Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

Programación II

Hoja de ejercicios – Colas

Una implementación en C del TAD Cola tiene como único interfaz las primitivas:

- `Cola *cola_crear()`, que crea (reservando memoria), inicializa y devuelve una cola.
- `void cola_liberar(Cola *pq)`, que libera la memoria de una cola y sus elementos.
- `boolean cola_vacia(Cola *pq)`, que comprueba si una cola está vacía.
- `boolean cola_llena(Cola *pq)`, que comprueba si una cola está llena.
- `status cola_insertar(Cola *pq, Elemento *pe)`, que inserta un elemento en la cola.
- `Elemento *cola_extraer(Cola *pq)`, que extrae y devuelve un elemento en la cola.

Asumir que un `insertar` precedido de un `extraer` sobre una cola no causa error.

Ejercicio 1

Dar el código C de una función `int cola_tamanio(Cola *pq)` que, usando las primitivas de arriba, devuelva el número de elementos almacenados en la cola de entrada `pq`.

```
int cola_tamanio(Cola *pq) {
	Cola *colaAux = NULL;
	Elemento *elem = NULL;
	int tam;

	if (!pq) {
		return -1;
	}

	// Creamos una cola colaAux
	colaAux = cola_crear();
	if (!colaAux) {
		return -1;
	}

	// Extraemos elemento a elemento de pq, insertándolos en colaAux y calculando el tamaño
	while (cola_vacia(pq)==FALSE) {
		elem = cola_extraer(pq);
		cola_insertar(colaAux, elem);
		elemento_liberar(elem);
		tam++;
	}

	// Recuperamos los elementos de pq sacándolos de colaAux
	while (cola_vacia(colaAux)==FALSE) {
		elem = cola_extraer(colaAux);
		cola_insertar(pq, elem);
		elemento_liberar(elem);
	}

	cola_liberar(colaAux);

	return tam;
}
```

Ejercicio 2

Dar el código C de una función `status cola_insertarConPrioridad(Cola *pq, Elemento *pe)` que actualice la cola `pq` poniendo `pe` detrás de todos los elementos con mayor o igual prioridad y delante de todos los elementos con menor prioridad. Los elementos que ya estaban en la cola deben mantenerse tras la inserción en el mismo orden en el que se encontraban. Asumir que el TAD Elemento tiene una primitiva `int elemento_prioridad(Elemento *pe)`, que devuelve un entero indicando el nivel de prioridad. Asumir que a mayor valor de ese entero, mayor prioridad del elemento, e.g., 4 tiene más prioridad que 3, 2 y 1.

```
// Esta función se puede implementar de forma similar a cola_insertarEnOrden del ejercicio 3
// La siguiente es una solución alternativa

status cola_insertarConPrioridad(Cola *pq, Elemento *pe) {
      Cola *colaAux = NULL;
      Elemento *elem = NULL;
      int insertado;

      if (!pq || cola_llena(pq)==TRUE) {
            return ERROR;
      }

      // Si pq está vacía, insertamos pe y salimos
      if (cola_vacia(pq)==TRUE) {
            cola_insertar(pq, pe);
            return OK;
      }

      // Creamos cola auxiliar
      colaAux = cola_crear();
      if (colaAux==NULL) {
            return ERROR;
      }

      // Extraemos todos los elementos de pq insertándolos en colaAux
      while (cola_vacia(pq)==FALSE) {
            elem = cola_extraer(pq);
            cola_insertar(colaAux, elem);
            elemento_liberar(elem);
      }

      // Extraemos elemento a elemento de colaAux y lo comparamos su prioridad con la de pe
      // para decidir cuál de ellos va antes en pq
      insertado = 0;
      while (cola_vacia(colaAux)==FALSE) {
            // Extraemos elem de colaAux
            elem = cola_extraer(colaAux);

            // En caso de mayor prioridad de pe, éste se inserta en pq
            if (insertado == 0 && elemento_prioridad(pe) > elemento_prioridad(elem)) {
                  cola_insertar(pq, pe);
                  insertado = 1;
            }

            // Se inserta elem en pq
            cola_insertar(pq, elem);
            elemento_liberar(elem);
      }

      cola_liberar(colaAux);

      return OK;
}
```

Ejercicio 3

Dar el código C de una función `status cola_insertarEnOrden(Cola *pq, Elemento *pe)` que inserte un elemento `pe` en una cola `pq` manteniendo un orden creciente de los elementos de `pq`.

