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

Segundo Parcial de Programación Imperativa

12/06/2025

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

- **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 **administrar los distintos cursos** de una universidad, para un cuatrimestre. Cada carrera se identifica con una letra del alfabeto inglés, y cada curso se identifica con un código numérico y una descripción. Además para cada curso se registra la cantidad de estudiantes inscriptos.

En una misma carrera puede haber dos materias con la misma descripción pero en ese caso deben tener distinto código.

Una misma materia se puede dictar en distintas carreras.

Si bien no debería haber dos materias con el mismo código en una carrera, esto el TAD no lo valida.

Para ello se definió la siguiente interfaz:

```c
typedef struct coursesCDT * coursesADT;

struct courseData {
   char * name;
   int code;
   unsigned int students;
};

coursesADT newCourses(void);

void freeCourses(coursesADT course);

/**
* Agrega -si no estaba- una nueva materia a la carrera. Si ya existe una materia con esa
* descripción en la carrera, debe tener un código distinto
* El parámetro students indica cuántos alumnos están inscriptos en esa materia
* Retorna 1 si se agregó la materia, 0 si no (si la carrera era inválida o ya existía la
* materia)
*/
int addCourse(coursesADT course, char degree, int courseCode, const char * courseName,
unsigned int students);

/**
* Retorna la cantidad de materias que hay en una carrera.
*/
size_t courses(const coursesADT course, char degree);
```

```c
/**
* Retorna la cantidad de alumnos que hay en una carrera.
* Si la carrera es inválida retorna cero
*/
size_t students(const coursesADT course, char degree);

// Funciones para iterar por carrera, alfabéticamente por materia (en forma eficiente)
void toBeginByDegree(coursesADT course, char degree);
/**
* Si hay siguiente materia para una carrera, retorna 1 y deja en data la info sobre la
* materia. Si no hay más materias para esa carrera retorna 0 y no cambia data
*/
int nextCourse(coursesADT course, char degree, struct courseData * data);
```

Se pide **implementar únicamente las siguientes funciones**, junto con todas las funciones que sean invocadas por las mismas: **newCourses**, **freeCourses**, **addCourse**, **toBeginByDegree** y **nextCourse**.

Programa de prueba:

```c
int main(void) {
   coursesADT course = newCourses();
   // No es válida la carrera '2', sólo letras
   assert(addCourse(course, '2', 10, "Economia", 90)==0);
   assert(students(course, 's')==0); // Aún no hay cursos para la carrera 'S'
   assert(students(course, 'S')==0);
   assert(students(course, '+')==0); // carrera inválida

   // Dos materias con el mismo nombre pero distinto código
   assert(addCourse(course, 'S', 10, "Economia", 90)==1);
   assert(addCourse(course, 'S', 5, "Economia", 20)==1);
   assert(students(course, 's')==110);

   // Ya existe "Economia" con el código 10
   assert(addCourse(course, 'S', 10, "Economia", 20)==0);
   // "Economia" con código 10 pero para otra carrera, es válido
   assert(addCourse(course, 'X', 10, "Economia", 90)==1);

   // Si bien no deberia pasar que tenga el mismo código que "Economia",
   // el TAD no lo valida
   assert(addCourse(course, 's', 10, "Programacion Imperativa", 40)==1);
   assert(students(course, 'S')==150);

   char s[100] = "Optativa 1";
   assert(addCourse(course, 'S', 400, s, 0)==1); // Agregamos "Optativa 1", sin alumnos
   strcpy(s,"Optativa regalada");
   assert(addCourse(course, 'S', 110, s, 120)==1);

   toBeginByDegree(course, 'S');
   struct courseData aux;
   assert(nextCourse(course, 'S', &aux)==1);
   assert(aux.code == 5 && aux.students == 20 && strcmp(aux.name, "Economia")==0);
   free(aux.name);
   assert(nextCourse(course, 'S', &aux)==1);
   assert(aux.code == 10 && strcmp(aux.name, "Economia")==0);
   free(aux.name);
   assert(nextCourse(course, 'S', &aux)==1);
   assert(strcmp(aux.name, "Optativa 1")==0);
   free(aux.name);
   assert(nextCourse(course, 'S', &aux)==1);
   assert(strcmp(aux.name, "Optativa regalada")==0);
   free(aux.name);
   assert(nextCourse(course, 'S', &aux)==1);
```

```
    assert(strcmp(aux.name, "Programacion Imperativa")==0);
    free(aux.name);
    assert(nextCourse(course, 'S', &aux)==0);

    assert(courses(course, 'S')==5); assert(courses(course, 's')==5);
    assert(courses(course, 'T')==0); assert(courses(course, '?')==0);

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

Ejercicio 2

Una gran ciudad o zona urbana (por ejemplo el AMBA) mantiene **información sobre sus colectivos**, cada uno de ellos identificado con un número entero no negativo.

La información puede ser diversa (promedio de pasajeros mensuales, comentarios sobre la calidad del servicio, una estructura que reúna varias estadísticas, etc.) por lo que **se usará un tipo genérico**.

Para ello se definió la siguiente interfaz:

```
typedef struct busCDT * busADT;

typedef ... elemType;

busADT newBusADT(???);

void freeBusADT(busADT adt);

/**
* Registra la información info para la línea de colectivos busNumber.
* Si ya existe, se reemplaza.
*/
void busSet(busADT adt, unsigned int busNumber, elemType info);

/**
* Registra la información info para la línea de colectivos busNumber sólo si no existe.
* Si ya existe, no hace nada.
*/
void busSetIfAbsent(busADT adt, unsigned int busNumber, elemType info);

/**
* Si existe una línea de colectivos con el número busNumber retorna 1 y deja en *info
* una copia de la info asociada. Si no existe retorna cero y no modifica *info
*/
int busInfo(const busADT adt, unsigned int busNumber, elemType * info);

/**
* Elimina una línea de colectivos. Si no existe no hace nada
*/
void removeBus(busADT adt, unsigned int busNumber);

/**
* Retorna la cantidad de líneas de colectivo que se registraron
*/
size_t busSize(const busADT adt);

/**
* Retorna un vector con todos los números de líneas de colectivos que tienen esa info
* asociada. Si no hay ningún colectivo con esa info retorna NULL y deja *dim en cero
*/
int * busCodes(const busADT adt, elemType info, size_t * dim);
```

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 **busSize**, **busSet**, **busSetIfAbsent**, **busInfo** y **removeBus** sean lo más eficientes posibles.
- **Implementar únicamente las siguientes funciones**, junto con todas las funciones que sean invocadas por las mismas: **newBusADT**, **busSet**, **busSetIfAbsent** y **busInfo**.

Programa de prueba, **en este caso se decidió que elemType sea int:**

```
int main(void) {
   busADT buses = newBusADT(_____________);

   assert(busSize(buses)==0);
   int aux;
   assert(busInfo(buses, 104, &aux)==0);

   busSet(buses, 104, 1040); // A la línea 140 se le asocia el entero 1040
   busSetIfAbsent(buses, 104, 500); // No hace nada porque la línea 140 ya tiene info

   assert(busInfo(buses, 104, &aux)==1);
   assert(aux==1040);
   assert(busSize(buses)==1);

   busSetIfAbsent(buses, 29, 10); // Lo agrega pues no había línea 29
   assert(busSize(buses)==2);
   assert(busInfo(buses, 103, &aux)==0);
   assert(busInfo(buses, 28, &aux)==0);

   freeBusADT(buses);
   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 **toDigits** que **reciba un número entero sin signo** (unsigned long), y **retorne una lista con los dígitos del número en orden inverso**.

Si recibe el número 13329 debe retornar la lista 9->2->3->3->1. Si recibe el valor 0 retorna la lista vacía.

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

NO DEFINIR MACROS NI FUNCIONES AUXILIARES.