Nombre y Apellido: ..... N° Legajo: .....

Segundo Parcial de Programación Imperativa

17/11/2023

	Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*
Calificación	**/3.5**	**/3.5**	**/3**	

- **Condición mínima de aprobación: Sumar 5 (cinco) puntos.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **No usar variables globales ni static.**
- **No es necesario escribir los #include**
- **Escribir en esta hoja Nombre, Apellido y Legajo**

Ejercicio 1

Se desea implementar un TAD para la **venta de tickets para un museo**, que permita saber cuántos tickets hay por día y por año, como así también obtener los nombres de las personas que reservaron tickets para un día determinado. Cada día se identifica por el orden dentro del año, al primero de enero le corresponde 1, al 2 de enero 2, al 1 de febrero 32, etc.

Para ello se definió la siguiente interfaz:

```
typedef struct museumTicketCDT * museumTicketADT;

/**
 * Reserva los recursos para administrar las ventas de tickets para visitar un museo en
 * un año determinado
 * Nota: Como no se indica el año para el que se lo utilizará se asume que el
 * año será siempre bisiesto
 */
museumTicketADT newMuseumTicket(void);

/**
 * Se registra un ticket para que #visitor visite el museo el día #dayOfYear del año
 * Retorna la cantidad actual de tickets registrados para visitar el museo ese día
 * Retorna 0 si #dayOfYear es igual a 0 o mayor a 366
 * Retorna 0 si ya se había registrado un ticket para ese #visitor y ese #dayOfYear
 */
int addTicket(museumTicketADT museumTicketADT, size_t dayOfYear, const char * visitor);

/**
 * Retorna la cantidad de tickets registrados para visitar el museo el día #dayOfYear
 * del año
 * Retorna -1 si #dayOfYear es igual a 0 o mayor a 366
 */
int dayTickets(const museumTicketADT museumTicketADT, size_t dayOfYear);

/**
 * Retorna la cantidad total de tickets registrados para visitar el museo (todos los
 * días del año)
 */
int yearTickets(const museumTicketADT museumTicketADT);
```

```
/**
 * Funciones de iteración para poder consultar, para un día #dayOfYear del año,
 * los nombres de los visitantes de los tickets registrados para visitar el museo
 * ese día en orden alfabético por nombre de visitante.
 * IMPORTANTE: Permitir utilizar estas funciones para distintos días del año
 * EN SIMULTANEO (ver ejemplo en programa de prueba)
 * Abortar si cualquiera de las funciones es invocada con un #dayOfYear igual a 0
 * o mayor a 366
 */
void toBeginByDay(museumTicketADT museumTicketADT, size_t dayOfYear);

size_t hasNextByDay(museumTicketADT museumTicketADT, size_t dayOfYear);

char * nextByDay(museumTicketADT museumTicketADT, size_t dayOfYear);

/**
 * Libera los recursos utilizados para administrar las ventas de tickets para visitar
 * un museo en un año determinado
 */
void freeMuseumTicket(museumTicketADT museumTicketADT);
```

Se pide:

- **Implementar todas las estructuras necesarias**, de forma tal que **todas** las funciones puedan ser implementadas de la forma más eficiente posible (no sólo las que les pedimos implementar)
- **Implementar las siguientes funciones:**
  - **newMuseumTicket**
  - **addTicket**
  - **dayTickets**
  - **toBeginByDay**