Para ello asumir la existencia de una función `int elemento_comparar(Elemento *pe1, Elemento *pe2)` que devuelve –1 si el contenido de `pe1` es "menor" que el de `pe2`, 0 si el contenido de `pe1` es igual al de `pe2`, y 1 si el contenido de `pe1` es "mayor" que el de `pe2`.

```
// Esta función se puede implementar de forma similar a cola_insertarConPrioridad del ejercicio 2
// La siguiente es una solución alternativa

status cola_insertarEnOrden(Cola *pq, Elemento *pe) {
      Cola *colaAux = NULL;
      Elemento *elem = NULL;

      if (!pq || cola_llena(pq)==TRUE) {
            return ERROR;
      }

      // Creamos cola auxiliar
      colaAux = cola_crear();
      if (colaAux==NULL) {
            return ERROR;
      }

      // Extraemos de pq los elementos "menores" que pe, insertándolos en colaAux
      while (cola_vacia(pq)==FALSE) {
            elem = cola_extraer(pq);

            if (elemento_comparar(elem, pe)<=0) {      // elem va antes que pe
                  cola_insertar(colaAux, elem);        // insertamos elem
                  elemento_liberar(elem);
                  continue;
            }

            cola_insertar(colaAux, pe);         // insertarmos pe
            cola_insertar(colaAux, elem);       // insertamos elem

            elemento_liberar(elem);
            break;
      }

      // Extraemos los elementos restantes de pq, insertándolos en colaAux
      while (cola_vacia(pq)==FALSE) {
            elem = cola_extraer(pq);
            cola_insertar(colaAux, elem);
            elemento_liberar(elem);
      }

      // Recuperamos los elementos de pq, extrayéndolos de colaAux
      while (cola_vacia(colaAux)==FALSE) {
            elem = cola_extraer(colaAux);
            cola_insertar(pq, elem);
            elemento_liberar(elem);
      }

      cola_liberar(colaAux);

      return OK;
}
```

Ejercicio 4

Dar el código C de una función `status cola_juntar(Cola *pq1, Cola *pq2)` que reciba dos colas `pq1`, `pq2` y modifique `pq1` situando a continuación de su último elemento los de la cola `pq2` (que debe de quedar vacía si la unión de colas se efectúa con éxito) en su orden propio. La función sólo debe utilizar las primitivas anteriores y hacer el control y recuperación de errores pertinente.

```
status cola_juntar(Cola *pq1, Cola *pq2) {
      Cola *_pq1 = NULL, *_pq2 = NULL, *colaAux = NULL;  // en colaAux guardamos todos los elementos de pq1 y pq2
      Elemento *elem = NULL;
      status st;

      if (!pq1 || !pq2) {
            return ERROR;
      }

      // Creamos colas auxiliares
      _pq1 = cola_crear();
      if (_pq1==NULL) {
            return ERROR;
      }
      _pq2 = cola_crear();
      if (_pq2==NULL) {
            cola_liberar(_pq1);
            return ERROR;
      }
      colaAux = cola_crear();
      if (colaAux==NULL) {
            cola_liberar(_pq1);
            cola_liberar(_pq2);
            return ERROR;
      }

      // Extraemos los elementos de pq1 insertándolos en _pq1 y colaAux
      while (cola_vacia(pq1)==FALSE) {
            elem = cola_extraer(pq1);
            cola_insertar(_pq1, elem);
            cola_insertar(colaAux, elem);
            elemento_liberar(elem);
      }

      // Extraemos los elementos de pq2 insertándolos en _pq2
      while (cola_vacia(pq2)==FALSE) {
            elem = cola_extraer(pq);
            cola_insertar(_pq2, elem);
            elemento_liberar(elem);
      }

      // Insertamos los elementos de _pq2 en colaAux y pq2
      // En caso de error, recuperamos pq1 y pq2
      while (cola_vacia(_pq2)==FALSE) {
            elem = cola_extraer(_pq2);
            cola_insertar(pq2, elem);
            st = cola_insertar(colaAux, elem);
            elemento_liberar(elem);

            if (st==ERR) {
                  while (cola_vacia(_pq2)==FALSE) {
                        elem = cola_extraer(_pq2);
                        cola_insertar(pq2, elem);
                        elemento_liberar(elem);
                  }
                  while (cola_vacia(_pq1)==FALSE) {
                        elem = cola_extraer(_pq1);
                        cola_insertar(pq1, elem);
                        elemento_liberar(elem);
                  }
                  cola_liberar(_pq1);
                  cola_liberar(_pq2);
                  cola_liberar(colaAux);
                  return ERROR;
            }
      }

      // Volcamos colaAux en pq1 (que está vacía)
      while (cola_vacia(colaAux)==FALSE) {
            elem = cola_extraer(colaAux);
            cola_insertar(pq1, elem);
            elemento_liberar(elem);
      }

      cola_liberar(_pq1);
      cola_liberar(_pq2);
      cola_liberar(colaAux);
      return OK;
}
```

