Programación II, 2015-2016
Escuela Politécnica Superior, UAM

Práctica 3: TAD Cola y TAD Lista

OBJETIVOS

- Familiarización con los TADs Cola y Lista, conocer su funcionamiento y potencial.
- Implementación en C de los TADs Cola y Lista.
- Utilización de los TADs Cola y Lista para resolver algunos problemas de programación en el contexto de la resolución de laberintos.

NORMAS

Igual que en prácticas anteriores, los ***programas*** que se entreguen deben:

- Estar escritos en ***ANSI C,*** siguiendo las **normas de programación** establecidas.
- Compilar sin errores ni *warnings*, estableciendo 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.
- La ***memoria*** que se entregue debe elaborarse sobre el modelo propuesto y entregado con la práctica.

PLAN DE TRABAJO

Semana 1: código de P3_E1 y parte de P3_E2. Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.
Semana 2: terminar P3_E2 y código de P3_E3.
Semana 3: código de P3_E4.
Semana 4: 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 subir 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 11 de abril*** *(cada grupo puede realizar la entrega hasta las 23:30 h. de la noche anterior a su clase de prácticas*), excepto los grupos del martes (2101, 2102, 2111) y del viernes (2163) que entregan la semana del 18 de abril para recuperar las clases perdidas (el viernes 11 de marzo no hay clase, y el martes 29 de marzo hay docencia de lunes).
- Los alumnos de ***Evaluación Final,*** según lo especificado en la normativa.

PARTE 1. EJERCICIOS

Ejercicio 1 (P3_E1). Colas de puntos

1. Definición del TAD Cola (Queue) circular. Implementación: selección de estructura de datos e implementación de primitivas.

Se pide implementar en el fichero queue.c las primitivas del TAD Cola que se definen en el archivo de cabecera queue.h (ver Apéndice 1) cuyo tipo de dato a guardar en la misma se define en el fichero elequeue.h (ver Apéndice 2). Los tipos Status y Bool son los definidos en types.h. La constante MAXQUEUE indica el tamaño máximo de la cola. Su valor debe permitir desarrollar los ejercicios propuestos.

La estructura de datos elegida para implementar el TAD COLA consiste en un array de elementos de tipo genérico y referencias a los elementos situados en la cabeza y al final de la cola. Dicha estructura se muestra a continuación:

```
/* En queue.h */
typedef struct _Queue Queue;

/* En queue.c */
struct _Queue {
        EleQueue** head;
        EleQueue** end;
        EleQueue* item[MAXQUEUE];
};
```

2. Encapsulación del tipo del TAD Cola: el TAD EleQueue

Para esta práctica, la cola debe almacenar puntos. Por tanto, se necesitará un TAD similar al codificado en el módulo elestack-point.c (y su correspondiente elestack.h) de la práctica anterior. Este TAD se llamará **EleQueue** y se codificará en un módulo C en ficheros de nombre elequeue-point.c y elequeue.h

En el Apéndice 2 se encuentra el contenido del fichero elequeue.h y la descripción de las funciones que hay que implementar.

