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

Segundo Parcial de Programación Imperativa

15/11/2024

	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 el ingreso de un secundario**. A este secundario se presentan cada año más de mil postulantes. A cada postulante se le asigna un número de legajo correlativo que consta de los dos últimos dígitos del año seguido de un número de 4 dígitos que comienzan en 1, por ejemplo 240001, 240002, 241543 para el año 2024.

Para cada postulante se registra el nombre y además las notas de cada una de las materias de las que consta el ingreso. No todos los años tienen la misma cantidad de materias, por ejemplo en el 2023 fueron 3, en el 2024 fueron 4, pero se puede asumir que nunca serán muchas materias.

Cada materia se identifica con un código múltiplo de 10, por ejemplo si son 3 materias, serán la 10, la 20 y la 30.

Para cada materia se consigna una única nota entera entre 0 y 10.

El número de legajo se asigna cuando el alumno se inscribe al curso, si luego se da de baja (con la función correspondiente) ese legajo se le puede asignar luego a otro postulante. Es por eso que pueden quedar “huecos” en la numeración a medida que algunos alumnos se den de baja.

<u>Para ello se definió la siguiente interfaz:</u>

```
typedef struct courseCDT * courseADT;

/* Crea un sistema de administración del curso de ingreso para un año #year
 * con una cantidad #subjects de materias
 */
courseADT newCourse(size_t year, size_t subjects);

/* Agrega un alumno al curso
 * Si el id es inválido para el año no lo agrega y retorna cero
 * Si ya existía un alumno con ese id no hace nada y retorna cero
 * En cualquier otro caso lo agrega con todas las notas en cero y retorna 1
 */
int addStudent(courseADT course, size_t id, const char * name);

/* Retorna una copia del nombre del estudiante o NULL si el id es inválido o no existe */
char * studentName(const courseADT course, size_t id);

/* Retorna la cantidad de estudiantes en el curso */
size_t students(const courseADT course);
```

```
/* Establece la nota en una materia para un estudiante
 * Si el legajo del estudiante es incorrecto o no existe el estudiante retorna 0
 * Si el código de materia es incorrecto o la nota no está entre 0 y 10 retorna 0
 * En ambos casos no realiza ningún cambio
 * En caso contrario registra la nota y retorna 1
 * Si la materia ya tenía una nota, la pisa con la nueva nota recibida
 */
int setGrade(courseADT course, size_t id, size_t subject, unsigned char grade);

/* Retorna la nota de un estudiante en una materia. La nota de una materia es cero
 * hasta tanto se registre una nueva nota
 * Si el legajo del estudiante es incorrecto o no existe el estudiante retorna -1
 * Si el código de materia es incorrecto retorna -1
 */
int gradeSubject(const courseADT course, size_t id, size_t subject);

/* Si el id es válido y existe el alumno, lo elimina del curso y retorna 1
 * En otro caso no hace nada y retorna cero
 */
int deleteStudent(courseADT course, size_t id);

/* Libera los recursos utilizados por el TAD */
void freeCourse(courseADT course);
```

Se pide:

- **Implementar <u>todas las estructuras</u> necesarias**, de forma tal que **<u>todas las funciones puedan ser implementadas de la forma más eficiente posible</u>** (no sólo las que les pedimos implementar)
- **Implementar las siguientes funciones,** junto con todas las funciones que sean invocadas por las mismas**:**
    - **newCourse**
    - **addStudent**
    - **studentName**
    - **students**
    - **setGrade**

<u>Ejemplo de programa de prueba:</u>

```
int main(void) {
    // Curso de ingreso para el 2025 con 3 materias
    courseADT course = newCourse(2025, 3);

    // El id es inválido
    assert(addStudent(course, 1, "Jose")==0);

    assert(addStudent(course, 250002, "Han")==1);
    assert(addStudent(course, 250001, "Felix")==1);
    assert(addStudent(course, 250101, "Lisa")==1);
    assert(addStudent(course, 250101, "Otra Lisa")==0);

    char name[] = "Radames";
    assert(addStudent(course, 250020, name)==1);
    strcpy(name, "Tosca");
    assert(addStudent(course, 250022, name)==1);

    assert(students(course)==5);

    assert(studentName(course, 250900)==NULL);

    char * s = studentName(course, 250020);
```

```
    assert(strcmp(s, "Radames")==0);
    free(s);
    assert(deleteStudent(course, 250020)==1);
    assert(studentName(course, 250020)==NULL);
    assert(students(course)==4);

    // Las 3 notas de "Han"
    assert(setGrade(course, 250002, 10, 2)==1);
    assert(setGrade(course, 250002, 20, 9)==1);
    assert(setGrade(course, 250002, 30, 8)==1);
    assert(setGrade(course, 250002, 10, 6)==1);  // Era un 2, ahora es 6

    // Códigos inválidos de materia
    assert(setGrade(course, 250002, 15, 5)==0);
    assert(setGrade(course, 250002, 40, 5)==0);

    freeCourse(course);

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

Ejercicio 2

Se desea implementar un TAD para **almacenar elementos de cualquier tipo** (pero todos del mismo tipo) donde los elementos se pueden **insertar en una posición determinada o al final**. La primera posición se indica con el valor cero.

La colección acepta elementos repetidos.

Una vez agregado a la colección, un elemento no se elimina.

Para ello se definió la siguiente interfaz:

```
typedef struct collectionCDT * collectionADT;

typedef __________ elemType;

/* Crea una colección vacía de elementos */
collectionADT newCollection(¿?);

/* Inserta el elemento #elem, que pasa a ocupar la posición #idx en la colección,
 * desplazando una posición hacia atrás a todos los elementos que estaban a partir de la
 * posición #idx.
 * Si #idx es 0, será el primer elemento de la colección
 * Si #idx es mayor o igual a la cantidad de elementos, lo inserta al final
*/
void addElement(collectionADT collection, elemType elem, size_t idx);

/* Retorna la cantidad de elementos presentes en la colección */
size_t sizeCollection(const collectionADT collection);

/* Devuelve la posición de la primer aparición del elemento,
 * o -1 si el elemento no está */
int positionFirst(const collectionADT collection, elemType elem);

/* Funciones de iteración para que se puedan consultar todos los elementos
** registrados en forma ordenada por posición
*/
void toBegin(collectionADT collection);

size_t hasNext(const collectionADT collection);

elemType next(collectionADT collection);
```

```
/* Libera todos los recursos utilizados por el TAD */
void freeCollection(collectionADT collection);
```

Se pide:

- **Implementar <u>todas las estructuras</u> necesarias**.
- **Implementar las siguientes funciones,** junto con todas las funciones que sean invocadas por las mismas
  - **newCollection**
  - **addElement**
  - **positionFirst**
  - **toBegin**
  - **hasNext**
  - **next**

<u>Ejemplo de programa de prueba, asumiendo que `elemType` es int:</u>

```
int main(void) {
    collectionADT c = newCollection(_________)

    // Agrega el elemento 10 a la colección.
    // No importa el índice que le pongamos, será el primero (posición cero)
    addElement(c, 10, 1);  // 10
    // Agrega el elemento 20 al principio, desplazando una posición hacia atrás al 10
    addElement(c, 20, 0);  // 20 - 10
    addElement(c, 30, 0);  // 30 - 20 - 10
    addElement(c, 40, 1);  // 30 - 40 - 20 - 10

    assert(sizeCollection(c)==4);

    addElement(c, 60, 10); // 30 - 40 - 20 - 10 - 60
    // Acepta repetidos
    addElement(c, 10, 10); // 30 - 40 - 20 - 10 - 60 - 10

    assert(positionFirst(c, 70)==-1); // No está en la colección
    assert(positionFirst(c, 30)==0);
    assert(positionFirst(c, 20)==2);
    assert(positionFirst(c, 10)==3);

    assert(sizeCollection(c)==6);

    toBegin(c);
    assert(hasNext(c)==1);
    assert(next(c)==30); assert(next(c)==40); assert(next(c)==20);
    assert(next(c)==10); assert(next(c)==60); assert(next(c)==10);
    assert(!hasNext(c));

    freeCollection(c);

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

Ejercicio 3

Dada la siguiente definición de estructuras para una lista lineal

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

typedef node * TList;
```

Escribir la función **recursiva** **`addElem`** que **reciba únicamente la lista actual y un elemento de tipo `char`**. La función debe:

- Si el elemento no está en la lista, agregarlo al final, con el valor `hits` en 0
- Si el elemento está, debe incrementar en uno el campo `hits`

  La lista siempre debe estar ordenada en forma descendente por cantidad de `hits`

No definir macros ni funciones auxiliares
No usar ciclos dentro de la función

Ejemplos:

Si la lista fuera **`{'X', 2}`**→**`{'Z', 1}`** y se agrega el elemento **'A'** la lista queda **`{'X', 2}`**→**`{'Z', 1}`**→**`{'A', 0}`**

Si se vuelve a agregar el elemento **'A'** la lista puede quedar de dos formas, ambas correctas: **`{'X', 2}`**→**`{'Z', 1}`**→**`{'A', 1}`** o bien **`{'X', 2}`**→**`{'A', 1}`**→**`{'Z', 1}`**

Si la lista fuera **`{'X', 2}`**→**`{'Z', 2}`**→**`{'A', 2}`** y se agrega una **'A'** la lista que debe quedar es **`{'A', 3}`**→**`{'X', 2}`**→**`{'Z', 2}`**

Ejemplo de programa de prueba:

```
int main(void) {
    TList l = NULL;
    l = addElem(l, 'X');
    l = addElem(l, 'X');
    l = addElem(l, 'Z');
    // (X,1) -> (Z,0)
    assert(l->head == 'X'); assert(l->tail->head == 'Z');
    l = addElem(l, 'A');
    // (X,1) -> (Z,0) -> (A,0) o bien (X,1) -> (A,0) -> (Z,0)
    assert(l->head == 'X');
    assert((l->tail->head == 'Z' && l->tail->tail->head == 'A')
           || (l->tail->head == 'A' && l->tail->tail->head == 'Z'));
    l = addElem(l, 'A');
    l = addElem(l, 'A');
    assert(l->head == 'A');
    assert(l->tail->head == 'X');
    assert(l->tail->tail->head == 'Z');
    assert(l->tail->tail->tail == NULL);

    // Se liberan todos los nodos de la lista

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