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

Recuperatorio Segundo Parcial de Programación Imperativa

02/07/2025

- **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 (3,75 puntos)

Una gran ciudad (podría ser también el AMBA) mantiene información sobre sus colectivos, cada uno de ellos identificado con un número entero, y **sobre los choferes de cada línea**.

Basarse en el programa de prueba para determinar lo que se espera de cada función.

El contrato con el TAD es el siguiente:

busDriverADT.h

```
typedef struct busDriverADT * busDriverADT;

busDriverADT newBusDriverADT();

void freeBusDriverADT(busDriverADT adt); // NO IMPLEMENTAR

/**
* Agrega (si no existía) una línea de colectivos
* retorna 1 si la agrega, 0 si no la agrega (ya estaba)
*/
int newBusLine(busDriverADT adt, unsigned int busLine);

/**
* Retorna la cantidad de líneas de colectivos
*/
int busLinesCount(const busDriverADT adt); // NO IMPLEMENTAR

/**
* Retorna la cantidad de choferes que hay en la línea busLine
* Si no existe la línea, retorna 0
*/
int driversCount(const busDriverADT adt, unsigned int busLine);

/**
* Retorna un vector con la copia de los nombres de los choferes de una línea
* en orden alfabético (ascendente). La dimensión se obtiene con la función driversCount
* Si la línea no existe o no tiene choferes retorna NULL
*/
char ** drivers(const busDriverADT adt, unsigned int busLine);

/**
* Agrega un chofer a una línea.
* Si la línea no existe o el chofer ya estaba en esa línea, no hace nada
*/
// NO IMPLEMENTAR
void addDriver(busDriverADT adt, unsigned int busLine, const char * driver);
```

```
/**
* Si existe el chofer para esa línea, lo elimina
*/
// NO IMPLEMENTAR
void removeDriver(busDriverADT adt, unsigned int busLine, const char * driver);
```

Se pide:

- **Implementar <u>todas las estructuras</u> necesarias**, de forma tal que las funciones **`newBusLine`**, **`busLinesCount`** y **`driversCount`** <u>sean lo más eficientes posibles.</u>
- **<u>Implementar únicamente las siguientes funciones</u>**, junto con todas las funciones que sean invocadas por las mismas:
  - **`newBusDriverADT`**
  - **`newBusLine`**
  - **`driversCount`**
  - **`drivers`**

Las estructuras definidas deben servir para poder implementar **<u>todas</u>** las funciones del TAD, no solo las pedidas.

<u>Programa de prueba:</u>

```
int main(void) {
   busDriverADT adt = newBusDriverADT();
   assert(busLinesCount(adt) == 0); // Inicialmente no hay líneas

   // Agregamos líneas nuevas
   assert(newBusLine(adt, 60) == 1);
   assert(newBusLine(adt, 168) == 1);
   assert(newBusLine(adt, 60) == 0); // ya estaba

   assert(busLinesCount(adt) == 2);
   assert(driversCount(adt, 60) == 0);
   assert(driversCount(adt, 999) == 0); // línea inexistente
   assert(driversCount(adt, 19) == 0);  // línea inexistente

   // Agregamos choferes a la línea 60: Carlos, Ana y Bruno
   addDriver(adt, 60, "Carlos");
   char auxName[20] = "Ana";
   addDriver(adt, 60, auxName);
   strcpy(auxName, "Bruno");
   addDriver(adt, 60, auxName);
   addDriver(adt, 60, "Ana"); // repetido, no se agrega

   assert(driversCount(adt, 60) == 3);

   // Agregamos chofer a otra línea
   addDriver(adt, 168, "Diana");
   assert(driversCount(adt, 168) == 1);

   // No debería hacer nada si la línea no existe
   addDriver(adt, 9, "Pedro");
   assert(driversCount(adt, 9) == 0);

   // Verificamos orden alfabético de choferes línea 60: Ana, Bruno, Carlos
   char ** names = drivers(adt, 60);
   assert(names != NULL);
   assert(strcmp(names[0], "Ana") == 0);
   assert(strcmp(names[1], "Bruno") == 0);
   assert(strcmp(names[2], "Carlos") == 0);
   free(names[0]); free(names[1]); free(names[2]);
   free(names);
```

```
    // Verificamos choferes de línea inexistente
    assert(drivers(adt, 1234) == NULL);
    assert(drivers(adt, 15) == NULL);

    // Quitamos un chofer de la línea 60
    removeDriver(adt, 60, "Bruno");
    assert(driversCount(adt, 60) == 2);

    // Intentamos quitar chofer inexistente o en línea inexistente
    removeDriver(adt, 60, "Juan");  // no está
    removeDriver(adt, 99, "Ana");  // línea no existe

    // Verificamos choferes actuales en 60: Ana, Carlos
    names = drivers(adt, 60);
    assert(names != NULL);
    assert(strcmp(names[0], "Ana") == 0);
    assert(strcmp(names[1], "Carlos") == 0);
    free(names[0]); free(names[1]);
    free(names);

    freeBusDriverADT(adt);
    puts("OK");
    return 0;
}
```

