on the list module you have implemented in the previous lab :

— it is a doubly linked list : each node is linked to the next one but also to the previous one ;

— this list will have two sentinel nodes, *ie.* one dummy node at the beginning of the list and one another at the end. It will help to handle easily insertion and removal of nodes in the list ;

— the type node element, which will be a *MemoryBlock* defined like this :

```
typedef struct {
    char * file;      /*!< the filename where the memory has been allocated*/
    char * function; /*!< the function name the memory has been allocated*/
    int line;         /*!< the number of the line where is the malloc call*/
    void * address;   /*!< the address of the allocated block*/
}MemoryBlock;
```

To handle this structure, two functions are provided :

```
static MemoryBlock * initMemoryBlock(MemoryBlock *b,const char *f,  const char
    * fn, int l, void * a);
static void freeMemoryBlock(MemoryBlock *b);
```

What does the *static* keyword associated to a function mean ?
What does the *const* keyword associated to a parameter in a function mean ?

It leads to this new list structure :

```
typedef struct NodeList{
    MemoryBlock b;               /*!< information about the allocated block*/
    struct NodeList * next;      /*!< the address of the next element in the
       list*/
    struct NodeList * previous; /*!< the address of the previous element in
       the list*/
}NodeList;

typedef struct{
  NodeList sentinel_begin; /*!< the beginning sentinel node*/
  NodeList * current;      /*!< the address of the current element in the
     list*/
  NodeList sentinel_end;   /*!< the sentinel node at the end*/
}List;
```

— the list which will be sorted : as a consequence, there will be only one function available to a user that enables to insert an item : *insertSort()*. It will insert an item at the right place in a list that is supposed to be sorted.

As a consequence of these specifications, this list should provide the following functionalities[1] :

```
void initList(List * l);

int isEmpty(List * l);
int isFirst(List * l);
int isLast(List * l);
int isOutOfList(List * l);

void setOnFirst(List * l);
void setOnLast(List * l);
void setOnNext (List * l);
void setOnPrevious (List * l);
void * getCurrentAddress (List * l);

void printList(List * l);
int countElement(List * l);

int insertSort(List * l, const char *f, const char * fn, int ln, void * a);

int find(List * l, void * a);
void * deleteValue(List *l, void * a);
```

To implement it properly, you will have to code also the following functions :

```
static NodeList *newNodeList(const char *f,  const char * fn, int l, void * a,
    NodeList *n, NodeList *p);
static void freeNodeList(NodeList *n);
```

Why are they static ?

The files "memoryList.h" and "memoryList.c" are provided. This latter contains the function to init and free the fields of a memoryBlock structure.

Create a Clion project with a Makefile with these two files. Implement all these functions in the "memoryList.c" file.

When inserting a new block in the list, you should insert such as the filenames are sorted. If filenames are identical by function name and if the function names are also identical by line of code.

Test your functions. It is highly recommended to use the test module from previous labs to design your tests.

8.2.B Basic leak memory detector

A basic leak memory detector is provided. Download the source code, write the associated Makefile. Test your Makefile and then create a Clion project from this Makefile.

The code provided in the "myAlloc.h/.c" files aims to count the number of dynamically allocated blocks when the program is running. This number is increased each time the *malloc()* function is called and decreased each time the *free()* function is called[2]. At the end of the program, it tests if every allocated block has been freed, otherwise a memory leak is detected.

Study the code to fully understand how the *malloc()* and *free()* functions are overloaded. Especially, answer the following questions :

— What are __FILE__, __FUNCTION__ and __LINE__ ?

1. doc is provided in *doc/html/index.html*
2. The *calloc()* and *realloc()* functions are not considered here to make it simpler but it should be for a correct allocation management

— What is the "#define MEMCHECK" in the "mainc.c" file for ?
— Why must MEMCHECK not be defined in "myAlloc.c" file ?

Once you have studied the code and tested it, you can note that if a memory leak is detected, it is not possible to fix it, for example with a garbage collector, since no information about the memory block is stored. In order to provide it, you will use the list you have implemented.

8.2.C Add functionalities to the leak memory detector using the list of allocated blocks

The new memory leak detector will manage a list of allocated blocks. This list will be declared as a global variable in the "myAlloc.c" file, for example by adding at the beginning of "myAlloc.c" :

```
  List l = {
             {{NULL,NULL,0,0,NULL},&(l.sentinel_end),NULL},   /* sentinel_begin*/
             &(l.sentinel_begin),                             /* current*/
             {{NULL,NULL,0,0,NULL},NULL,&(l.sentinel_begin)} /* sentinel_end*/
           };
```

This line enables to init the List as an empty list, so there is no need to call *initList()*.

