Ejercicios con estructuras enlazadas lineales

Bloque 1. Creación de código

Salvo que se diga lo contrario explícitamente, en todos los ejercicios de este documento se usarán las siguientes definiciones para representar listas de enteros mediante estructuras enlazadas de simple enlace con cabecera, es decir, con una celda de encabezamiento precediendo a las celdas que realmente contengan los elementos de la lista:

```
typedef int Elemento;

struct Nodo {
      Elemento elem;
      struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;
```

1. Completa el código de la siguiente función para que construya y devuelva una lista vacía.

   ```
   NodoPtr crea()
   ```

2. Completa el código de la siguiente función para que construya y devuelva una nueva lista con un único elemento, el pasado como parámetro.

   ```
   NodoPtr crea( Elemento d )
   ```

3. Completa el código de la siguiente función para que construya y devuelva una nueva lista con todos los números enteros en el intervalo [a,b] en orden.

   ```
   NodoPtr crea( Elemento a, Elemento b )
   ```

4. Completa el código de la siguiente función para que construya y devuelva una nueva lista con n números enteros generados aleatoriamente en el intervalo [a,b].

   ```
   NodoPtr crea( int n, Elemento a, Elemento b )
   ```

5. Completa el código de la siguiente función para que construya y devuelva una nueva lista con los n valores contenidos en el array a.

   ```
   NodoPtr crea( Elemento a[], int n )
   ```

6. Suponiendo que l apunta a la cabecera de una lista, completa el código de la siguiente función para que libere toda la memoria asociada a la misma.

   ```
   void libera( NodoPtr l )
   ```

7. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que muestre todos los elementos separados por comas.

```
void muestra( NodoPtr l )
```

8. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que inserte como primer elemento de la lista `l` un nuevo nodo con el dato `d`.

```
void inserta( NodoPtr l, Elemento d )
```

9. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que inserte como último elemento de la lista `l` un nuevo nodo con el dato `d`.

```
void inserta( NodoPtr l, Elemento d )
```

10. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están repetidos ni ordenados, completa el código de la siguiente función para que añada un nuevo nodo con el dato `d` si este no se encuentra ya en la lista.

```
void inserta( NodoPtr l, Elemento d )
```

11. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor pero entre los que sí puede haber repetidos, completa el código de la siguiente función para que añada un nodo a la lista con el dato `d` dejando la lista ordenada.

```
void inserta( NodoPtr l, Elemento d )
```

12. Suponiendo que `l` apunta a la cabecera de una lista, y `p` apunta a un nodo de la misma, completa el código de la siguiente función para que añada un nodo a la lista con el dato `d` a continuación del nodo apuntado por `p`.

```
void inserta( NodoPtr l, NodoPtr p, Elemento d )
```

13. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor pero entre los que sí puede haber repetidos, completa el código de la siguiente función para que inserte los `n` elementos del array `a` dejando la lista ordenada.

```
void inserta( NodoPtr l, Elemento a[], int n )
```

14. Suponiendo que `l` apunta a la cabecera de una lista, y que `0 < i <= n+1`, siendo `n` la longitud de la lista `l`, completa el código de la siguiente función para que añada un nodo a la lista con el dato `d` en la posición `i`-esima.

```
void inserta( NodoPtr l, int i, Elemento d )
```

15. Suponiendo que `l` apunta a la cabecera de una lista, y que `0 < i <= n`, siendo `n` la longitud de la lista `l`, completa el código de la siguiente función para que añada un nodo a la lista con el dato `d` en la posición `i`-esima empezando a contar por el final.

```
void inserta( NodoPtr l, int i, Elemento d )
```

16. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si el elemento `d` se encuentra en la lista y 0 en caso contrario.

```
int existe( NodoPtr l, Elemento d )
```

17. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor y no repetidos, completa el código de la siguiente función para que devuelva 1 si el elemento `d` se encuentra en la lista y 0 en caso contrario.

```
int existe( NodoPtr l, Elemento d )
```

18. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si el elemento `d` se encuentra en la lista, al menos `n` veces, y 0 en caso contrario.

```
int existe( NodoPtr l, int n, Elemento d )
```

19. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si ninguno de los `n` elementos contenidos en el array `a` se encuentra en la lista y 0 en caso contrario.

```
int noexiste( NodoPtr l, Elemento a[], int n )
```

20. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre el primer nodo de la lista.

```
void suprime( NodoPtr l )
```

21. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre el último nodo de la lista.

```
void suprime( NodoPtr l )
```

22. Suponiendo que `l` apunta a la cabecera de una lista, y `p` apunta a cualquier nodo de la misma, completa el código de la siguiente función para que borre el nodo siguiente al apuntado por `p` siempre que `p` no apunte al último nodo de la lista.

```
void suprime( NodoPtr l, NodoPtr p )
```

23. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre todos los elementos de la lista dejando la lista vacía.

```
void suprime( NodoPtr l )
```

24. Suponiendo que `l` apunta a la cabecera de una lista, y que `0 < i <= n`, siendo `n` la longitud de la lista `l`, completa el código de la siguiente función para que borre el i-esimo elemento de la lista.

```
void suprime( NodoPtr l, int i )
```

25. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre el primer elemento cuyo valor sea igual a `d`.

```
void suprime( NodoPtr l, Elemento d )
```

26. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor y sin repetir, completa el código de la siguiente función para que borre el primer elemento cuyo valor sea igual a `d`.

```
void suprime( NodoPtr l, Elemento d )
```

27. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están ordenados y pueden estar repetidos, completa el código de la siguiente función para que borre el último de los elementos cuyo valor sea igual a `d`.

```
void suprime( NodoPtr l, Elemento d )
```

28. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre todos los elementos cuyo valor sea igual a `d`.

```
void suprime( NodoPtr l, Elemento d )
```

29. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre todos los elementos situados en posiciones pares.

```
void suprime( NodoPtr l, Elemento d )
```

30. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre todos los elementos cuyo valor esté en el intervalo [a,b].

```
void suprime( NodoPtr l, Elemento a, Elemento b )
```

31. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre, como máximo, los primeros `n` elementos cuyo valor **no** esté en el intervalo [a,b].

```
void suprime( NodoPtr l, int n, Elemento a, Elemento b )
```

32. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre todos los elementos cuyo valor coincida con alguno de los `n` elementos del array `a`.

```
void suprime( NodoPtr l, Elemento a[], int n )
```

33. Suponiendo que l apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva el número de elementos de la lista.

```
int cuenta( NodoPtr l )
```

34. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva la suma de todos los elementos.

```
int suma( NodoPtr l )
```

35. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos son todos mayores o iguales a cero, completa el código de la siguiente función para que devuelva el mayor valor o -1 si la lista está vacía.

```
int maximo( NodoPtr l )
```

36. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva el número de elementos iguales a `d`.

```
int cuenta( NodoPtr l, Elemento d )
```

37. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo los primeros `n` elementos de `l` o todos si no hay suficientes.

```
NodoPtr recupera( NodoPtr l, int n )
```

38. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo todos los elementos situados en posiciones pares.

```
NodoPtr recupera( NodoPtr l )
```

39. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo todos los elementos de `l` cuyo valor esté en el intervalo [`a`,`b`].

```
NodoPtr recupera( NodoPtr l, Elemento a, Elemento b )
```

40. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo todos los elementos de `l` cuyo valor sea un número primo.

```
NodoPtr recupera( NodoPtr l )
```

41. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que rellene el array a con un máximo de `n` elementos de `l` cuyo valor sea mayor que `min`. La función devolverá el número de elementos almacenados en `a`.

```
int recupera( NodoPtr l, Elemento min, Elemento a[], int n )
```

42. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo los últimos `n` elementos de `l` o todos si no hay suficientes.

```
NodoPtr recupera( NodoPtr l, int n )
```

43. Suponiendo que `l` apunta a la cabecera de una lista completa, y que sus elementos están ordenados de menor a mayor y no repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo, como máximo `n` elementos de `l` que sean menores que `max`.

```
NodoPtr recupera( NodoPtr l, int n, Elemento max )
```

44. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que sustituya por `d` todos los elementos cuyo valor sea igual a alguno de los `n` elementos del array `a`.

```
void sustituye( NodoPtr l, Elemento a[], int n, Elemento d )
```

45. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que sustituya por 0, como máximo, los primeros `n` elementos cuyo valor sea igual a `d`.

```
void sustituye( NodoPtr l, int n, Elemento d )
```

46. Suponiendo que `l` apunta a la cabecera de una lista, y que contiene dos o más elementos, completa el código de la siguiente función para que intercambie el primer elemento con el último de la lista.

```
void intercambia( NodoPtr l )
```

47. Suponiendo que `l` apunta a la cabecera de una lista, y que contiene dos o más elementos, completa el código de la siguiente función para que intercambie el situado en la posición `i` por el situado en la posición `j` siempre que ambas posiciones sean válidas, es decir, `0 < i < j <= n` siendo `n` la longitud de `l`.

```
void intercambia( NodoPtr l, int i, int j )
```

48. Suponiendo que `a` y `b` apuntan a la cabecera de sendas listas, completa el código de la siguiente función para que construya y devuelva una nueva lista con todos los elementos de `a`, y a continuación, todos los de `b`.

```
NodoPtr concatena( NodoPtr a, NodoPtr b )
```

49. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista con el mismo contenido y ordenación de sus elementos de la primera.

```
NodoPtr duplica( NodoPtr l )
```

50. Suponiendo que `a` y `b` apuntan a la cabecera de sendas listas, completa el código de la siguiente función para que construya y devuelva una nueva lista con todos los elementos de `a` intercalados con los de `b`. Por ejemplo, si `a`={1,2,3} y `b`={5,6,7} la función devolverá una nueva lista con {1,5,2,6,3,7}

```
NodoPtr intercala( NodoPtr a, NodoPtr b )
```

51. Suponiendo que `a` y `b` apuntan a la cabecera de sendas listas, completa el código de la siguiente función para que construya y devuelva una nueva lista con todos los elementos de ambas listas intercalados de forma aleatoria.

```
NodoPtr baraja( NodoPtr a, NodoPtr b )
```

52. Suponiendo que `a` y `b` apuntan a las cabeceras de sendas listas, completa el código de la siguiente función para que inserte todos los elementos de `b` al final de `a`.

```
void inserta( NodoPtr a, NodoPtr b )
```

53. Suponiendo que `a` y `b` `b` apuntan a las cabeceras de sendas listas, completa el código de la siguiente función para que inserte todos los elementos de `b` al principio de `a`.

```
void inserta( NodoPtr a, NodoPtr b )
```

54. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que inserte una copia de si misma al final. Por ejemplo, si la lista original es {1,2,3,4} tras ejecutar la función la lista debe ser {1,2,3,4,1,2,3,4}.

```
void inserta( NodoPtr l )
```

55. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que inserte a continuación de cada elemento otro igual a él. Por ejemplo, si la lista original es {1,2,3} tras ejecutar la función la lista será {1,1,2,2,3,3}.

```
void inserta( NodoPtr l )
```

56. Suponiendo que `l` apunta a la cabecera de una lista, y que `0 < i <= n`, siendo `n` la longitud de la lista `l`, completa el código de la siguiente función para que borre el i-esimo elemento de la lista empezando por el final.

```
void suprime( NodoPtr l, int i )
```

57. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor y sin repetir, completa el código de la siguiente función para que borre el mayor de sus elementos que sea menor que `max`.

```
void suprime( NodoPtr l, Elemento max )
```

58. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que borre el menor de sus elementos.

```
void suprime( NodoPtr l )
```

59. Suponiendo que `l` apunta a la cabecera de una lista, cuyos elementos no están repetidos ni ordenados, completa el código de la siguiente función para que borre los tres mayores elementos.

```
void suprime( NodoPtr l )
```

60. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que deje la lista sin elementos repetidos.

```
void suprime( NodoPtr l )
```

61. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si los `n` elementos contenidos en el array `a` se encuentra en la lista en el mismo orden y de forma consecutiva y 0 en caso contrario.

```
int existe( NodoPtr l, Elemento a[], int n )
```

62. Suponiendo que `a` y `b` apuntan a las cabeceras de sendas listas, completa el código de la siguiente función para que devuelva 1 si la secuencia de elementos de la segunda se encuentra en la primera y 0 en caso contrario.

```
int existe( NodoPtr a, NodoPtr b )
```

63. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si contiene algún elemento repetido y 0 en caso contrario.

```
int existe( NodoPtr l )
```

64. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que devuelva 1 si contiene algún elemento repetido más de `n` veces y 0 en caso contrario.

```
int existe( NodoPtr l, int n )
```

65. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo, como máximo, sus 3 mayores elementos.

```
NodoPtr recupera( NodoPtr l )
```

66. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo, como máximo sus `n` mayores elementos.

```
NodoPtr recupera( NodoPtr l, int n )
```

67. Suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo, como máximo los `n` mayores elementos de `l` que sean menores que `max`.

```
NodoPtr recupera( NodoPtr l, int n, Elemento max )
```

68. Suponiendo que `l` apunta a la cabecera de una lista, y que sus elementos están ordenados de menor a mayor y no repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista conteniendo, como máximo, los `n` mayores elementos de `l` cuyo valor esté en el intervalo [`a`,`b`].

```
NodoPtr recupera( NodoPtr l, Elemento a, Elemento b, int n )
```

69. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva un array en memoria dinámica conteniendo, como máximo, los primeros `n` elementos de `l` que sean menores que `max`.

```
Elemento * recupera( NodoPtr l, int n, Elemento max )
```

70. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que sustituya los 3 menores elementos por `d`.

```
void sustituye( NodoPtr l, Elemento d )
```

71. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que sustituya por , como máximo, los `n` menores elementos por `d`.

```
void sustituye( NodoPtr l, int n, Elemento d )
```

72. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que intercambie cada elemento con su siguiente.

```
void intercambia( NodoPtr l )
```

73. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que intercambie el menor elemento por el mayor de la lista.

```
void intercambia( NodoPtr l )
```

74. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que la ordene de menor a mayor repitiendo para cada nodo el siguiente proceso. Compara el valor del nodo actual con todos los que haya a continuación e intercámbialos cuando alguno sea menor.

```
void ordena( NodoPtr l )
```

75. Suponiendo que `l` apunta a la cabecera de una lista, completa el código de la siguiente función para que construya y devuelva una nueva lista con los elementos de `l` en el orden contrario.

```
NodoPtr invierte( NodoPtr l )
```

76. Suponiendo que `a` y `b` apuntan a la cabecera de sendas listas cuyos elementos no están ordenados y entre los que puede haber repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista sólo con los elementos que estén ambas listas.

```
NodoPtr intersecta( NodoPtr a, NodoPtr b )
```

77. Suponiendo que `a` y `b` apuntan a la cabecera de sendas listas cuyos elementos están ordenados de menor a mayor y entre los que no hay repetidos, completa el código de la siguiente función para que construya y devuelva una nueva lista ordenada con los elementos de ambas listas.

```
NodoPtr mezcla( NodoPtr a, NodoPtr b )
```

