Práctico 12 - Memoria dinámica. Punteros y Listas

Programación 1
InCo - Facultad de Ingeniería, Udelar

1. Dadas las siguientes declaraciones:

```
type
    PunteroInt  = ^integer;
    PunteroChar = ^char;
var
    apun1, apun2: PunteroInt;
    apun3, apun4: PunteroChar;
```

   (a) Determine cuáles de las siguientes instrucciones son válidas:

   - √ `new(apun1)`
   - ☐ `new(apun1^)`
   - ☐ `apun1 := apun3`
   - √ `apun2^ := apun2^ + apun1^`
   - ☐ `writeln(apun2, apun3)`
   - √ `read(apun1^, apun4^)`
   - ☐ `apun2 := new(apun1)`
   - √ `dispose(apun3)`
   - √ `apun1 := NIL`
   - ☐ `apun3^ := NIL`
   - ☐ `apun3 := apun4 and (apun3 = NIL)`

   (b) Determine la salida que despliega el siguiente fragmento de código:

```
new(apun1);
new(apun2);
new(apun3);
apun2 := apun1;
apun1^ := 2;
apun2^ := 3;
apun3^ := 'A';
writeln(apun1^, apun2^, apun3^)
```

> Ejecutar el código. Notar que en este código se deja colgada la celda de memoria resevada por `new(apun2)`.

   (c) ¿Tiene algún error el siguiente fragmento de código?

```
new(apun1);
read(apun1^);
writeln(apun1^);
dispose(apun1);
writeln(apun1^)
```

> Si, tiene un error. En la instrucción `dispose(apun1)` se libera la memoria a la que apunta `apun1`. Sin embargo, en la instrucción `writeln(apun1^)` se intenta acceder a una posición de memoria no reservada.

(d) Determine la salida que despliega el siguiente fragmento de código:

```
new(apun3);
new(apun1);
apun3^ := 'Z';
apun2 := NIL;
apun4 := NIL;
if (apun3 <> NIL) and (apun2 = NIL) then
    writeln ('A');
if apun3^ = 'Z' then
    writeln ('Z')
else
    writeln ('X')
```

Ejecutar el código.

2. Dado el siguiente programa:

```
Program
type
   TipoVehiculo = (barco, camion);
   Transporte = record
                    capacidad : integer;
                    case vehiculo : TipoVehiculo of
                         barco  : (habitaciones : integer);
                         camion : ();
                end;
   PunteroTransporte = ^Transporte;

var a, b, c : PunteroTransporte;

begin
   new(a);
   a^.capacidad := 30;
   a^.vehiculo := barco;
   a^.habitaciones := 4;

   new(b);
   b^.capacidad := 5;
   b^.vehiculo := camion;

   new(c);
   c^.capacidad := 5;
   c^.vehiculo := camion;
end.
```

(a) ¿Cuál de las figuras (Figura 1) representa el estado de la memoria tras ejecutarse las instrucciones anteriores?

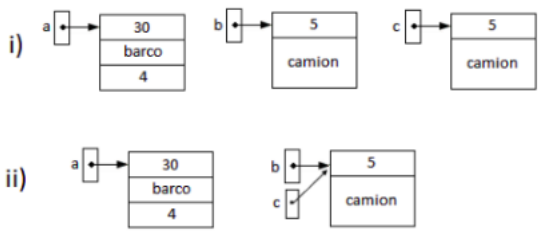

Figura 1: Ejercicio 2a

√ **i**
☐ ii

(b) Suponiendo que al final del programa se agrega la instrucción: `b := c`

I) ¿Cuál de las figuras (Figura 2) representa el estado de la memoria posterior a la ejecución de dicha instrucción?

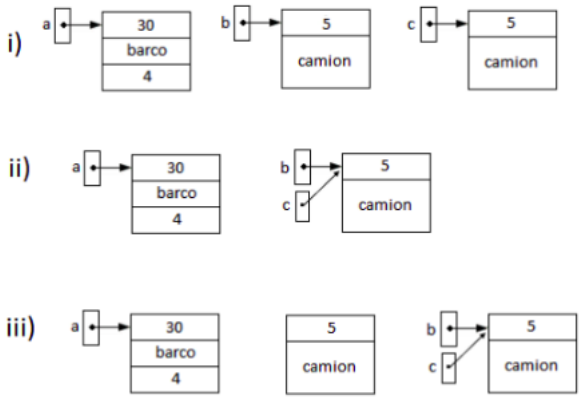

Figura 2: Ejercicio 2b

☐ i
☐ ii
√ **iii**

II) ¿Cuál o cuáles de las afirmaciones son correctas?

