# Programación II, 2018-2019

# Escuela Politécnica Superior, UAM

### Practice 4: Binary Search Trees

## GOALS

- Familiarization and implementation in C of the TAD binary search tree.
- Apply the binary search trees to a real case. The application that is proposed aims to confront the student with two relevant aspects: the creation of the TAD itself and the manipulation of an important volume of information.

## NORMS

As in practice 1, in this practice delivered **programs** must:

- Be written in **ANSI** C, following established programming **rules**.
- Compile without errors or warnings including "-ansi" and "-pedantic" flags when compiling.
- **Run without problem in a command console**.
- Include adequate error control; it is justifiable that a program does not admit inadequate values, but not that it behaves abnormally with these values.
- It must be accompanied by a **report** that must be prepared based on the proposed model and delivered with practice.

## WORKING PLAN

**Week 1: code of P4_E1 and P4_E2.** Each teacher will indicate in class if a delivery has to be made and how: paper, e-mail, Moodle, etc.
**Week 2: code of P4_E3 and P4_E4**, all exercises.

**Week 2: all the exercises.**

The final delivery will be made through Moodle, following scrupulously the instructions indicated in the statement of practice 0 referring to the organization and nomenclature of files and projects. Remember that the compressed file must be called Px_Prog2_Gy_Pz, where '**x**' is the number of the practice, '**y**' is the practice group and '**z**' is the pair number (example of delivery of pair 5 of group 2161: P1_Prog2_G2161_P05.zip).
Moodle upload dates for the file are as follows:

- Students of continuous Evaluation, on **May 8** (until 11:30 p.m).
- **Final Evaluation** students, as specified in the regulations.

# PART 1. EXERCISES

**EXERCISE 1 (P4_E1). Binary Search Trees of Integers**

### 1. TAD implementation of the Binary Search Trees.

As in previous TAD implementations, we start with a generic structure that allows us to isolate the TAD definition from the types that it allows to store. In this case, the structure is as follows:

```
/* In tree.h */

typedef struct _Tree Tree;


/* En tree.c */

typedef struct _NodeBT {
        void* info;
        struct _NodeBT* left;
        struct _NodeBT* right;
} NodeBT;


struct _Tree {
    NodeBT *root;

    destroy_element_function_type   destroy_element_function;
    copy_element_function_type      copy_element_function;
    print_element_function_type     print_element_function;
    cmp_element_function_type       cmp_element_function;
};
```

The prototypes of the functions of this exercise are shown in Appendix 1.

### 2. Test of the TAD Tree with integer data.

In order to evaluate the working of the coded functions, develop a program **p4_e1.c** that works with binary integer search trees. This program will receive as an argument a file, where each line will contain a number, and will introduce them one by one in the tree. Once all the numbers have been read, the number of nodes as well as their depth will be displayed on the screen. At the end, it will allow you to search for a number entered by console.

Expected output for the file:

```
> cat numeros.txt
5
7
6
3
1
2
4
> ./p4_e1 numeros.txt
 Number of nodes: 7
Depth: 3
> Enter a number: 5
Number entered: 5
The data 5 is inside the Tree
```

**Exercise 2 (P4_E2). Traverse functions of a tree**

### 1. Implementation of functions to traverse the TAD Tree.

Complementing the functions of the TAD Tree prototype, it is requested to implement different traversing functions of a binary tree. The different routes will be in previous order, medium order and later order as explained in theory. In all cases, the traverse functions will dump the result of their output into a file (see Appendix 1 for the definition of the functions).

### 2. Test.

To test this functions, it is sufficient to extend the tests carried out in the P4_E1 exercise so that, in addition to the number of nodes and depth, the three routes of the tree are shown on the screen..

Write a new program named p4_e2.c that does just the same as p4_e1 but before asking for the number that should be searched, shows the tree in the three orders (pre, post and in).

Expected output for the case of an integer tree:

```
> ./p4_e2 numeros.txt
Number of nodes: 7
Depth: 3
Previous order: 5 3 1 2 4 7 6
Average order: 1 2 3 4 5 6 7
Later order: 2 1 4 3 6 7 5
> Enter a number: 9
Number entered: 9
The data 9 is NOT found inside the Tree
```

## Exercise 3 (P4_E3). Node Search Binary Trees

### 1. Modification of the TAD Node).

The Node TAD serves to represent a node in a graph. In this exercise we will modify the definition used until now in the practices for the following one, which allows a TAD with one of the most generic fields (not limited to a maximum size for the name, thanks to the use of dynamic memory for it):

```
/* In node_p4.c */

struct _Node {

   char* name;

   int id;

};
```

### 2. TAD Tree test with node type data.

Make a test program of this TAD Tree called p4_test-node-tree.c that reads a file of nodes like the following.

```
> cat nodos.txt
5 five
7 seven
6 six
3 three
1 one
2 two
4 four
```

### 3. Implementation check using an external program.

In this section, you are requested to test the prototypes of the TADs implemented so far in a comprehensive manner, using a program that we give you (p4_e3.c) that should work without changing anything from it (and compiled using the **makefile_ext file**.) can be used with the files dict.dat (and its versions dict10.dat, dict1K.dat, dict10K.dat and dict1M.dat), loading the nodes contained in them and inserting said nodes in the tree after processing them in a specific way according to how it is called (basically, if it is called **"./p4_e3 <name_file**> B" the nodes are ordered in memory and inserted so that the tree is created in the most balanced way possible, instead if it is called "./ **p4_e3 <name_file**> N ", they are inserted in the tree in the order they are read from the file).