3. Comprobación de la corrección de la definición del tipo Queue y sus funciones.

Con el objetivo de evaluar el funcionamiento de las funciones anteriores, se desarrollará un programa **p3_e1.c** que trabajará con colas de puntos. Este programa recibirá como argumento un fichero, cuya primera línea indicará el número de puntos que se deben leer (siguiendo el formato que podéis ver en la salida esperada), y los introducirá uno a uno en una cola. A continuación, irá sacando de dicha cola la mitad de los puntos introducidos y los introducirá en una cola, introduciendo la otra mitad en una cola distinta. Durante el proceso el contenido de las tres colas se irá imprimiendo como se muestra a continuación en la salida esperada:

```
> cat puntos.txt
3
1,1,+
2,2,v
3,3,

> ./p3_e1 puntos.txt
Cola 1: Queue vacia.
Cola 2: Queue vacia.
Cola 3: Queue vacia.

Cola 1: Cola con 1 elementos:
[(1, 1): +]
Cola 2: Queue vacia.
Cola 3: Queue vacia.

Cola 1: Cola con 2 elementos:
[(1, 1): +]
[(2, 2): v]
Cola 2: Queue vacia.
Cola 3: Queue vacia.

Cola 1: Cola con 3 elementos:
[(1, 1): +]
[(2, 2): v]
[(3, 3):  ]
Cola 2: Queue vacia.
Cola 3: Queue vacia.
<<<Pasando la primera mitad de Cola 1 a Cola 2
Cola 1: Cola con 2 elementos:
[(2, 2): v]
[(3, 3):  ]
Cola 2: Cola con 1 elementos:
[(1, 1): +]
Cola 3: Queue vacia.
<<<Pasando la segunda mitad de Cola 1 a Cola 3
Cola 1: Cola con 1 elementos:
[(3, 3):  ]
Cola 2: Cola con 1 elementos:
[(1, 1): +]
Cola 3: Cola con 1 elementos:
[(2, 2): v]
Cola 1: Queue vacia.
Cola 2: Cola con 1 elementos:
[(1, 1): +]
Cola 3: Cola con 2 elementos:
[(2, 2): v]
[(3, 3):  ]
Cola 1: Queue vacia.
Cola 2: Cola con 1 elementos:
[(1, 1): +]
Cola 3: Cola con 2 elementos:
[(2, 2): v]
[(3, 3):  ]
```

Ejercicio 2 (P3_E2). Resolución de un laberinto usando una cola.

En la práctica anterior (P2_E3) se ha realizado esta funcionalidad apoyándose en una pila. Se trata de sustituir en ese algoritmo la pila por una cola. De esta forma se cambiará el orden en el que se exploran los movimientos pendientes. En este ejercicio, por tanto, se trata de que se comprueben esas diferencias.

El programa implementado se escribirá en un fichero de nombre **p3_e2.c** y los argumentos de entrada así como el formato de salida serán los mismos que los del ejercicio P2_E3.

Ejercicio 3 (P3_E3). Resolución de laberintos para según diferentes estrategias

En este mismo contexto, se pretende generalizar el estudio de los laberintos mediante la comprobación de la eficiencia de diferentes estrategias.

Entenderemos por **estrategia** simplemente un orden entre los movimientos posibles del laberinto. Nos limitaremos a estrategias que utilicen exactamente 4 movimientos y sólo uno cada vez. Por lo tanto, una estrategia no es más que una permutación en el conjunto de 4 movimientos.

Para su implementación en C se utilizará un vector que podrá contener 4 elementos de tipo Move según, por ejemplo, esta declaración

```
Move strategy[] = { RIGHT, LEFT, UP, DOWN };
```

Cuando se desee probar más de una estrategia se seguirá este esquema, utilizando una matriz 2D como en el siguiente ejemplo:

```
Move strategies [4][4] = {
                          { RIGHT, LEFT, UP, DOWN },
                          { DOWN, RIGHT, LEFT, UP },
                          { UP, DOWN, RIGHT, LEFT },
                          { LEFT, UP, DOWN, RIGHT }
                         };
```

Para completar esta parte hay que seguir los siguientes pasos:

- Crear un módulo propio para las funciones relacionadas con la resolución del laberinto. Este módulo se llamará solve_mazes (.c y .h).

- Implementar en dicho módulo las funciones

  ```
  int mazesolver_stack(const char* maze_file, const Move strat[4]);
  ```

  y

  ```
  int mazesolver_queue(const char* maze_file, const Move strat[4]);
  ```

  Las cuales devuelven un número positivo si el laberinto tiene solución o uno negativo si no, utilizando en el primer caso una pila para resolver el laberinto (como en P2_E3) o una cola en el segundo (como en P3_E2). Este número indicará **el número de movimientos que se realiza para resolver cada laberinto** según la estrategia y el TAD (pila o cola), el cual será mayor o igual a la longitud del camino óptimo.

