

## Primer Parcial de Estructuras de Datos y Algoritmos 2007 2do C

Ejercicio 1	Ejercicio 2	Ejercicio 3	Nota

**Condición Mínima de Aprobación: Tener por lo menos dos ejercicios con B-**

**Realizar cada ejercicio en una hoja separada**  
**Definir todos los tipos de datos utilizados en cada ejercicio**  
**No usar variables globales ni static**  
**Respetar estrictamente los enunciados**  
**Se tendrán en cuenta la eficiencia y el estilo de programación elegido**  
(Recordar las consideraciones sobre cantidad de líneas de una función)

### Ejercicio 1

Se tiene la siguiente interface para un TAD lista lineal doblemente encadenada sin ordenar, sin repeticiones con header.

```
#ifndef _LISTALINEAL_DOBLEMENTEENCADENADA_CONHEADER_SINREPETICIONES_H
#define _LISTALINEAL_DOBLEMENTEENCADENADA_CONHEADER_SINREPETICIONES_H

typedef void* listElementT;

typedef struct listCDT *listADT;

/* Operaciones con las que se puede acceder al TAD */

listADT NewList(int (*fn)(listElementT elem1, listElementT elem2));

int Insert(listADT list, listElementT element);

int Delete(listADT list, listElementT element);

int ListIsEmpty(listADT list);

int ElementBelongs(listADT lista, listElementT element);

void SetBegin(listADT list);

listElementT GetDato(listADT list, int direccion);

/* Funcion:      SwapNodes
 * Uso:          listADT lista;
 *              ...
 *              SwapNodes(lista, 5, 10);
 *              SwapNodes(lista, 8, 3);
 * -----
 * Descripción:  Intercambia dos nodos en la lista. Retorna 1 si pudo realizar la
operación
 *              y 0 en caso contrario.
 * -----
 * Precondicion: Lista no vacía.
 *              Indices que representen nodos existentes en la lista.
 * Postcondicion: Lista con nodos intercambiados.
 */

int SwapNodes(listADT lista, int indice1, int indice2);

void FreeList(listADT lista);

#endif
```

Para este ejercicio vamos a considerar el *índice de un nodo* como la cantidad de nodos previos a él en la lista. Se pide escribir la función **SwapNodes** que dados dos índices en una lista, intercambie los nodos correspondientes. Este intercambio deberá ser hecho de manera tal de **no modificar los contenidos de los nodos**.

Escribir el CDT y la función correspondiente.

Tener en cuenta que el puntero al nodo actual del iterador **debe quedar sin modificar**.

## Ejercicio 2

Un objeto está definido por un conjunto de propiedades. Cada propiedad tiene un nombre y un valor asociado, ambos strings. Cada objeto puede estar asociado a otro objeto o a ninguno. Ningún objeto puede estar asociado directamente a más de un objeto, pero sí indirectamente; es decir, el objeto *obj1* puede estar asociado al objeto *obj2* y este, el objeto *obj2*, estar asociado al objeto *obj3*. Entonces el objeto *obj1* está indirectamente asociado al objeto *obj3* a través del objeto *obj2*. De esta manera las propiedades de un objeto en particular son las definidas en él o las obtenidas de alguno de los objetos con los cuales está asociado. En el caso de necesitar conocer la propiedad de un objeto deberá buscarse primeramente como propiedad del objeto en cuestión y si no se encuentra se deberá buscar dicha propiedad en la cadena de objetos a los cuales está asociado, ya sea directa o indirectamente.

Por ejemplo: Los perros son objetos que tienen entre otras propiedades las de tener cuatro patas y dos orejas. Existe un perro con un nombre particular, “*huesos*”, sin ninguna otra característica que lo distinga de los restantes perros y otro perro denominado “*colita*” que tiene la característica de tener 3 patas, ya que fue atropellado por un colectivo de la línea 146. Sabemos además que “*huesos*” tuvo un hijo, llamado “*puchy*” con la característica de tener 3 orejas. Si quisiéramos conocer la cantidad de patas de “*colita*” deberíamos buscar entre sus propiedades, pero si quisiéramos conocer la cantidad de patas de “*huesos*” deberíamos ver la cantidad de patas de perros, ya que el número de patas no es una propiedad instanciada en el objeto “*huesos*”.