☐ Cada puntero referencia un lugar de memoria distinto
☐ `b` y `c` referencian al mismo lugar de memoria. No se desperdicia memoria
√ **`b` y `c` referencian al mismo lugar de memoria, queda memoria ocupada y se pierde su referencia**

> Es importante tener en cuenta que queda un espacio de memoria ocupado, pero que no es referenciado por ninguna variable y por lo tanto no va a poder liberarse.

III) ¿Sería correcto hacer `dispose(b)` antes de la instrucción `b := c`? Justifique.

> Correcto, de no hacerse la memoria reservada queda ocupada sin ser referenciada por ninguna variable.

(c) Suponiendo ahora que al final del programa dado al comienzo del ejercicio, se le agregan las instrucciones

```
c^.capacidad := 4;
b^ := c^
```

I) Dibuje el estado de la memoria posterior a la ejecución de dicha instrucción

II) ¿Qué valor tiene `b^.capacidad` tras la ejecución de dichas instrucciones?

☐ 30
√ **4**
☐ 5

Para los ejercicios que siguen considere la siguiente definición de tipos para representar una lista encadenada de enteros:

```
type
    Positivo = 1..MaxInt;
    ListaInt = ^TCelda;
```

```
TCelda   = record
    info : integer;
    sig  : ListaInt
end;
```

3. Escribir los siguientes subprogramas:
   (a) Una función que retorne la suma de todos los elementos de la lista. Si la lista es vacia retorna 0.

```
function suma(lst : ListaInt) : integer;
```

```
{Si la lista es vacia devuelve 0}
function suma(lst : ListaInt) : integer;
var
    sumaL : integer;
begin
    sumaL := 0;

    while lst <> NIL do
    begin
        sumaL := sumaL + lst^.info;
        lst := lst^.sig
    end;

    suma := sumaL
end;
```

   (b) Una función que retorne el mínimo de todos los elementos de la lista. Se asume que la lista no es vacía.

```
function minimo(lst : ListaInt) : integer;
```

```
{PRE: lst no es vacia}
function minimo(lst : ListaInt) : integer;
var
    min : integer;
begin
    min := lst^.info;
    lst := lst^.sig;

    while lst <> NIL do
    begin
        if lst^.info < min then
            min := lst^.info;
        lst := lst^.sig
    end;

    minimo := min
end;
```

   (c) Una función que retorne el producto de todos los elementos de la lista. Tenga en cuenta que alguno de los elementos puede ser 0. Si la lista es vacía, devuelve 1.

```
function producto(lst : ListaInt) : integer;
```

```
{Si la lista es vacia devuelve 1}
function producto(lst : ListaInt) : integer;
var
```

```
    prod : integer;
begin

    prod := 1;

    while (prod <> 0) and (lst <> NIL) do
    begin
        prod := prod * lst^.info;
        lst := lst^.sig
    end;

    producto := prod
end;
```

(d) Una función que retorne la cantidad de números pares que hay en la lista.

```
function cuantosPares(lst : ListaInt) : integer;
```

```
function cuantosPares(lst : ListaInt) : integer;
var
    pares : integer;
begin
    pares := 0;

    while lst <> NIL do
    begin
        if lst^.info mod 2 = 0 then
            pares := pares + 1;
        lst := lst^.sig
    end;

    cuantosPares := pares
end;
```

(e) Una función que retorne la primera posición en la cual aparece `valor` dentro de la lista `lst`. Las posiciones se cuentan desde `1` en adelante. Si valor no aparece en la lista, la función retorna `-1`.

```
function posicion(valor : integer; lst : ListaInt) : integer;
```

```
function posicion(valor : integer; lst : ListaInt) : integer;
var
    pos : integer;
begin
    pos := 1;

    while (lst <> NIL) and (lst^.info <> valor) do
    begin
        pos := pos + 1;
        lst := lst^.sig
    end;

    if lst = NIL then
        posicion := -1
    else
        posicion := pos
end;
```