78. Redefiniendo el tipo `Elemento` para que sea una cadena de 80 caracteres de longitud máxima, y suponiendo que `l` apunta a la cabecera de una lista cuyos elementos no están repetidos ni ordenados, completa el código de la siguiente función para que añada un nuevo nodo con el dato `d` si este no se encuentra ya en la lista.

```
void inserta( NodoPtr l, Elemento d )
```

79. Redefiniendo el tipo `Elemento` para que sea una cadena de 80 caracteres de longitud máxima, y suponiendo que `l` apunta a la cabecera de una lista cuyos elementos están ordenados de menor a mayor y entre los que puede haber repetidos, completa el código de la siguiente función para que borre el último de los elementos cuyo valor sea igual a `d`.

```
void suprime( NodoPtr l, Elemento d )
```

80. Suponiendo que el tipo `Elemento` se redefine para que sea un `char`. Completa la función para que construya y devuelva una lista cuyos elementos sean cada una de las letras incluidas en la cadena que se le pasa como parámetro en mismo orden que el de la cadena y sin incluir el carácter '\0'.

```
NodoPtr test( char * palabra )
```

Bloque 2. Comprensión de código

1. Si suponemos que p es una variable de tipo NodoPtr que apunta a un nodo de una lista ¿qué hace el siguiente código?

```
if ( p->sig != NULL ) {
      NodoPtr aux = p->sig
      p->sig = p->sig->sig;
      free( aux );
}
```

2. Si suponemos que p es una variable de tipo NodoPtr que apunta a un nodo de una lista ¿Cuándo fallaría el siguiente código?

```
NodoPtr aux = p->sig
p->sig = p->sig->sig;
free( aux );
```

3. Si suponemos que p es una variable de tipo NodoPtr que apunta a la cabecera de una lista ¿Cuándo fallaría el siguiente código?

```
NodoPtr aux = p->sig
p->sig = p->sig->sig;
free( aux );
```

4. Suponiendo que existe una variable p de tipo NodoPtr que apunta a un nodo de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = malloc( sizeof ( struct Nodo ) );
aux->elem = 0;
aux->sig = p->sig;
p->sig = aux;
```

5. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p;
while ( aux->sig != NULL ) {
      if ( aux->sig->elem > 10 ) {
            aux->sig->elem = 0;
      }
      aux = aux->sig;
}
```

6. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p->sig;
while ( aux != NULL ) {
      NodoPtr aux2 = aux;
      aux = aux->sig;
      free( aux2 );
}
```

7. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p;
while ( aux->sig != NULL ) {
      if ( aux->sig->elem < 10 ) {
            NodoPtr aux2 = aux->sig;
            aux->sig = aux2->sig;
            free( aux2 );
      } else {
            aux = aux->sig;
      }
}
```

8. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p->sig;
while ( aux != NULL ) {
      printf("%d\n", aux->elem);
      aux = aux->sig;
}
```

9. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p;
int suma = 0;
while ( aux->sig != NULL ) {
      if ( aux->sig->elem > 0 ) {
            suma = suma + aux->sig->elem;
      }
      aux = aux->sig;
}
printf("%d\n", suma);
```

10. Suponiendo que existe una variable p de tipo NodoPtr que apunta a la cabecera de una lista ¿Qué hace el siguiente código?

```
NodoPtr aux = p;
while (aux->sig != NULL && aux->sig->elem < 100 ) {
      aux = aux->sig;
}
if ( aux->sig != NULL && aux->sig->elem == 100 ) {
      printf("Ok\n");
}
```

11. La función libera tiene como objetivo borrar toda la memoria asociada a la lista pasada como parámetro. ¿Por qué falla?

```
void libera ( NodoPtr lista ) {
    NodoPtr borrar = lista;
    libera( lista->sig );
    free( borrar );
}
```

12. La función libera tiene como objetivo borrar toda la memoria asociada a la lista pasada como parámetro. ¿Por qué falla?

```
void libera ( NodoPtr lista ) {
    NodoPtr aux = lista;
    while ( aux != NULL ) {
        NodoPtr borrar = aux;
        free( borrar );
    }
}
```

