Programación II, 2019-2020. Ejercicios de colas.

1. Cola circular sin sacrificar una posición del array.

Considera un array utilizado de forma circular para implementar una cola. En estas condiciones es posible evitar el sacrificio de un elemento de la cola si se agrega el campo `is_empty` a la EdD, que tomará el valor `TRUE` si y sólo si la cola está vacía. Escribe en C la EdD correspondiente y las primitivas de la cola para esta solución, indicando los cambios con respecto a la versión estudiada en clase.

Solución:

```
//-----------------------------------------------------------------
// Data structure:
//-----------------------------------------------------------------
struct _Queue {
  Element *data[MAX_QUEUE];
  int front;
  int rear;
  Boolean is_empty;
};

//-----------------------------------------------------------------
// Create an initialize a new queue:
//-----------------------------------------------------------------
Queue *queue_new() {
  Queue *q = NULL;
  int i;

  q = (Queue *)malloc(sizeof(Queue));
  if (q == NULL) {
    return NULL;
  }

  for (i=0; i<MAX_QUEUE; i++) {
    q->data[i] = NULL;
  }

  q->front = 0;
  q->rear = 0;
  q->is_empty = TRUE;

  return q;
}

//-----------------------------------------------------------------
// Free a queue:
//-----------------------------------------------------------------
void queue_free(Queue *q) {
  Element *p;

  if (q != NULL) {
    while ((p = queue_pop(q)) != NULL) {
      element_free(p);
```

```
    }

    free(q);
  }
}

//------------------------------------------------------------------
// Check if the queue is empty:
//------------------------------------------------------------------
Boolean queue_isEmpty(const Queue *q) {
  if (q == NULL) {
    return TRUE;
  }
  return q->is_empty;
}

//------------------------------------------------------------------
// Check if the queue is full:
//------------------------------------------------------------------
Boolean queue_isFull(const Queue *q) {
  if (q == NULL) {
    return TRUE;
  }

  if (q->is_empty == FALSE && q->front == q->rear) {
    return TRUE;
  }

  return FALSE;
}

//------------------------------------------------------------------
// Push element into the queue:
//------------------------------------------------------------------
Status queue_push(Queue *q, const Element *e) {
  Element *aux = NULL;

  if (q == NULL || e == NULL || queue_isFull(q) == TRUE) {
    return ERR;
  }

  aux = element_copy(e);
  if (aux == NULL) {
    return ERR;
  }

  q->data[q->rear] = aux;
  q->rear = (q->rear+1) % MAX_QUEUE;
  q->is_empty = FALSE;

  return OK;
}
```

```c
//-----------------------------------------------------------------
// Pop element from the queue:
//-----------------------------------------------------------------
Element *queue_pop(Queue *q) {
  Element *e = NULL;

  if (q == NULL || queue_isEmpty(q) == TRUE) {
    return NULL;
  }

  e = q->data[q->front];
  q->data[q->front] = NULL;
  q->front = (q->front+1) % MAX_QUEUE;

  if (q->front == q->rear) {
    q->is_empty = TRUE;
  }

  return e;
}
```

2. Número de elementos en la cola.

Se desea crear una función derivada (a partir de las primitivas) del TAD `Queue` para calcular el número de elementos en la cola. Escribe el pseudocódigo y el código C de esa nueva función, de prototipo `int queue_size (Queue *q)`, que devuelva el número de elementos que tiene la cola `q` o `-1` si se produce algún error. Al salir de la función la cola debe tener el mismo contenido que al principio. No está permitido acceder directamente a la estructura de datos.

