Programación II, 2018-2019
Escuela Politécnica Superior, UAM

Práctica 1: Estructuras de Datos y Tipos Abstractos 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.

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.

PLAN DE TRABAJO

Semana 1: P1_E1 completo y funciones más importantes de P1_E2.
Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.

Semana 2: todos los ejercicios.
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 20 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.

Contenido del .zip para esta Práctica 1:
- Carpeta de la práctica, siguiendo al pie de la letra las instrucciones indicadas en la P0.
- Fichero donde se respondan a las cuestiones planteadas en la PARTE 2 de este documento, con nombre y apellidos de la pareja de prácticas. En esta primera práctica no será necesario elaborar ninguna memoria adicional.

PARTE 1: CREACIÓN DEL TAD NODO Y GRAFO

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 apéndice 1).

EJERCICIO 1 (P1_E1)

1. Definición del tipo de dato NODO (*Node*). Implementación: selección de estructura de datos e implementación de primitivas.

En esta práctica, un nodo se representará mediante un ***id*** (un entero) y un nombre (una cadena fija de caracteres).

- 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:

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

- Para poder interactuar con datos de tipo *Node* serán necesarias, al menos, las funciones públicas de su interfaz cuyos prototipos se encuentren declarados en el fichero *node.h* (ver apéndice 2). 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.

2. 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 (de nombre **p1_e1**) con las siguientes operaciones:

- Inicializar dos nodos de modo que el primero sea un nodo con nombre “first” e id 111 y el segundo otro nodo con nombre “second” e id 222.
- Imprimir 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 ambos nodos.
- Comprobar si los dos nodos son iguales
- Liberar ambos nodos.

  El programa debe gestionar correctamente la memoria

  Salida:

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

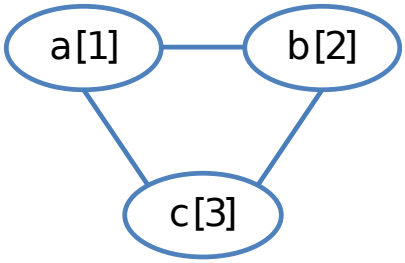
EJERCICIO 2 (P1_E2)

Definición del tipo abstracto de dato GRAFO.

1. Implementación: selección de estructura de datos e implementación de primitivas.

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.

Se desea **definir una estructura de datos para representar el TAD GRAFO.** 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). Es decir, el siguiente grafo tendría la matriz de adyacencia que se muestra a continuación:



	Nodo a	Nodo b	Nodo c
Nodo a	0	1	1
Nodo b	0	0	1
Nodo c	0	0	0

Para implementar los TADs relacionados con el grafo:

- Declarad en el fichero *graph.h* la interfaz del nuevo tipo de dato **Graph.** Este deberá incluir, al menos, las funciones indicadas en el apéndice 3.
- Definid en graph.c la estructura de datos **_Graph.**

Se facilitan las funciones privadas del grafo (ver apéndice 4)

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

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

2. Comprobación de la corrección de la definición del tipo Graph y sus primitivas.

Definid 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” e id 111 y el segundo con nombre “second” e id 222).
- 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? Sí
Insertando nodo 2....resultado: 0
 Grafo
[first, 111, 1]
[second, 222, 1]111
```

3. Comprobación del funcionamiento de un grafo a través de ficheros.

Junto con este enunciado, se os entrega una función que permite probar la interfaz del TaD Grafo usando ficheros. Esta función (**graph_readFromGraph**) permite cargar un grafo a partir de información leída en un fichero de texto que repete un formato dado.

Cree un programa 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).

Un ejemplo de fichero de datos de entrada es el siguiente, donde la primera línea indica el número de nodos que se van a introducir en el grafo y en las siguientes la información correspondiente a cada nodo (id y nombre); después de estas líneas aparecerán en lineas separadas pares de enteros indicando qué nodos están conectados con cuáles:

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

```
1 2
1 3
2 3
```

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

PARTE 2: PREGUNTAS SOBRE 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)

(a)

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

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

   /* inicializa campos */
   node_setId (n1, -1);
   node_setName (n1, "");
   node_setConnect (n1, 0);

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

(b)

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

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

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

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

   return OK;
}

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

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

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

(c)

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

// en node.c
Status node_ini (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;
}

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

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

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

3. ¿Sería posible implementar la función de copia de nodos empleando el siguiente prototipo
   *STATUS node_copy (Node nDest, const Node nOrigin)**;*** ? ¿Por qué?

4. ¿Es imprescindible el puntero Node* en el prototipo de la función i*nt 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 **S***TATUS node_copy(const Node* nSource, Node* nDest)*? ¿Lo siguiente sería válido: *STATUS node_copy(const Node* nSource, Node** nDest***)**? Discute las diferencias.

6. ¿Por que las funciones del apéndice 4 no deben ser funciones públicas? Justifica la respuesta.

Apéndice 1: types.h

```
/*
 * File:   types.h
 * Author: Profesores 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 */
```

Apéndice 2: node.h

```
#ifndef NODE_H_
#define NODE_H_

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

typedef struct _Node Node;
/* Inicializa un nodo, reservando memoria y devolviendo el nodo inicializado si
 * lo ha hecho correctamente, sino devuelve NULL en otro caso
 *  e imprime el string correspondiente al errorno en stderror */
Node * node_ini();