Ejemplo de programa de prueba:

```
int
main(void) {
    // Reserva los recursos para administrar las ventas de tickets
    museumTicketADT museum = newMuseumTicket();

    // Se registra un ticket para que John visite el museo el día 4 del año
    // Y retorna 1 porque es la cantidad de tickets para ese día
    assert(addTicket(museum, 4, "John") == 1);
    // Los siguientes fallan porque John ya cuenta con un ticket para el día 4 del año
    assert(addTicket(museum, 4, "John") == 0);
    assert(addTicket(museum, 4, "JOHN") == 0);
    assert(addTicket(museum, 4, "john") == 0);

    // Se registra un ticket para que John visite el museo el día 360 del año
    assert(addTicket(museum, 360, "John") == 1);

    // Falla porque el día del año es igual a 0
    assert(addTicket(museum, 0, "Katherine") == 0);
    // Falla porque el día del año es mayor a 366
    assert(addTicket(museum, 367, "Katherine") == 0);

    // Se registra un ticket para que Paul visite el museo el día 4 del año
    // Y retorna 2 porque es la cantidad de tickets para ese día
    assert(addTicket(museum, 4, "Paul") == 2);
    assert(addTicket(museum, 4, "Ariel") == 3);
```

```
    assert(addTicket(museum, 360, "Brenda") == 2);

    // Se inicializa el iterador para el día 4 del año
    toBeginByDay(museum, 4);
    // Quedan visitantes por recorrer para el día 4 del año
    assert(hasNextByDay(museum, 4) == 1);
    // Se obtiene el primer visitante para el día 4 del año en orden alfabético
    assert(strcmp(nextByDay(museum, 4), "Ariel") == 0);

    toBeginByDay(museum, 360);
    // Se obtiene el primer visitante para el día 360 del año en orden alfabético
    assert(hasNextByDay(museum, 360) == 1);
    assert(strcmp(nextByDay(museum, 360), "Brenda") == 0);

    assert(hasNextByDay(museum, 4) == 1);
    // Se obtiene el segundo visitante para el día 4 del año en orden alfabético
    assert(strcmp(nextByDay(museum, 4), "John") == 0);
    assert(hasNextByDay(museum, 360) == 1);
    // Se obtiene el segundo visitante para el día 360 del año en orden alfabético
    assert(strcmp(nextByDay(museum, 360), "John") == 0);

    assert(hasNextByDay(museum, 4) == 1);
    assert(strcmp(nextByDay(museum, 4), "Paul") == 0);
    // No quedan más visitantes por recorrer para el día 4 del año
    assert(hasNextByDay(museum, 4) == 0);
    // No quedan más visitantes por recorrer para el día 360 del año
    assert(hasNextByDay(museum, 360) == 0);

    // Se obtiene la cantidad de tickets para el día 4 del año
    assert(dayTickets(museum, 4) == 3);
    assert(dayTickets(museum, 360) == 2);
    assert(dayTickets(museum, 15) == 0);

    // Se obtiene la cantidad total de tickets (todos los días del año)
    assert(yearTickets(museum) == 5);

    // Libera los recursos utilizados para administrar las ventas de tickets
    freeMuseumTicket(museum);

    puts("OK!");
    return 0;
}
```

Ejercicio 2

Se desea implementar un TAD para **administrar sitios de interés sobre un camino** o ruta. En este TAD sólo se puede registrar un sitio de interés cada 100 metros (uno entre los 0 y 99 metros, otro entre los 100 y 199, etc.). Pero como un sitio de interés se designa con un tipo genérico, si el usuario desea registrar varios podría usar el siguiente tipo:

```
typedef struct {
      int dim;            // cantidad de sitios
      char * names[20];   // hasta 20 sitios de interés
} landmarkType;
```

No hay límite para la distancia máxima a la cual se encuentre un sitio de interés, ni tampoco para la cantidad de sitios a registrar.

Para ello se definió la siguiente interfaz:

```
#include <stdlib.h>

typedef ____ landmarkType;

typedef struct landmarkCDT * landmarkADT;

/**
 * Crea una nueva colección de sitios de interés. Sólo se registrará un sitio cada
 * 100 metros (uno entre los 0 y 99 metros, posiblemente otro entre los 100 y 199, etc.)
 */
landmarkADT newLandmark(¿?);

/**
 * Agrega un punto de interés #site a #meters metros del origen. Si ya había
 * un sitio, sólo queda #site (ver programa de testeo)
 */
void addLandmark(landmarkADT landmark, size_t meters, landmarkType site);

/*
 * Retorna 1 si a esa distancia aproximada hay un sitio de interés. Ejemplo: si
 * meters es 135 retorna 1 si se registró algún sitio de interés entre los 100 y 199
 * metros), 0 sinó
 */
int hasLandmark(const landmarkADT landmark, size_t meters);

/*
 * Retorna la distancia aproximada en la cual se encuentra el sitio de interés
 * o -1 si no existe. Ejemplo: si el sitio de interés se indicó que está a 135
 * metros retorna 100, si se indicó a 1240 metros retorna 1200
 */
int distance(const landmarkADT landmark, landmarkType site);

/*
 * Retorna cuántos sitios de interés se registraron
 */
size_t landmarkCount(const landmarkADT landmark);

/*
 * Retorna un vector con los sitios de interés, en orden ascendente por distancia
 * al origen. Si no hubiera sitios de interés retorna NULL
 */
landmarkType * landmarks(const landmarkADT landmark);

/*
 * Retorna un vector con los sitios de interés entre dos distancias,
 * en orden ascendente por distancia al origen, dejando en *dim la cantidad
 * Si from es mayor que to o no hay sitios de interés en ese rango retorna NULL y
 * deja *dim en cero
 */
landmarkType * landmarksBetween(const landmarkADT landmark, size_t from, size_t to, size_t
*dim);

void freeLandmark(landmarkADT landmark);
```

