

Cuaderno de problemas
Estructuras de Datos. TADs lineales

Prof. Isabel Pita

1 de marzo de 2022

Índice

1. Recorrido de listas enlazadas simples. Tipos genéricos y propiedades genéricas.	3
1.1. Objetivos del problema	4
1.2. Ideas generales.	4
1.3. Algunas cuestiones sobre implementación.	4
1.4. Errores frecuentes.	10
1.5. Implementación.	10
2. Modificación de listas enlazadas simples. Agrupar elementos iguales.	12
2.1. Objetivos del problema	14
2.2. Ideas generales.	14
2.3. Algunas cuestiones sobre implementación.	14
2.4. Errores frecuentes.	16
2.5. Implementación.	17
3. Listas enlazadas dobles. Agrupar elementos.	19
3.1. Objetivos del problema	20
3.2. Ideas generales.	20
3.3. Algunas cuestiones sobre implementación.	21
3.4. Errores frecuentes.	26
3.5. Implementación.	26
4. Pilas, invertir una secuencia.	29
4.1. Objetivos del problema	30
4.2. Ideas generales.	30
4.3. Algunas cuestiones sobre implementación.	30
4.4. Coste de la implementación realizada.	30
4.5. Implementación.	31
5. Pilas, guardar información relevante.	32
5.1. Objetivos del problema	33
5.2. Ideas generales.	33
5.3. Algunas cuestiones sobre implementación.	34
5.4. Coste de la implementación realizada.	34
5.5. Implementación.	34
6. Iteradores. My search	36
6.1. Objetivos del problema	38
6.2. Ideas generales.	38
6.3. Ideas detalladas.	38
6.4. Algunas cuestiones sobre implementación.	38
6.5. Errores frecuentes.	39
6.6. Coste de la solución	39
6.7. Implementación.	40

1. Recorrido de listas enlazadas simples. Tipos genéricos y propiedades genéricas.

Recorrer una lista enlazada simple

En este ejercicio se trata de practicar con la estructura de datos de listas enlazadas simples `linked_list.ed`. Se leerán una serie de valores de la entrada que se guardarán en una lista enlazada simple. A continuación, se recorrerá la lista mostrando por pantalla todos los valores que cumplan una cierta propiedad. En este problema la propiedad pedida es que sean múltiplos de un número dado.

Requisitos de implementación.

Para implementar el ejercicio se extenderá la clase `linked_list.ed` con un método que la recorra y muestre los valores pedidos.

El método debe permitir mostrar los elementos que cumplen cualquier propiedad.

Al llamar al método se especificará que la propiedad pedida sea que los elementos son múltiplos de un número.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se muestra en dos líneas. En la primera se muestra el número de elementos de la lista y el valor del que se quieren mostrar los múltiplos. En la línea siguiente se muestran todos los valores de la lista. Los valores se añaden a la lista por la derecha.

Los valores de la lista son números enteros entre cero y 10^6

Salida

Para cada caso de prueba se escribirán en una línea todos los valores de la lista múltiplos del valor dado.

Entrada de ejemplo

```
6 2
2 4 6 8 10 12
6 2
1 3 5 7 9 11
6 2
1 2 3 4 5 6
```

Salida de ejemplo

```
2 4 6 8 10 12
2 4 6
```

Autor: Isabel Pita

1.1. Objetivos del problema

- Conocer una implementación de lista enlazada simple.
- Practicar el uso de las listas enlazadas simples. Crear una lista y recorrerla.
- Uso de la herencia para crear nuevas clases que añadan operaciones a un TAD.
- Practicar las clases genéricas y los objetos función.

1.2. Ideas generales.

- En el problema nos piden que construyamos una lista enlazada simple con los elementos de la entrada y que después la recorramos.
- Utilizaremos la implementación de la lista enlazada simple `linked_list_ed` que se proporciona en el campus virtual. El tipo de datos nos ofrece:
 1. la estructura `Nodo` con un campo genérico para guardar el valor del nodo y un puntero al siguiente `Nodo`,
 2. dos punteros a la lista: `prim` nos permite acceder al primer elemento de la lista y `ult` nos permite acceder al último elemento de la lista.
 3. Operaciones para construir una lista vacía, destruir la lista, copiar y asignar una lista, añadir elementos por el principio (izquierda) de la lista y por el final (derecha) de la lista, eliminar el primer elemento de la lista y comprobar si la lista está vacía.
- Para construir la lista utilizaremos la operación `push_back` que añade los elementos por el final.
- La operación de recorrer la lista mostrando los valores que cumplen una cierta propiedad es una operación nueva del TAD. En este curso no modificaremos los TADs proporcionados en el campus, sino que utilizaremos herencia para crear nuevas clases que tengan la funcionalidad de la clase de la que heredan e incorporen la funcionalidad de las operaciones que se van realizando en los ejercicios.
- La operación que recorre la lista lo hará mediante un puntero que empieza en el primer elemento de la lista y va siguiendo los nodos hasta llegar al último.
- Para comprobar la propiedad que se pide se utilizarán funciones parametrizadas respecto a la propiedad de los elementos que se muestran.

