Programación II, 2018-2019

Escuela Politécnica Superior, UAM

Practice 1: Data Structures and Abstract Data Types

GOALS

- Deepen the concept of **TAD (Abstract Data Type).**
- Learn to choose the appropriate **data structure** to implement a TAD.
- Encode its **primitives** and use it in a main program.

NORMS

Delivered programs **must**:

- Be written in **ANSI C**, following the established **programming rules**.
- Compile without errors or warnings including "-ansi" and "-pedantic" flags when compiling.
- **Run without problems in a command console**.
- Incorporate an adequate **error control**. It is justifiable that a program does not admit inadequate values, but not that it behaves abnormally with these values.

WORKPLAN

Week 1: P1_E1 completed and most important functions of P1_E2.
Each teacher will indicate in class if a delivery has to be made and how: paper, e-mail, Moodle, etc..

Week 2: all EXERCISES.
The final delivery will be made through Moodle, scrupulously **following the instructions** indicated in the statement of practice 0 referring to the organization and nomenclature of files and projects. Remember that the compressed file that must be delivered must be named Px_Prog2_Gy_Pz, where '**x**' is the number of the practice, '**y**' the group of practices and '**z**' the number of the pair (example of delivery of pair 5 of group 2161 : P1_Prog2_G2161_P05.zip).

Moodle **upload dates** of the file are the following:

- Students on **Continuous Evaluation**, the Week of February 20th (*each group can make the delivery until 23:55 of the night before their practice class).*
- **Final Evaluation** students, as specified in the regulations.

.zip content for this Practice 1:

- The practice folder, following <u>to the letter</u> the instructions given in P0.
- A file answering the questions raised in PART 2 of this document, with name and surnames of the pair of practices. In this first practice it will not be necessary to elaborate any additional memory.

PART 1: CREATION OF THE TAD NODE AND GRAPH

In this practice, we will implement a graph of nodes. To do this, we will first begin by defining the type NODE (Node), and then work on the TAD GRAPG (Graph). Note that the contents of some of the files necessary for this practice are provided at the end of this file (see appendix 1).

EXERCISE 1 (P1_E1)

1. **Definition of the NODE data type (Node). Implementation: selection of data structure and implementation of primitives.**

In this practice, a node will be represented by an id (an integer) and a name (a fixed string of characters).

- In order to define the data structure necessary to represent the TAD NODE according to the methodology of hidden types seen in class, the following statement must be included in *node.h*:

```
typedef struct _Node Node;
```

  Besides, in node.c you have to include the implementation of the abstract data type, this is its data structure:

```
struct _Node {
   char name[100];
   int id;
   int nConnect;         };
```

- To be able to interact with data of type Node, at least the public functions of its interface whose prototypes are declared in the file node.h (see appendix 2) will be necessary. Write the code associated with its definition in the file *node.c*.

- File node.c may also include the definition of those private functions that facilitate the implementation of the public functions of the interface.

2. **Checking the correctness of the type Node definition and its primitives.**

You must create a file **p1_e1**.c that defines a program (named **p1_e1**) with the following operations:

- Initialize two nodes so that the first one is a node with name "first" and id 111 and the second node with name "second" and id 222.
- Print both nodes and then print a line break.
- Check if the two nodes are equal.
- Print the id of the first node along with an explanatory phrase (see example)
- Print the name of the second node (see example below)
- Copy the first node in the second.
- Print both nodes.
- Check if the two nodes are equal
- Free both nodes.

  El programa debe gestionar correctamente la memoria

  Output:

```
[111, first, 0][222, second, 0]
Are they equal? No
Id of the first node: 111
The name of the second node is: second
[111, first][111, first]
```

Are they equal? Sí

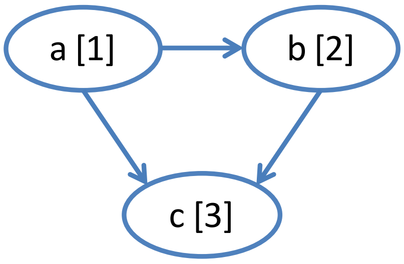
EXERCISE 2 (P1_E2)

Definition of the abstract type of data GRAPH.

