

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 C, siguiendo las normas de programación establecidas.
- Compilar sin errores ni warnings incluyendo las banderas `-Wall` 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.
- **Semana 2:** Ejercicios 2.1 y 2.2.
- **Semana 3:** Ejercicio 2.3.

Cada profesor indicará en clase cómo hacer el seguimiento. Si se pedirán **entregas parciales semanales** por moodle o email, preguntas en clase, etc.

La entrega final se realizará a través de Moodle, siguiendo escrupulosamente las instrucciones indicadas en el enunciado 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`).

El fichero comprimido debe contener la siguiente organización de ficheros:

```
--- P1_Prog2_Gy_Pz/  
    |--- vertex.c  
    |--- vertex.h  
    |--- graph.c  
    |--- graph.h  
    |--- p1_e1.c  
    |--- p1_e2.c  
    |--- p1_e3.c
```

```
|--- compile.sh  
|--- g1.txt
```

Las fechas de entrega son las siguientes:

- Los alumnos de Evaluación Continua, la semana del **27 de febrero** hasta las **23:55 h.**
- Los alumnos de Evaluación Final, según lo especificado en la normativa.

EJERCICIO 1.

En este ejercicio definiremos el Tipo Abstracto de Dato (TAD) **Vertex** que se representará mediante un **id** (un **long int**), un **tag** (una cadena estática de caracteres) y un **estado** (de tipo **Label**).

El tipo de datos **Label** está definido en el fichero **vertex.h** (ver Apéndice 2). Otros tipos de datos útiles están definidos en el fichero **types.h**.

Definición del tipo abstracto de dato **Vertex** e implementación: Selección de su estructura de datos e implementación de su interfaz.

Para definir la estructura de datos necesaria para representar el TAD **Vertex** conforme la metodología de encapsulamiento vista en clase, debe incluirse la siguiente declaración en **vertex.h**:

```
typedef struct _Vertex Vertex;
```

Para implementar el TAD **Vertex** deberás:

- Crear el fichero **vertex.h** con la declaración del tipo **Vertex**, los prototipos de las funciones de la interfaz pública y las directivas **#include** necesarias.
- Definir en el fichero **vertex.c** la estructura de datos **_Vertex**:

```
#define TAG_LENGTH 64

struct _Vertex {
    long id;
    char tag[TAG_LENGTH];
    Label state;
};
```

- Implementar en el fichero **vertex.c** las funciones de la interfaz pública declaradas en el fichero **vertex.h**.
- Incluir en el fichero **vertex.c** las funciones privadas que consideres oportunas.

Comprobación de la corrección de la definición del tipo **Vertex** y de su interfaz.

Crea un programa en un fichero de nombre **p1_e1.c** cuyo ejecutable se llame **p1_e1**, y que realice las siguientes operaciones:

- Crear e inicializar dos vértices.
- El primero tendrá **id 10**, **tag one** y estado **WHITE**.
- El segundo tendrá **id 20**, **tag two** y estado **BLACK**.
- Imprimir ambos vértices y después un salto de línea.
- Comparar ambos vértices y mostrar un mensaje que diga el resultado.
- Imprimir el **tag** del segundo vértice.
- Copiar el primer vértice en un tercero.

- Imprimir el *id* del tercer vértice.
- Imprimir el primer y el tercer vértice y después un salto de línea.
- Comparar el primer y tercer vértices y mostrar un mensaje que diga el resultado.
- Liberar memoria.

Salida esperada:

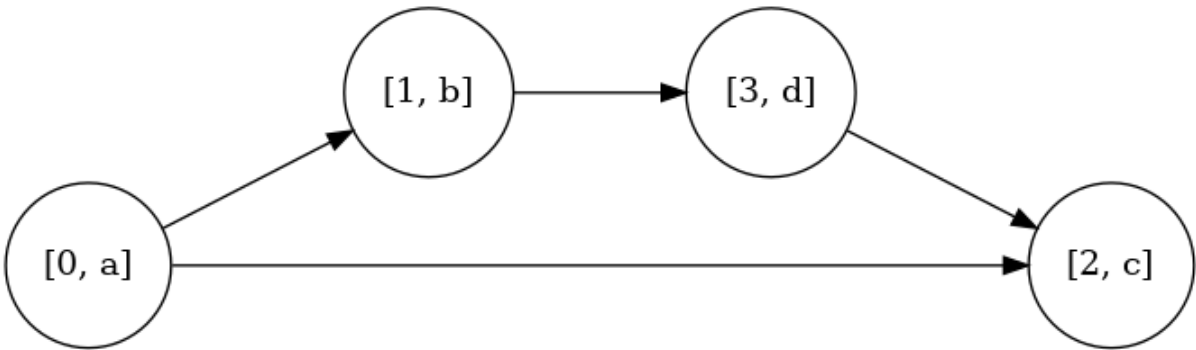
```
[10, one, 0][20, two, 1]
Equals? No
Vertex 2 tag: two
Vertex 3 id: 10
[10, one, 0][10, one, 0]
Equals? Yes
```

EJERCICIO 2.

