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

Segundo Parcial de Programación Imperativa

08/06/2018

	Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*	*Firma Docente*
Entregado				-----	
Calificación	/2	/4	/4		

- **Condición mínima de aprobación: Sumar 5 puntos**
- **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 Apellido, Legajo, Número de hoja y Total de hojas entregadas.**

Ejercicio 1:

Dada una **lista de enteros** cuya definición de tipos es la siguiente:

```
typedef struct node * TList;
typedef struct node {
      int elem;
      struct node * tail;
} TNode;
```

y donde una lista vacía se representa con el valor `NULL`.

Escribir una función recursiva **`elimina`** que recibe únicamente una lista y una función criterio, y elimine de la misma aquellos elementos que cumplen el criterio.

Por ejemplo, el siguiente programa usa la lista {1, 3, 4, 5, 6} y la función `esPar`

```
int
main(void)
{
       TNode * node5 = malloc(sizeof(*node5));
       node5->elem = 6;
       node5->tail = NULL;
       TNode * node4 = malloc(sizeof(*node4));
       node4->elem = 5;
       node4->tail = node5;
       TNode * node3 = malloc(sizeof(*node3));
       node3->elem = 4;
       node3->tail = node4;
       TNode * node2 = malloc(sizeof(*node2));
       node2->elem = 3;
       node2->tail = node3;
       TNode * node1 = malloc(sizeof(*node1));
       node1->elem = 1;
       node1->tail = node2;
       TList res = elimina(node1, esPar);
       while(res != NULL) {
             printf("%d -> ", res->elem);
             res = res->tail;
       }
       printf("NULL");
}
```

```
int
esPar(int n) {
       return n % 2 == 0;
}
```

Luego de la invocación, deja la lista como {1, 3, 5} e imprime `1 -> 3 -> 5 -> NULL`

Ejercicio 2:

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 vacío no hace nada.
** hasNext() retorna 1 (uno) si una invocación a nextElement() retorna un
**     evento válido. Retorna cero si no hay más elementos por consumir en el iterador.
** 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 aborta la ejecución.
*/
void toBegin(calADT cal);
int hasNext(calADT cal);
tEvent nextElement(calADT cal);
```

Implementar el TAD completo

Con el siguiente ejemplo de invocación

```
calADT cal = newCal();

tDate d1 = {8, 6, 2018};
tEvent e1 = {"Segundo Parcial", d1};
addEvent(cal, e1);

tDate d2 = {13, 4, 2018};
tEvent e2 = {"Primer Parcial", d2};
addEvent(cal, e2);

tDate d3 = {30, 6, 2018};
tEvent e3 = {"Entrega de notas de cursada", d3};
addEvent(cal, e3);

tDate d4 = {13, 4, 2018}; // Misma fecha que el evento Primer Parcial
tEvent e4 = {"Clase de consulta", d4};
addEvent(cal, e4);

toBegin(cal);

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

se obtiene la siguiente salida:

```
13/04/2018: Primer Parcial
08/06/2018: Segundo Parcial
30/06/2018: Entrega de notas de cursada
```

Ejercicio X: NO SE PUEDE

Escribir la función genérica **`diferencia`** que recibe dos vectores del mismo tipo y ordenados por el mismo criterio. Los vectores no tienen elementos repetidos.

La función debe retornar un nuevo vector con los elementos que estén en el primer vector pero no en el segundo. El vector de respuesta tiene que estar ordenado por el mismo criterio que los vectores recibidos.

Agregar los parámetros que sean estrictamente necesarios.

Ejemplos:

Si los vectores fueran

```
int v1[] = {8, 7, 6, 5, 4, 3, 2, 1};
int v2[] = {10, 8, 6, 4, 2};
```

La diferencia entre `v1` y `v2` resulta en `{7, 5, 3, 1}`.
La diferencia entre `v2` y `v1` resulta en `{10}`.

Si los vectores fueran

```
char * v1[] = {"abc", "def", "ghi"};
char * v2[] = {"def"};
```

La diferencia entre `v1` y `v2` resulta en `{"abc", "ghi"}`.
La diferencia entre `v2` y `v1` da como resultado el vector vacío.

Ejercicio 1

```
typedef struct nodo * TList;
typedef  struct nodo{
    int elem;
    struct nodo * tail,
}TNodo;

TList elimina (TList l, (int)(*PFun)(int elem)){
    if(l==NULL)
       return;
    if ((*PFun)(l->elem)){
        TNodo *aux =malloc(sizeof(TNodo));
        aux = l->tail;
        free(l)
        return elimina(aux, PFun)
    }
    l->tail = elimina (l->tail, PFun);
    return l;
 }
```

Ejercicio 2

```
struct node{
      tEvent head;
      struct node * tail;
};
typedef struct node * TNode;
struct colCDT{
    TNode first;
    TNode next;
    size_t size;
};

colADT newCol(){
    colADT col = calloc(1, sizeof(struct colCDT));
    if (col == NULL)
        return NULL;
    return col;
}
void toBegin(colADT col){
    col->next = col->first;
    return;
}
int hasNext(colADT col){
    return col->next != NULL;
}
tEvent nextElement (colADT col){
    tEvent aux = col->next->head;
    col->next = col->next->tail;
    return aux;
}
static int compare (tEvent e1, tEvent e2){
    if(e1.date.year == e2.date.year){
        if(e1.date.month == e2.date.month){
            if(e1.date.day > e2.date.day)
                return 1;
            if(e1.date.day < e2.date.day)
                return -1;
            return 0;
        }
        if(e1.date.month > e2.date.month)
            return 1;
        return -1;
    }
    if(e1.date.year > e2.date.year)
        return 1;
    return -1;
}
```

```
static TNode addRec(TNode first, tEvent event, size_t * size){
    int c;
    if(first ==NULL || (c= compare(first->head, event)) ==1){
        TNode aux = malloc(sizeof(struct node));
        aux->head= event;
        aux -> tail = first;
        (* size)++;
        return aux;
    }
    if(c == 0)
       return first;
    first->tail = addRec(first->tail, event, size);
    return first;
}
int addEvent(colADT col, tEvent event){
    size_t aux= col->size;
    col->first= addRec(col->first, event, &(col->size));
    if (aux == col->size)
        return 0;
    return 1;
}
```