Ejercicio 5

Una **bi-cola con entrada restringida** es una cola especial en la que se puede extraer tanto por el inicio como por el final, y la inserción siempre se hace por el final.

Suponer una bi-cola con entrada restringida en la que los elementos que se almacenan tienen asociada una prioridad, de forma que cuando se desea extraer un elemento de la bicola se extrae, de entre los dos que están en los sendos extremos de la cola, aquel que tenga mayor prioridad. Asumir que todos los elementos son distintos.

Dadas las siguientes interfaces:

TAD bicola

```
Bicola *bicola_crear();

void bicola_liberar(Bicola *bc);

STATUS bicola_insert(const Elemento *ele, Bicola *bc); // Devuelve ERROR si la bicola está llena, o si alguno
                                                       // de los argumentos (ele, bc) es NULL

Elemento *bicola_extract(Bicola *bc); // Extrae el elemento del extremo que tenga mayor prioridad.
                                      // Devuelve NULL si la bicola está vacía o si bc es NULL

BOOLEAN bicola_vacia(const Bicola *bc);

BOOLEAN bicola_llena(const Bicola *bc);
```

TAD elemento

```
Elemento *elemento_crear();

void elemento_liberar(Elemento  *pe);

int elemento_imprimir(FILE *pf, const Elemento *pe);

Elemento *elemento_copiar(Elemento *porigen); // Crea un elemento copia del elemento origen

int elemento_comparaPrioridad(const Elemento *pe1, const Elemento *pe2);
// Devuelve -1, 0 ó 1 si la prioridad del elemento 1 es mayor, igual o menor a la del elemento 2
// respectivamente
```

Proporciona el código C de las primitivas **bicola_crear** y **bicola_extraer** suponiendo que el TAD bicola se implementa con un array como estructura de datos (EdD).