The output expected for one of the files that are delivered, dict10K.dat, is shown below (creation and search times may vary, since it depends on the machine used):

```
> ./p4_e3 dict10K.dat N
10000 lines read

Tree creation time: 10000 ticks (0.010000 seconds)
Number of nodes: 10000
Depth: 30
Enter a node to search the tree (following the same format as in the input file):
3 a
Element NOT found!

Search time in the tree: 0 ticks (0.000000 seconds)
```

```
> ./p4_e3 dict10K.dat B
10000 lines read
Sorted data

Tree creation time: 10000 ticks (0.010000 seconds)
Number of nodes: 10000
Depth: 13
Enter a node to search the tree (following the same format as in the input file):
3 a
Element NOT found!

Search time in the tree: 0 ticks (0.000000 seconds)
```

**Note: to indicate that you have finished entering the node, you must press <enter> on the keyboard when you finish writing the data of the node.**

Below is the output for the largest file ***dict.dat***:

```
> ./p4_e3 dict.dat N
4001906 lines read

Tree creation time: 10980000 ticks (10.980000 seconds)
Number of nodes: 4001906
Depth: 54
Enter a node to search the tree (following the same format as in the input file):
3 a
Element NOT found!

Search time in the tree: 0 ticks (0.000000 seconds)
```

```
> ./p4_e3 dict.dat B
4001906 lines read
Sorted data

Tree creation time: 3710000 ticks (3.710000 seconds)
Number of nodes: 4001906
Depth: 21
Enter a node to search the tree (following the same format as in the input file):
3 a
Element NOT found!

Search time in the tree: 0 ticks (0.000000 seconds)
```

**Exercise 4 (P4_E4). Binary search trees for character strings**

**1. Implementation of a tree that allows to store strings of characters.**

In this exercise you are asked to implement a tree where the elements that are stored are strings of characters. To do this, we must comply that:

- The implementation of binary trees is not modified.

- Character strings do not have a predefined size.

Explain in the memory the decisions that have had to be taken to solve this exercise.

**2. Test it works.**

In a similar way to the first exercise, it is requested to code a program p4_e4.c where all the functions of the prototype of the TAD Tree are tested, reading the data of a file. A possible output of this exercise would be (showing the average order of the resulting tree):

```
> cat cadenas.txt
arbol
abridor
antes
abatir
ahora
barcelona
madrid
bueno
```

```
> ./p4_e4 cadenas.txt
Number of nodes: 8
Depth: 3
abatir opener now before tree barcelona good madrid
Enter a string to search the tree (following the same format as in the input file):
Madrid
Element found!
```

# PART 2. QUESTIONS ABOUT THE PRACTICE

1. Is the tree that is created from the node files (dict * .dat) complete or almost complete? Justify your answer.

2. a) What is the relationship between the "shape" of a tree and its traversing modes?

b) Can you tell if a binary search tree is well constructed according to how it is traversed?

3. Compare and describe the differences between the trees generated by the executables p4_e3 with the last argument B or N (number of nodes, depth, routes, etc.).

## Appendix 1: tree.h

```
typedef struct _Tree Tree;

typedef void (*destroy_elementtree_function_type)(void*);

typedef void (*(*copy_elementtree_function_type)(const void*));

typedef int (*print_elementtree_function_type)(FILE *, const void*);

typedef int (*cmp_elementtree_function_type)(const void*, const void*);

/* Initialize an empty tree. */

Tree*   tree_ini(

   destroy_element_function_type f1,

   copy_element_function_type f2,

   print_element_function_type f3,

   cmp_element_function_type f4);

/* Indicates if the tree is empty or not. */

Bool    tree_isEmpty( const Tree *pa);

/* Frees the memory used by a Tree.*/

void    tree_free(Tree *pa);

/* Inserts an element into a binary search tree by copying it into new memory. */

Status  tree_insert(Tree *pa, const void *po);

/*Writes in the file f the path of a tree in previous order without modifying it. */

Status    tree_preOrder(FILE *f, const Tree *pa);

/ * Writes in the file f the path of a tree in medium order without modifying it */

Status    tree_inOrder(FILE *f, const Tree *pa);

/ * Writes in file f the path of a tree in post order without modifying it. */

Status    tree_postOrder(FILE *f, const Tree *pa);

/* Computes the depth of a tree. An empty tree has depth -1. */

int    tree_depth(const Tree *pa);

/* Computes the number of nodes in a tree. An empty tree has 0 nodes.*/

int    tree_numNodes(const Tree *pa);

/* Returns TRUE if the element pe can be found in the tree pa */

Bool    tree_find(Tree* pa, const void* pe);
```