PRÁCTICA 3: Tipos Abstracto de Datos Cola (Queue) y Lista (List)

OBJETIVOS

- Aprender a utilizar el TAD Cola para resolver problemas.
- Aprender a elegir la estructura de datos apropiada para implementar el TAD Cola.
- Aprender a elegir la estructura de datos apropiada para implementar el TAD Lista.
- Codificar sus primitivas y utilizarlas en un programa principal.

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. Prueba y aplicación de la biblioteca de Cola a un problema concreto.
- **Semana 2:** Ejercicio 2. Implementación y prueba del TAD Cola.
- **Semana 3:** Ejercicio 3. Implementación y prueba del TAD Lista.

Cada profesor indicará en clase cómo realizar las **entregas parciales semanales**: papel, e-mail, Moodle, etc, en caso de que las haya.

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_EDAT_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: `P3_EDAT_G2161_P05.zip`).

El fichero comprimido debe contener la siguiente organización de ficheros:

```
--- P3_EDAT_Gy_Pz/
 |--- delivery.c
 |--- delivery.h
 |--- file_utils.c
 |--- file_utils.h
 |--- graph.c
 |--- graph.h
 |--- libqueue.a
 |--- list.c
 |--- list.h
 |--- Makefile
 |--- p3_e1.c
 |--- p3_e2b.c
 |--- p3_e3.c
 |--- queue.h
 |--- queue.c
 |--- stack.c
 |--- stack.h
 |--- types.h
 |--- vertex.c
 |--- vertex.h
```

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

- Los alumnos de Evaluación Continua, el 17 de abril hasta las 23.55 horas.
- Los alumnos de Evaluación Final, según lo especificado en la normativa.

Ejercicio 1.

En este ejercicio vamos a simular el funcionamiento de un plan de reparto de vacunas por distintas ciudades de una comunidad autónoma: los ayuntamientos que lo desean (que se van a encapsular en el TAD `Vertex`, con su correspondiente id y nombre), solicitan las vacunas (insertando su solicitud en una cola) y después se procede a repartir dichas vacunas entre las ciudades que lo hayan solicitado, por orden de espera según su solicitud en la cola.

Para ello, se va a definir un nuevo TAD `Delivery` que incluya un nombre (en este caso, de la comunidad autónoma), el nombre del producto que se va a repartir (en este ejemplo, distintas vacunas) y, por último, se almacenarán las solicitudes de vacunas en una cola para poder procesarlas en el orden en el que llegaron. La nueva estructura de datos tendrá la siguiente forma:

```
struct _Delivery {
  char *name;
  char *product_name;
  Queue *plan;
};
```

La mayoría de las funciones de dicho TAD ya están implementadas en el módulo *Delivery*, y toda la información relativa a ellas se puede leer en *delivery.h*. Sin embargo, las siguientes funciones faltan por implementar:

1. La función `delivery_add` añadirá una nueva solicitud en el plan del reparto (haciendo uso de las funciones del TAD Queue, descritas en *queue.h*), siendo esta de tipo genérico `void *`. La función irá indicando por pantalla los datos del elemento añadido, tal y como se aprecia en el ejemplo mostrado después.

```
/**
 * @brief Adds a product to a delivery.
 *
 * @param pf File descriptor where the added product information will
    be printed.
 * @param d Delivery pointer
 * @param p A pointer to a generic product
 * @param f Function pointer to print elements in the delivery plan
 *
 * @return Returns OK or ERROR.
 */
Status delivery_add (FILE *pf, Delivery *d, void *p, p_element_print f
   );
```

2. La función `delivery_run_plan` mostrará primero el contenido actual de la cola e irá

procesando las solicitudes provenientes de las ciudades en el orden en que llegaron, extrayendo de la cola cada solicitud para atenderla y mostrando un mensaje para indicar que se está procediendo al reparto en esa ciudad (ver ejemplo después). Para ello, la función irá extrayendo los elementos de la cola, ejecutando así el plan de reparto, y los irá liberando según los saca.

```
/**
 * @brief Simulates running the plan associated to a delivery.
 *
 * @param pf File descriptor where the simulation will be shown.
 * @param d Delivery pointer
 * @param fprint Function pointer to print elements in the delivery
    plan
 * @param ffree Function pointer to free elements in the delivery plan
 *
 * @return Returns OK or ERROR.
 */
Status delivery_run_plan (FILE *pf, Delivery *d, p_element_print
   fprint, p_element_free ffree);
```

Ejercicio 1a.

Implementar las funciones previas en el TAD `Delivery`.

Observación: Se debe tener en cuenta que el TAD `Delivery` almacena en la cola solicitudes de tipo genérico `void *`.

Ejercicio 1b.

Las dos funciones se probarán en un programa principal de nombre *p3_e1.c*. El programa, a partir de una lista de solicitudes de vacunas, montará un `Delivery` y ejecutará un plan de reparto. Para ello,recibirá un fichero por argumento donde la primera línea contendrá el nombre de la comunidad y del producto solicitado; la siguiente línea indicará cuántos productos se van a insertar en el plan de reparto, y, para cada uno de ellos, se indicará la descripción de la ciudad (en formato `Vertex`).