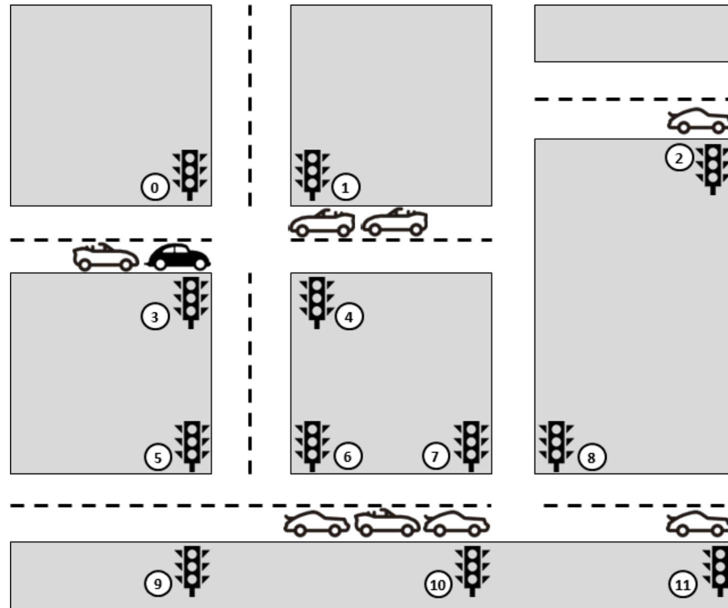
```
# define MAXQUEUE 100

struct _Bicola {
Elemento *datos[MAXQUEUE];
    int front; //primer elemento de la cola, extremo de salida
    int rear;  //primer hueco libre en la cola, extremo de entrada y salida
};

Bicola * bicola_crear() {
    Bicola *bc;
    bc = (Bicola *) malloc(sizeof(Bicola));
    if (!bc) return NULL

    for(i=0; i<MAXQUEUE; i++) bc->datos[i] = NULL;
    bc->front = bc->rear = 0;

    return bc;
}
```

```
Elemento *bicola_extraer(const Bicola *bc){
    Elemento *pe;
    int i;

    if (!bc) return NULL;
    if (esVacia(bc) == TRUE) return NULL;
    i = elemento_comparar(bc->a[bc->front], bc->a[bc->rear-1]);

    if (i==1) {
        pe = bc->datos[bc->front];
        bc->datos[bc->front] = NULL;
        bc->front = (bc->front + 1) % MAXQUEUE;
    }
    else {
        bc->rear - -;
                    if ( (bc->rear) < 0)
            bc->rear = MAXQUEUE-1;
        pe = bc->datos[bc->rear];
        bc->datos[bc->rear] = NULL;
    }
        return pe;
}
```

Ejercicio 6

Formas parte de un equipo de programadores que está desarrollando una aplicación de monitorización y gestión del tráfico de una ciudad. Actualmente tu equipo está implementando un simulador que posee un mapa con un número fijo de semáforos cuyos estados (verde o rojo) van modificándose cada cierto tiempo.

Cada semáforo tiene una cola de vehículos, y cada vehículo tiene una trayectoria a seguir semáforo a semáforo. En el dibujo se muestra un ejemplo de mapa con 12 semáforos –con identificadores (IDs) de 0 a 11– y vehículos encolados en ciertos semáforos: 2 en el semáforo 1, 1 en el semáforo 2, 2 en el semáforo 3, 3 en el semáforo 10, y 1 en el semáforo 11. La trayectoria del vehículo negro podría consistir en 4 tramos [3, 5, 10, 11] correspondientes a los semáforos 3, 5, 10 y 11.



Asumiendo que **existen las primitivas básicas de colas**, (p.e., `status cola_insertar(Cola *pc, Vechiculo *pv)`) y dadas las siguientes estructuras de datos:

```
typedef struct Vehiculo {
    int tramos[MAX_SEMAFOROS];  // array de IDs de los tramos (semáforos) en la trayectoria
    int numTramos;              // número de tramos (semáforos) en la trayectoria
    int tramoActual;            // índice en el array del tramo actual
};

typedef struct Semaforo {
    int estado;                 // SEMAFORO_VERDE o SEMAFORO_ROJO
    Cola *colaVehiculos;
    boolean alarma;
};

typedef struct MapaSemaforos {
    Semaforo *semaforos[MAX_SEMAFOROS];       // el ID de un semáforo es el índice en este
array
    int numSemaforos;
};
```

Proporciona el código C de la función `simularCambioSemaforo` que recibe el mapa del simulador, el identificador de un semáforo y el estado (`SEMAFORO_VERDE` o `SEMAFORO_ROJO`) al que hay que cambiar ese semáforo.

```
void simularCambioSemaforo(MapaSemaforos *mapa, int idSemaforo, int estadoSemaforo);
```