13. La función libera tiene como objetivo borrar toda la memoria asociada a la lista pasada como parámetro. ¿Por qué falla?

```
void libera ( NodoPtr lista ) {
    NodoPtr aux = lista;
    while ( aux == NULL ) {
        NodoPtr borrar = aux;
        aux = aux->sig;
        free( borrar );
    }
}
```

14. La función borra tiene como objetivo borrar el siguiente elemento al apuntado por p dejando la estructura enlazada bien conectada. ¿Por qué falla?

```
void borra( NodoPtr p ) {
    NodoPtr aux = p->sig;
    p->sig = aux->sig;
    free( p->sig );
}
```

15. La función borra tiene como objetivo borrar el siguiente elemento al apuntado por p dejando la estructura enlazada bien conectada. ¿Por qué falla?

```
void borra( NodoPtr p ) {
    NodoPtr aux = p;
    p = p->sig;
    free( aux->sig );
}
```

16. La función borra tiene como objetivo borrar todos los nodos de la lista cuyo dato coincida con d dejando la estructura enlazada bien conectada. ¿Por qué falla?

```
void borra( NodoPtr lista, Elemento d ) {
    NodoPtr aux = lista;
    while ( aux->sig != NULL ) {
        if ( aux->sig->elem == d ) {
            NodoPtr borrar = aux->sig;
            aux->sig = borrar->sig;
            free(borrar);
        }
        aux = aux->sig;
    }
}
```

17. La función borra tiene como objetivo borrar todos los nodos de la lista cuyo dato coincida con d dejando la estructura enlazada bien conectada. ¿Por qué falla?

```
void borra( NodoPtr lista, Elemento d ) {
    NodoPtr aux = lista;
    while ( aux->sig != NULL && aux->sig->elem != d ) {
        if ( aux->sig->elem == d ) {
            NodoPtr borrar = aux->sig;
            aux->sig = borrar->sig;
            free(borrar);
        } else {
            aux = aux->sig;
        }
    }
}
```

18. La función inserta tiene como objetivo añadir un nuevo nodo con el dato d al final de la lista con cabecera pasada como parámetro. ¿Por qué falla?

```
void inserta( NodoPtr cabecera, int d ) {
    NodoPtr aux = cabecera;
    while ( aux != NULL ) {
        aux = aux->sig;
    }
    aux = malloc( sizeof( Nodo ) );
    aux->dato = d;
    aux->sig = NULL;
}
```

19. La función inserta tiene como objetivo añadir un nuevo nodo con el dato d al final de la lista con cabecera pasada como parámetro. ¿Por qué falla?

```
void inserta( NodoPtr cabecera, int d ) {
    NodoPtr aux = cabecera;
    while ( aux->sig != NULL ) {
        aux = aux->sig;
    }
    aux = malloc( sizeof( Nodo ) );
    aux->dato = d;
    aux->sig = NULL;
}
```

20. La función inserta tiene como objetivo añadir un nuevo nodo con el dato d al final de la lista con cabecera pasada como parámetro. ¿Por qué falla en algunos casos?

```
void insertaFinal( NodoPtr cabecera, int d ) {
    NodoPtr aux = cabecera->sig;
    while ( aux->sig != NULL ) {
        aux = aux->sig;
    }
    aux->sig = malloc( sizeof( Nodo ) );
    aux->sig->dato = d;
    aux->sig->sig = NULL;
}
```

Bloque 3. Estructuras doblemente enlazadas

1. Repite todos los ejercicios del bloque 1 pero con estructuras enlazadas de doble enlace definidas del siguiente modo:

```
typedef int Elemento;

struct Nodo {
     Elemento elem;
     struct Nodo * sig;
     struct Nodo * ant;
};
typedef struct Nodo * NodoPtr;
```