1.3. Algunas cuestiones sobre implementación.

- Código de la clase `linked_list_ed`.
 1. La clase `linked_list_ed` es una clase genérica, es decir los tipos y métodos definidos en la clase dependen de uno o más parámetros que se instancian en el momento en que se crea un objeto de la clase. Para indicar que una clase es genérica se utiliza la palabra reservada `template` como se muestra en el código siguiente. Entre angulos se indica el identificador del tipo genérico a utilizar en la implementación de la clase. El identificador puede ser cualquier identificador válido, normalmente se utiliza `T`. Se permite utilizar la palabra reservada `class` en lugar de `typename` para mantener la compatibilidad de las versiones anteriores de C++.

```
template <typename T>
class linked_list_ed {
    ....
};
```
 2. Para crear un objeto de esta clase indicaremos el tipo de los elementos de la lista.

```
linked_list_ed<int> lista;
```
 3. Si se quiere utilizar más de un tipo genérico, estos se muestran separados por comas.

```
template <typename T, typename S>
class linked_list_ed {
.....
};
```

y al instanciarlo se indicarán todos los tipos:

```
linked_list_ed<int, char> lista;
```

4. Como utilizaremos herencia para implementar las nuevas operaciones, los atributos de la clase los declaramos protegidos: `protected`. De esta forma la clase que hereda tiene acceso a los atributos. (En la clase que hereda los atributos se deben referenciar siempre con `this`).
5. El `struct Nodo` consta de dos campos, el elemento que será del tipo genérico y un puntero a una estructura de tipo `Nodo`. Ofrece dos constructores, uno sin parámetros que inicializa el puntero al siguiente nodo al valor `nullptr`, y otro constructor con uno/dos parámetros que serán el valor del elemento y el valor del puntero. Observad que el valor del puntero por defecto es `nullptr`.
6. La clase `linked_list_ed` tiene dos atributos: un puntero al primer nodo, `prim`, y un puntero al último nodo de la lista, `ult`.
7. Las operaciones públicas de la lista son:

- a) Todos los TADs que utilicen memoria dinámica deben tener los siguientes 4 métodos: constructor, destructor, constructor por copia y operador de asignación.
- b) Un constructor sin parámetros que inicializa los dos punteros: `prim` y `ult` al valor `nullptr`.

```
linked_list_ed() : prim(nullptr), ult(nullptr) {}
```

- c) El destructor de la lista. El destructor es necesario porque debemos devolver al sistema toda la memoria utilizada por los nodos de la lista. Esto se hace mediante la función privada `libera` implementada en la parte privada de la clase. El destructor tiene el mismo nombre que la clase, precedido del símbolo `~`. No se indica valor de retorno y no tiene parámetros. Una clase solo puede tener un destructor.

```
~linked_list_ed() {
    libera();
}
```

- d) Un constructor por copia, que recibe un objeto del mismo tipo y realiza una *copia profunda* de la lista, es decir una copia de todos los nodos que no comparta memoria con la lista original. Utiliza la función privada `copia` que reserva memoria para todos los nodos de la lista a copiar y copia los valores.

```
linked_list_ed(linked_list_ed<T> const& other) {
    copia(other);
}
```

- e) El operador de asignación que nos permite realizar una *copia profunda* de la lista. Para ello se libera la memoria de la lista en la que se va a realizar la copia y a continuación mediante la función privada `copia` se reserva memoria para todos los nodos de la lista a copiar y se copian los valores. Observad que se devuelve el contenido del objeto, para que se puedan anidar varias instrucciones de asignación.

```
linked_list_ed<T>& operator= (
    linked_list_ed<T> const& other) {
    if (this != &other) {
        libera();
        copia(other);
    }
    return *this;
}
```

- f) Añadir un elemento al final de la lista: **push_back**. Se reserva memoria para un nuevo nodo con el operador **new** que llama al constructor del **Nodo** con el valor del elemento como parámetro. Conecta el nuevo nodo al final de la lista, accediendo al último puntero de la lista por medio del puntero **ult**. Actualiza el valor del puntero **ult** para que quede en el último nodo de la lista.

```
void push_back(T const& elem) {
    Nodo * nuevo = new Nodo(elem);
    if (ult != nullptr)
        ult->sig = nuevo;
    ult = nuevo;
    if (prim==nullptr) // lista vacia
        prim = nuevo;
}
```

Hay que tener en cuenta que si la lista inicial era vacía hay que dar valor al puntero **prim** para que quede en el primer nodo de la lista (si la lista estaba vacía tenía el valor **nullptr**). Si la lista no estaba vacía no hay que modificar este puntero.

- g) Añadir un elemento al principio de la lista: **push_front**. Se reserva memoria para un nuevo nodo con el operador **new** que llama al constructor del **Nodo** con dos parámetros: el valor del elemento y el valor que se asignará al puntero **sig**. Actualiza el valor del puntero **prim** para que quede en el primer nodo de la lista. Si la lista inicial era vacía actualiza el valor del puntero **ult**.

```
void push_front(T const& elem) {
    Nodo* nuevo=new Nodo(elem, prim);
    prim = nuevo;
    if (ult==nullptr) // lista vacia
        ult = nuevo;
}
```