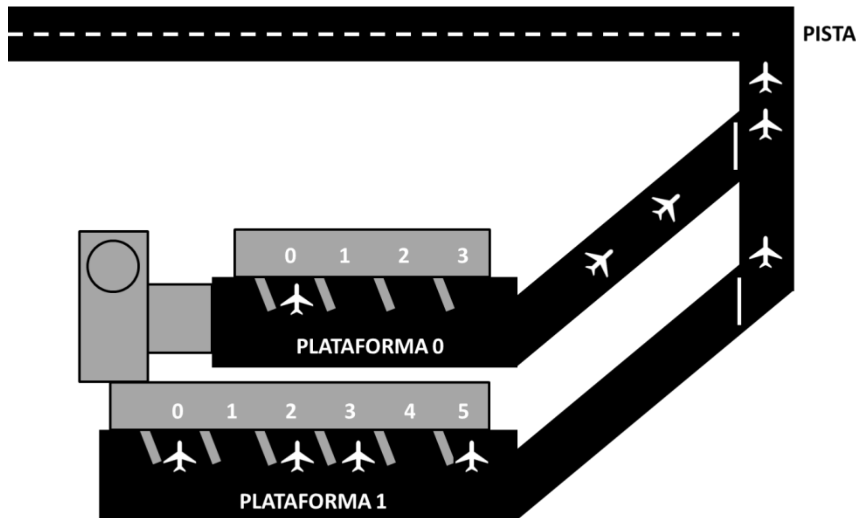
En caso de que el estado cambie a verde:

- La función ha de modificar el mapa para que uno a uno los vehículos encolados en el semáforo avancen un tramo de sus trayectorias.
- Si para uno de esos vehículos la cola de destino quedase llena, se acaba la ejecución de la función dejando pendiente el movimiento del resto de vehículos y habiendo puesto el campo `alarma` del semáforo a TRUE.

```c
void simularCambioSemaforo(MapaSemaforos *mapa, int idSemaforo, int estadoSemaforo) {
    Semaforo *semaforo = NULL, *semaforoSiguiente = NULL;
    Vehiculo *vehiculo = NULL;
    int idSemaforoSiguiente;

    if( !mapa || idSemaforo < 0 || idSemaforo > mapa->numSemaforos ) {
        return;
    }

    semaforo = mapa->semaforos[idSemaforo]

    // Si el nuevo estado del semaforo es rojo, lo actualizamos y salimos
    if ( estadoSemaforo == SEMAFORO_ROJO ) {
        semaforo->estado = SEMAFORO_ROJO;
        return;
    }

    // Si el nuevo estado del semáforo es verde, lo actualizamos y movemos sus
    // vehículos a sus siguientes tramos
    semaforo->estado = SEMAFORO_VERDE;

    // Movemos los vehiculos del semáforo
    while( cola_vacia(semaforo->colaVehiculos) == FALSE ) {
        vehiculo = cola_extraer(semaforo->colaVehiculos);

        if( vehiculo->tramoActual < vehiculo->numTramos-1 ) {
            vehiculo->tramoActual++;

            idSemaforoSiguiente = vehiculo->tramos[vehiculo->tramoActual];
            semaforoSiguiente = mapa->semaforos[idSemaforoSiguiente];

            cola_insetar(semaforoSiguiente->colaVehiculos, vehiculo);

            if (cola_llena(semaforoSiguiente->colaVehiculos) == TRUE ) {
                semaforo->alarma = TRUE;
                return;
            }
        }

        vehiculo_eliminar(vehiculo);
    }
}
```

Ejercicio 7

Formas parte de un equipo de programadores que está desarrollando una aplicación de monitorización y gestión del tráfico aéreo de un aeropuerto. Se te pide implementar un módulo controlador de despegues para **una única pista**. Los aviones inicialmente están estacionados en **puertas de embarque**. Cada puerta de embarque pertenece a una plataforma de estacionamiento dada. Así, el aeropuerto tiene **varias plataformas de estacionamiento** con acceso a la pista de despegue. La figura de abajo muestra un ejemplo de aeropuerto con dos plataformas, la 0 y la 1, con 4 y 6 puertas de embarque respectivamente.



