PRÁCTICA 1: Tipos Abstractos de Datos y Estructuras de datos

OBJETIVOS

- Profundizar en el concepto de TAD (Tipo Abstracto de Dato).
- Aprender a elegir la estructura de datos apropiada para implementar un TAD.
- Codificar sus primitivas y utilizarlo en un programa principal.
- Entender el tipo de dato `void *` del lenguaje C

NORMAS

Los programas que se entreguen deben:

- Estar escritos en ANSI C, siguiendo las normas de programación establecidas.
- Compilar sin errores ni warnings incluyendo las banderas "-ansi" y "-pedantic" al compilar.
- Ejecutarse sin problema en una consola de comandos.
- Incorporar un adecuado control de errores. Es justificable que un programa no admita valores inadecuados, pero no que se comporte de forma anómala con dichos valores.
- No producir fugas de memoria al ejecutarse

PLAN DE TRABAJO

- Semana 1: Ejercicio 1 completo y funciones más importantes del Ejercicio 2. Cada profesor indicará en clase cómo realizar la entrega: papel, e-mail, Moodle, etc.
- Semana 2: Ejercicio 2.
- Semana 3: Ejercicio 2 (OPTATIVO).

La entrega final se realizará a través de Moodle, siguiendo escrupulosamente las instrucciones indicadas en el enunciado de la práctica 0 referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido que se debe entregar debe llamarse `Px_Prog2_Gy_Pz`, siendo 'x' el número de la práctica, 'y' el grupo de prácticas y 'z' el número de pareja (ejemplo de entrega de la pareja 5 del grupo 2161: `P1_Prog2_G2161_P05.zip`).

Las fechas de subida a Moodle del fichero son las siguientes:

- Los alumnos de Evaluación Continua, la semana del **24 de febrero** (cada grupo puede realizar la entrega hasta las **23:55 h**. de la noche anterior a su clase de prácticas).
- Los alumnos de Evaluación Final, según lo especificado en la normativa.

PARTE 1: CREACIÓN DE LOS TAD NODO (NODE) Y GRAFO (GRAPH)

En esta práctica se va a implementar un grafo de nodos. Para ello, primero se comenzará definiendo el tipo NODO (Node), y a continuación se trabajará sobre el TAD GRAFO (Graph). Observe que el contenido de algunos de los ficheros necesarios para esta práctica se proporcionan al final de este enunciado (ver el enlace Files -> File List en Doxygen).

Ejercicio 1

Definición del tipo abstracto de dato Node. implementación: selección de estructura de datos e implementación de primitivas.

En esta práctica, un nodo se representará mediante un id (un long int), un nombre (una cadena fija de caracteres), una etiqueta (de tipo Label) y un número dado de conexiones (de tipo int).

- Para definir la estructura de datos necesaria para representar el TaD NODO conforme la metodología de tipos ocultos vista en clase, debe incluirse la siguiente declaración en node.h:

```
typedef struct _Node Node;
```

Además, en node.c hay que incluir la implementación del tipo abstracto de datos, esto es su estructura de datos _Node

```
#define NAME_L 64   /*!< Maximum node name length */

struct _Node {
    char name[NAME_L]; /*!<Node name */
    long id;            /*!<Node id */
    int nConnect;      /*!<Node number of connections */
    Label label;       /*!<Node state */
};
```

- Para poder interactuar con datos de tipo Node serán necesarias, al menos, las funciones de su interfaz cuyos prototipos se encuentren declarados en el fichero node.h. Escribid el código asociado a su definición en el fichero node.c.
- El fichero node.c podrá incluir, además, la definición de aquellas funciones privadas que faciliten la implementación de las funciones públicas de la interfaz.

Comprobación de la corrección de la definición del tipo Node y sus primitivas.

Se deberá crear un fichero `p1_e1.c` que defina un programa ejecutable (de nombre `p1_e1`) con las siguientes operaciones:

- Inicializar dos nodos de modo que el primero sea un nodo con nombre "first", id 111 y label WHITE y el segundo otro nodo con nombre "second", id 222 y label WHITE.
- Imprimir el contenido de ambos nodos e imprimir después un salto de línea.
- Comprobar si los dos nodos son iguales.
- Imprimir el id del primer nodo junto con una frase explicativa (ver ejemplo).
- Imprimir el nombre del segundo nodo (ver ejemplo más abajo).
- Copiar el primer nodo en el segundo.
- Imprimir el contenido de ambos nodos e imprimir después un salto de línea.
- Comprobar si los dos nodos son iguales.

El programa deberá gestionar correctamente la memoria y hacer los oportunos controles de errores.