1. Implementation: selection of the data structure and implementation of primitives.

In this part of the practice the Abstract Data Type (TaD) GRAPH will be defined as a set of homogeneous elements (of the same type) and a set of connections (edges) that define a binary relation between the elements.

You need to **define a data structure to represent the** TAD **GRAPH**. Assume that the data to be stored in the graph are of **Node** type and that its maximum capacity is 4096 elements. The information about the connections will be stored in an adjacency matrix (a matrix of 0's and 1's indicating if the node corresponding to the row is connected to the node corresponding to that column). That is, the following graph would have the adjacency matrix shown below:



	Node a	Node b	Node c
Node a	0	1	1
Node b	0	0	1
Node c	0	0	0

To implement the TADs related to the graph:

- Declare in the file graph.h the interface of the new Graph data type. This must include, at least, the functions indicated in Appendix 3.
- Defined in graph.c the data structure **_Graph.**

Private functions of the graph are provided (see Appendix 4)

```
int find_node_index (const Graph * g, int nId1)

int* graph_getConectionsIndex (const Graph * g, int index)
```

2. Checking the correctness of the definition of the type Graph and its primitives.

Define a program in a file named **p1_e2**.c whose executable is called **p1_e2**, and that performs the following operations:

- Initialize two nodes. The first one with name "first" and id 111 and the second one with name "second" and id 222).
- Initialize a graph.
- Insert node 1 and verify if the insertion was successful.
- Insert node 2 and verify if the insertion was successful.
- Insert a connection between node 2 and node 1.

- Check if node 1 is connected to node 2 (see message below).
- Check if node 2 is connected to node 1 (see message below).
- Insert node 2 and verify the result.
- Print the graph.
- Free the resources by destroying the nodes and the graph.

Output:

```
Inserting node 1 ... result ...: 1
Inserting node 2 ... result ...: 1
Inserting edge: node 2  ---> node 1
Connected node 1 and node 2? No
Connected node 2 and node 1? Yes
Inserting node 2 ... result ...: 0
Graph
[first, 111, 1]
[second, 222, 1]111
```

3. **Checking a graph behaviour through files.**

Along with this file, you are given a function that allows you to test the TaD Graph interface using files. This function (**graph_readFromGraph**) allows to load a graph from information read in a text file that repeats a given format.

Create a main program that as an input parameter (remember the parameters argc and argv of the main function) receives the name of a file. This file should be read and printed on the screen using the Tad Graph interface developed in the previous EXERCISES.

The execution of this program should run smoothly, including, proper memory management (ie, valgrind should not show memory leaks when running).

An example of an input data file is the following, where the first line indicates the number of nodes to be entered in the graph and in the following the information corresponding to each node (id and name); after these lines, pairs of integers will appear in separate lines indicating which nodes are connected with which:

```
3
1 a
2 b
3 c
1 2
1 3
2 3
```

In this way, this file and the graph of the figure in the previous section are equivalent.

PART 2: QUESTIONS ABOUT THE PRACTICE

Answer the following questions by **completing the file available in the .zip**. Rename the file so that it corresponds to your group number and practice pair and attach it to the .zip file that you deliver.

1. It provides a script that includes the commands necessary to compile, link and create an executable with the gcc compiler, indicating what action is performed in each of the sentences in the script. <u>ATTENTION, a Makefile file is not being requested</u>

2. Briefly justify if the following implementations of these functions are correct and if they are not, justify the reason why (suppose that the rest of the functions have been declared and implemented as in practice)

(a)

```
int main() {
   Node *n1;

   n1 = (Node*) malloc(sizeof(Node));
   if (!n1) return EXIT_FAILURE;

   /* initialize fields */
   node_setId (n1, -1);
   node_setName (n1, "");
   node_setConnect (n1, 0);

   node_ destroy (n1);
   return EXIT_SUCCESS;
}
```

(b)

```
// in node.h
Status node_ini (Node *n);

// in node.c
Status node_ini (Node *n) {

   n = (Node *) malloc(sizeof(Node));
   if (!n) return ERROR;

   /* initialize fields */
   node_setId (n, -1);
   node_setName (n, "");
   node_setConnect (n, 0);

   return OK;
}

// in main.c
int main() {
   Node *n1;

   if (node_ini (n) == ERROR)
         return EXIT_FAILURE;

   node_ destroy (n1);
   return EXIT_SUCCESS;
}
```