  Nótese que estas funciones ahora reciben un vector de movimientos, de manera que al explorar los vecinos se tendrá que usar el orden indicado en este vector, y no como se hubiera hecho en los ejercicios previos.

- Por último, implementar una función que llame a las anteriores con una matriz de movimientos:

```
void mazesolver_run(const char* maze_file, const Move strat[][4], const int num_strategies);
```

Esta función probará todas las estrategias que se pasen como argumento usando tanto una pila como una cola, llamando a las funciones descritas previamente.

Para entender la declaración de esta función es necesario recordar las peculiaridades del uso de matrices 2D en el lenguaje de programación.

Tanto **la evolución del proceso como la solución final se mostrarán por la salida estándar** (`stdout`):

- Cuando se comienza la prueba de una estrategia con el siguiente mensaje: "ESTRATEGIA 0213" (correspondiente al array {RIGHT,LEFT,UP,DOWN}).
- Cada vez que se ha visitado un nodo se debe mostrar el TAD con movimientos pendientes y el estado actual del laberinto.
- Al finalizar, se indica si ha habido solución ("SALIDA ENCONTRADA" o "SALIDA NO ENCONTRADA") así como el número de movimientos realizados.

Ejercicio 4 (P3_E4). Lista de enteros incluyendo inserción en orden

1. Definición del TAD Lista (List). Implementación: selección de estructura de datos e implementación de primitivas.

Se pide implementar en el fichero list.c las primitivas del TAD Lista que se definen en el archivo de cabecera list.h el cual, a su vez, hace uso de los elementos de elelist.h. En esta ocasión, la estructura de datos elegida para implementar el TAD Lista consistirá en una estructura con un campo capaz de almacenar el dato y un apuntador al siguiente elemento de la lista. Dicha estructura se muestra a continuación:

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

/* En list.c */
typedef struct _Node {
    EleList* data;
    struct _Node *next;
} Node;
struct _List {
    Node *node;
};
```

Las primitivas a implementar en este apartado están indicadas en los apéndices 3 y 4.

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

Con el objetivo de evaluar el funcionamiento de las primitivas anteriores, se desarrollará un programa **p3_e4.c** que trabajará con listas de enteros. Este programa recibirá como argumento el nombre del fichero que contendrá los números, todos a continuación unos de otros en una misma línea y separados por espacios.

Este fichero se leerá una vez: por cada número que se lea del fichero, si es par se introducirán por el principio de la lista y si es impar, por el final. En cada inserción irá imprimiendo por pantalla el estado de la lista en ese momento. Cuando se ha terminado de insertar, el programa irá extrayendo los números de la lista de uno en uno. La primera mitad los extraerá por el principio y la segunda mitad por el final. De nuevo, tras cada extracción se imprimirá el elemento extraído y la lista en el estado en que se encuentre. Además, a la vez que se extrae, se insertará en otra lista pero de manera ordenada y se imprimirá el estado de esta segunda lista.

Un ejemplo de la salida esperada se puede ver a continuación:

```
> cat datos.txt
12
11
10
9
8
7
6
5
4
3
2
1

> ./p3_e4 datos.txt
Lista con 12 elementos:
[2]
[4]
[6]
[8]
[10]
[12]
[11]
[9]
[7]
[5]
[3]
[1]