La salida del programa deberá ser:

```
[111, first, 0, 0][222, second, 0, 0]
Son iguales? No
Id del primer nodo: 111
Nombre del segundo nodo es: second
[111, first, 0, 0][111, first, 0, 0]
Son iguales? Sí
```

Apéndice 1

Contenido del fichero `node.h`

```
/**
 * @file  node.h
 * @author Profesores Prog2
 * @date 29 January 2020
 * @version 1.0
 * @brief Library to manage TAD Node
 *
 * @details
 *
 * @see
 */
```

```
#ifndef NODE_H_
#define NODE_H_

#include "types.h"

/**
 * @brief Label to characterize the node state (to be used in P2)
 *
 **/
typedef enum {
    WHITE, /*!< node not visited */
    BLACK,  /*!< nodo visited */
    ERROR_NODE   /*!< not valid node */
} Label;

/**
 * @brief Data structure to implement the ADT node. To be defined in node.c
 *
 **/
typedef struct _Node Node;


/**
 * @brief Constructor. Initialize a node, reserving memory.
 *
 * This function initiates the node structure fields to "", -1, 0 and WHITE
 * respectively
 * @code
 * Node *n;
 * n = node_init ();
 * @endcode
 * @return Return the initialized node if
  * it was done correctly, otherwise return NULL and print the corresponding
  * string to the error in stderror
*/
Node * node_init ();

/**
 * @brief Destructor. Free the dynamic memory reserved for a node
 * @param n Node to free
 */
void node_free (void * n);
```

```
/**
 * @brief  Gets the node id
 * @param n Node address
 * @return  Returns the id of a given node, or -1 in case of error
 *  */
long node_getId (const Node * n);

/**
 * @brief Gets the node name
 * @param n Node address
 * @return Returns a pointer to the name of the node, or NULL in case of error
 */
const char* node_getName (const Node * n);

/**
 * @brief Gets the number of connections of a given node.
 * @param n Node address
 * @return Returns the number of connections of a given node, or -1 in case
 * of error
 */
int node_getNConnect (const Node * n);

/**
 * @brief Gets the label of a given node.
 * @param Node address
 * @return Returns the label of a given node, or ERROR_NODE in case of error
 */
Label node_getLabel (const Node* n);


/**
 * @brief Modifies the label of a given node
 * @param n Node address
 * @param id New node label
 * @return Returns OK or ERROR in case of error
 */
Status node_setLabel (Node *n, const Label l);

/**
 * @brief Modifies the id of a given node
 * @param n Node address
 * @param id New node id
 * @return Returns OK or ERROR in case of error
 */
Status node_setId (Node * n, const long id);
```

```
/**
 * @brief Modifies the name of a given node
 * @param n Node address
 * @param id New node name
 * @return  Returns OK or ERROR in case of error
 */
Status node_setName (Node *n, const char *name);

/**
 * @brief Modifies the number of connections of a given node
 * @param n Node address
 * @param id Number of connections for the node
 * @return  Returns OK or ERROR in case of error
 */
Status node_setNConnect (Node *n, const int cn);


/**
 * @brief Compares two nodes by id and then by name.
 * @param n1,n2 Nodes to compare.
 * @return return an integer less than, equal to, or greater than zero
 * depending on their ids. If they are the same, then their names must
 * be compared.
 */
int node_cmp (const void *n1, const void *n2);

/**
 * @brief Reserves memory for a node where it copies the data from the node src.
 * @code
 * Node *trg, *src;
 * src = node_init ();
 * trg = node_copy(src);
 * // .... aditional code ...
 * // free nodes
 * node_free (src);
 * node_free (trg);
 * @endcode
 * @param  Original node
 * @return Returns the address of the copied node if everything went well,
 * or NULL otherwise
 */
void * node_copy (const void *src);

/**
 * @brief Prints in pf the data of a node
 *
```

```
 * The format will be: [id, name, label, nConnect]
 * @code
 * Node *n;
 * n = node_init ();
 * node_print (stdout, n);
 * @endcode
 * @param pf File descriptor
 * @param n Node to be printed
 * @return 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 and returns -1.
 */
int node_print (FILE *pf, const void *n);

#endif /* NODE_H_ */
```

Apéndice 2

Contenido del fichero `types.h`

```
/**
 * @file types.h
 * @author Profesores Programación 2
 * @date 2 February 2020
 * @brief ADT Boolean and Status
 *
 * @details Here typically goes a more extensive explanation of what the header
 * defines. Doxygens tags are words preceeded by @.
 *
 * @see
 */

#ifndef TYPES_H_
#define TYPES_H_

/**
 * @brief ADT Boolean
 */
typedef enum {
    FALSE=0, /*!< False value */
    TRUE=1  /*!< True value  */
} Bool;
```