(f) Un procedimiento que obtiene el valor del elemento de la lista que está en la posición `pos`. Más precisamente el procedimiento retorna un registro con variante de tipo `PosibleElem` ya que puede no existir el elemento buscado.

```
type
PosibleElem = record case ok : boolean of
              true  : (elem : integer);
              false : ()
            end;

procedure elemEnPos(pos : Positivo; lst : ListaInt; var resultado : PosibleElem);
```

```
procedure elemEnPos(pos : Positivo; lst : ListaInt; var resultado : PosibleElem);
var
    iterPos : integer;
begin
    iterPos := 1;

    while (lst <> NIL) and (iterPos <> pos) do
    begin
        iterPos := iterPos + 1;
        lst := lst^.sig
    end;

    if lst = NIL then
        resultado.ok := false
    else
    begin
        resultado.ok := true;
        resultado.elem := lst^.info
    end
end;
```

(g) Una función que retorna el último elemento de la lista. Se asume que la lista no es vacía.

```
function ultimo(lst : ListaInt) : integer;
```

```
{PRE: lst no es vacia}
function ultimo(lst : ListaInt) : integer;
begin
    while lst^.sig <> NIL do
        lst := lst^.sig;

    ultimo := lst^.info
end;
```

(h) Una función que determina si la lista está ordenada de menor a mayor:

```
function ordenada(lst : ListaInt) : boolean;
```

```
{Si la lista es vacia retorna TRUE}
function ordenada(lst : ListaInt) : boolean;
begin
    if lst = NIL then
        ordenada := TRUE
    else
    begin
```

```
        while (lst^.sig <> NIL) and (lst^.info < lst^.sig^.info) do
            lst := lst^.sig;

        ordenada := lst^.sig = NIL
    end
end;
```

4. Escribir los siguientes subprogramas:
   (a) Insertar un elemento luego del segundo elemento de la lista. Si no hubiera segundo, la lista no cambia.

   ```
   procedure insertarTercero(elem : integer; var lst : ListaInt);
   ```

```
procedure insertarTercero(elem : integer; var lst : ListaInt);
var
     nuevo : ListaInt;
begin
    if (lst <> NIL) and (lst^.sig <> NIL) then
    begin
        new(nuevo);
        nuevo^.info := elem;
        nuevo^.sig := lst^.sig^.sig;
        lst^.sig^.sig := nuevo
    end
end;
```

   (b) Insertar un elemento antes del último. Si no hubiera último, la lista no cambia.

   ```
   procedure insertarPenultimo(elem : integer; var lst : ListaInt);
   ```

```
procedure insertarPenultimo(elem : integer; var lst : ListaInt);
var
     nuevo, iter : ListaInt;
begin
    if (lst <> NIL) then
    begin
        new(nuevo);
        nuevo^.info := elem;

        if (lst^.sig = NIL) then
        begin
            nuevo^.sig := lst;
            lst := nuevo
        end
        else
        begin
            iter := lst;
            while (iter^.sig^.sig <> NIL) do
                iter := iter^.sig;

            nuevo^.sig := iter^.sig;
            iter^.sig := nuevo
        end
    end
end;
```

   (c) Insertar el elemento nuevo luego del elemento que está en la posición pos. Si no existiera un elemento en tal posición, la lista no cambia.

   ```
   procedure InsertarLuegoPos(nuevo : integer; pos : Positivo; var lst : ListaInt);
   ```

```
procedure InsertarLuegoPos(nuevo : integer; pos : Positivo; var lst : ListaInt);
var
 iterPos : integer;
 celda, iterL : ListaInt;
begin
    if (lst <> NIL) then
    begin
        iterPos := 1;
        iterL := lst;
        while (iterL^.sig <> NIL) and (iterPos <> pos) do
        begin
            iterPos := iterPos + 1;
            iterL := iterL^.sig
        end;

        if iterPos = pos then
        begin
            new(celda);
            celda^.info := nuevo;
            celda^.sig := iterL^.sig;
            iterL^.sig := celda
        end
    end
end;
```

(d) Insertar un elemento `nuevo` antes de la primera aparición del elemento `valor`. Si no existiera tal elemento, la lista no cambia.