Lista con 12 elementos:
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
```

Ejercicio 5 (P3_E5). Lista de soluciones con diferentes estrategias

Como se ha podido analizar en ejercicios anteriores, diferentes estrategias obtienen soluciones distintas (distintos caminos óptimos pero también distinto número de movimientos para llegar a soluciones equivalentes). Se trata en este ejercicio de extender la funcionalidad de los anteriores para **guardar en una lista las soluciones ordenadas según su longitud** y poder comparar cuál de ellas resulta más eficiente.

Para ello, se seguirán los siguientes pasos:

- Definición del **TAD Solution**. Este TAD se puede implementar de distintas formas, pero por lo menos almacenará el array de estrategias utilizado, el número de movimientos empleado y si se ha utilizado una cola o una pila para la resolución; otros atributos que puede contener son el camino obtenido, longitud del camino óptimo, etc.
- Implementación de primitivas del TAD Solution y del correspondiente **EleList** que contenga un puntero a Solution. La lista concreta de primitivas se espera que sea similar a las de otros TADs que ya hemos visto, siguiente un formato y notación similar, por lo que no se incluye en este enunciado.
- Definir una nueva función en el módulo solve_mazes que tenga una funcionalidad parecida a la que se definió en P3_E3 pero que en este caso, además de resolver los laberintos, almacene esos datos en el TAD Solution y lo vaya insertando en una lista en orden (ordenado por el número de movimientos, de menor a mayor).
- Escribir un programa main de nombre **p3_e5.c** con el mismo formato que en P3_E3 pero donde, al final de la ejecución, se muestre el estado de la lista ordenada de forma que se pueda analizar qué estrategias obtienen mejores resultados según el TAD (pila, cola) utilizado. Una salida sugerida para este programa se puede ver a continuación:

```
...
Lista con 8 elementos:
[2301, pila, 11]
[3012, pila, 11]
[1230, pila, 14]
[0123, pila, 15]
[0123, cola, 18]
[1230, cola, 18]
[2301, cola, 18]
[3012, cola, 18]
...
```

PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

1. Suponed que en lugar de la implementación del TAD Pila elaborado para la práctica 2 con memoria estática, se desea hacer una implementación basada en listas. Indicad cómo influye el uso de esta nueva implementación desde los puntos de vista sintáctico, semántico y uso de memoria.

2. Suponed que se disponen de dos implementaciones diferentes del TAD lista (no ordenada). La primera de ellas es la que se ha elaborado en la presente práctica con memoria dinámica. La segunda es una implementación con memoria estática, donde se emplea un array para almacenar las direcciones a los elementos que contienen la lista, y una referencia (puntero) a la posición de cabeza de la misma.

Si se desea eliminar el elemento que se encuentra justo en la mitad de la lista, indique los pasos que debe realizarse en cada caso y calcule el número de accesos a memoria en cada uno de los mismos.

Si esta operación tuviera que repetirse con mucha asiduidad, ¿por cuál de las dos implementaciones optaría?

PARTE 2.B. MEMORIA SOBRE LA PRÁCTICA

1. Decisiones de diseño

Explicad las decisiones de diseño y alternativas que se han considerado durante la práctica para cada uno de los ejercicios propuestos.

2. Informe de uso de memoria

Elaborad un informe sobre la salida del análisis de memoria realizado por la herramienta memcheck de valgrind para cada uno de los ejercicios propuestos. Debe tenerse en cuenta que **en la ejecución del código entregado no deben producirse ningún aviso ni alerta sobre uso inapropiado de memoria**.

3. Conclusiones finales

Se reflejará al final de la memoria unas breves conclusiones sobre la práctica, indicando los beneficios que os ha aportado la práctica, qué aspectos vistos en las clases de teoría han sido reforzados, qué apartados de la práctica han sido más fácilmente resolubles y cuáles han sido los más complicados o no se han podido resolver, qué aspectos nuevos de programación se han aprendido, dificultades encontradas, etc.

Apéndice 1: queue.h

```
typedef struct _Queue Queue;