- h) Eliminar el primer nodo de la lista: **pop_front**. Esta operación es parcial, lanza una excepción cuando la lista es vacía, ya que no puede eliminar el elemento. Si la lista no es vacía, declaramos un puntero al primer nodo de la lista para no perder su dirección de memoria. A continuación modificamos el valor del puntero **prim** para que apunte al segundo elemento de la lista, que pasará a ser el primero una vez que eliminemos el primer nodo. Observamos que si la lista tenía un único elemento debemos dar el valor **nullptr** al puntero **ult**. Por último devolvemos la memoria del nodo que eliminamos al sistema con la instrucción **delete**.

```

void pop_front() {
    if (empty())
        throw std::domain_error("lista vacia");
    Nodo * a_borrar = prim;
    prim = prim->sig;
    if (prim==nullptr) // ha quedado vacia
        ult = nullptr;
    delete a_borrar;
}

```

- i) No se implementa la operación de eliminar por la parte derecha de la lista porque esta operación tiene coste $\mathcal{O}(n)$ siendo n el número de elementos de la lista en las listas enlazadas simples. Esto es debido a la necesidad de recorrer la lista para que el puntero `ult` quede al final de la lista una vez eliminado el elemento.
- j) Comprobar si una lista es vacía: `empty`. Comprueba si el puntero al principio de la lista tiene el valor `nullptr`.

```

bool empty() const {
    return this->prim == nullptr;
}

```

8. Se definen dos operaciones protegidas en las listas:

- a) **libera**. Libera la memoria de todos los nodos de la lista. Para ello recorre la lista utilizando el puntero `prim`. En cada vuelta del bucle, guarda la dirección de memoria del primer nodo en un puntero `a_borrar`, avanza el puntero `prim` para que guarde la dirección de memoria del siguiente nodo y devuelve al sistema la memoria del primer nodo con la instrucción `delete`. El bucle termina cuando no hay más nodos que eliminar en la lista.

```

void libera() {
    while (prim != nullptr) {
        Nodo * a_borrar = prim;
        prim = prim->sig;
        delete a_borrar;
    }
    ult = nullptr;
}

```

- b) **copia**. Crea una lista a partir de otra duplicando los nodos. Si la lista que copiamos es vacía, se crea una lista vacía. En otro caso recorre la lista que se copia desde el primer nodo hasta el último utilizando un puntero auxiliar `act`. Para cada nodo reserva memoria en la nueva lista y le da el valor del elemento de la lista a copiar en el constructor del `Nodo`. El puntero `ult` de la lista se va manteniendo en todo momento al final de la lista que se va creando.

```

void copia(linked_list_ed const& other) {
    if (other.empty()) {
        prim = ult = nullptr;
    }
    else {
        Nodo * act = other.prim; // recorre la cola original
        ult = new Nodo(act->elem); // ultimo nodo copiado
        prim = ult;
        while (act->sig != nullptr) {
            act = act->sig;
            ult->sig = new Nodo(act->elem);
            ult = ult->sig;
        }
    }
}

```

- Código de la clase `linked_list_ed_plus` que usaremos para implementar las nuevas operaciones de la lista.

1. La nueva clase es también una clase genérica, por lo que utilizaremos la cabecera `template <typename T>`.
2. La clase hereda de forma pública de la clase `linked_list_ed`. Debemos también indicar que utilizaremos el tipo `Nodo` en la nueva clase con el mismo nombre, para ello utilizamos la declaración `using`. Observad que en la clase que hereda nos referimos siempre con `this` a los atributos y métodos de la clase heredada.

```
template <typename T>
class linked_list_ed_plus : public linked_list_ed<T> {
    using Nodo = typename linked_list_ed<T>::Nodo;
public:
    ....
};
```

3. La operación que se nos pide se implementa en la parte pública. Vemos primero la operación que muestra los múltiplos de un valor y a continuación la operación que muestra los elementos que cumplen una propiedad genérica, de forma que se puedan comparar ambas implementaciones.

a) Operación que muestra los valores múltiplos de uno dado:

- La operación recibe como primer parámetro el valor del que queremos obtener los múltiplos y como segundo parámetro el buffer en el que se escribirán.
- La operación se declara `const` ya que no se modifican los atributos de la clase.
- Para evitar escribir un blanco al final de la salida buscamos el primer valor múltiplo del valor dado con un bucle. Observad que siempre hay que comprobar que no se ha terminado la lista antes de acceder al elemento para comprobar si es múltiplo. Se avanza en la lista con la instrucción `aux = aux->sig` hasta que se termina la lista o se encuentra un valor múltiplo del dado.
- A continuación si existe el valor, lo mostramos en el buffer de salida que nos pasaron como parámetro y continuamos recorriendo la lista mostrando el resto de los valores múltiplos del valor dado.

```
void mostrarMultiplos(
    int n, std::ostream & o = std::cout) const {
    Nodo * aux = this->prim;
    while (aux != nullptr && aux->elem % n != 0)
        aux = aux->sig;
    if (aux != nullptr) {
        o << aux->elem;
        aux = aux->sig;
        while (aux != nullptr) {
            if (aux->elem % n == 0)
                o << ' ' << aux->elem;
            aux = aux->sig;
        }
    }
    o << '\n';
}
```

b) Operación que muestra los valores que cumplen una propiedad:

- Declaramos un tipo genérico por medio de la palabra reservada `template`. La propiedad genérica será del tipo declarado y es un parámetro de la función.

```
template <typename Predicate>
void mostrarPropiedad(
    Predicate pred, std::ostream& o=std::cout) const {
    ...
}
```