`procedure InsertarAntes(nuevo,valor : integer; var lst : ListaInt);`

```
procedure InsertarAntes(nuevo,valor : integer; var lst : ListaInt);
var
    p,q : ListaInt;
begin
    if (lst <> nil)  then
        if (lst^.info = valor) then
        begin
        new(q);
            q^.info:= nuevo;
            q^.sig:= lst;
            lst:= q
        end
        else
        begin
            p:= lst;
            while (p^.sig <> nil) and (p^.sig^.info <> valor) do
            p:= p^.sig;

            if p^.sig <> nil then
            begin
                new(q);
                q^.info:= nuevo;
                q^.sig:= p^.sig;
                p^.sig:= q
            end
        end
end;
```

(e) Insertar un elemento en una lista ordenada. El orden debe mantenerse luego de la inserción.

```
procedure InsertarOrdenado(nuevo : integer; var lst : ListaInt);
```

```
{PRE: lst esta ordenada}
procedure InsertarOrdenado(nuevo : integer; var lst : ListaInt);
var
    celda, iterL : ListaInt;
begin
    new(celda);
    celda^.info := nuevo;

    if (lst = NIL) or (nuevo <= lst^.info) then
    begin
        celda^.sig := lst;
        lst := celda
    end
    else
    begin
        iterL := lst;
        while (iterL^.sig <> NIL) and (iterL^.sig^.info < nuevo) do
            iterL := iterL^.sig;

        celda^.sig := iterL^.sig;
        iterL^.sig := celda
    end
end;
```

(f) Borrar el segundo elemento de la lista. Si no hubiera segundo, la lista no cambia.

```
procedure BorrarSegundo(var lst : ListaInt);
```

```
procedure BorrarSegundo(var lst : ListaInt);
var
    borrar : ListaInt;
begin
    if (lst <> NIL) and (lst^.sig <> NIL) then
    begin
        borrar := lst^.sig;
        lst^.sig := borrar^.sig;
        dispose(borrar)
    end
end;
```

(g) Escribir procedimientos para las siguientes operaciones sobre una lista de enteros:

i) borrar el último

ii) borrar el primer número impar

iii) borrar todos los números pares

```
procedure BorrarUltimo(var lst : ListaInt);
var
    iterL : ListaInt;
begin
    if (lst <> NIL) then
    begin
        if (lst^.sig = NIL) then
        begin
            dispose(lst);
            lst := NIL
        end
```

```
        else
        begin
            iterL := lst;
            while (iterL^.sig^.sig <> NIL) do
                iterL := iterL^.sig;

            dispose(iterL^.sig);
            iterL^.sig := NIL
        end
    end
end;
```

```
procedure BorrarPrimerImpar(var lst : ListaInt);
var
    iterL, borrar : ListaInt;
begin
    if (lst <> NIL) then
    begin
        if (lst^.info mod 2) <> 0 then
        begin
            borrar := lst;
            lst := borrar^.sig;
            dispose(borrar)
        end
        else
        begin
            iterL := lst;
            while (iterL^.sig <> NIL) and (iterL^.sig^.info mod 2 = 0) do
                iterL := iterL^.sig;

            if (iterL^.sig <> NIL) then
            begin
                borrar := iterL^.sig;
                iterL^.sig := borrar^.sig;
                dispose(borrar)
            end
        end
    end
end;
```

```
procedure BorrarTodosPares(var lst : ListaInt);
var
    iterL, borrar : ListaInt;
begin
    while (lst <> NIL) and (lst^.info mod 2 = 0) do
    begin
        borrar := lst;
        lst := borrar^.sig;
        dispose(borrar)
    end;

    if (lst <> NIL) then
    begin
        iterL := lst;
        while (iterL^.sig <> NIL) do
            if iterL^.sig^.info mod 2 = 0 then
            begin
```

```
                borrar := iterL^.sig;
                iterL^.sig := borrar^.sig;
                dispose(borrar)
            end
            else
                iterL := iterL^.sig
    end
end;
```

(h) Borrar la primera aparición de valor en la lista. Si no existiera tal elemento, la lista no cambia.

```
procedure BorrarPrimeraAparicion(valor : integer; var lst : ListaInt);
```

```
procedure BorrarPrimeraAparicion(valor : integer; var lst : ListaInt);
var
    iterL, borrar :  ListaInt;
begin
    if (lst <> nil)  then
        if (lst^.info = valor) then
        begin
            borrar:= lst;
            lst:= lst^.sig;
            dispose(borrar)
        end
        else
        begin
            iterL:= lst;
            while (iterL^.sig <> nil) and (iterL^.sig^.info <> valor) do
                iterL:= iterL^.sig;

            if iterL^.sig <> nil then
            begin
                borrar:= iterL^.sig;
                iterL^.sig:= borrar^.sig;
                dispose(borrar)
            end
        end
end;
```

5. Indique qué **errores** presentan los siguientes procedimientos, en los que hay una o más instrucciones incorrectas. Explique.