/**----------------------------------------------------------------
Inicializa la cola: reserva memoria para ella e inicializa todos sus elementos. Es importante que no se reserve memoria para los elementos de la cola.
----------------------------------------------------------------*/
Queue* queue_ini();
/**----------------------------------------------------------------
Libera la cola y todos sus elementos.
----------------------------------------------------------------*/
void queue_free(Queue *q);
/**----------------------------------------------------------------
Comprueba si la cola está vacía.
----------------------------------------------------------------*/
Bool queue_isEmpty(const Queue *q);
/**----------------------------------------------------------------
Comprueba si la cola está llena.
----------------------------------------------------------------*/
Bool queue_isFull(const Queue* queue);
/**----------------------------------------------------------------
Inserta un elemento en la cola realizando para ello una copia del mismo, reservando memoria nueva para él.
----------------------------------------------------------------*/
Queue* queue_insert(Queue *q, const EleQueue* pElem);
/**----------------------------------------------------------------
Extrae un elemento de la cola. Es importante destacar que la cola deja de apuntar a este elemento por lo que la gestión de su memoria debe ser coherente: devolver el puntero al elemento o devolver una copia liberando el elemento en la cola.
----------------------------------------------------------------*/
EleQueue* queue_extract(Queue *q);
/**----------------------------------------------------------------
Devuelve el número de elementos de la cola.
----------------------------------------------------------------*/
int queue_size(const Queue *q);
/**----------------------------------------------------------------
Imprime toda la cola, devolviendo el número de caracteres escritos.
----------------------------------------------------------------*/
int queue_print(FILE *pf, const Queue *q);
```

Apéndice 2: elequeue.h

```
typedef struct _EleQueue EleQueue;

/* Inicializa un EleQueue reservando memoria e inicializando todos sus elementos. */
EleQueue* elequeue_ini();
/* Libera un EleQueue y todos sus elementos. */
void elequeue_free(EleQueue * ele);

/* Modifica los datos de un EleQueue. */
EleQueue* elequeue_setInfo(EleQueue * e, void* p);
/* Devuelve el contenido de un EleQueue. */
void* elequeue_getInfo(EleQueue * e);

/* Copia reservando memoria un EleQueue. */
EleQueue* elequeue_copy(const EleQueue * src);

/* Devuelve un número positivo, negativo o cero según si ele1 es mayor, menor o igual que ele2. */
int elequeue_cmp(const EleQueue * ele1, const EleQueue * ele2);

/* Imprime un EleQueue devolviendo el número de caracteres escritos. */
int elequeue_print(FILE* pf, const EleQueue * ele);
```

Apéndice 3: list.h

```
typedef struct _List List;

/* Inicializa la lista reservando memoria e inicializa todos sus elementos. */
List* list_ini();
/* Libera la lista y todos sus elementos. */
void list_free(List* list);

/* Inserta al principio de la lista realizando una copia del elemento. */
List* list_insertFirst(List* list, const EleList *elem);
/* Inserta al final de la lista realizando una copia del elemento. */
List* list_insertLast(List* list, const EleList *elem);
/* Inserta en orden en la lista realizando una copia del elemento. */
List* list_insertInOrder(List *list, const EleList *pElem);

/* Extrae del principio de la lista realizando una copia del elemento almacenado en dicho nodo. */
EleList* list_extractFirst(List* list);
/* Extrae del final de la lista realizando una copia del elemento almacenado en dicho nodo. */
EleList* list_extractLast(List* list);

/* Comprueba si una lista está vacía o no. */
Bool list_isEmpty(const List* list);

/* Devuelve el tamaño de una lista. */
int list_size(const List* list);

/* Imprime una lista devolviendo el número de caracteres escritos. */
int list_print(FILE *fd, const List* list);
```

Apéndice 4: elelist.h

```
typedef struct _EleList EleList;

/* Inicializa un EleList reservando memoria e inicializando todos sus elementos. */
EleList* elelist_ini();
/* Libera un EleList y todos sus elementos. */
void elelist_free(EleList * ele);

/* Modifica los datos de un EleList . */
EleList* elelist_setInfo(EleList * e, void* p);
/* Devuelve el contenido de un EleList . */
void* elelist_getInfo(EleList * e);

/* Copia reservando memoria un EleList . */
EleList* elelist_copy(const EleList * src);

/* Devuelve un número positivo, negativo o cero según si ele1 es mayor, menor o igual que ele2. */
int elelist_cmp(const EleList * ele1, const EleList * ele2);

/* Imprime un EleList devolviendo el número de caracteres escritos. */
int elelist_print(FILE* pf, const EleList * ele);
```