

Programación II, 2016-2017

Escuela Politécnica Superior, UAM

Práctica 3: TAD Cola y TAD Lista

OBJETIVOS

- Familiarización con los TADs Cola y Lista, aprendiendo su funcionamiento y potencial.
- Implementación en C de los TADs Cola y Lista.
- Utilización de los TADs Cola y Lista para resolver problemas.

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. Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.

Semana 2: código de P3_E2 y P3_E3.

Semana 3: 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 18 de abril** (cada grupo puede realizar la entrega hasta las 23:30 h. de la noche anterior a su clase de prácticas).
- Los alumnos de **Evaluación Final**, según lo especificado en la normativa.

PARTE 1. EJERCICIOS

Ejercicio 1 (P3 E1). Cola (genérica) de nodos

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) utilizando punteros a función, como se hizo en el último ejercicio de la P2. 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 punteros genéricos (void*) y referencias a los elementos situados en la cabeza y al final de la cola, así como punteros a las funciones de destrucción, copia e impresión de los elementos (recuerda que se puede usar un módulo **functions.c** como en la práctica anterior). Dicha estructura se muestra a continuación:

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

/* En queue.c */
struct _Queue {
    void** head;
    void** end;
    void* item[MAXQUEUE];
    destroy_elementqueue_function_type destroy_element_function;
    copy_elementqueue_function_type copy_element_function;
    print_elementqueue_function_type print_element_function;
};
```

2. 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 nodos. Este programa recibirá como argumento un fichero, cuya primera línea indicará el número de nodos que se deben leer (siguiendo el formato que podéis ver a continuación), y los introducirá uno a uno en una cola. A continuación, irá sacando de dicha cola la mitad de los nodos introducidos y los introducirá en otra cola, introduciendo la otra mitad en una tercera cola. Durante el proceso el contenido de las tres colas se irá imprimiendo como se muestra a continuación en la salida esperada:

```
> cat nodos.txt
3
1 uno
2 dos
3 tres

> ./p3_e1 nodos.txt
Cola 1: Queue vacía.
Cola 2: Queue vacía.
Cola 3: Queue vacía.
```

```
Cola 1: Cola con 1 elementos:
[1, uno]
Cola 2: Queue vacia.
Cola 3: Queue vacia.

Cola 1: Cola con 2 elementos:
[1, uno]
[2, dos]
Cola 2: Queue vacia.
Cola 3: Queue vacia.

Cola 1: Cola con 3 elementos:
[1, uno]
[2, dos]
[3, tres]
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, dos]
[3, tres]
Cola 2: Cola con 1 elementos:
[1, uno]
Cola 3: Queue vacia.
<<<Pasando la segunda mitad de Cola 1 a Cola 3
Cola 1: Cola con 1 elementos:
[3, tres]
Cola 2: Cola con 1 elementos:
[1, uno]
Cola 3: Cola con 1 elementos:
[2, dos]
Cola 1: Queue vacia.
Cola 2: Cola con 1 elementos:
[1, uno]
Cola 3: Cola con 2 elementos:
[2, dos]
[3, tres]
Cola 1: Queue vacia.
Cola 2: Cola con 1 elementos:
[1, uno]
Cola 3: Cola con 2 elementos:
[2, dos]
[3, tres]
```

Ejercicio 2 (P3_E2). Búsqueda de caminos en un grafo 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 nodos 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). 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**. 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 _NodeList {
    void* data;
    struct _NodeList *next;
} NodeList;
struct _List {
    NodeList *node;

    destroy_elementlist_function_type  destroy_element_function;
    copy_elementlist_function_type     copy_element_function;
    print_elementlist_function_type    print_element_function;
    cmp_elementlist_function_type      cmp_element_function;
};
```

Las primitivas de este apartado están indicadas en el apéndice 2.

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_e3.c** que trabajará con listas de enteros. Este programa recibirá como argumento el nombre del fichero que contendrá los números, uno en cada línea.