/* Libera la memoria dinámica reservada para un nodo */
void node_destroy(Node * n);

/* Devuelve el id de un nodo dado, o -1 en caso de error */
int node_getId(const Node * n);

/* Devuelve un puntero al nombre de un nodo dado, o NULL en caso de error */
char* node_getName(const Node * n);

/* Devuelve el número de conexiones de un nodo dado, o -1 en caso de error */
int node_getConnect(const Node * n);

/* Modifica el id de un nodo dado, devuelve NULL en caso de error */
Node * node_setId(Node * n, const int id);

/* Modifica el nombre de un nodo dado, devuelve NULL en caso de error */
Node * node_setName(Node * n, const char* name);

/* Modifica el número de conexiones de un nodo dado, devuelve NULL en caso de
error */
Node * node_setConnect(Node * n, const int cn);

/* Compara dos nodos por el id y después el nombre.
 * Devuelve 0 cuando ambos nodos tienen el mismo id, un número menor que
 * 0 cuando n1 < n2 o uno mayor que 0 en caso contrario. */
int node_cmp (const Node * n1, const Node * n2);

/* Reserva memoria para un nodo en el que copia los datos del nodo src.
 * Devuelve la dirección del nodo copia si todo ha ido bien, o NULL en otro caso
*/
Node * node_copy(const Node * src);

/* Imprime en  pf los datos de un nodo con el formato: [id, name, nConnect]
 * Devuelve el número de caracteres que se han escrito con éxito.
 * Comprueba si ha habido errores en el flujo de salida, en ese caso imprime
 * mensaje de error en stderror*/
int node_print(FILE *pf, const Node * n);

#endif /* NODE_H_ */
```

Apéndice 3: graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include "node.h"

typedef struct _Graph Graph;

/* Inicializa un grafo, reservando memoria y devolviendo la dirección del grafo
 * si lo ha hecho correctamente, o si no devuelve NULL e imprime el string
 * asociado al errorno en stderror  */
Graph * graph_ini();

/* Libera la memoria dinámica reservada para el grafo */
void graph_destroy(Graph * g);

/* Se añade un nodo al grafo (reservando memoria nueva para dicho nodo) siempre
 * y cuando no hubiese ya otro nodo de igual id en el grafo. Actualiza
 * los atributos del grafo que sean necesarios. Devuelve OK o ERROR. */
Status graph_insertNode(Graph * g, const Node* n);

/* Se añade una arista entre los nodos de id "nId1" y "nId2".
 * Actualiza los atributos del grafo y de los nodos que sean necesarios.
 * Devuelve OK o ERROR. */
Status graph_insertEdge(Graph * g, const int nId1, const int nId2);

/* Devuelve una copia del nodp de id "nId" */
Node *graph_getNode (const Graph *g, int nId);

/* Actualiza el nodo del grafo que tiene el mismo id que el nodo n, con la
información de n */
Status graph_setNode (Graph *g, const Node *n);

/* Devuelve la dirección de un array con los id de todos los nodos del grafo.
 * Reserva memoria para el array. */
int * graph_getNodesId (const Graph * g);

/* Devuelve el número de nodos del grafo. -1 si ha habido algún error*/
int graph_getNumberOfNodes(const Graph * g);

/* Devuelve el número de aristas del grafo. -1 si ha habido algún error*/
int graph_getNumberOfEdges(const Graph * g);

/* Determina si dos nodos están conectados*/
Bool graph_areConnected(const Graph * g, const int nId1, const int nId2);

/* Devuelve el número de conexiones del nodo de id fromId */
int graph_getNumberOfConnectionsFrom(const Graph * g, const int fromId);

/* Devuelve la dirección de un array con los id de todos los nodos del grafo.
 * Reserva memoria para el array. */
int* graph_getConnectionsFrom(const Graph * g, const int fromId);

/* Imprime en el flujo pf los datos de un grafo, devolviendo el número de
caracteres impresos.
```

```
 * Comprueba si ha habido errorres en el flujo de salida. Si es así imprime
mensaje
 * de error en stderror y devuelve el valor -1.
 * El formato a seguir es: imprimir una linea por nodo con la información asociada
al nodo y
 * los id de sus conexiones. La salida para el grafo del ejercicio 2.3 de la parte
1 es:
 * [1, a, 2]  2 3
 * [2, b, 2] 1 3
 * [3, c, 2]] 1 2  */
int graph_print(FILE *pf, const Graph * g);

/* Lee de un flujo de entrada la información asociada a un grafo */
Status graph_readFromFile (FILE *fin, Graph *g);

#endif /* GRAPH_H */
```

Apéndice 4: Funciones privadas

En el fichero graph.c se incluirán las siguientes funciones privadas.

```
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;
    }

    // asigno valores al array con los indices de los nodos conectados
    for(i = 0; i< g->num_nodes; i++) {
        if (g->connections[index][i] == TRUE) {
            array[j] = i;
            j++;
        }
    }

    return array;
}
```

-

Apéndice 5: Función para leer un grafo desde un fichero

```
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;
}
```