Solución:

```c
int queue_size (Queue *q) {
  Queue *qaux = NULL;
  Element *e = NULL;
  int size=0;

  if (q == NULL) {
    return -1;
  }

  qaux = queue_new();
  if (qaux == NULL) {
    return -1;
  }

  // Pop from q and push into qaux (errors are not checked for clarity):
  while (queue_isEmpty(q) == FALSE) {
    e = queue_pop(q);
    queue_push(qaux, e);
    element_free(e);
    size++;
```

```
  }

  // Restore q (errors are not checked for clarity):
  while (queue_isEmpty(qaux) == FALSE) {
    e = queue_pop(qaux);
    queue_push(q, e);
    element_free(e);
  }

  queue_free(qaux);

  return size;
}
```

3. Tirando las cartas.

Basado en https://onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1876

Imagina que tienes una baraja ordenada de n cartas numeradas de 1 a n. La carta 1 es la primera y la carta n es la última. Vas a jugar al siguiente juego mientras queden al menos dos cartas en el mazo:

1. Coges la primera carta del mazo y la tiras.
2. Coges la carta que está ahora en primera posición y la pones al final del mazo.

Escribe el código de la función `int last_card(int n)` que recibe el número de cartas en la baraja, n, y devuelve el número de la última carta que queda cuando el juego termina. Por ejemplo, si $n = 5$ la función debe devolver 2.

- La función debe devolver el número de la última carta que queda en el mazo, o -1 si se produce algún error.
- La función debe hacer uso de una cola con la siguiente interfaz.
- Puedes añadir otras funciones si lo consideras necesario.

Interfaz:

```
typedef struct _Queue Queue;

typedef void (*P_queue_ele_free)(void*);
typedef void *(*P_queue_ele_copy)(const void*);
typedef int (*P_queue_ele_print)(FILE *, const void*);

Queue *queue_new(P_queue_ele_free f1, P_queue_ele_copy f2, P_queue_ele_print f3);
void queue_free(Queue *q);
Boolean queue_isEmpty(const Queue *q);
Boolean queue_isFull(const Queue *q);
Status queue_push(Queue *q, const void *ele);
void *queue_pop(Queue *q);
```

Solución:

```
//------------------------------------------------------------------------
void int_free(void *p) {
```

```
  free(p);
}

//-----------------------------------------------------------------------------
void *int_copy(const void *p) {
  int *x = (int *)p;
  int *c = (int *)malloc(sizeof(int));

  if (c == NULL) {
    return NULL;
  }

  *c = *x;
  return c;
}

//-----------------------------------------------------------------------------
int int_print(FILE *f, const void *p) {
  int *c = (int *)p;
  return fprintf(f, "%d", *c);
}

//-----------------------------------------------------------------------------
int last_card(int n) {
  int i, x;
  int *p;
  Queue *q;

  // Check argument:
  if (n <= 0) return -1;

  // Create queue:
  q = queue_new(&int_free, &int_copy, &int_print);
  if (!q) return -1;

  // Insert cards in the queue:
  for (i=1; i<=n; i++) {
    if (ERROR == queue_push(q, &i)) {
      queue_free(q);
      return -1;
    }
  }

  // Play the game:
  while (queue_isEmpty(q) == FALSE) {
    p = (int *)queue_pop(q);
    x = *p;
    free(p);
    if (queue_isEmpty(q)) {
      queue_free(q);
      return x;
    }
    p = (int *)queue_pop(q);
```

```
    x = *p;
    free(p);
    if (ERROR == queue_push(q, &x)) {
      queue_free(q);
      return -1;
    }
  }

  return -1;
}
```

4. Implementación de una cola usando dos pilas.

Basado en https://www.geeksforgeeks.org/queue-using-stacks/

Proporciona una implementación del TAD `Queue` a partir del TAD `Stack`. La estructura de datos contará con dos pilas `s1` y `s2`. La pila `s1` contiene los datos ordenados de tal modo que el `top` corresponda con el `front` de la cola. La pila `s2` se usa como auxiliar.

- Para extraer un elemento de la cola simplemente hacemos `pop` de `s1`.
- Para insertar un elemento en la cola tenemos que mover todos los elementos de `s1` a `s2`, hacer `push` en `s1` y traer los elementos de vuelta.