Ejercicio 2 (3,75 puntos)

Se desea guardar una colección de elementos **no repetidos**, en la cual los elementos más "populares" (los que más se consultan) estén al principio de la colección. De esta forma, será más rápido acceder a los elementos que más veces se consulten. Para ello se definió que el conjunto de datos opere de la siguiente forma:

- Cuando se inserta un elemento (no repetido) se lo inserta **al final**
- Cuando se consulta un elemento (con la función **exist**) el mismo es enviado **al principio** de la colección.

El contrato con el TAD es el siguiente:

mostPopularADT.h

```
typedef struct mostPopularCDT * mostPopularADT;

// Tipo de elemento a insertar
typedef ... elemType;

/* Retorna un nuevo conjunto de elementos genéricos. */
mostPopularADT new(???);

/* Libera todos los recursos del TAD */
void freeMostPopular(mostPopularADT tad); // NO IMPLEMENTAR

/* Inserta un elemento si no está. Lo inserta al final.
** Retorna 1 si lo agregó, 0 si no.
*/
unsigned int add(mostPopularADT tad, elemType elem); // NO IMPLEMENTAR

/* Retorna la cantidad de elementos que hay en la colección */
unsigned int size(const mostPopularADT tad);  // NO IMPLEMENTAR

/* Se ubica al principio del conjunto, para poder iterar sobre el mismo */
void toBegin(mostPopularADT tad);
```

```c
/* Retorna 1 si hay un elemento siguiente en el iterador, 0 si no */
int hasNext(const mostPopularADT tad);

/* Retorna el siguiente elemento. Si no hay siguiente elemento, aborta */
elemType next(mostPopularADT tad);

/* Retorna 1 si el elemento está en la colección, 0 si no está
** Si el elemento estaba lo ubica al principio.
*/
int exist(mostPopularADT tad, elemType elem);
```

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 que se puedan implementar todas las funciones, no solo las pedidas**
- **Implementar únicamente las siguientes funciones**, junto con todas las funciones que sean invocadas por las mismas:
  - **new**
  - **toBegin**
  - **hasNext**
  - **next**
  - **exist**