Modify the "myAlloc.c" file to use your list functions (no direct change of the list members) in order to maintain a list of allocated blocks, *ie.* by replacing the block counter by the use of the block memory list :

— when a *malloc()* is performed, instead of increasing the counter, a node list should be added in this list, with the associated block data
— when a *free()* is performed, the corresponding node list should be deleted from the list
— checking if there is no memory leak amounts to checking if the list is empty.

The advantage of the new memory leak detector is that the allocated blocks are recorded. Therefore, if a memory leak is detected, it can be forced to be freed. You will add a function in the "myAlloc.c" file that frees every memory block registered in the list :

```
        void myGarbageCollector();
```

As a consequence, your *main()* function has to be now :

```
int main()
{
    char word [] = "bonjour";
    char * t = NULL;
    char * s = NULL;
    int check;

    t= (char * ) malloc(50);
    s= (char * ) malloc(50);

    mystrcpy(&t,word);
    mystrinv(&s,word);

    free(t);

#ifdef MEMCHECK
    printf("There are %d unfreed memory blocks \n", check=myCheck());
    if(check)
    {
        myGarbageCollector();
        printf("There are %d unfreed memory blocks \n", check=myCheck());
    }
#endif
    return 0;
}
```

8.2.D Pointer of function + ellipsis : generic code of *insertSort* (optionnal this special year)

Provide a function *insertSortGeneric* with an additional parameter being a pointer of function defining the way two elements are compared, :

```
int insertSortGeneric(List * l, const char *f, const char * fn, int ln, void *
    a,  int (*comp) (void *,void*) );
```

Test it.

LAB 9

Balanced trees : AVL trees

Objectives : (*1h30*)
- — manipulate binary trees ;
- — use recursive programming to implement an AVL tree.

Requirement :
- — read the course document on trees ;
- — read the whole practical subject ;
- — write the Makefile for the project before the practical session.

9.1 Positioning of this practical work

The studied structure is a sorted binary tree of double. Each node of the tree is associated to up to two child nodes : a left child, and a right child. Each child node can itself be the root of a subtree, called left and right subtrees, respectively. The sorted binary tree is sorted such as the value of a node is greater than the maximum value contained in its left subtree and smaller than the minimum value of its right subtree. The structure definition is provided with some basic functionalities already presented during the lessons.

For your information, this lab subject was the final test in 2016.

9.2 Work for this session

A basic tree module is provided.

The work for this session is split in 3 steps :
- — add basic functionalities to the tree module
- — add functionalities to the tree module to get an AVL tree
- — (optionnal) build a balanced tree from data stored in an array : require pointer of functions and binary file use

9.2.A Add basic functionalities to the tree module

Question 1 : the height of a node is the number of edges on the longest downward path between that node and a leaf. The height of a tree is the height of its root. Figure 9.1 describes an example.
Write a function *getHeight()* that returns the height of the tree given as a parameter.

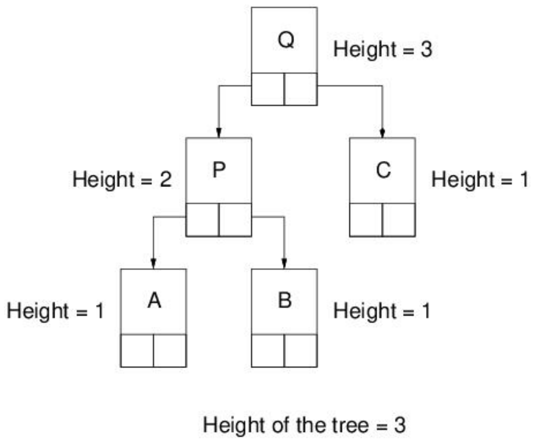

FIGURE 9.1 – The height of a binary tree

```
int getHeight(DTree t);
```

Question 2 : write a function that returns the maximal value found in such a sorted tree.

```
double findMax(DTree t);
```

9.2.B Add functionalities to the tree module to get an AVL tree

To perform efficient searches in a tree, it is important to maintain a balanced tree. A solution is proposed by the AVL tree [1] which is a self-balancing binary search tree. It is named after its inventors Georgy Adelson-Velsky and Evgenii Landis. The idea is that **the heights of the two child subtrees of any node differ by at most one**. If this requirement is not met, rotations in the tree structure are performed to get a balanced tree. In this part, you will code functionalities to get an AVL tree.