Solución:

```
struct _Queue {
  Stack *s1;
  Stack *s2;
  P_queue_ele_free pfree;
  P_queue_ele_copy pcopy;
  P_queue_ele_print pprint;
};

//------------------------------------------------------------------------
Queue *queue_new(P_queue_ele_free f1, P_queue_ele_copy f2, P_queue_ele_print f3) {
  Queue *q = NULL;

  // Allocate queue:
  q = (Queue *)malloc(sizeof(Queue));
  if (q == NULL) {
    return NULL;
  }

  // Init stacks:
  q->s1 = stack_new(f1, f2, f3);
  if (q->s1 == NULL) {
    queue_free(q);
    return NULL;
  }
  q->s2 = stack_new(f1, f2, f3);
  if (q->s2 == NULL) {
    queue_free(q);
```

```
    return NULL;
  }

  // Set function pointers:
  q->pfree = f1;
  q->pcopy = f2;
  q->pprint = f3;

  return q;
}

//------------------------------------------------------------------------
void queue_free(Queue *q) {
  void *p;

  if (q != NULL) {
    // Free stacks:
    stack_free(q->s1);
    stack_free(q->s2);

    // Free queue:
    free(q);
  }
}

//------------------------------------------------------------------------
Status queue_push(Queue *q, const void *ele) {
  void *aux = NULL;

  if (q == NULL || ele == NULL || queue_isFull(q) == TRUE) {
    return ERROR;
  }

  // Move elements from s1 to s2:
  while (stack_isEmpty(q->s1) == FALSE) {
    aux = stack_pop(q->s1);
    stack_push(q->s2, aux); // Check error !!!
    (*(q->pfree))(aux);
  }

  // Push ele into s1:
  stack_push(q->s1, ele);

  // Move elements back to s1:
  while (stack_isEmpty(q->s2) == FALSE) {
    aux = stack_pop(q->s2);
    stack_push(q->s1, aux); // Check error !!!
    (*(q->pfree))(aux);
  }

  return OK;
}
```

```
//---------------------------------------------------------------------------
void *queue_pop(Queue *q) {
  void *e = NULL;

  if (q == NULL || queue_isEmpty(q) == TRUE) {
    return NULL;
  }

  return stack_pop(q->s1);
}

//---------------------------------------------------------------------------
void *queue_getFront(const Queue *q) {
  if (q == NULL || queue_isEmpty(q) == TRUE) {
    return NULL;
  }

  return stack_top(q->s1);
}

//---------------------------------------------------------------------------
void *queue_getBack(const Queue *q) {

  // TO DO.....................

  return NULL;
}

//---------------------------------------------------------------------------
Boolean queue_isEmpty(const Queue *q) {
  if (q == NULL) {
    return TRUE;
  }
  return stack_isEmpty(q->s1);
}

//---------------------------------------------------------------------------
Boolean queue_isFull(const Queue *q) {
  if (q == NULL) {
    return TRUE;
  }
  return stack_isFull(q->s1);
}

//---------------------------------------------------------------------------
size_t queue_size(const Queue *q) {
  if (q == NULL) return 0;
  return stack_size(q->s1);
}

//---------------------------------------------------------------------------
int queue_print(FILE *fp, const Queue *q) {
  int i;
```

```
  if (fp == NULL || q == NULL || queue_isEmpty(q) == TRUE) {
    return 0;
  }

  return stack_print(fp, q->s1);
}
```

5. Implementación de una pila usando una cola.

Basado en https://www.geeksforgeeks.org/implement-a-stack-using-single-queue/

Proporciona una implementación del TAD `Stack` a partir del TAD `Queue`. La estructura de datos contará con una única cola `q`, con los datos ordenados de modo que el `front` sea el `top` de la pila.

- Para hacer un `pop` simplemente extraemos de la cola.
- Para hacer un `push` insertamos en la cola y recolocamos los elementos para que el último elemento quede en el `front`.