Programa de prueba, **en este caso se decidió que elemType sea char *** (pero podría ser de cualquier tipo, este es un ejemplo)

```c
int main(void) {
   mostPopularADT tad = new(...);
   assert(size(tad) == 0); // Al principio, está vacío

   // Agregamos elementos
   assert(add(tad, "perro") == 1);
   assert(add(tad, "gato") == 1);
   assert(add(tad, "loro") == 1);
   assert(add(tad, "gato") == 0); // ya estaba

   assert(size(tad) == 3);

   // Verificamos el orden: perro, gato, loro
   const char * expected1[] = {"perro", "gato", "loro"};
   toBegin(tad);
   for (int i = 0; i < 3; i++) {
       assert(hasNext(tad));
       const char * elem = next(tad);
       assert(strcmp(elem, expected1[i]) == 0);
   }
   assert(!hasNext(tad));

   // Consultamos "gato" → debe pasar al principio
   int ok = exist(tad, "gato");
   assert(ok == 1);

   // Verificamos nuevo orden: gato, perro, loro
   const char * expected2[] = {"gato", "perro", "loro"};
   toBegin(tad);
   for (int i = 0; i < 3; i++) {
       assert(hasNext(tad));
       const char * elem = next(tad);
       assert(strcmp(elem, expected2[i]) == 0);
   }
```

```
    assert(!hasNext(tad));

    // consulta de uno que no está → 0
    assert(exist(tad, "pez") == 0);

    // Consultamos "loro" → pasa al principio
    ok = exist(tad, "loro");
    assert(ok == 1);

    // Nuevo orden: loro, gato, perro
    const char * expected3[] = {"loro", "gato", "perro"};
    toBegin(tad);
    for (int i = 0; i < 3; i++) {
        assert(hasNext(tad));
        const char * elem = next(tad);
        assert(strcmp(elem, expected3[i]) == 0);
    }
    assert(!hasNext(tad));

    freeMostPopular(tad);
    puts("OK");
    return 0;
}
```

Ejercicio 3 (2,50 puntos)

Dada la siguiente definición de estructuras para una **lista lineal** en la cual se almacenan nombres de personas (const char *) acompañados con su popularidad (unsigned int):

```
typedef struct node {
    const char * person;
    unsigned int popularity;
    struct node * tail;
} node;

typedef node * TList;
```

donde **cuánto mayor es el valor del campo `popularity`, más popular es la persona**. La lista se mantiene ordenada descendente por **`popularity`** (los más populares aparecerán primero).

Implementar la función recursiva `upPop` que **reciba como únicos parámetros:**

- **La lista actual** *(Donde NULL representa una lista vacía)*
- **El nombre exacto de una persona** *(“Paco” y “PACO” son dos personas distintas)*
- **Cuánto se incrementará su popularidad** *(No se invocará con el valor cero)*

e **incrementa la popularidad de la persona en la cantidad indicada**, dejando la lista correctamente ordenada.

Si la persona no está en la lista se la agrega, en el orden que corresponda de acuerdo a su popularidad. **Ver el ejemplo de uso para determinar qué se espera de la función.**

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

NO DEFINIR MACROS NI FUNCIONES AUXILIARES.

Ejemplo de uso:

```
int main(void) {
   TList myList = NULL;                  // Lista Vacía
   myList = upPop(myList, "Cato", 5);    // {"Cato",5}
   myList = upPop(myList, "Paco", 3);    // {"Cato",5} -> {"Paco",3}
   myList = upPop(myList, "Mon" , 4);    // {"Cato",5} -> {"Mon" ,4} -> {"Paco",3}
   myList = upPop(myList, "Cato", 1);    // {"Cato",6} -> {"Mon" ,4} -> {"Paco",3}
   myList = upPop(myList, "Mon" , 2);    // {"Cato",6} -> {"Mon" ,6} -> {"Paco",3}
   myList = upPop(myList, "Paco", 2);    // {"Cato",6} -> {"Mon" ,6} -> {"Paco",5}
   myList = upPop(myList, "Paco", 2);    // {"Paco",7} -> {"Cato",6} -> {"Mon" ,6}
   myList = upPop(myList, "Mon" , 1);    // {"Paco",7} -> {"Mon" ,7} -> {"Cato",6}
   // Se agrega otro (PACO no es lo mismo que Paco)
   myList = upPop(myList, "PACO" , 7);
                             // {"Paco",7} -> {"Mon",7} -> {"PACO",7} -> {"Cato",6}
   myList = upPop(myList, "Cato" , 8);
                            // {"Cato",14} -> {"Paco",7} -> {"Mon",7} -> {"PACO",7}
   // TODO Liberar la lista
   puts("OK");
   return 0;
}
```