Donde **¿?** indica que para resolver este problema **se debe indicar el o los parámetros resultantes**

Se pide:

- **Implementar <u>todas las estructuras</u> necesarias,** de forma tal que <u>las funciones `addLandmark`, `hasLandmark`, `landmarkCount` y `freeLandmark` puedan ser implementadas de la forma más eficiente posible</u>.
- **Implementar las siguientes funciones:**
  - **`newLandmark`**
  - **`addLandmark`**
  - **`landmarks`**

<u>Ejemplo de programa de prueba.</u> En este caso usamos `char *` para `landmarkType`

```
typedef char * landmarkType;
```

```
int
main(void) {
    landmarkADT lm = newLandmark(_____________);

    assert(landmarkCount(lm)==0);

    landmarkType * v = landmarks(lm);
    assert(v==NULL);

    size_t dim;
    v = landmarksBetween(lm, 0, 300, &dim);
    assert(v==NULL && dim==0);

    addLandmark(lm, 50, "Atalaya");
    addLandmark(lm, 70, "YPF abandonada");
    addLandmark(lm, 99, "El estadio del Chelsea");
    assert(landmarkCount(lm)==1);

    addLandmark(lm, 650, "Restos de un OVNI");

    addLandmark(lm, 350, "Pulpería de 1800");

    addLandmark(lm, 200, "Otra pulperia");

    v = landmarksBetween(lm, 0, 299, &dim);
    assert(dim==2);
    assert(strcmp(v[0], "El estadio del Chelsea")==0);
    assert(strcmp(v[1], "Otra pulperia")==0);
    free(v);

    v = landmarksBetween(lm, 100, 199, &dim);
    assert(v==NULL && dim==0);

    // El siguiente ciclo imprime:
    //     El estadio del Chelsea
    //     Otra pulpería
    //     Pulpería de 1800
    //     Restos de un OVNI
    v = landmarks(lm);
    for(int i=0; i < landmarkCount(lm); i++) {
        puts(v[i]);
    }
    free(v);

    assert(distance(lm, "Atalaya")==-1);
    assert(distance(lm, "El estadio del Chelsea")==0);
    assert(distance(lm, "Restos de un OVNI")==600);
```

```
    for(int i=600; i<=699; i++)
        assert(hasLandmark(lm, i));

    for(int i=0; i<=99; i++)
        assert(hasLandmark(lm, i));

    for(int i=100; i<=199; i++)
        assert(!hasLandmark(lm, i));

    freeLandmark(lm);

    puts("OK!");
    return 0;
}
```

Ejercicio 3

a) Escribir la función **recursiva** **`contarGrupos`** que reciba como **únicos parámetros** un string (cadena de chars *null terminated*) y un carácter. La función debe retornar la **cantidad de grupos de caracteres consecutivos iguales al carácter pasado por parámetro**. Un grupo está formado por uno o más caracteres iguales de manera consecutiva.

Si el string es "**`abbbccaadaaaa`**" y el carácter es 'a', la función debe devolver 3, ya que hay 3 grupos de caracteres 'a' formados por 1, 2 y 4 caracteres consecutivos respectivamente.

Si el string es "**`34abcaa33 30 0`**" y el carácter es '3', la función debe devolver 3, ya que hay 3 grupos de caracteres '3' formados por 1, 2 y 1 caracteres consecutivos respectivamente.

Si el string es "**`abbbccaadaaaa`**" y el carácter es 'x', la función debe retornar cero.

b) Dada la siguiente definición de listas

```
typedef struct node {
      char head;
      struct node * tail;
};

typedef struct node * TList;
```

Escribir la función **recursiva** **`countGroups`** que reciba como **únicos parámetros** una lista de tipo TList y un carácter. La función debe retornar la **cantidad de grupos de caracteres consecutivos iguales al carácter pasado por parámetro**. Un grupo está formado por uno o más caracteres iguales en nodos consecutivos.

Las funciones **`contarGrupos`** y **`countGroups`** **NO pueden invocar a otras funciones ni macros.**