Question 3 : write a function that tests if a tree is an AVL tree or not, ie if the heights of the two child subtrees of any node differ by at most one.

```
int isUnbalancedTree(DTree t);
```

Question 4 : we would like to add some operations on the tree. The first one is the right rotation. When performing a right rotation on a tree, if the root Q of the tree has a left child P, the tree changes as described by Figure 9.2.

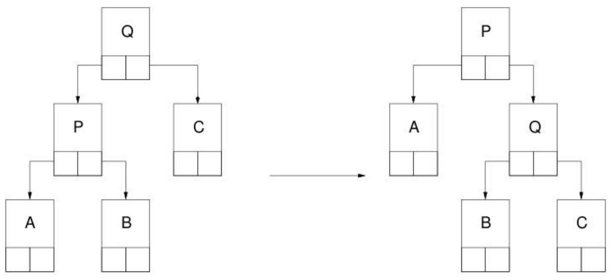

FIGURE 9.2 – The right rotation on a binary tree

1. cf https ://fr.wikipedia.org/wiki/Arbre_AVL

As you can see, the right child B of P becomes the left child of Q. Q becomes the right child of P and P becomes the new root of the tree. The rest of the tree remains the same.

Write the function *rightRotation()* that performs such an operation on a tree. It returns the new root of the tree. Its prototype is given by :

```
DTree rightRotation(DTree t);
```

You can note that the tree remains sorted with such an operation.

Question 5 : The next one is the left rotation. When performing a left rotation on a tree, if the root P of the tree has a right child Q, the tree changes as described by Figure 9.3.

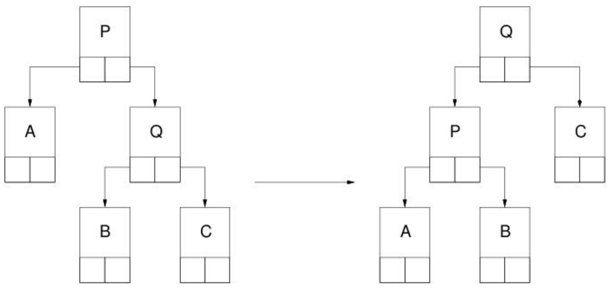

FIGURE 9.3 – The left rotation on a binary tree

As you can see, the left child B of Q becomes the right child of P. P becomes the left child of Q and Q becomes the new root of the tree. The rest of the tree remains the same.

Write the function *leftRotation()* that performs such an operation on a tree. It returns the new root of the tree. Its prototype is given by :

```
DTree leftRotation(DTree t);
```

You can note that the tree remains sorted with such an operation.

NB : when performing a right rotation and then a left one, you should get the tree as it was before the two operations.

Question 6 : This function aims to test if the tree is well balanced and if not, rebuild it to balance it. In order to do so, it starts from the root of the tree to the leaves. At each node, it tests if the absolute difference between the height of each child is greater than 1.

- If so, it means the tree is not balanced. In this case :
  - if the height of the left child is greater than the right one, it performs a right rotation on the current node ;
  - if the height of the right child is greater than the left one, it performs a left rotation on the current node ;
- if the absolute difference of height is smaller or equal to 1, then, nothing has to be done on this node.

Write the function *reBalance()* that performs this task. It returns the new root of the tree. Its prototype is given by :

```
  DTree reBalance(DTree t);
```

In order to test it, the function *addDouble()* is provided to build an unbalanced tree.

Question 7 (optionnal this special year) : We want to remove a value from a sorted tree and keep it sorted. Since this value may be the root, the tree will change. Therefore, the function needs to return the new root of the tree. Its prototype is then :

```
DTree removeNode(DTree t, double value);
```

If the value is not found, the tree will remain unchanged.

The removal will be performed using the recursive property of a tree. We will test the current node value :

1. if it is not the value to be deleted, we call *removeNode* on the left of right child depending on the value to be removed.
2. if the node is the one to be removed, there are several cases. If it has no child, it is easy to handle. We only need to free the node and return an empty tree. Otherwise, we can consider two cases :
   (a) the node with the value to be removed has only one child : we need to return this child after deleting the current node. You can find two examples in Figures 9.4 and 9.5. With such a change, the binary remains sorted ;
   (b) if the node has two children, a solution is to :
      i. find the maximal value in its left child and store this value,
      ii. call *removeNode* on the tree to remove this maximal value,
      iii. replace the current value (the one to be removed) by the maximal value that was deleted at the previous step,
      iv. return the current tree

      You can find two examples in Figures 9.6 and 9.7. With such a change, the binary remains sorted.