Para leerlo, se puede implementar una función que construya un `Delivery` (añadiendo la información de la comunidad autónoma, el tipo de vacuna y la cola de peticiones provenientes de las ciudades), a partir de un fichero de texto. Esta función será una función privada del fichero donde se encuentre el programa principal.

```
/**
 * @brief Builds a delivery from the information read on the file.
 *
 * @param pf File descriptor. The file should contains the following
    information:
 *      Firt line: delivery_name product_name
 *      Second line: number of cities/vertices
 *      Following lines: vertices description
 *
 * @return A pointer to the fullfilled delivery. If there have been
    errors returns NULL.
 */
Delivery* build_delivery(FILE * pf);
```

Si, por ejemplo, se recibe el fichero *requests.txt* cuyo contenido es:

```
castilla_y_leon vacuna_pfizer
4
id:200 tag:Valladolid
id:100 tag:Burgos
id:400 tag:Salamanca
id:550 tag:Leon
```

La salida obtenida por el programa sería:

```
Adding: [200, Valladolid, 0, 0] to delivery: castilla_y_leon
Adding: [100, Burgos, 0, 0] to delivery: castilla_y_leon
Adding: [400, Salamanca, 0, 0] to delivery: castilla_y_leon
Adding: [550, Leon, 0, 0] to delivery: castilla_y_leon

Running delivery plan for queue:
[200, Valladolid, 0, 0][100, Burgos, 0, 0][400, Salamanca, 0, 0][550,
    Leon, 0, 0]
Delivering vacuna_pfizer requested by castilla_y_leon to [200,
   Valladolid, 0, 0]
Delivering vacuna_pfizer requested by castilla_y_leon to [100, Burgos
   , 0, 0]
Delivering vacuna_pfizer requested by castilla_y_leon to [400,
   Salamanca, 0, 0]
Delivering vacuna_pfizer requested by castilla_y_leon to [550, Leon,
   0, 0]
```

Ejercicio 2.

Ejercicio 2a.

En este ejercicio, se va a realizar una implementación propia del TAD `Queue`, pero usando la variante donde el front y el rear son de tipo puntero, es decir, en un fichero *queue.c* se implementarán todas las funciones definidas en *queue.h* usando la siguiente definición del TAD:

```
#define MAX_QUEUE 100

struct _Queue{
  void *data[MAX_QUEUE];
  void **front;
  void **rear;
}
```

Para probar este ejercicio, se debe generar un ejecutable de nombre `p3_e2a` a partir del programa diseñado en el ejercicio 1b.

Observación: no hace falta volver a codificar nada para los programas principales, simplemente enlazar con la nueva implementación de Queue en lugar de con la biblioteca *libqueue.a*.

Ejercicio 2b.

En los ejercicios anteriores de esta práctica se ha visto cómo realizar un plan de entregas de vacunas por una serie de ciudades. En este caso nos vamos a centrar en detectar caminos para llegar de una ciudad a otra. Para ello podemos utilizar, como ya hicimos en la Práctica 2, el algoritmo DFS, definido utilizando una pila. El camino encontrado por la búsqueda en profundidad del grafo puede compararse con el algoritmo de búsqueda en anchura de ese grafo.

El algoritmo de búsqueda en anchura o BFS se diferencia con DFS en que se van visitando todos los nodos de un mismo nivel de profundidad antes de pasar al siguiente. Para conseguir este objetivo se debe utilizar un cola en lugar de una pila. Este sencillo cambio es suficiente para ver una implementación de este famoso algoritmo.

En este ejercicio, la primera tarea consiste en añadir el algoritmo BFS sobre el grafo, para lo cual se añadirá al fichero `graph.c` la siguiente función:

```
Status graph_breathSearch (Graph *g, long from_id, long to_id)
```

Para probarla, se definirá un programa principal *p3_e2a* que cargue un grafo de ciudades de un fichero, que se recibe como argumento, por ejemplo el fichero *city_graph*:

```
7
id:100 tag:Madrid
id:200 tag:Toledo
id:300 tag:Segovia
id:400 tag:Valladolid
id:500 tag:Burgos
id:600 tag:Palencia
id:700 tag:Santander
100 200
100 300
300 400
300 500
400 600
500 600
500 700
600 700
```

El programa recibirá también por argumento el id del vértice desde el cual se desea iniciar el camino y el id del vértice de llegada.

Posteriormente se llamará a la función que calcula y pinta el resultado del algoritmo DFS, así como el del algoritmo BFS, de forma que se puedan comparar ambos.