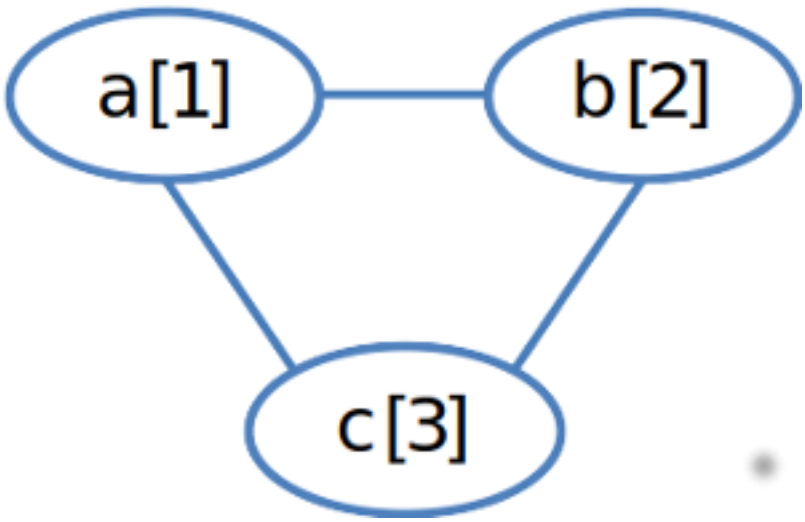
```
/**
 * @brief ADT Status
 */
typedef enum {
    ERROR=0, /*!< To codify an ERROR output */
    OK=1     /*!< OK output */
} Status;

#endif /* TYPES_H_ */
```

EJERCICIO 2.

En esta parte de la práctica se definirá el Tipo Abstracto de Datos (TaD) GRAFO como un conjunto de elementos homogéneos (del mismo tipo) y un conjunto de conexiones (aristas) que definen una relación binaria entre los elementos.

Definición del tipo abstracto de dato Graph e implementación: selección de su estructura de datos e implementación de su interfaz.

Se desea definir una estructura de datos para representar el TAD Graph. Asumir que los datos que hay que almacenar en él grafo son de tipo Node y que su capacidad máxima son 4096 elementos. La información sobre las conexiones se almacenará en una matriz de adyacencia (una matriz de 0's y 1's indicando si el nodo correspondiente a la fila está conectado con el nodo correspondiente a esa columna). Por ejemplo, el siguiente grafo tendría la matriz de adyacencia que se muestra a continuación:



	Nodo A	Nodo B	Node C
Nodo A	0	1	1
Nodo B	0	0	1
Nodo C	0	0	0

Para implementar el TADs GRAFO deberás:

- Crear el fichero graph.h con los prototipos de las funciones de la interfaz del tipo de dato Graph y los includes necesarios.
- Definir en graph.c la estructura de datos _Graph.

```
# define MAX_NODES 1064

struct _Graph {
    Node *nodes[MAX_NODES]; /*!<Array with the graph nodes */
    Bool connections[MAX_NODES][MAX_NODES];  /*!<Adjacency matrix  */
    int num_nodes;          /*!<Total number of nodes in te graph  */
    int num_edges;          /*!<Total number of connection in the graph  */
};
```

- Implementar en graph.c las funciones de la interfaz declaradas en el fichero graph.h.
- Incluir en graph.c las funciones privadas que consideres oportunas. Se facilitan las siguientes dos funciones privadas (para ver su implementación consultar el apéndice al final de este documento) por si fuesen de tu interés

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

Comprobación de la corrección de la definición del tipo Graph y de su interfaz.

Crear un programa en un fichero de nombre `p1_e2.c` cuyo ejecutable se llame `p1_e2`, y que realice las siguientes operaciones:

- Inicialice dos nodos. El primero con nombre “first”, id 111 y label WHITE y el segundo con nombre “second”, id 222 y label WHITE).
- Inicialice un grafo.
- Insertar el nodo 1 y verifique si la inserción se realizó correctamente
- Insertar el nodo 2 y verifique si la inserción se realizó correctamente
- Insertar una conexión entre el nodo 2 y el nodo 1.
- Comprobar si el nodo 1 está conectado con el nodo 2 (ver mensaje más abajo).
- Comprobar si el nodo 2 está conectado con el nodo 1 (ver mensaje más abajo).
- Insertar nodo 2 y verificar el resultado.

- Imprimir el grafo
- Liberar los recursos destruyendo los nodos y el grafo.