Solución:

```
struct _Stack {
  Queue *q;
  P_stack_ele_free pfree;
  P_stack_ele_copy pcopy;
  P_stack_ele_print pprint;
};

//------------------------------------------------------------------------
Stack *stack_new(P_stack_ele_free f1, P_stack_ele_copy f2, P_stack_ele_print f3) {
  Stack *s = NULL;

  // Allocate stack:
  s = (Stack *)malloc(sizeof(Stack));
  if (s == NULL) {
    return NULL;
  }

  // Initialize queue:
  s->q = queue_new(f1, f2, f3);
  if (s->q == NULL) {
    free(s);
    return NULL;
  }

  // Set function pointers:
  s->pfree = f1;
  s->pcopy = f2;
  s->pprint = f3;

  return s;
}
```

```c
//------------------------------------------------------------------------
void stack_free(Stack *s) {
  int i;
  if (s != NULL) {
    // Free queue:
    queue_free(s->q);

    // Free stack:
    free(s);
  }
}

//------------------------------------------------------------------------
Status stack_push(Stack *s, const void *ele) {
  void *aux = NULL;
  int qsize;
  int i;

  if (s == NULL || ele == NULL || stack_isFull(s) == TRUE) {
    return ERROR;
  }

  // Get queue size:
  qsize = queue_size(s->q);

  // Push into queue:
  queue_push(s->q, ele); // Check error !!!

  // Move elements in queue so that ele is the first:
  for (i=0; i<qsize; i++) {
    aux = queue_pop(s->q);
    queue_push(s->q, aux); // Check error !!!
    (*(s->pfree))(aux);
  }

  return OK;
}

//------------------------------------------------------------------------
void *stack_pop(Stack *s) {
  if (s == NULL || stack_isEmpty(s) == TRUE) {
    return NULL;
  }

  return queue_pop(s->q);
}

//------------------------------------------------------------------------
void *stack_top(Stack *s) {
  if (s == NULL || stack_isEmpty(s) == TRUE) {
    return NULL;
  }
```

```
  return queue_getFront(s->q);
}

//------------------------------------------------------------------------
Boolean stack_isEmpty(const Stack *s) {
  if (s == NULL) {
    return TRUE;
  }

  return queue_isEmpty(s->q);
}

//------------------------------------------------------------------------
Boolean stack_isFull(const Stack *s) {
  if (s == NULL) {
    return TRUE;
  }

  return queue_isFull(s->q);
}

//------------------------------------------------------------------------
size_t stack_size(const Stack *s) {
  if (s == NULL) {
    return 0;
  }

  return queue_size(s->q);
}

//------------------------------------------------------------------------
int stack_print(FILE *fp, const Stack *s) {
  if (fp == NULL || s == NULL) {
    return 0;
  }

  return queue_print(fp, s->q);
}
```

6. Intercalar los elementos de una cola.

Basado en https://www.geeksforgeeks.org/interleave-first-half-queue-second-half/

Implementar la función `Queue *interleave(Queue *q)` que recibe una cola `q` y devuelve otra cola con los elementos de la primera mitad y la segunda mitad de `q` intercalados.

Ejemplos:

Entrada: 1, 2, 3, 4, 5, 6
Salida: 1, 4, 2, 5, 3, 6

Entrada: 1, 2, 3, 4, 5
Salida: 1, 4, 2, 5, 3

Solución:

```
Queue *interleave(Queue *q) {
  Queue *qaux;
  Queue *qret;
  int mid;
  int i;
  void *p;

  // Check argument:
  if (q == NULL) {
    return NULL;
  }

  // Create new queues:
  qaux = queue_new(ffree, fcopy, fprint);
  if (qaux == NULL) {
    return NULL;
  }
  qret = queue_new(ffree, fcopy, fprint);
  if (qret == NULL) {
    queue_free(qaux);
    return NULL;
  }

  // Get middle point:
  mid = (queue_size(q) + 1) / 2;

  // Move first half into aux queue:
  for (i=0; i<mid; i++) {
    p = queue_pop(q);
    queue_push(qaux, p); // Check error !!!
    (*ffree)(p);
  }

  // Interleave:
  while (queue_isEmpty(qaux) == FALSE) {
    p = queue_pop(qaux);
    queue_push(qret, p); // Check error !!!
    (*ffree)(p);

    if (queue_isEmpty(q) == FALSE) {
      p = queue_pop(q);
      queue_push(qret, p); // Check error !!!
      (*ffree)(p);
    }
  }

  // Free qaux:
  queue_free(qaux);

  return qret;
}
```