-

Programación II. Examen final. 22 de mayo 2017. Parcial 2

Apellidos (en mayúsculas): _____ Nombre: _____

Ejercicio 5 (2,5 puntos) Considerar la definición de lista enlazada (Lista), utilizada en la asignatura:

```
struct _Nodo {
    Elemento *info;
    struct _Nodo *next;
};
typedef struct _Nodo Nodo;

struct _Lista {
    Nodo *first;
};
typedef struct _Lista Lista;
```

Y considerar la siguiente de lista doblemente enlazada (ListDE), donde uno nodo apunta no sólo al nodo siguiente (next), sino también al previo (prev) en una lista.

```
struct _NodoDE {
    Elemento *info;
    struct _NodoDE *prev;
    struct _NodoDE *next;
};
typedef struct _NodoDE NodoDE;

struct _ListaDE {
    NodoDE *first;
};
typedef struct _ListaDE ListaDE;
```

Sin usar primitivas de una Lista, asumir que se puede acceder a la información de sus nodos.

Asumir también que sólo están disponibles las siguientes 4 funciones:

```
Elemento *elemento_copiar(Elemento *);
// devuelve una copia del Elemento pasado como argumento de entrada
NodoDE *nodoDE_crear();
// reserva memoria para un NodoDE y pone sus campos info, prev y next a NULL
ListaDE *listaDE_crear();
// reserva memoria para una ListaDE y pone su campo first a NULL
void listaDE_liberar(ListaDE *);
// libera los nodos y memoria reservada de una ListaDE pasada como argumento de entrada
```

Se pide implementar una función `listaDE_convertir` que recibe una lista enlazada `pl`, y devuelve una lista doblemente enlazada con los elementos de `pl`, sin realizar modificación alguna sobre esta última.

```
ListaDE *listaDE_convertir(Lista *pl);
```

-

Ejercicio 6 (2.5 puntos en total)

a) Crea, a partir de la siguiente expresión sufijo, su árbol de expresión (AdE) correspondiente, mostrando la evolución del algoritmo paso a paso: 2 3 - 7 9 5 * + / ;

b) Considera la siguiente implementación:

```
// En nodoAB.h

#define info(pnodo)  ((pnodo)->info)
#define izq(pnodo)   ((pnodo)->izq)
#define der(pnodo)   ((pnodo)->der)

struct _NodoAB {
        Elemento * info;
        struct _NodoAB *izq;
        struct _NodoAB *der;
};
// En nodoAB.c
typedef struct _NodoAB NodoAB;

NodoAB *nodoab_crear();
void nodoab_liberar(NodoAB *pn);
```

```
// En arbolbinario.c
#define root(pab) ((pab)->root)
struct _ArbolBinario {
        NodoAB *root;
};

// En arbolbinario.h
typedef struct _ArbolBinario ArbolBinario;

ArbolBinario *ab_crear();
boolean ab_vacio(ArbolBinario *pa);
void ab_liberar(ArbolBinario *pa);
```

y las primitivas del TAD elemento:

```
boolean elemento_es_operador(const Elemento *pe);
status elemento_imprime(const Elemento *pe, FILE *f);
```

Implementa una función:

```
status imprime_infijo(const ArbolBinario *pa, FILE *f)
```

que toma como entrada un árbol de expresión e imprime en el fichero f su expresión equivalente en formato infijo, incluyendo los paréntesis asociados a cada operador. Controla los diferentes tipos de error que se puedan dar.

-

Ejercicio 7 (2.5 puntos)

Dada la siguiente definición de árboles binarios:

```
typedef struct _ArbolBinario ArbolBinario;

ArbolBinario *ab_crear();
boolean ab_vacio(ArbolBinario *pa);
void ab_liberar(ArbolBinario *pa);
```

```
#define root(pab) ((pab)->root)
#define info(pnodo)  ((pnodo)->info)
#define izq(pnodo)    ((pnodo)->izq)
#define der(pnodo)   ((pnodo)->der)

struct _NodoAB {
        Elemento * info;
        struct _NodoAB *izq;
        struct _NodoAB *der;
};

/* Funciones privadas "/
NodoAB *nodoab_crear();
void nodoab_liberar(NodoAB *pn);

struct _ArbolBinario {
          NodoAB *root;
};
```

a) Imagina que tienes una empresa y usas un árbol para guardar valores numéricos que representan el sueldo de tus empleados al mes (versión súper simplificada del asunto, con solo un dato a guardar en cada nodo, que es un número entero que representa un sueldo). Como tus empleados están trabajando muy bien, has decidido subirles el sueldo. Escribe el pseudocódigo de un algoritmo que, dado un árbol binario de números enteros, incremente los valores del campo info de todos sus nodos en 200 y devuelva un valor entero que indique cuánto dinero necesitarás en total para poder pagar los nuevos sueldos este próximo mes (no hace falta tener en cuenta posibles errores de desbordamiento en la suma). Imagina que se ha definido la constante INCR y que existe la función incrementar.

b) Proporciona el código C del algoritmo anterior

-

Ejercicio 8. (2.5 puntos). Se desea ordenar una serie de valores de menor a mayor. El vector con los valores a ordenar es:

V 5 10 7 2 15 8

a) Construya el heap insertando los valores del vector paso a paso. Dibuja el estado del heap en cada paso, marcando las operaciones realizadas en él tras cada inserción.

b) Partiendo del vector inicial V, construye un heap, usando ahora el vector directamente como representación del árbol, modificándolo como corresponda hasta obtener un heap. Dibuja el estado del árbol paso a paso, marcando las operaciones realizadas en cada paso.

c) ¿Cuál es la complejidad de los algoritmos de los apartados a y b? Justifica la respuesta.

d) Partiendo del heap obtenido en el apartado a), obtén el vector de datos ordenados de menor a mayor. Dibuja el heap y el vector de valores ordenados tras cada paso del algoritmo correspondiente.

e) ¿Cuál sería la complejidad de un algoritmo para ordenar un array? Justifica la respuesta.

g) ¿Cuál es la estructura de datos más eficiente para implementar el TAD Heap? ¿Por qué? Se podría implementar el TAD Heap con la estructura de datos basada en nodos utilizada en clase para implementar ABs? Justifica la respuesta.

h) Implementa la estructura de C struct _Heap que debería figurar en una librería para que sea genérica (independiente del tipo de datos que se deseen almacenar en un heap, tal y como has hecho en la última práctica). Si su implementación requiriese la definición de punteros a funciones, incluya los typedef apropiados.