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 (omitiendo impresiones intermedias por claridad) se puede ver a continuación:

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

> ./p3_e3 datos.txt

...                               ← se imprime el estado de la primera lista con
                                   cada inserción

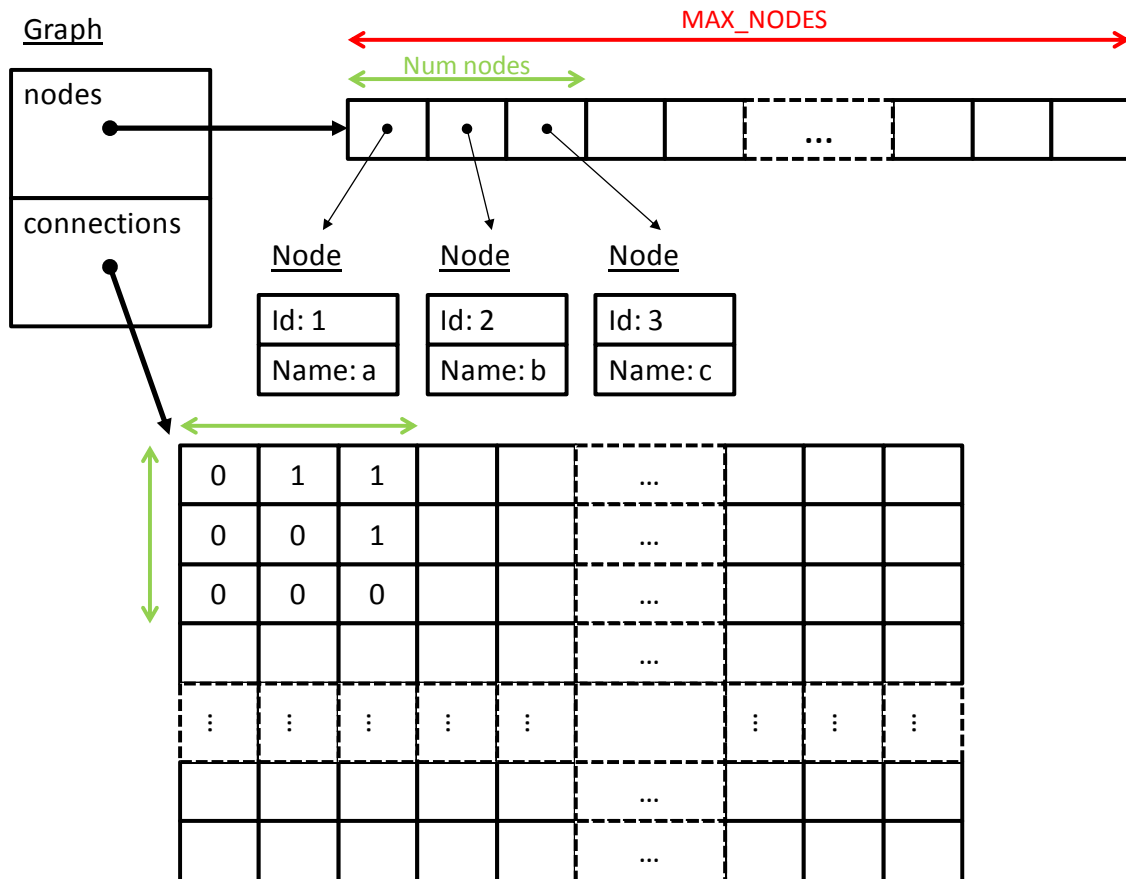
Lista con 12 elementos:
[2]
[4]
[6]
[8]
[10]
[12]
[11]
[9]
[7]
[5]
[3]
[1]

...                               ← se imprime el estado de la primera lista con cada
                                   extracción y el de la segunda lista según va
                                   creciendo en tamaño

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

Ejercicio 4 (P3 E4). Modificación del TAD Grafo utilizando el TAD Lista

El TAD Grafo utilizado hasta ahora contiene (como mínimo) dos atributos: un array de nodos y una matriz de adyacencia. Gráficamente, se puede representar como sigue:

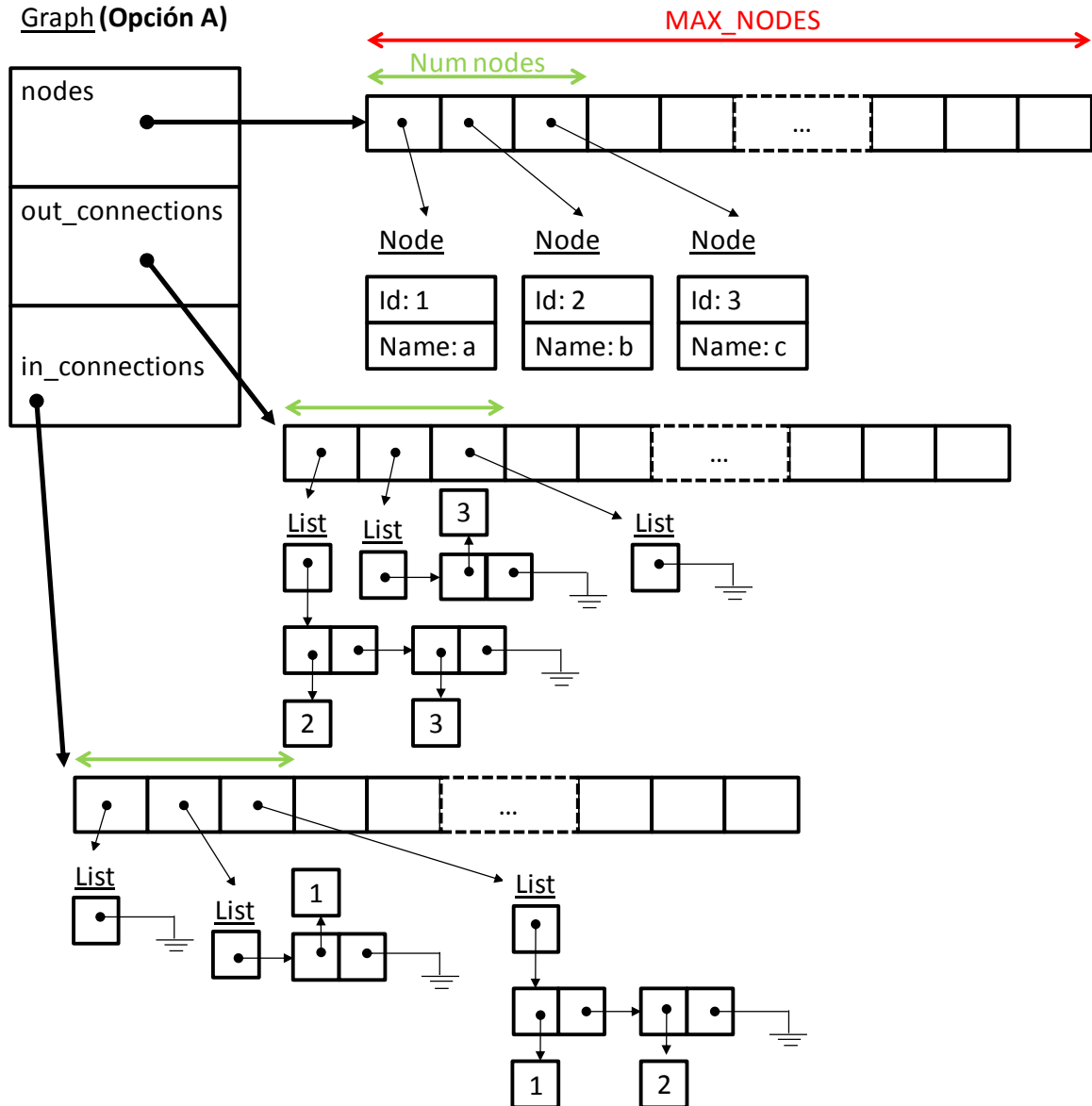


En este ejercicio, se quiere utilizar el TAD Lista implementado anteriormente para modificar el TAD Grafo y **evitar el malgasto de memoria** que se realiza utilizando arrays estáticos (especialmente en la matriz de adyacencia).