Test your function on the four examples.

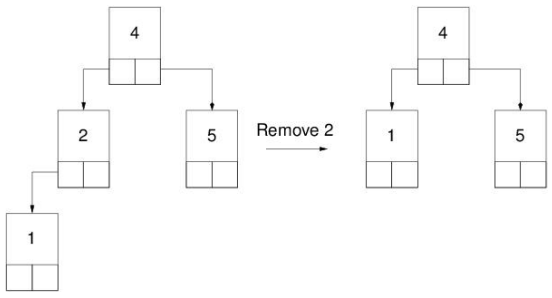

FIGURE 9.4 – Removing a value with only a left child.

9.2.C (Optionnal) Build a balanced tree from data stored in an unsorted array

This exercise is a good training to use function pointers and binary files. It was given during an exam.

As seen during the courses, it is possible to get a balanced tree by using the provided function *buildBalancedTreeFromSortedArray()*. However this function requires a sorted array, which is not always the case. Moreover, data are often stored in a file. So here are a few exercises to be able to load unsorted data for a binary sorted tree from a binary file.

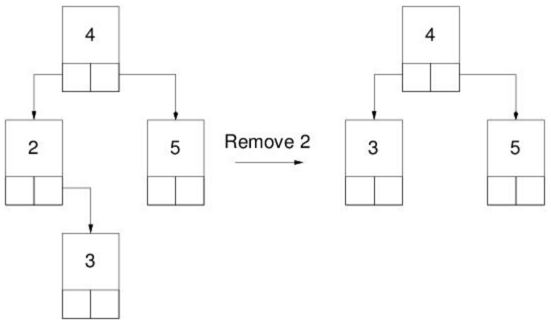

FIGURE 9.5 – Removing a value with only a right child.

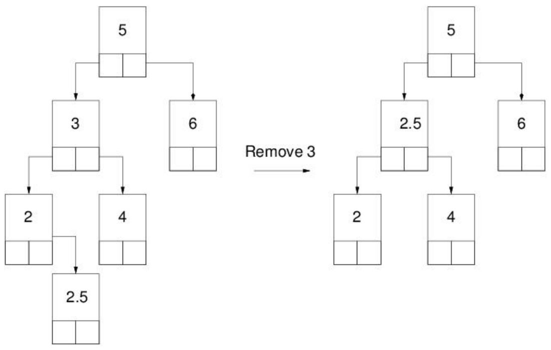

FIGURE 9.6 – Removing a value with two children : case 1.

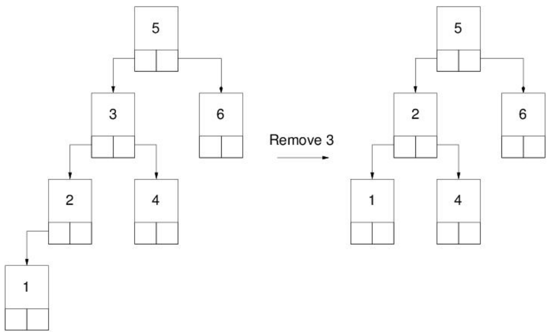

FIGURE 9.7 – Removing a value with two children : case 2.

Question 8 : write a function *buildBalancedTreeFromUnsortedArray(double * data, int nb)* that sorts the array *data* using the function *qsort()*, build the tree with the provided function *buildBalancedTreeFromSortedArray()* and return it. Here is the prototype of the function you have to write :

```
DTree buildBalancedTreeFromUnsortedArray(double * data, int n);
```

As you have to use *qsort()* to sort the array, here is a reminder on this function :

```
   void qsort(void* base, size_t nmemb, size_t size, int(*compar)(const void*,
       const void*));
```

The *qsort()* function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

Question 9 : the data of a sorted array of double can be provided in a binary file. There are only the double values stored in the file.

Write the function *readFromFile()* that returns an array with the data from such a binary file.

```
double * readArrayFromFile(char *s, int * nb);
```

where :
— *s* the filename where the data is stored ;
— *nb* an address to store the number of *double* read in this file ;
— the *double ** value returned by the function is the address of the beginning of the created array filled by the read values.

You can test your function on the provided "data.bin" file. You should get the following array with it : 1 ; 2.1 ; 3.5 ; 4.6 ; 5.1

Question 10 : use the two previous functions to provide a function that fill a tree from data in a binary file.

```
DTree readDTreeFromFile(char *s);
```

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7

Tél. +33 [0] 2 23 23 82 00
Fax +33 [0] 2 23 23 83 96

www.insa-rennes.fr