(c)

```
// in node.h
Status node_ini (Node **n);

// in node.c
Status node_ini (Node **n) {

   *n = (Node *) malloc(sizeof(Node));
   if (*n == NULL) return NULL;

   /* initialize fields */
   node_setId (*n, -1);
   node_setName (*n, "");
   node_setConnect (*n, 0);

   return OK;
}

// in main.c
int main() {
   Node *n1;

   if (node_ini (&n) == ERROR)
         return EXIT_FAILURE;

   node_ destroy (n1);
   return EXIT_SUCCESS;
}
```

3. Would it be possible to implement the nodes copy function using the following prototype
4. STATUS node_copy (Node nDest, const Node nOrigin); ? Why?

5. Is the pointer Node * essential in the prototype of the function int node_print (FILE * pf, const Node * n); or it could be int node_print (FILE * pf, const Node p); ?

If the answer is yes: Why?
If the answer is no: Why is it used, then?

6. What changes should be made in the function of copying nodes if we want it to receive a node as an argument where the information should be copied? That is, how should it be implemented if instead of Node * node_copy (const Node * nOrigin), it would have been defined as STATUS node_copy (const Node * nSource, Node * nDest)? Would the following be valid: STATUS node_copy (const Node * nSource, Node ** nDest)? Discuss the differences?.

7. Why should the functions of Appendix 4 not be public functions? Justify the answer.

Appendix 1: types.h

```
/*
 * File:   types.h
 * Author: Professors de PROG2
 */

#ifndef TYPES_H
#define     TYPES_H

typedef enum {
    ERROR = 0, OK = 1
} Status;

typedef enum {
    FALSE = 0, TRUE = 1
} Bool;

#endif      /* TYPES_H */
```

Appendix 2: node.h

```
#ifndef NODE_H_
#define NODE_H_

#include <stdio.h>
#include "types.h"

typedef struct _Node Node;
/ * Initialize a node, reserving memory and returning the initialized node if
  * it was done correctly, otherwise return NULL and print the corresponding
  * string to the error in stderror */
Node * node_ini();

/* Free the dynamic memory reserved for a node */
void node_destroy(Node * n);

/* Returns the id of a given node, or -1 in case of error */
int node_getId(const Node * n);

/* Returns a pointer to the name of a given node, or NULL in case of error */
char* node_getName(const Node * n);

/* Returns the number of connections of a given node, or -1 in case of error */
int node_getConnect(const Node * n);

/* Modifies the id of a given node, returns NULL in case of error */
Node * node_setId(Node * n, const int id);

/* Modifies the name of a given node, returns NULL in case of error */
Node * node_setName(Node * n, const char* name);

/* Modifies the number of connections of a given node, returns NULL in case of
error */
Node * node_setConnect(Node * n, const int cn);

/* Compares two nodes by the id and then the name.
  * Returns 0 when both nodes have the same id, a smaller number than
  * 0 when n1 <n2 or one greater than 0 otherwise. */
int node_cmp (const Node * n1, const Node * n2);

/* Reserves memory for a node where it copies the data from the node src.
  * Returns the address of the copied node if everything went well, or NULL
otherwise */
Node * node_copy(const Node * src);

/* Prints in pf the data of a node with the format: [id, name, nConnect]
  * Returns the number of characters that have been written successfully.
  * Checks if there have been errors in the Output flow, in that case prints
  * an error message in stderror*/
int node_print(FILE *pf, const Node * n);

#endif /* NODE_H_ */
```

Appendix 3: graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include "node.h"

typedef struct _Graph Graph;

/* Initializes a graph, reserving memory and returns the graph address
  * if it has done it correctly, otherwise it returns NULL and prints the string
  * associated with error in stderror */
Graph * graph_ini();

/* Frees the dynamic memory reserved for the graph */
void graph_destroy(Graph * g);

/* Adds a node to the graph (reserving new memory for that node) only
  * when there was no other node with the same id in the graph. It updates
  * the necessary graph’s attributes. Returns OK or ERROR. */
Status graph_insertNode(Graph * g, const Node* n);

/* Adds an edge between the nodes of id "nId1" and "nId2".
  * Updates the necessary graph’s attributes.
  * Returns OK or ERROR. */
Status graph_insertEdge(Graph * g, const int nId1, const int nId2);