Se proponen dos alternativas para esta modificación:

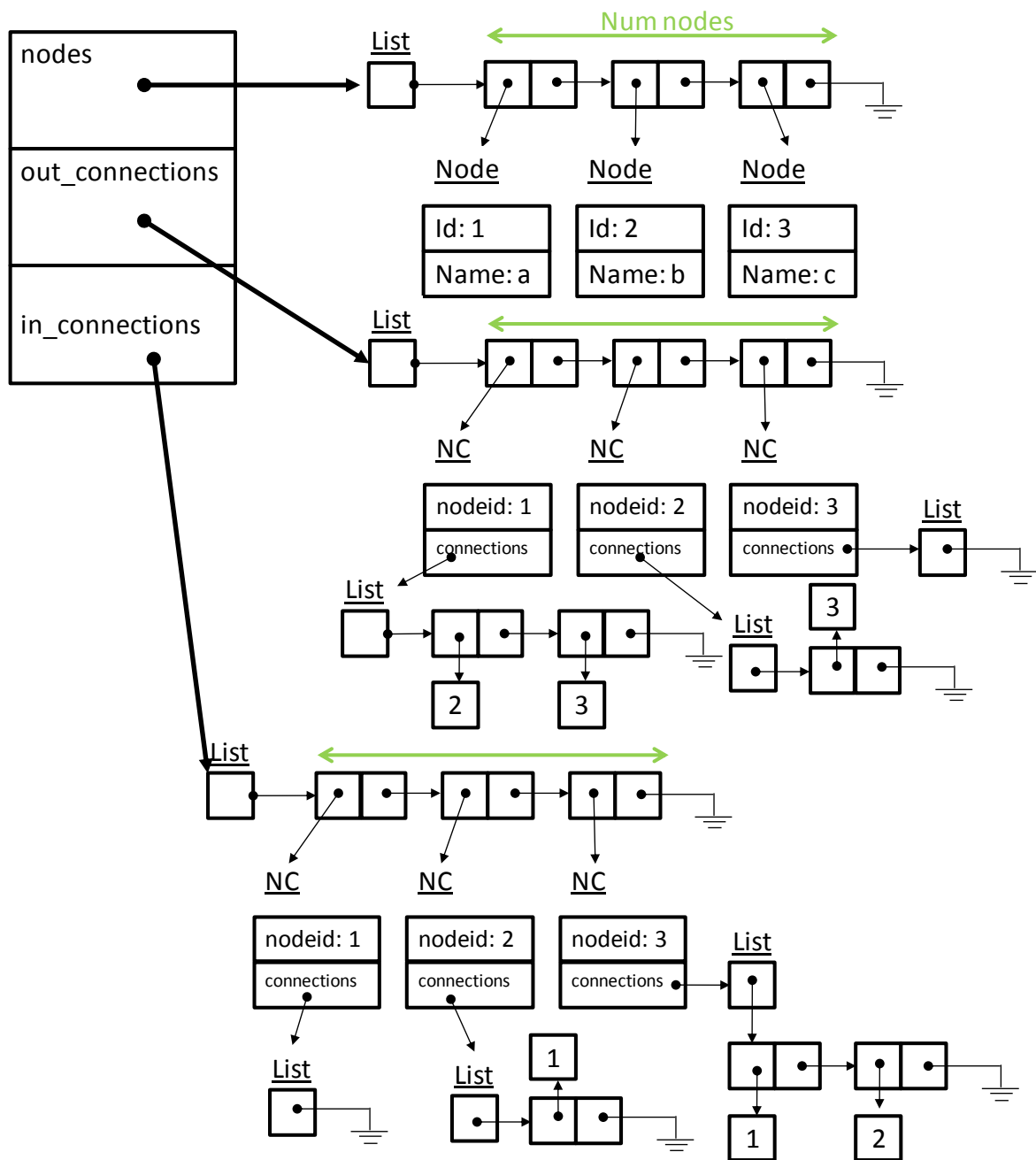
- En la primera opción (**Opción A**) sólo se modifica cómo se guardan las conexiones en la estructura del Grafo: en lugar de tener una matriz de adyacencia se tendrán dos arrays de listas, donde cada posición del array almacena una lista con los ids de los nodos a los que se conecta el nodo asociado a dicha posición (bien los nodos salientes, *out_connections*, o los nodos entrantes, *in_connections*).
- En la segunda opción (**Opción B**) se eliminan todos los arrays estáticos y se sustituyen por listas de distintos tipos de datos; por ejemplo, los nodos se almacenan en listas de nodos, mientras que las conexiones se almacenan como listas de un nuevo tipo de datos que guarda un id y una lista de enteros (los ids de los nodos conectados con el nodo correspondiente).