- Cuando necesitemos comprobar que un valor cumple la propiedad utilizaremos el parámetro.

```
template <typename Predicate>
void mostrarPropiedad(
    Predicate pred, std::ostream& o=std::cout) const {
    Nodo * aux = this->prim;
    while (aux != nullptr && !pred(aux->elem)) <==
        aux = aux->sig;
    if (aux != nullptr) {
        o << aux->elem;
        aux = aux->sig;
        while (aux != nullptr) {
            if (pred(aux->elem)) <==
                o << ' ' << aux->elem ;
            aux = aux->sig;
        }
    }
    o << '\n';
}
```

- Código para resolver el problema:

- Debemos declarar un objeto de clase `linked_list_ed_plus` para poder llamar al método de la clase. En nuestro problema la lista es de valores enteros (`int`).

```
linked_list_ed_plus<int> lista;
```

- Se leen los datos de la lista y se añaden con el método `push_back`.
- Se llama al método `mostrarPropiedad`. Para ello necesitamos tener implementada la propiedad que nos piden. Esto se puede hacer de dos formas:
 - Utilizando la sobrecarga del operador paréntesis. Este operador se declara siempre dentro de una clase y debe ser público. Si la propiedad que nos piden requiere un valor, como en el caso de *ser múltiplo de...* guardaremos este valor en un atributo de la clase. Para dar valor al atributo necesitamos un constructor con parámetros.

```
class multiplo {
public:
    multiplo(int num) : n(num) {}
    ....
private:
    int n;
};
```

La sobrecarga del operador paréntesis es una función: `operator()` En nuestro caso recibe un valor y devuelve cierto si el valor es el múltiplo del atributo de la clase y falso en caso contrario.

```
class multiplo {
public:
    multiplo(int num) : n(num) {}
    bool operator()(int valor) {
        return valor % n == 0;
    }
private:
    int n;
};
```

- Utilizando una *lambda expression*. Estas permiten implementar pequeñas funciones como instrucciones del código. La *lambda expression* que calcula si un valor es múltiplo de otro dado es:

```
[n](int valor){return valor % n == 0;}
```

1. Una *lambda expression* comienza con `[]` y entre los dos corchetes se pueden indicar variables del programa que se van a utilizar en la *lambda expression*. En nuestro caso `[n]` se utiliza la variable `n` que se ha leído de la entrada.
2. Entre paréntesis se indican los parámetros de la función, igual que ocurre con las funciones *normales*.
3. El cuerpo de la función se escribe entre llaves. En nuestro caso se calcula el módulo del parámetro de entrada con el valor `n` y se compara con cero.

- Si se ha utilizado la sobrecarga del operador paréntesis, la llamada a la función es:

```
lista.mostrarPropiedad(multiplo(n));
```

siendo `n` el valor del que se quieren mostrar los múltiplos.

- Si se utiliza una *lambda expression* la llamada al método de la clase es:

```
lista.mostrarPropiedad(
    [n](int valor){return valor % n == 0;});
```

1.4. Errores frecuentes.

No se conocen

1.5. Implementación.

La implementación de la clase `linked_list_ed` se proporciona en el campus virtual.

```
#include "linked_list_ed.h"

template <typename T>
class linked_list_ed_plus : public linked_list_ed<T> {
    using Nodo = typename linked_list_ed<T>::Nodo;
public:

    // Recorrer la lista y mostrar los elementos que
    // cumplan una propiedad
    template <typename Predicate>
    void mostrarPropiedad(Predicate pred, std::ostream& o=std::cout) const{
        Nodo * aux = this->prim;
        while (aux != nullptr && !pred(aux->elem))
            aux = aux->sig;
        if (aux != nullptr) {
            o << aux->elem;
            aux = aux->sig;
            while (aux != nullptr) {
                if (pred(aux->elem)) o << ' ' << aux->elem ;
                aux = aux->sig;
            }
            o << '\n';
        }
    };

    class multiplo {
    public:
        multiplo(int num) : n(num) {}
        bool operator()(int valor) {
            return valor % n == 0;
        }
    private:
        int n;
    };
};
```

```

bool resuelveCaso() {
    int numElem, n;
    std::cin >> numElem >> n;
    if (!std::cin) return false;
    linked_list_ed_plus<int> lista;
    int num;
    for (int i = 0; i < numElem; ++i) {
        std::cin >> num;
        lista.push_back(num);
    }
    // LLamada al metodo con una lambda expression
    //lista.mostrarPropiedad([n](int valor){return valor % n == 0;});

    // LLamada al metodo con sobrecarga del operador parentesis
    lista.mostrarPropiedad(multiplo(n));

    return true;
}

int main() {

#ifdef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifdef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}

```

2. Modificación de listas enlazadas simples. Agrupar elementos iguales.

Agrupar los elementos iguales de una lista enlazada simple

En este ejercicio se trata de practicar con la estructura de datos de listas enlazadas simples `linked_list.ed`. Dada una lista de caracteres, se agruparán todos los caracteres iguales a uno dado, quedando todos ellos consecutivos a la primera aparición del carácter.