Estando en una puerta de embarque, un avión puede solicitar a tu controlador el despegue. Al hacerlo, el controlador ha de cambiar el estado de la puerta de embarque de OCUPADA a DESPEGUE_SOLICITADO.

Acto seguido, si el acceso de la plataforma a la pista tiene espacio, el avión deja LIBRE la puerta de embarque y pasa a esperar en la (posible) cola de aviones del acceso de la plataforma. En la figura, la plataforma 0 tiene dos aviones esperando en cola, y la plataforma 1 no tiene aviones esperando. Si el acceso de la plataforma no tiene espacio, el controlador devuelve AVION_EN_PUERTA.

En caso de que el avión haya entrado en la cola de la plataforma, se comprueba si hay espacio en pista. En el ejemplo de la figura hay 3 aviones esperando en cola en pista. Si la cola en pista no está llena, uno de los aviones de la plataforma pasa a pista. Si resulta que el avión que pasa a pista es el que solicitó despegue, el controlador devuelve AVION_EN_PISTA. En caso contrario, devuelve AVION_EN_PLATAFORMA.

Asumiendo que **existen las primitivas básicas de colas**, (p.e., `status cola_insertar(Cola *pc, Avion *pa)`) y **del TAD Avion** (p.e., `void avion_eliminar(Avion *pa)`), y usando las siguientes estructuras de datos:

```
typedef enum Status{ERROR, OK, AVION_EN_PUERTA, AVION_EN_PLATAFORMA, AVION_EN_PISTA};
typedef enum EstadoPuerta{LIBRE, OCUPADA, DESPEGUE_SOLICITADO};

typedef struct PlataformaEstacionamiento {
    EstadoPuerta puertas[MAX_NUM_PUERTAS_PLATAFORMA];         // ids de puertas: 0, 1, 2,...
    int numPuertas;                                           // número de puertas en la plataforma
    Cola *colaEnPlataforma;                                   // aviones en espera en plataforma
};
typedef struct Aeropuerto {
    PlataformaEstacionamiento *plataformas[NUM_PLATAFORMAS];  // ids de plataformas: 0, 1, 2,...
    Cola *colaEnPista;                                        // aviones en espera en pista
};
```

proporciona el código C de la función `avionSolicitaDespegue` que realiza el proceso explicado arriba:

```
Status avionSolicitaDespegue(Aeropuerto *pAeropuerto, Avion *pAvion, int idPlataforma, int idPuerta);
```

En la función haz un control de argumentos de entrada, pero no un control de errores al usar las colas.

```c
Estado avionSolicitaDespegue(Aeropuerto *pAeropuerto, Avion *pAvion, int idPlataforma, int idPuerta) {
    PlataformaEstacionamiento *plataforma;
    Avion *avion;
    int i;

    if ( !pAeropuerto || !pAvion || idPlataforma<0 || idPlataforma>=NUM_PLATAFORMAS ||
          idPuerta<0 || idPuerta>=MAX_NUM_PUERTAS_PLATAFORMA ) {
        return ERROR;
    }

    plataforma = pAeropuerto->plataformas[idPlataforma];
    if ( idPuerta >= plataforma->numPuertas || plataforma->puertas[idPuerta] != OCUPADA ) {
        return ERROR;
    }

    plataforma->puertas[idPuerta] = DESPEGUE_SOLICITADO;

    if ( cola_llena(plataforma->colaEnPlataforma) == TRUE ) {
        return AVION_EN_PUERTA;
    }

    cola_insertar(plataforma->colaEnPlataforma, pAvion);
    plataforma->puertas[idPuerta] = LIBRE;

    if ( cola_llena(pAeropuerto->colaEnPista) == FALSE ) {
        avion = cola_extraer(plataforma->colaEnPlataforma);
        cola_insertar(pAeropuerto->colaEnPista);
        avion_eliminar(avion);

        if (cola_vacia(plataforma->colaEnPlataforma) == TRUE) {
            return AVION_EN_PISTA;
        }
    }

    return AVION_EN_PLATAFORMA;
}
```