En esta parte de la práctica se definirá el Tipo Abstracto de Datos (TAD) **Graph** como un conjunto de *vértices* junto con un conjunto de conexiones o *aristas* que definen una relación binaria entre los vértices.

2.1. Definición e implementación del TAD **Graph**: Estructura de datos e interfaz.

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



	Vertex A	Vertex B	Vertex C	Vertex D
Vertex A	0	1	1	0
Vertex B	0	0	0	1
Vertex C	0	0	0	0
Vertex D	0	0	1	0

Para implementar el TAD **Graph** deberás:

- Crear el fichero **graph.h** con la definición del tipo **Graph**, los prototipos de las funciones de la interfaz (ver Apéndice 3) y las directivas **#include** necesarias.
- Definir en el fichero **graph.c** la estructura de datos **_Graph**:

```
# define MAX_VTX 4096
struct _Graph {
    Vertex *vertices[MAX_VTX];
    Bool connections[MAX_VTX][MAX_VTX];
    int num_vertices;
    int num_edges;
};
```

- Implementar en el fichero `graph.c` las funciones de la interfaz declaradas en el fichero `graph.h`.
- Incluir en el fichero `graph.c` las funciones privadas que consideres oportunas.

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

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

- Inicializar un `Graph`.
- Insertar un vértice con `tag Madrid`, `id 111` y `state WHITE`, y verificar si la inserción se realizó correctamente.
- Insertar un vértice con `tag Toledo`, `id 222` y `state WHITE`, y verificar si la inserción se realizó correctamente.
- Insertar una arista desde el vértice con `id 222` hasta el vértice con `id 111`.
- Comprobar si el vértice con `id 111` está conectado con el vértice con `id 222` (ver salida esperada más abajo).
- Comprobar si el vértice con `id 222` está conectado con el vértice con `id 111` (ver salida esperada más abajo).
- Obtener e imprimir el número de conexiones desde el vértice con `id 111`.
- Obtener e imprimir el número de conexiones desde el vértice con `tag Toledo`.
- Obtener e imprimir la lista de conexiones del vértice con `tag Toledo`.
- Imprimir el grafo.
- Liberar todos los recursos y salir.

Salida esperada:

```
Inserting Madrid... result...: 1
Inserting Toledo... result...: 1
Inserting edge: 222 --> 111
111 --> 222? No
222 --> 111? Yes
Number of connections from 111: 0
Number of connections from Toledo: 1
Connections from Toledo: 111
Graph:
[111, Madrid, 0]:
[222, Toledo, 0]: [111, Madrid, 0]
```

2.3. Lectura de un grafo desde fichero.

Implementa la siguiente función de la interfaz de `Graph`, que permite cargar un grafo a partir de la información leída de un fichero de texto. El formato del fichero está definido en la documentación de la función.\

```
/**
 * @brief Reads a graph definition from a text file.
 */
```

```

* Reads a graph description from the text file pointed to by fin,
* and fills the graph g.
*
* The first line in the file contains the number of vertices.
* Then one line per vertex with the vertex description.
* Finally one line per connection, with the ids of the origin and
* the destination.
*
* For example:
*
* 4
* id:1 tag:Madrid
* id:2 tag:Toledo
* id:3 tag:Avila
* id:4 tag:Segovia
* 1 2
* 1 3
* 2 4
* 4 3
*
* @param fin Pointer to the input stream.
* @param g Pointer to the graph.
*
* @return OK or ERROR
*/
Status graph_readFromFile (FILE *fin, Graph *g);

```

Para probar la función crea el programa `p1_e3.c` que reciba como argumento el nombre de un fichero de texto, cree un `Graph` de acuerdo a la descripción contenida en ese fichero e imprima el grafo por pantalla. La ejecución de este programa debe funcionar sin problemas, incluyendo una gestión adecuada de la memoria (*valgrind* no debería mostrar fugas).

A continuación se muestra un ejemplo de fichero de datos de entrada:\

```

4
id:100 tag:Madrid
id:200 tag:Toledo
id:300 tag:Avila
id:400 tag:Segovia
100 200
100 300
200 400
400 300

```

Y la correspondiente salida:

```

[1, Madrid, 0]: [2, Toledo, 0] [3, Avila, 0]
[2, Toledo, 0]: [4, Segovia, 0]

```

```
[3, Avila, 0]:  
[4, Segovia, 0]: [3, Avila, 0]
```