Por ejemplo dada la lista: `d r t r d t r t d`, si queremos agrupar el carácter `r`, obtendremos la lista: `d r r r r t d t t d`. En la que todos los caracteres `r` de la lista se han trasladado junto al primero.

Requisitos de implementación.

Para implementar el ejercicio se extenderá la clase `linked_list.ed` con un método que realice la operación pedida. Se podrán realizar las operaciones privadas que sean necesarias.

Se implementará un método público `print(std::ostream & o)` en la clase que muestre en el buffer `o` los elementos de la lista.

Se realizará la sobrecarga del operador de insercción (`<<`) para mostrar el contenido de la lista. En la implementación se hará uso del método `print` anterior.

No se puede reservar nueva memoria, salvo la necesaria para construir la lista inicial, ni devolver memoria al sistema, salvo la reservada para la lista inicial.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se muestra en dos líneas. En la primera se muestra el número de elementos de la lista y el carácter que se quiere agrupar. En la línea siguiente se muestran todos los valores de la lista. Los valores se añaden a la lista por la derecha.

Los valores de la lista son caracteres del alfabeto anglosajón

Salida

Para cada caso de prueba se escribirá en una línea el contenido de la lista una vez agrupados los caracteres pedidos.

Entrada de ejemplo

```
5 b
b h j b u
4 f
f f f f
6 k
t t t t t t
6 k
t t t t t k
6 k
t k k l q k
10 r
d r r t r d t r t d
```

Salida de ejemplo

```
b b h j u  
f f f f  
t t t t t t  
t t t t t k  
t k k k l q  
d r r r r t d t t d
```

Autor: Isabel Pita

2.1. Objetivos del problema

- Practicar el uso de listas enlazadas simples. Modificar la posición en la lista de algunos elementos.
- Uso de la herencia para crear nuevas clases que añadan operaciones a un TAD.

2.2. Ideas generales.

- En el problema nos piden que construyamos una lista enlazada simple con los elementos de la entrada y que después la modifiquemos agrupando los valores iguales a uno dado en la entrada.
- Utilizaremos la implementación de la lista enlazada simple `linked_list_ed` que se proporciona en el campus virtual y explicada en el problema anterior (recorrer una lista enlazada simple), e implementaremos la nueva operación en una clase que herede de esta.
- Para construir la lista inicial a partir de los datos de entrada utilizaremos la operación `push_back` que añade los elementos al final de la lista.
- La operación de modificar la lista debe encontrar en primer lugar la posición en que se deben enlazar los valores que se van a mover, en este problema se trata de la primera posición del carácter de entrada. Para agrupar los valores podemos enlazar el nodo que movemos delante de la primera aparición del valor, o detrás. De ambas formas los valores quedarán agrupados. La decisión es importante, ya que influye en la posición en que se debe dejar el puntero que nos permitirá enlazar el nodo. Debemos pensar cuál de las dos posibilidades es más sencilla de implementar.
- A continuación recorreremos el resto de la lista buscando los caracteres que debemos agrupar. Cuando se encuentra uno de estos valores debemos romper los enlaces que mantienen este nodo en este lugar de la lista, sin perder la memoria reservada para el nodo, y a continuación enlazar el nodo en su nueva posición, a través del puntero del apartado anterior. No se debe devolver memoria al sistema, ni reservar nueva memoria ya que no es necesario y son operaciones que tienen un coste, ya que se copian valores. El problema se debe resolver modificando los enlaces de la lista. Observad que para romper los enlaces que mantienen un nodo en la lista se debe situar un puntero en el nodo anterior de la lista.

2.3. Algunas cuestiones sobre implementación.

- El código de la clase `linked_list_ed` se explicó en el problema anterior *Recorrido de una lista enlazada simple*.
- Código de la clase `linked_list_ed_plus` que usaremos para implementar las nuevas operaciones de la lista enlazada.
 1. La nueva clase es también una clase genérica, por lo que utilizaremos la cabecera `template <typename T>`.
 2. La clase hereda de forma pública de la clase `linked_list_ed`. Debemos también indicar que utilizaremos el tipo `Nodo` en la nueva clase con el mismo nombre, para ello utilizamos la declaración `using`. Observad que en la clase que hereda nos referimos siempre con `this` a los atributos y métodos de la clase heredada.

```
template <typename T>
class linked_list_ed_plus : public linked_list_ed<T> {
    using Nodo = typename linked_list_ed<T>::Nodo;
public:
    ....
};
```
 3. El método que agrupa los elementos iguales a uno dado se implementa en la parte pública.
 - a) Se busca la primera aparición del carácter que nos piden. Decidimos añadir los elementos que se desplazan detrás de la primera aparición del carácter para no tener que diferenciar el caso en que el carácter pedido esté en la primera posición de la lista, por lo tanto se marca con un puntero, `pos`, la posición de la primera aparición del carácter pedido.

```

// Busca la primera aparicion del elemento
Nodo* pos = this->prim;
while (pos != nullptr && pos->elem != ch)
    pos = pos->sig;

```

Al finalizar el bucle el puntero `pos` se encuentra apuntando al primer nodo con el carácter pedido. Si el carácter no existe apuntará al valor `nullptr`.