Cada vez que un objeto se asocia a otro ya existente, este último incrementa en uno un contador de objetos asociados a él. Esto nos permitirá no liberar un objeto si tiene otros objetos asociados a él. Solamente va a ser posible liberar definitivamente la memoria ocupada por el objeto cuando nadie lo referencie. Para liberar un objeto, entonces, el contador de objetos asociados deberá estar en cero. Si un objeto puede ser liberado el objeto inmediato asociado a él decrementará la cantidad de objetos asociados en 1.

Para representar estos objetos se define el *objectTAD* con el siguiente contrato:

```
#ifndef      _OBJECT_H
#define      _OBJECT_H

typedef struct objectCDT *objectADT;

/* Operaciones con las que se puede acceder al TAD */

/* Funcion:      NewObject
 * Uso:          objectADT perros;
 *              ...
 *              perros = NewObject();
 * -----
 * Descripción:   Crea un objeto. Retorna NULL en caso de error.
 * -----
 * Precondicion:  -
 * Postcondicion: Objeto creado.
 */
objectADT NewObject(void);

/* Funcion:      SetProperty
 * Uso:          objectADT perros;
 *              ...
 *              perros = NewObject();
 *              SetProperty (perros, "patas", "cuatro");
 *              SetProperty (perros, "orejas", "dos");
 * -----
 * Descripción:   Setea una propiedad del objeto. Si ya existe la reemplaza.
 *              No hay límite para la cantidad de propiedades por objeto.
 * -----
 * Precondicion:  Objeto existente. name y value distintos de NULL.
 * Postcondicion: Propiedad seteada.
```

```

*/
int SetProperty(objectADT obj, char * name, char * value);

/* Funcion:          SetProperty
 * Uso:              objectADT perros;
 *                  char * patas;
 *                  ...
 *                  patas = SetProperty (perros, "patas");
 * -----
 * Descripción:      Obtiene el valor de una propiedad. Si la propiedad no está seteada
en el objeto
 *                  busca dicha propiedad en la jerarquía encima de él.
 *                  De no existir devuelve NULL.
 * -----
 * Precondicion:     Objeto existente. Nombre y valor distintos de NULL.
 * Postcondicion:    Propiedad seteada.
*/
char * SetProperty(objectADT obj, char * name);

/* Funcion:          Inherit
 * Uso:              objectADT colita, perros, huesos;
 *                  ...
 *                  colita = Inherit(perros);
 *                  SetProperty(colita, "patas", "tres");
 *                  huesos = Inherit(perros);
 * -----
 * Descripción:      Crea un nuevo objeto asociándolo a otro objeto ya definido.
 * -----
 * Precondicion:     Objeto existente.
 * Postcondicion:    Objeto asociado al original.
*/
objectADT Inherit(objectADT obj);

/* Funcion:          FreeObject
 * Uso:              objectADT perros;
 *                  int valor;
 *                  ...
 *                  valor = FreeObject(perros);
 * -----
 * Descripción:      Libera un objeto siempre que no tenga objetos asociados.
 *                  Devuelve 1 si el objeto pudo ser liberado y 0 en caso contrario.
 * -----
 * Precondicion:     Objeto existente.
 * Postcondicion:    Objeto liberado.
*/
int FreeObject(objectADT obj);

#endif

```

Se pide definir el **tipo de dato concreto**, la función **GetProperty**, **Inherit** y **FreeObject**.

### Ejemplo

```
perro = NewObject()
```

```
perro
```

refs = 0

```
SetProperty(perro, "patas", "cuatro")
```

```
perro
```

refs = 0  
Props  
patas = cuatro

```
colita = Inherit(perro)
```


```
perro
```

refs = 1  
Props  
patas = cuatro

```
colita
```

refs = 0



```
SetProperty(colita, "patas", "tres")
SetProperty(colita, "atropellado por", "colectivo 146")
```


```
perro
```

refs = 1  
Props  
patas = cuatro

```
colita
```

refs = 0  
Props  
patas = tres  
atropellado por = colectivo 146



```
SetProperty(perro, "orejas", "dos")
```

```
huesos = Inherit(perro)
```


```
perro
```

refs = 2  
Props  
patas = cuatro  
orejas = dos


```
colita
```

refs = 0  
Props  
patas = tres  
atropellado por = colectivo 146

```
huesos
```

refs = 0



```
GetProperty(colita, "patas")
```

```
tres
```

```
GetProperty(huesos, "patas")
```