(a) Se desea crear una lista de un solo elemento con valor 1.

```
procedure crearListaUnitaria (var nuevo : ListaInt);
begin
    new(nuevo);
    nuevo := NIL;
    nuevo^.dato := 1;
    nuevo^.sig := NIL;
end;
```

La instrucción `nuevo := NIL` no debe ir. Este código produce un error en tiempo de ejecución al intentar desreferenciar una variable cuyo valor es NIL.

(b) Se desea insertar un elemento al final de una lista.

```
procedure agregarAlFinal (dato : Integer; var lista : ListaInt);
var it, nuevo : ListaInt;
begin
```

```
    new(nuevo);
    nuevo^.dato := dato;
    if lista = NIL then
        lista := nuevo
    else
    begin
            new(it);
            it := lista;
            while it^.sig <> NIL do
                it := it^.sig;
            it^.sig := nuevo
    end
end;
```

> Este código presenta dos errores. El primer error es que falta indicar que la nueva celda de la lista es la última, es decir, falta la instrucción `nuevo^.sig := NIL`. El segundo error es que la instrucción `new(it)` no debe ir. En esta instrucción se reserva memoria para la variable `it`. Sin embargo la siguiente instrucción define `it` como un alias a lista y esto hace que la memoria previamente reservada quede inaccesible.

(c) Lo mismo que en la parte anterior, pero con otro error.

```
procedure agregarAlFinal (dato : Integer; var lista : ListaInt);
var it, nuevo : ListaInt;
begin
    new(nuevo);
    nuevo^.dato := dato;
    nuevo^.sig := NIL;
    if lista = NIL then
        lista := nuevo
    else begin
            it := lista;
            while it <> NIL do
                it := it^.sig;
            it := nuevo
    end
end;
```

> Este código presenta dos errores. El primer error está en la condición de iteración del `while`. La iteración debe parar en el últmo elemento de la lista, es decir, la variable `it` debe quedar apuntando a la celda cuyo puntero al siguiente vale NIL. La condición de iteración debe ser `it^.sig <> NIL`. El segundo error es en la instrucción `it := nuevo`. Lo correcto es modificar el puntero `it^.sig` y apuntarlo a la nueva celda de memoria creada.

6. Implementar los siguientes subpogramas:

(a) Retornar una lista con los primeros `k` múltiplos positivos de `num`. La lista debe estar ordenada de menor a mayor. Se supone `num > 1`

```
function multiplos(k : Positivo; num : Positivo) : ListaInt;
```

```
function multiplos(k : Positivo; num : Positivo) : ListaInt;
var
    res, celda : ListaInt;
    i : Integer;
begin
    res := NIL;

    for i := k downto 1 do
    begin
```

```
        new(celda);
        celda^.info := num*i;
        celda^.sig := res;
        res := celda
    end;

    multiplos := res
end;
```

(b) Retornar una copia limpia de la lista `lst`. Esto es, una lista con el mismo contenido que `lst` pero que no comparte ninguna celda con esta.

`function copia(lst : ListaInt) : ListaInt;`

```
function copia(lst : ListaInt) : ListaInt;
var
    copiaL, iterC : ListaInt;
begin
    if (lst = NIL) then
        copiaL := NIL
    else
    begin
        new(copiaL);
        copiaL^.info := lst^.info;
        iterC := copiaL;

        lst := lst^.sig;
        while (lst <> NIL) do
        begin
            new(iterC^.sig);
            iterC := iterC^.sig;
            iterC^.info := lst^.info;
            lst := lst^.sig
        end;

        iterC^.sig := NIL
    end;

    copia := copiaL
end;
```

(c) Retornar la lista `lst` invertida. La lista resultado no comparte memoria con la lista `lst`.