- b) Si el carácter existe, se procede a saltar todos los caracteres iguales para que el puntero `pos` se sitúe al final del primer tramo de valores iguales al pedido. Después de desplazar los demás nodos, el puntero `pos` se situará siempre al final del primer tramo de valores consecutivos (invariante).

```

// saltamos todos los elementos iguales para
// anadir los elementos detras de la
// posicion de pos
while (pos->sig!=nullptr&&pos->sig->elem==ch)
    pos = pos->sig;

```

- c) A continuación, buscamos todas las apariciones siguientes para agruparlas. En este caso debemos marcar con un puntero, `otro`, el nodo anterior a la aparición del carácter, para poder desligar el nodo de la lista. Observad como se pregunta por el elemento del nodo siguiente al apuntado por el puntero, en lugar del elemento del propio nodo. Esto exige comprobar que existe el nodo siguiente antes de preguntar por el elemento.

```

// Busca caracteres posteriores para agruparlos
Nodo* otro = pos->sig; // salta el nodo apuntado por pos
while (otro->sig != nullptr){
    if (otro->sig->elem == ch) { // si contiene el elemento
        moverNodo(pos, otro); // desplaza el nodo
        pos = pos->sig; // para que quede detras del ultimo elemento
    }
    else otro = otro->sig;
}

```

- d) La función que traslada el nodo siguiente al nodo apuntado con un puntero `antMover` a continuación del nodo apuntado con un puntero `pos` es la siguiente

```

void moverNodo(Nodo* pos, Nodo* antMover) {
    Nodo* mover = antMover->sig;
    antMover->sig = mover->sig;
    mover->sig = pos->sig;
    pos->sig = mover;
}

```

- e) Por último debemos colocar el puntero `ult` de la lista si se ha movido el último nodo. Para ello contamos con el puntero `otro` que debe apuntar al nodo anterior al que se ha movido, y por lo tanto debe estar en la última posición de la lista si el nodo desplazado es el último. Por otra parte, podemos saber si el último nodo ha sido desplazado preguntando por el puntero `ult`. Recordemos que al desplazar los nodos, dejamos el puntero `pos` apuntando al nodo que hemos desplazado, por lo tanto si `ult` es la misma dirección de memoria que `pos` es que el nodo ha sido desplazado.

```

if (this->ult == pos) // se ha movido el ultimo elemento
    this->ult = otro;

```

4. Es conveniente implementar un método que muestre el contenido de la lista, para utilizarlo posteriormente en la sobrecarga del operador `<<`. Este método declara un puntero al primer nodo y lo va desplazando por la lista hasta completarla.

■ Código para resolver el problema:

- Debemos declarar un objeto de clase `linked_list_ed_plus` para poder llamar al método de la clase. En nuestro problema la lista es de caracteres (`char`).

```
linked_list_ed_plus<char> lista;
```

- Se leen los datos de la lista y se añaden con el método `push_back`.
- Se llama al método `agruparIgualesA` con la letra leída de consola.
- Se escribe la lista mediante la sobrecarga del operador de inserción `<<`. En la implementación de este método se llama a una función pública de la clase que muestre todos los elementos de la lista.

2.4. Errores frecuentes.

No se conocen.

2.5. Implementación.

La implementación de la clase `linked_list_ed` se proporciona en el campus virtual.

```
#include "linked_list_ed.h"

template <typename T>
class linked_list_ed_plus : public linked_list_ed<T> {
    using Nodo = typename linked_list_ed<T>::Nodo;
public:

    void print(std::ostream & o) const {
        Nodo* aux = this->prim;
        if (aux != nullptr) {
            o << aux->elem;
            aux = aux->sig;
        }
        while (aux != nullptr) {
            o << ' ' << aux->elem;
            aux = aux->sig;
        }
        o << '\n';
    }

    void agruparIgualesA(T const& ch){
        if (this->prim == nullptr) return; // lista vacia
        // Busca la primera aparicion del elemento
        Nodo* pos = this->prim;
        while (pos != nullptr && pos->elem != ch)
            pos = pos->sig;
        if (pos != nullptr) { // encontro primer elemento
            // saltamos todos los elementos iguales para anadir los elementos
            // detras de la posicion de pos
            while (pos->sig != nullptr && pos->sig->elem == ch)
                pos = pos->sig;

            // Busca elementos posteriores para agruparlos
            Nodo* otro = pos; // pos situado al final del primero bloque de elementos
            while (otro->sig != nullptr){
                if (otro->sig->elem == ch) {
                    moverNodo(pos, otro);
                    pos = pos->sig; // para que quede detras del ultimo elemento
                }
                else otro = otro->sig;
            }
            if (this->ult == pos) // se ha movido el ultimo elemento
                this->ult = otro;
        }
    }

private:
    void moverNodo(Nodo* pos, Nodo* antMover) {
        Nodo* mover = antMover->sig;
        antMover->sig = mover->sig;
        mover->sig = pos->sig;
        pos->sig = mover;
    }
};

template <typename T>
inline std::ostream& operator<< (std::ostream& o, linked_list_ed_plus<T> const& l) {
    l.print(o);
    return o;
}
```

```
}
```

```
bool resuelveCaso() {  
    int numElem; char letra;  
    std::cin >> numElem;  
    if (!std::cin) return false;  
    std::cin >> letra;  
    linked_list_ed_plus<char> lista;  
    char c;  
    for (int i = 0; i < numElem; ++i) {  
        std::cin >> c;  
        lista.push_back(c);  
    }  
    lista.agruparIgualesA(letra);  
    std::cout << lista;  
    return true;  
}
```

```
int main() {  
  
#ifndef DOMJUDGE  
    std::ifstream in("datos.txt");  
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt  
#endif  
  
    while (resuelveCaso()) {} //Resolvemos todos los casos  
  
#ifndef DOMJUDGE // para dejar todo como estaba al principio  
    std::cin.rdbuf(cinbuf);  
    system("PAUSE");  
#endif  
  
    return 0;  
}
```