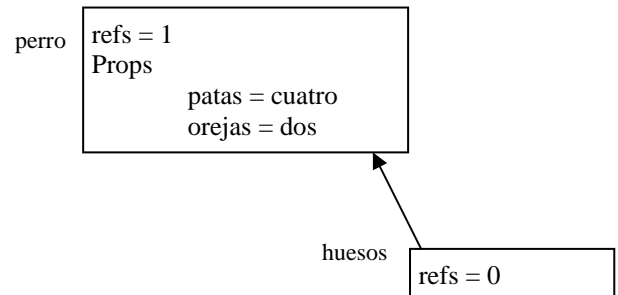
```
cuatro
```

```
FreeObject(perro)
estructura)
```

0 (no se ve modificada la

```
FreeObject(colita)
```

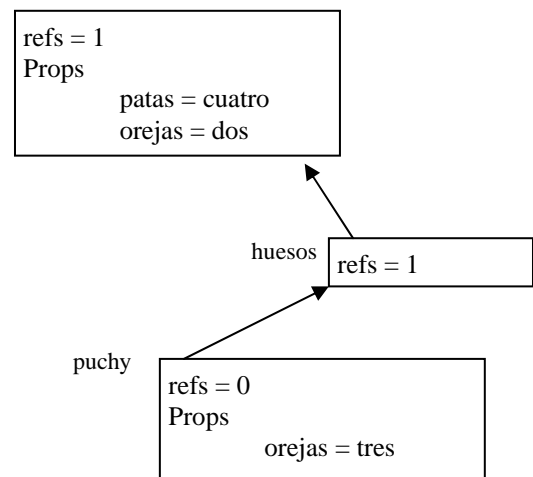
1



```
puchy = Inherit(huesos)
```

perro

```
SetProperty(puchy, "orejas", "tres")
```



```
GetProperty(puchy, "patas")
```

cuatro

```
GetProperty(puchy, "dientes")
```

NULL

### Ejercicio 3

3.1) Escribir la función **join** que recibe como entrada dos matrices (**NO arreglos de arreglos**) de elementos homogéneos de cualquier tipo y devuelve una matriz (**puede ser un arreglo de arreglos**) donde los elementos son del mismo tipo que las originales. Las dimensiones de ambas matrices pueden ser diferentes, por ejemplo, la primer matriz de  $M \times N$  y la segunda de  $K \times L$ . El proceso a realizar es el siguiente: dada una fila en la primer matriz, F1, y una fila en la segunda matriz, F2, si el valor del primer elemento de F1 es igual al valor del último elemento en F2, se genera una fila en la matriz de salida cuyos valores serán todos los elementos de F1 y los primeros  $L-1$  elementos de F2. Este proceso se repite para toda combinación posible de filas de ambas matrices. Recordar el JOIN de SQL.

La función **join** debe contener como **mínimo** los siguientes parámetros:

- Los parámetros formales **matrix1** y **matrix2**. Los elementos de cada matriz son homogéneos pero de cualquier tipo: enteros, doubles, estructuras, punteros a estructuras, punteros a char, etc. Ambas matrices tienen el mismo tipo de elemento.
- El parámetro formal **M** que se corresponde con la cantidad de filas en la primer matriz.
- El parámetro formal **N** que se corresponde con la cantidad de columnas en la primer matriz.
- El parámetro formal **K** que se corresponde con la cantidad de filas en la segunda matriz.
- El parámetro formal **L** que se corresponde con la cantidad de columnas en la segunda matriz.

De tener que agregar algún otro parámetro, por considerarlo necesario, indicar su tipo y descripción.

3.2) Escribir el programa principal que utilice la función anterior. El mismo debe generar el caso mostrado en el próximo ejemplo. (o sea invocar a la función **join** y definir todas las variables y funciones que sean necesarias). Luego de la invocación a la función **join** se deberá imprimir la matriz resultante.

#### Ejemplo de función join

matrix1 (char \*)

matrix2 (char \*)

salida (char \*)

**matrix1 y matrix2 son matrices de punteros a char. La salida queda a elección, aunque obviamente**

aa	bbb	cccc	dd
ee	ff	gg	hh
ii	jj	ll	mmmm
oo	pp	qq	tt
ii	mq	tt	xx
rr	ss	kk	tt

ss	kk	aa
rr	nn	jj
uu	vv	ii
ww	zz	aa
oo	qq	ss

aa	bbb	cccc	dd	ss	kk
aa	bbb	cccc	dd	ww	zz
ii	jj	ll	mmmm	uu	vv
ii	mq	tt	xx	uu	vv

**el elemento debe ser puntero a char.**