Nombre y Apellido: _____ Nº Legajo: _____

Segundo Parcial de Programación Imperativa

24/06/2017

	Ejercicio 1	Ejercicio 2	Ejercicio 3	Nota	Firma docente
Entregado				-----	
Calificación					

- **Condición mínima de aprobación: Tener BIEN o BIEN- dos de los tres ejercicios.**
- **Se tendrá en cuenta en la calificación el ESTILO y la EFICIENCIA de los algoritmos.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **Puede entregarse en lápiz.**
- **No usar variables globales ni static.**
- **En caso de necesitar usar malloc o similar, asumir que nunca retornan NULL.**
- **No es necesario escribir los #include**
- **Escribir en cada hoja Nombre, Apellido, Legajo, Número de hoja, Total de hojas entregadas**

Ejercicio 1:

En los dos casos la función debe ser recursiva, recibir únicamente los parámetros mencionados y no definir macros o funciones auxiliares.

Se cuenta con la siguiente definición para listas de enteros, donde una lista vacía se representa con el valor NULL.

```
typedef struct node {
    int elem;
    struct node * tail;
} Tnode;

typedef struct node * Tlist;
```

a) Escribir una función que reciba un vector de enteros positivos, donde la marca de final es el valor -1. El vector está ordenado de menor a mayor y no contiene elementos repetidos. La función debe retornar una lista de enteros (de tipo `Tlist`) ordenada en forma ascendente y sin repetidos.

b) Escribir una función que reciba una lista de enteros ordenada de menor a mayor pero que puede contener elementos repetidos. La función debe retornar una **nueva** lista de enteros ordenada y sin repetidos.

Ejercicio 2:

Un "diccionario genérico" es una colección de datos que asocia una clave con un valor.

- Todas las claves son del mismo tipo
- Todos los valores son del mismo tipo
- No puede haber dos claves iguales
- Puede haber valores repetidos
- No hay límite para la cantidad de pares claves-valor que se inserten

Se desea escribir un TAD para manejar diccionarios genéricos. Para ello se cuenta con la siguiente interfaz:

Archivo dict.h

```
typedef struct dictCDT * dictADT;

/*
** Crea un nuevo diccionario para asociar pares de clave / valor
** Inicialmente está vacío
** Se inserta una copia tanto de la clave como del valor
*/
dictADT newDict( ¿? );

/* Libera todos los recursos reservados por el TAD */
void freeDict(dictADT m);

/*
** Agrega un elemento al diccionario. Si key o value son NULL, no los agrega
**     key: clave. Se inserta una COPIA de la clave recibida
**     value: valor asociado a la clave. Se inserta una COPIA del valor recibido
** Si ya existía la clave, el valor se pisa con value (se lo considera una actualización)
*/
void addKey(dictADT d, const void * key, const void * value);

/*
** Retorna una copia del valor asociado a key o NULL si no estaba la clave
** en el diccionario
*/
void * getValue(dictADT d, const void * key);

/*
** Vector que contenga una copia de todas las claves
** Cada elemento del vector es una COPIA de las claves en el diccionario
** El vector debe finalizar en NULL
** Si no hay claves (el diccionario está vacío) retorna un vector que sólo contiene NULL
 */
void ** keys(dictADT d);
```

Donde ¿? en una lista de parámetros indica que usted (programador) debe definir cuáles son los parámetros necesarios para esa función, en base a las características del TAD.

Implementar el TAD completo

Ejercicio 3:

Se desea implementar un TAD de un calendario de eventos. Un evento consiste en su nombre y la fecha del mismo. Los campos de la fecha del evento son día, mes y año. El calendario admite hasta un evento por fecha. Se cuenta con el siguiente contrato

calADT.h

```
typedef struct calCDT * calADT;

typedef struct tDate {
        unsigned char day;
        unsigned char month;
        unsigned short year;
} tDate;

typedef struct tEvent {
        char * eventName;
        tDate date;
} tEvent;

/* Crea la estructura que dará soporte al almacenamiento de eventos. */
calADT newCal();

/*
** Agrega un evento al calendario.
** Si ya existe un evento en el calendario para la fecha del evento, la función no lo agrega y
** retorna cero. Si pudo agregar el evento retorna 1.
** Se asume que la fecha recibida es válida
*/
int addEvent(calADT cal, tEvent event);

/*
** Funciones de iteración para que el usuario pueda consultar todas los eventos del calendario
** en orden cronológico (del más antiguo al más reciente).
** toBegin() inicializa el iterador en el evento de fecha más antigua del calendario.
**     Si el calendario es vacio no hace nada.
** nextElement() retorna el evento del calendario al que apunta el iterador y hace
**     apuntar el iterador al siguiente evento cronológico del calendario.
**     Si no hay más eventos en el calendario retorna una estructura con todos sus campos de
**     tipo char y short en cero y los punteros en NULL.
*/
void toBegin(calADT cal);
tEvent nextElement(calADT cal);
```

Implementar el TAD completo

Con el siguiente ejemplo de invocación

```
calADT cal = newCal();

tDate d1 = {24, 6, 2017};
tEvent e1 = {"Segundo Parcial", d1};
addEvent(cal, e1);

tDate d2 = {28, 4, 2017};
tEvent e2 = {"Primer Parcial", d2};
addEvent(cal, e2);

tDate d3 = {3, 7, 2017};
tEvent e3 = {"Entrega de notas de cursada", d3};
addEvent(cal, e3);

tDate d4 = {28, 4, 2017}; // Misma fecha que el evento Primer Parcial
tEvent e4 = {"Clase de consulta", d4};
addEvent(cal, e4);

toBegin(cal);

tEvent aux;
do{(aux = nextElement(cal))) {
    if ( aux.date.day > 0 )
        printf("%02d/%02d/%d: %s\n", aux.date.day, aux.date.month, aux.date.year, aux.eventName);
} while ( aux.date.day > 0 )
```

se obtiene la siguiente salida:

```
28/04/2017: Primer Parcial
24/06/2017: Segundo Parcial
03/07/2017: Entrega de notas de cursada
```