3. Listas enlazadas dobles. Agrupar elementos.

Agrupar los elementos iguales de una lista enlazada doble

En este ejercicio se trata de practicar con la estructura de datos de listas enlazadas dobles `double_linked_list.ed`. Dada una lista cuyos valores son de tipo `Horas`, se agruparán aquellos nodos que tengan la misma hora, pudiendo diferir en el número de minutos y segundos, quedando todos ellos consecutivos a la primera aparición de la hora indicada. La posición relativa de las horas una vez agrupadas será la misma que tenían en la lista original.

Requisitos de implementación.

Para implementar el ejercicio se declarará una clase que herede de la clase `double_linked_list.ed` con un método que realice la operación pedida. Se podrán realizar las operaciones privadas que sean necesarias. La clase `double_linked_list.ed` no se debe modificar.

Se implementará un método público `print(std::ostream & o)` en la clase que hereda que muestre en el buffer o los elementos de la lista.

Se realizará la sobrecarga del operador de insercción (`<<`) para mostrar el contenido de la lista. En la implementación se hará uso del método `print` anterior.

No se puede reservar nueva memoria, salvo la necesaria para construir la lista inicial, ni devolver memoria al sistema, salvo la reservada para la lista inicial.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se muestra en dos líneas. En la primera se muestra el número de elementos de la lista y la hora que se quiere agrupar. En la línea siguiente se muestran todos los valores de la lista. Los valores se añaden a la lista por la derecha.

Los valores de la lista están dados en el formato HH:MM:SS, siendo HH el número de horas, MM el número de minutos y SS el número de segundos. Se garantiza que las horas son válidas.

Salida

Para cada caso de prueba se escribirá en una línea el contenido de la lista una vez agrupada la hora pedida.

Entrada de ejemplo

```
5 15
15:30:45 02:40:35 12:01:00 15:02:02 10:30:25
4 20
10:50:00 20:40:01 20:55:40 20:32:12
6 10
00:00:00 00:00:00 00:00:00 00:00:00 00:00:00 00:00:00
6 10
22:10:10 10:59:59 10:58:58 00:40:40 00:20:20 10:57:57
10 22
12:00:00 22:00:30 22:20:20 14:10:10 02:40:40 15:00:00 16:00:00 22:05:05 20:41:41 22:22:22
```

Salida de ejemplo

```
15:30:45 15:02:02 02:40:35 12:01:00 10:30:25
10:50:00 20:40:01 20:55:40 20:32:12
00:00:00 00:00:00 00:00:00 00:00:00 00:00:00 00:00:00
22:10:10 10:59:59 10:58:58 10:57:57 00:40:40 00:20:20
12:00:00 22:00:30 22:20:20 22:05:05 22:22:22 14:10:10 02:40:40 15:00:00 16:00:00 20:41:41
```

3.1. Objetivos del problema

- Conocer una implementación de lista enlazada doble.
- Practicar el uso de listas enlazadas dobles. Modificar la posición en la lista de algunos elementos.
- Uso de la herencia para crear nuevas clases que añadan operaciones a un TAD.

3.2. Ideas generales.

- En el problema nos piden que construyamos una lista enlazada doble con los elementos de la entrada y que después la modifiquemos agrupando los valores iguales a uno dado en la entrada.
- Utilizaremos la implementación de la lista enlazada doble `double_linked_list_ed` que se proporciona en el campus virtual. Se trata de una lista enlazada doble, circular y con nodo *fantasma*. El nodo *fantasma* marca el comienzo de la lista. Es un nodo sin valor, que no guarda elementos de la lista. Se utiliza para facilitar la implementación de las operaciones. Siempre existe, incluso en la lista vacía.

El tipo de datos nos ofrece:

