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

Segundo Parcial de Programación Imperativa

17/01/2025

	Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*
Calificación	**/3.75**	**/3.75**	**/2.5**	

- **<u>Condición mínima de aprobación: Sumar 5 (cinco) puntos.</u>**
- **Los ejercicios que no se ajusten estrictamente al enunciado, <u>no serán aceptados</u>.**
- **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 **administrar influencers de una red social**. <u>Cada influencer se identifica con un string case sensitive</u>. Para cada influencer se registra la cantidad de seguidores que cuenta, pero sólo nos interesa registrar los que tienen al menos 1000 followers.

Se crea el siguiente TAD para manejar la información.

Si bien no debería haber dos influencers con el mismo id y distinta cantidad de seguidores, esto el TAD no lo valida.

```
typedef struct socialCDT * socialADT;

typedef struct influencerData {
   char * id;
   size_t followers;
} influencerData;

socialADT newSocial(???);  // Implementar

void freeSocial(socialADT social);   // No implementar

/**
* Agrega un influencer
* id: identifica al influencer
* followers: cantidad de seguidores
* El influencer no debería existir, pero eso el TAD no lo valida. Si ya existe lo
* vuelve a agregar
* Si followers es menor a 1000, no lo agrega
*/
int addInfluencer(socialADT social, const char * id, size_t followers);

/**
* Le agrega una cantidad de seguidores al influencer. Si no existe retorna cero
*/
int addFollowers(socialADT social, const char * id, size_t newFollowers); // NO implementar

/**
* Retorna un vector con los influencers que tienen "n miles" de seguidores
* Por ejemplo si n es 1 los que tienen entre 1000 y 1999 seguidores
* si n es 11, los que tienen entre 11000 y 11999 seguidores
```

```
* si n es 0 o no hay influencers en ese rango, retorna NULL y asigna cero a *dim
* En cada posición del vector deja una copia del id y la cantidad de seguidores
* El vector debe estar ordenado en forma ascendente por cantidad de seguidores
* Si dos o más influencers tienen la misma cantidad de seguidores no es necesario
* desempatar por otro criterio
*/
influencerData * influencers(const socialADT social, size_t n, size_t * dim);
```

Donde **???** significa que el que implemente el TAD debe decidir qué parámetros son necesarios para la función

Se pide **implementar todos los structs y únicamente las siguientes funciones**, junto con todas las funciones que sean invocadas por las mismas: **`newSocial`**, **`addInfluencer`**, **`influencers`**.

Programa de prueba:

```
int main(void) {
   socialADT social = newSocial(______);
   assert(addInfluencer(social, "Nerd123", 999) == 0);
   assert(addInfluencer(social, "Marito", 3100) == 1);
   assert(addInfluencer(social, "Cuzco", 3100) == 1);
   char aux[20] = "Pachi";
   assert(addInfluencer(social, aux, 21100) == 1);
   strcpy(aux, "Luli");
   assert(addInfluencer(social, aux, 3001) == 1);
   int dim;
   influencerData * v = influencers(social, 3, &dim);
   assert(dim == 3);
   assert(v[0].followers == 3001); // es Luli
   assert(v[1].followers == 3100); // es Cuzco o Marito
   assert(v[2].followers == 3100); // es Marito o Cuzco
   for(int i=0; i< dim; i++) {
       free(v[i].id);
   }
   free(v);
   freeSocial(social);
   puts("OK!");
   return 0;
}
```

Ejercicio 2

Un museo de ciencias naturales clasifica a cada espécimen que se agrega al mismo con un número entero mayor a cero. El más antiguo lleva el código 1, luego el 2, 3, etc.

Se irán agregando al TAD todos los especímenes pero no necesariamente en orden.

La información asociada a cada espécimen puede ser cualquier cosa, por lo que se usará un tipo genérico.

Para ello se definió la siguiente interfaz:

```
typedef struct museumCDT * museumADT;

typedef ... ElemType;

museumADT newMuseumADT(???);

void freeMuseumADT(museumADT adt);

/**
* Registra la información info para el espécimen con código itemCode
* Si ya existe, no hace nada y retorna 0.
* Si no existe lo agrega y retorna 1
*/
int addItem(museumADT adt, unsigned int itemCode, ElemType info);

/**
* Registra la información info para el espécimen con código itemCode
* Si ya existe, reemplaza la información que tenía asociada por la nueva.
*/
void addOrUpdate(museumADT adt, unsigned int itemCode, ElemType info);

/**
* Retorna la cantidad de especímenes que hay registrados
*/
size_t countItems(const museumADT adt);

/**
* Retorna el código de item con la info asociada. Si no existe retorna -1
* Si hay más de un espécimen con la mismo info, retorna el de menor código
*/
int itemCode(const museumADT adt, ElemType info); // NO implementar
```

Donde **???** significa que el que implemente el TAD debe decidir qué parámetros son necesarios para la función

Se pide:

- **Implementar todas las estructuras necesarias**, de forma tal que las funciones **addItem**, **addOrUpdate**, **countItems** sean lo más eficientes posibles.
- **Implementar únicamente las siguientes funciones**, junto con todas las funciones que sean invocadas por las mismas: **newMuseumADT**, **addItem**, y **addOrUpdate**.

Programa de prueba, **en este caso se decidió que ElemType sea char *:**

```
int main(void) {
   museumADT cnat = newMuseumADT(________);
   assert(addItem(cnat, 0, "Mosca de la fruta")==0); // 0 es inválido
   assert(addItem(cnat, 10, "Mosca de la fruta")==1);
   assert(addItem(cnat, 10, "Otra mosca")==0);
   addOrUpdate(cnat, 10, "Garza blanca"); // Pisa "Mosca de la fruta"
```

```
    assert(addItem(cnat, 15, "Brontosaurio")==1);
    assert(addItem(cnat, 6, "Carpincho")==1);
    assert(addItem(cnat, 123, "Carpincho")==1); // Inserta igual
    assert(countItems(cnat)==4);
    assert(itemCode(cnat, "Mosca de la fruta")==-1);
    assert(itemCode(cnat, "Garza blanca")==10);

    freeMuseumADT(cnat);
    puts("OK");
    return 0;
}
```

Ejercicio 3

Dada la siguiente definición de estructuras para una **lista lineal**:

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

typedef node * TList;
```

Implementar la función recursiva **`removeIf`** que **reciba**

- **`list`: una lista (puede estar vacía)**
- **`criteria`: una función que reciba un entero y retorne 1 ó 0**
- **`count`: un puntero a entero. No está inicializado**

La función debe eliminar de list todos los elementos que al ser evaluados por la función criteria retorne 1. En el parámetro *dim debe indicar cuántos elementos se eliminaron

Ejemplo: si la lista fuera 2 -> 5 -> 1 -> 4 -> 3 -> 7 y la función **`criteria`** retorna 1 si el elemento es par, la lista ahora es 5 -> 1 -> 3 -> 7, y se eliminaron 2 elementos

NO SE ADMITIRÁ UNA SOLUCIÓN QUE TENGA UN CICLO DENTRO DE LA FUNCIÓN.

NO DEFINIR MACROS NI FUNCIONES AUXILIARES.