`function  invertir(lst : ListaInt) : ListaInt;`

```
function  invertir(lst : ListaInt) : ListaInt;
var
    invL, celda : ListaInt;
begin
    invL := NIL;
    while (lst <> NIL) do
    begin
        new(celda);
        celda^.info := lst^.info;
        celda^.sig := invL;
        invL := celda;
        lst := lst^.sig
    end;
```

```
    invertir := invL
end;
```

(d) Invertir la lista `lst`, modificando directamente los punteros dentro de la lista sin crear nuevas celdas.

`procedure invertir(var lst : ListaInt);`

```
procedure  invertir(var lst : ListaInt);
var
    nueva,q :  ListaInt;
begin
   nueva:= nil;
   while lst <> nil do
   begin
      { quito primera celda de lst}
      q:= lst;
      lst:= lst^.sig;

      { inserto q al principio de nueva}
      q^.sig:= nueva;
      nueva:= q
   end;

   lst:= nueva
end;
```

(e) Realizar la concatenación de dos listas `l1` y `l2`. En `l1` quedan todos los elementos originales seguidos de los elementos de `l2`. No se deben utilizar celdas nuevas.

`procedure concatenar(var l1 : ListaInt; l2 : ListaInt);`

```
procedure concatenar(var l1 : ListaInt; l2 : ListaInt);
var
    iterL : ListaInt;
begin
    if (l1 = NIL) then
        l1 := l2
    else
    begin
        iterL := l1;

        while (iterL^.sig <> NIL) do
            iterL := iterL^.sig;

        iterL^.sig := l2
    end
end;
```

(f) Partir una lista `lst` en dos listas de tal manera que `l1` contenga los primeros `k` elementos de `lst` y `l2` contenga los restantes. Si `lst` tuviera menos de `k` elementos, `l2` queda vacía y `l1` queda igual a `lst`. No se deben utilizar celdas nuevas.

```
procedure partir(k : Positivo; lst : ListaInt;
         var l1,l2 : ListaInt);
```

```
procedure partir(k : Positivo; lst : ListaInt;
                 var l1,l2 : ListaInt);
```

```
var
   p    : ListaInt;
   cont : integer;
begin
   p:= lst;
   cont:= 1;
   { buscar k-esimo }
   while (p <> nil) and (cont < k) do
   begin
      cont:= cont + 1;
      p:= p^.sig
   end;
   l1 := lst;
   if p <> nil then
   begin { p apunta al k-esimo}

      l2:= p^.sig;     { l2 comienza en el k+1 }
      p^.sig:= nil     { p pasa a ser último de l1}
   end
   else { la lista tiene menos de k elementos }
      l2:= nil
end;
```

(g) Escriba una función que, dadas dos listas de números enteros `l1` y `l2`, ordenadas de manera ascendente, obtenga una nueva lista que tenga los elementos de ambas listas ordenados de manera ascendente. La nueva lista no debe compartir memoria ni con `l1` ni con `l2`.

`function IntercalarListas (l1, l2: ListaInt) : ListaInt;`

```
function IntercalarListas (l1, l2: ListaInt) : ListaInt;
var
    resL, iterRes : ListaInt;
begin
    {Se crea la primer celda dummy}
    new(resL);
    iterRes := resL;

    while (l1 <> NIL) and (l2 <> NIL) do
    begin
        new(iterRes^.sig);
        iterRes := iterRes^.sig;
        if (l1^.info < l2^.info) then
        begin
            iterRes^.info := l1^.info;
            l1 := l1^.sig
        end
        else
        begin
            iterRes^.info := l2^.info;
            l2 := l2^.sig
        end
    end;

    while (l1 <> NIL) do
    begin
        new(iterRes^.sig);
        iterRes := iterRes^.sig;
        iterRes^.info := l1^.info;
        l1 := l1^.sig
```

```
    end;

    while (l2 <> NIL) do
    begin
        new(iterRes^.sig);
        iterRes := iterRes^.sig;
        iterRes^.info := l2^.info;
        l2 := l2^.sig
    end;

    iterRes^.sig := NIL;

    iterRes := resL;
    resL := resL^.sig;
    {se libera la memoria de la celda dummy}
    dispose(iterRes);

    IntercalarListas := resL
end;
```