En el siguiente ejemplo se ve el resultado tras ejecutar con el id de partida `100` y el de llegada `700`:

```
Input graph:
[100, Madrid, 0, 0]: [200, Toledo, 0, 1] [300, Segovia, 0, 2]
[200, Toledo, 0, 1]:
[300, Segovia, 0, 2]: [400, Valladolid, 0, 3] [500, Burgos, 0, 4]
[400, Valladolid, 0, 3]: [600, Palencia, 0, 5]
[500, Burgos, 0, 4]: [600, Palencia, 0, 5] [700, Santander, 0, 6]
[600, Palencia, 0, 5]: [700, Santander, 0, 6]
[700, Santander, 0, 6]:
--------DFS------------

From vertex id: 100
To vertex id: 700
Output:
[100, Madrid, 1, 0]
[300, Segovia, 1, 2]
[500, Burgos, 1, 4]
[700, Santander, 1, 6]
```

```
--------BFS------------

From vertex id: 100
To vertex id: 700
Output:
[100, Madrid, 1, 0]
[200, Toledo, 1, 1]
[300, Segovia, 1, 2]
[400, Valladolid, 1, 3]
[500, Burgos, 1, 4]
[600, Palencia, 1, 5]
[700, Santander, 1, 6]
```

Otro ejemplo se muestra tras ejecutar con un id de partida `100` y `400` de llegada:

```
Input graph:
[100, Madrid, 0, 0]: [200, Toledo, 0, 1] [300, Segovia, 0, 2]
[200, Toledo, 0, 1]:
[300, Segovia, 0, 2]: [400, Valladolid, 0, 3] [500, Burgos, 0, 4]
[400, Valladolid, 0, 3]: [600, Palencia, 0, 5]
[500, Burgos, 0, 4]: [600, Palencia, 0, 5] [700, Santander, 0, 6]
[600, Palencia, 0, 5]: [700, Santander, 0, 6]
[700, Santander, 0, 6]:
--------DFS------------

From vertex id: 100
To vertex id: 400
Output:
[100, Madrid, 1, 0]
[300, Segovia, 1, 2]
[500, Burgos, 1, 4]
[700, Santander, 1, 6]
[600, Palencia, 1, 5]
[400, Valladolid, 1, 3]

--------BFS------------

From vertex id: 100
To vertex id: 400
Output:
[100, Madrid, 1, 0]
[200, Toledo, 1, 1]
[300, Segovia, 1, 2]
[400, Valladolid, 1, 3]
```

¿Qué diferencias se ven entre ambos algoritmos y por qué?

Ejercicio 3.

En este ejercicio, se pide implementar en el fichero *list.c* las primitivas del TAD Lista que se definen en el archivo de cabecera *list.h*. La estructura de datos elegida para implementar este TAD consistirá en una estructura con un único campo: el puntero al **último** elemento de la lista (es decir, se implementará una lista circular). Por su parte, cada elemento de la lista será una estructura capaz de almacenar el dato y un puntero al siguiente elemento de la lista. Dicha estructura se muestra a continuación:

```
/* In list.h */
typedef struct _List List;

/* In list.c */
typedef struct _NodeList {
    void* data;
    struct _NodeList *next;
} NodeList;

struct _List {
    NodeList *last;
};
```

Al igual que en implementaciones previas, al insertar un elemento en una lista no se reserva memoria para el dato a insertar ni se hace una copia, simplemente se almacena en el campo de datos correspondiente la referencia (el puntero) al elemento que se desea insertar.

Comprobación del TAD

Para comprobar que se ha implementado correctamente el TAD, se trabajará de nuevo con el fichero de notas de la asignatura EDAT de la práctica anterior. Se leerán notas del fichero de manera que la primera, tercera, quinta, etc. se insertarán por el final de una lista, mientras que las demás (segunda, cuarta, sexta, etc.) se insertarán por el principio. Cuando se ha terminado de insertar, el programa irá extrayendo los elementos de la lista de uno en uno: la primera mitad desde el principio y la segunda mitad por el final. Cada vez que se extraiga una nota de la lista (ya sea del principio o del final) se insertará en orden en otra lista de acuerdo a una función de comparación. El orden será creciente o decreciente, según indique un argumento recibido en línea de comandos (por terminal): 1 indicará que los elementos se han de insertar de manera creciente en la lista y -1 indicará orden decreciente.

Para trabajar con los floats se pueden utilizar las funciones de la práctica anterior, especificadas

en *file_utils.h*.

Ejemplo

```
$ ./p3_e3 grades.txt 1
SIZE: 10
4.200000 9.750000 8.000000 9.500000 6.500000 5.000000 7.300000
   2.500000 9.200000 9.500000
Finished inserting. Now we extract from the beginning and insert in
   order:
4.200000 9.750000 8.000000 9.500000 6.500000
Now we extract from the end and insert in order:
9.500000 9.200000 2.500000 7.300000 5.000000
SIZE: 10
2.500000 4.200000 5.000000 6.500000 7.300000 8.000000 9.200000
   9.500000 9.500000 9.750000

$ ./p3_e3 grades.txt -1
SIZE: 10
4.200000 9.750000 8.000000 9.500000 6.500000 5.000000 7.300000
   2.500000 9.200000 9.500000
Finished inserting. Now we extract from the beginning and insert in
   order:
4.200000 9.750000 8.000000 9.500000 6.500000
Now we extract from the end and insert in order:
9.500000 9.200000 2.500000 7.300000 5.000000
SIZE: 10
9.750000 9.500000 9.500000 9.200000 8.000000 7.300000 6.500000
   5.000000 4.200000 2.500000
```