Salida

```
Insertando nodo 1...resultado...: 1
Insertando nodo 2...resultado...: 1
Insertando edge: nodo 2  ---> nodo 1
Conectados nodo 1 y nodo 2? No
Conectados nodo 2 y nodo 1? Si
Insertando nodo 2....resultado: 0
 Grafo
[first, 111, 0, 0]
[second, 222, 0, 1]111
```

Comprobación del funcionamiento de un grafo leído de un fichero.

Implementar la función de la interfaz

```
Status graph_readFromFile (FILE *fin, Graph *g);
```

que permite cargar un grafo a partir de información leída en un fichero de texto que respete un formato dado.

Cree un programa **p1_e3.c** main que como parámetro de entrada (recuerda los parámetros `argc` y `argv` de la función main) reciba el nombre de un fichero. Este fichero deberá leerse e imprimirse en pantalla haciendo uso de la interfaz del Tad Graph desarrollada en los ejercicios anteriores.

La ejecución de este programa debería funcionar sin problemas, incluyendo, la gestión adecuada de memoria (es decir, `valgrind` no debería mostrar fugas de memoria al ejecutarse). A continuación se muestra un ejemplo de fichero de datos de entrada. La primera línea indica el número de nodos del grafo y en las siguientes la información correspondiente a cada nodo (id, nombre y label); después de estas líneas aparecerán en lineas separadas pares de enteros indicando qué nodos (a partir de sus ids) están conectados con cuáles:

```
3
1 a 0
2 b 0
3 c 0
1 2
1 3
2 3
```

De esta forma, este fichero y el grafo de la figura de la sección anterior son equivalentes.

Apéndice 3.

Funciones privadas del TAD Graph

```
// It returns the index of the node with id nId1
int find_node_index(const Graph * g, long 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 found
   return -1;
}

// It returns an array with the indices of the nodes connected to the node
// whose index is index
// It also allocates memory for the array.
int* graph_getConnectionsIndex(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
    size = node_getConnect (g->nodes[index]);

    array = (int *) malloc(sizeof (int) * size);
    if (!array) {
        // print error message
        fprintf (stderr, "%s\n", strerror(errno));
        return NULL;
    }

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

    return array;
}
```

Apéndice 4.

Contenido del fichero graph.h

```
/**
 * @file graph.h
 * @author Profesores Prog2
 * @date Created on 29 January 2020, 15:03
 * @version 1.0
 * @brief Library to manage TAD Graph
 *
 * @see
 */

#ifndef GRAPH_H
#define GRAPH_H

#include "node.h"

typedef struct _Graph Graph;

/**
 * @brief Initializes a graph, reserving memory
 * @return 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_init ();

/**
 * @brief Frees the dynamic memory reserved for the graph
 **/
void graph_free (Graph *g);


/**
 * @brief 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.
 * @param g Pointer to the graph where the new node to be inserted
 * @param n Node to be inserted
 * @return Returns OK or ERROR.
 **/
Status graph_insertNode (Graph *g, const Node *n);
```

```
/**
 * @brief Adds an edge from the node with id "nId1" to the node "nId2" in a directed graph.
 * @param Pointer to the graph where the new edge to be inserted
 * @param nId1, nId2  Nodes ids
 * @return Returns OK or ERROR.
 **/
Status graph_insertEdge (Graph *g, const long nId1, const long nId2);

/**
 * @brief  Returns a copy of the node of id "nId" storaged in the graph
 * @param Pointer to the graph
 * @param nId1 Node id
 * @return Returns a ponter to the copy or NULL if the node does not exit in the graph.
 **/
Node *graph_getNode (const Graph *g, long nId);

/**
 * @brief Actualize the graph node with the same id
 * @param g Pointer to the graph
 * @param n
 * @return Returns OK or ERROR.
 **/
Status graph_setNode (Graph *g, const Node *n);

/**
 * @brief Returns the address of an array with the ids of all nodes in the graph.
 * Reserves memory for the array.
 * @param g Pointer to the graph
 * @return Returns the address of an array with the ids of all nodes in the graph or NULL if
 **/
long * graph_getNodesId (const Graph *g);

/**
 * @brief Returns the total number of nodes in the graph.
 * @param g Pointer to the graph
 * @return Returns the number of nodes in the graph. -1 if there have been errors
 **/
int graph_getNumberOfNodes (const Graph *g);

/**
 * @brief Returns the total number of edges in the graph.
 * @param g Pointer to the graph
 * @return Returns the number of nodes in the graph. -1 if there have been errors
 **/
int graph_getNumberOfEdges (const Graph *g);
```

```
/**
 * @brief Determines if two nodes are connected.
 * @param nId1, nId2 Nodes id
 * @return Returns TRUE or FALSE
 **/
Bool graph_areConnected (const Graph *g, const long nId1, const long nId2);

/**
 * @brief Get the total number of connections from the node with a given id.
 * @param fromId Node id
 * @param g Graph
 * @return Returns the total number of connections or -1 if there are any error
 **/
int graph_getNumberOfConnectionsFrom (const Graph *g, const long fromId);

/**
 * @brief Returns an array with the id of the nodes connected to a given node. Allocates mem
 * @param g The graph
 * @param fromId The id of the given node
 * @return Returns the number of connections from the id node fromId or -1 if there are any
 */
long* graph_getConnectionsFrom (const Graph *g, const long fromId);

/**
 * @brief Prints in the file pf the data of a graph, returning the number of printed charact
 *
 * The format to be followed is: print a line by node with the information associated with t
 * the id of their connections:
 * @code
 * [1, a, 0, 2]  2 3
 * [2, b, 0, 2] 1 3
 * [3, c, 0, 2]] 1 2
 * @endcode
 * @param pf File descriptor
 * @param g Pointer to the graph
 * @return Return the number of characters printed.
 * If there have been errors in the Output flow
 * prints an error message in stderror and returns the value -1.
 */
int graph_print (FILE *pf, const Graph *g);


/**
 * @brief Read from the stream pointer fin the graph information.
 *
 * This function is optative. See execirse 3.
```

```
 * @param pf Stream pointer
 * @param g Pointer to the graph
 * @return OK or ERROR
 */
Status graph_readFromFile (FILE *fin, Graph *g);

#endif /* GRAPH_H */
```