/* Returns a copy of the node of id "nId"*/
Node *graph_getNode (const Graph *g, int nId);

/* Actualize the graph node with the same id */
Status graph_setNode (Graph *g, const Node *n);

/* Returns the address of an array with the ids of all nodes in the graph.
  * Reserves memory for the array. */
int * graph_getNodesId (const Graph * g);

/* Returns the number of nodes in the graph. -1 if there have been errors */
int graph_getNumberOfNodes(const Graph * g);

/* Returns the number of edges of the graph. -1 if there have been errors */
int graph_getNumberOfEdges(const Graph * g);

/* Determines if two nodes are connected */
Bool graph_areConnected(const Graph * g, const int nId1, const int nId2);

/* Returns the number of connections from the id node fromId */
int graph_getNumberOfConnectionsFrom(const Graph * g, const int fromId);

/* Returns the address of an array with the ids of all nodes in the graph.
  * Reserves memory for the array.*/
int* graph_getConnectionsFrom(const Graph * g, const int fromId);

/* Prints in the flow pf the data of a graph, returning the number of printed
characters.
  * Checks if there have been errors in the Output flow. If so, prints Error
  * message Error in stderror and returns the value -1.
  * The format to be followed is: print a line by node with the information
associated with the node and
 * the id of their connections. The Output for the graph of the EXERCISE 2.3 of
*part 1 is:
 * [1, a, 2]  2 3
 * [2, b, 2] 1 3
 * [3, c, 2]] 1 2  */
int graph_print(FILE *pf, const Graph * g);

* Read from the stream fin the graph information *
Status graph_readFromFile (FILE *fin, Graph *g);

#endif /* GRAPH_H */
```

Appendix 4: Private functions

The following private functions will be included in the file graph.c.

```
int find_node_index(const Graph * g, int nId1) {
    int i;
    if (!g) return -1;

    for(i=0; i < g->num_nodes; i++) {
        if (node_getId (g->nodes[i]) ==  nId1) return i;
    }

    // ID not find
   return -1;
}

int* graph_getConectionsIndex(const Graph * g, int index) {
    int *array = NULL, i, j=0, size;

    if (!g) return NULL;
    if (index < 0 || index >g->num_nodes) return NULL;

    // get memory for the array with the connected nodes index
    size = node_getConnect (g->nodes[index]);
    array = (int *) malloc(sizeof(int) * size);
    if (!array) {
        // print errorr message
        fprintf (stderr, "%s\n", strerror(errno));
        return NULL;
    }

    // assign values to the array with the indexes of the connected nodes
    for(i = 0; i< g->num_nodes; i++) {
        if (g->connections[index][i] == TRUE) {
            array[j] = i;
            j++;
        }
    }

    return array;
}
```

-

Appendix 5: Function to read a graph from a file

```
Status graph_readFromFile (FILE *fin, Graph *g) {
    Node *n;
    char buff[MAX_LINE], name[MAX_LINE];
    int i, nnodes = 0, id1, id2;
    Status flag = ERROR;

    // read number of nodes
    if ( fgets (buff, MAX_LINE, fin) != NULL)
        if ( sscanf(buff, "%d", &nnodes) != 1) return ERROR;

    // init buffer_node
    n = node_ini();
    if (!n) return ERROR;

    // read nodes line by line
    for(i=0; i < nnodes; i++) {
        if ( fgets(buff, MAX_LINE, fin) != NULL)
            if (sscanf(buff, "%d %s", &id1, name) != NO_FILE_POS_VALUES) break;

        // set node name and node id
        node_setName (n, name);
        node_setId (n, id1);

        // insert node in the graph
      if ( graph_insertNode (g, n) == ERROR) break;
    }

    // Check if all node have been inserted
    if (i < nnodes) {
        node_destroy(n);
        return ERROR;
    }

    // read connections line by line and insert it
    while ( fgets(buff, MAX_LINE, fin) ) {
        if ( sscanf(buff, "%d %d", &id1, &id2) == NO_FILE_POS_VALUES )
            if (graph_insertEdge(g, id1, id2) == ERROR) break;
    }

    // check end of file
    if (feof(fin)) flag = OK;

    // clean up, free resources
    node_destroy (n);
    return flag;
}
```