Graph (Opción A)



```
struct _Graph {
    Node* nodes[MAX_NODES];
    List* out_connections[MAX_NODES]; /* Listas de enteros */
    List* in_connections[MAX_NODES]; /* Listas de enteros */
};
```


Graph (Opción B)



```

struct _Graph {
    List * nodes;           /* Lista de nodos */
    List* out_connections; /* Lista de NodeConnections */
    List* in_connections;  /* Lista de NodeConnections */
};

typedef struct _NodeConnections {
    int nodeid;
    List* connections;      /* Lista de enteros */
} NodeConnections;
    
```

Cualquiera de estas dos modificaciones no requiere cambiar las primitivas del TAD Grafo, y, por tanto, el fichero **graph.h** se mantiene intacto, el mismo que en la P1.

Por esta razón, no hace falta realizar ningún main de prueba, bastará con enlazar el programa incluido en **graph_test.c** con la nueva implementación de graph, por ejemplo (suponiendo que la modificación del TAD Grafo se ha hecho en un fichero de nombre **graph-list.c**):

```
> gcc -o graph_test2 graph_test.o node.o list.o functions.o graph-list.o
```

Para posteriormente ejecutar el nuevo programa **graph_test2**, que debería mostrar la misma salida que el **graph_test** original (enlazado con la definición del TAD Graph de la P1).

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?

3. Basándote en el análisis anterior, discute las diferencias (incluyendo ventajas y desventajas) de la definición del TAD Grafo de la P1 frente a las dos propuestas indicadas en el último ejercicio, el P3_E4, independientemente de que hayas implementado alguna de estas propuestas en la práctica.

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;

/* Tipos de los punteros a función soportados por la cola */
typedef void (*destroy_elementqueue_function_type)(void*);
typedef void (*copy_elementqueue_function_type)(const void*);
typedef int (*print_elementqueue_function_type)(FILE *, const void*);
/**-----*/
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(destroy_elementqueue_function_type f1, copy_elementqueue_function_type f2,
                 print_elementqueue_function_type f3);
/**-----*/
Libera la cola y todos sus elementos.
-----*/
void queue_destroy(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 void* 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.
-----*/
void * 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: list.h

```
typedef struct _List List;

/* Tipos de los punteros a función soportados por la lista */
typedef void (*destroy_elementlist_function_type)(void*);
typedef void (*copy_elementlist_function_type)(const void*);
typedef int (*print_elementlist_function_type)(FILE *, const void*);
/* La siguiente función permite comparar dos elementos, devolviendo un número positivo, negativo o
cero según si el primer argumento es mayor, menor o igual que el segundo argumento */
typedef int (*cmp_elementlist_function_type)(const void*, const void*);

/* Inicializa la lista reservando memoria e inicializa todos sus elementos. */
List* list_ini(destroy_elementlist_function_type f1, copy_elementlist_function_type f2,
               print_elementlist_function_type f3, cmp_elementlist_function_type f4);
/* 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 void *elem);
/* Inserta al final de la lista realizando una copia del elemento. */
List* list_insertLast(List* list, const void *elem);
/* Inserta en orden en la lista realizando una copia del elemento. */
List* list_insertInOrder(List *list, const void *pElem);

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

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

/* Devuelve el elemento i-ésimo almacenado en la lista. En caso de error, devuelve NULL. */
const void* list_get(const List* list, int i);
/* 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);
```