PARTE 2: PREGUNTAS ACERCA DE LA PRÁCTICA

Responded a las siguientes preguntas completando el fichero disponible en el .zip. Renombrad el fichero para que se corresponda con su nº de grupo y pareja de prácticas y adjuntadlo al fichero .zip que entreguéis.

1. Proporciona un script que incluya los comandos necesarios para compilar, enlazar y crear un ejecutable con el compilador `gcc`, indicando que acción se realiza en cada una de las sentencias del script. ATENCIÓN no se está pidiendo un fichero Makefile.
2. Justifique brevemente si son correctas o no las siguientes implementaciones de las funciones y en el caso de que no lo sean justifique el porqué (suponed que el resto de las funciones se han declarado e implementado como en la práctica)

   2.a)

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

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

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

    node_free (n1);
    return EXIT_SUCCESS;
}
```

2.b)

```
// In the file node.h
Status node_init (Node *n);

// In the file node.c
Status node_init (Node *n) {

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

    /* init fields */
    node_setId (n, -1);
    node_setName (n, "");
```

```
    node_setConnect (n, 0);
    node_setLabel (n, BLANCO);

    return OK;
}

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

    if (node_ini (n1) == ERROR)
          return EXIT_FAILURE;

    node_free (n1);

    return EXIT_SUCCESS;
}
```

2.c)

```
// In the file node.h
Status node_init (Node **n);

// In the file node.c
Status node_init (Node **n) {

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

    /* inicializa campos */
    node_setId (*n, -1);
    node_setName (*n, "");
    node_setConnect (*n, 0);

    return OK;
}

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

    if (node_ini (&n1) == ERROR)
          return EXIT_FAILURE;

    node_free (n1);
    return EXIT_SUCCESS;
}
```

3. ¿Sería posible implementar la función de copia de nodos empleando el siguiente prototipo?¿Por qué?

```
STATUS node_copy (Node nDest, const Node nOrigin);
```

4. ¿Es imprescindible el puntero `Node *` en el prototipo de la función int `node_print (FILE * pf, const Node * n);` o podría ser `int node_print(FILE * pf, const Node p);` ? Si la respuesta es sí: ¿Por qué? Si la respuesta es no: ¿Por qué se utiliza, entonces?

5. ¿Qué cambios habría que hacer en la función de copiar nodos si quisiéramos que recibiera un nodo como argumento donde hubiera que copiar la información? Es decir, ¿cómo se tendría que implementar si en lugar de `Node* node_copy(const Node* nOrigin)`, se hubiera definido como `STATUS node_copy(const Node* nSource, Node* nDest)`? ¿Lo siguiente sería válido: `STATUS node_copy(const Node* nSource, Node** nDest);`? Discute las diferencias.

6. ¿Por qué las funciones privadas incluidas en el apéndice no deben ser funciones públicas? Justifica la respuesta.

7. Podría ser el tipo devuelto por la función `node_copy` diferente a `void *`. Justifica la respuesta.