1. la estructura `Nodo` con un campo genérico para guardar el valor del nodo y dos punteros: uno al siguiente `Nodo`, y otro al nodo anterior.
 2. un puntero a la lista: `fantasma` nos permite acceder al nodo *fantasma*. A partir del nodo *fantasma* podemos acceder a cualquier otro nodo de la lista.
 3. Operaciones para construir una lista vacía, destruir la lista, copiar y asignar una lista, añadir elementos por el principio (izquierda) de la lista y por el final (derecha) de la lista, eliminar el primer elemento y el último de la lista, acceder al primer y al último elemento de la lista y comprobar si la lista está vacía.
- Para construir la lista inicial a partir de los datos de entrada utilizaremos la operación `push_back` que añade los elementos al final de la lista, de esta forma los elementos quedan en el mismo orden que la entrada de datos.
 - La operación de modificar la lista se implementa en una clase que hereda de la clase `double_linked_list_ed`. La clase `double_linked_list_ed` no debe modificarse.
 - En primer lugar se debe encontrar la posición en que se deben enlazar los valores que se van a mover y guardarla en un puntero `Nodo* pos`. El puntero guardará la dirección de memoria del nodo que tenga el primer valor que queremos agrupar. Al ser la lista doblemente enlazada se pueden añadir elementos delante y detrás del nodo mediante el puntero al nodo. En el enunciado nos indican que se guarde el orden de la entrada, por lo que añadiremos los nodos por detrás de los ya agrupados. Observad que si se añaden por delante, se obtienen en orden inverso al orden de entrada.
 - A continuación recorreremos el resto de la lista buscando los valores que debemos agrupar. Cuando se encuentra uno de estos valores debemos romper los enlaces que mantienen este nodo en este lugar de la lista, sin perder la memoria reservada para el nodo, y a continuación enlazar el nodo en su nueva posición, a través del puntero del apartado anterior. No se debe devolver memoria al sistema, ni reservar nueva memoria ya que no es necesario y son operaciones que tienen un coste, al copiarse los valores. El problema se debe resolver modificando los enlaces de la lista. Observad que los enlaces de una lista doblemente enlazada se pueden romper mediante un puntero que apunte al propio nodo, ya que se tiene acceso tanto al nodo anterior como al siguiente.

3.3. Algunas cuestiones sobre implementación.

■ Código de la clase `double_linked_list_ed`.

1. La clase `double_linked_list_ed` es una clase genérica, es decir los tipos y métodos definidos en la clase dependen de uno o más parámetros que se instancian en el momento en que se crea un objeto de la clase. Para indicar que una clase es genérica se utiliza la palabra reservada `template` como se muestra en el código siguiente. Entre angulos se indica el identificador del tipo genérico a utilizar en la implementación de la clase. El identificador puede ser cualquier identificador válido, normalmente se utiliza `T`. Se permite utilizar la palabra reservada `class` en lugar de `typename` para mantener la compatibilidad de las versiones anteriores de C++.

```
template <typename T>
class double_linked_list_ed {
    .....
};
```

2. Para crear un objeto de esta clase indicaremos el tipo de los elementos de la lista.

```
double_linked_list_ed<Horas> lista;
```

3. Como utilizaremos herencia para implementar las nuevas operaciones, los atributos de la clase los declaramos protegidos: `protected`. De esta forma la clase que hereda tiene acceso a los atributos. (En la clase que hereda los atributos se deben referenciar siempre con `this`).
4. El `struct` `Nodo` consta de tres campos, el elemento que será del tipo genérico y dos punteros a una estructura de tipo `Nodo`. Uno de ellos guardará la dirección de memoria del nodo anterior en la lista y el otro guardará la dirección de memoria del nodo siguiente de la lista. Ofrece dos constructores, uno sin parámetros que inicializa los punteros al valor `nullptr`, y otro constructor con tres parámetros que serán el valor del elemento y el valor de los punteros.
5. La clase `double_linked_list_ed` tiene un sólo atributo que es un puntero al primer elemento de la lista, el nodo fantasma.
6. Las operaciones públicas de la lista son:
 - a) Cómo se utiliza memoria dinámica debemos implementar los siguientes 4 métodos: constructor, destructor, constructor por copia y operador de asignación.
 - b) El constructor crea una lista vacía. Se debe crear una lista doble, circular y con nodo fantasma. Por lo tanto debe tener un nodo, sin valor en el elemento y cuyos punteros apunten al mismo nodo para obtener la circularidad. Observad como se reserva la memoria del nodo en la lista de inicialización y posteriormente se da valor a los punteros `ant` y `sig` en el cuerpo de la función.

```
double_linked_list_ed() : fantasma(new Nodo()) {
    fantasma->sig = fantasma->ant = fantasma; // circular
}
```

- c) El destructor de la lista. El destructor es necesario porque debemos devolver al sistema toda la memoria utilizada por los nodos de la lista. Esto se hace mediante la función privada `libera` implementada en la parte privada de la clase. El destructor tiene el mismo nombre que la clase, precedido del símbolo `~`. No se indica valor de retorno y no tiene parámetros. Una clase solo puede tener un destructor.

```
~double_linked_list_ed() {
    libera();
}
```

- d) Un constructor por copia, que recibe un objeto del mismo tipo y realiza una *copia profunda* de la lista, es decir una copia de todos los nodos que no comparta memoria con la lista original. Utiliza la función privada `copia` que reserva memoria para todos los nodos de la lista a copiar y copia los valores.

```
double_linked_list_ed(double_linked_list_ed<T> const& other) {
    copia(other);
}
```

- e) El operador de asignación que nos permite realizar una *copia profunda* de la lista. Para ello se libera la memoria de la lista en la que se va a realizar la copia y a continuación mediante la función privada `copia` se reserva memoria para todos los nodos de la lista a copiar y se copian los valores. Observad que se devuelve el contenido del objeto, para que se puedan anidar varias instrucciones de asignación.

```
double_linked_list_ed<T> & operator=(double_linked_list_ed<T> const& that) {
    if (this != &that) {
        libera();
        copia(that);
    }
    return *this;
}
```

- f) Añadir un elemento al principio o al final de la lista: `push_front`, `push_back`. Ambas operaciones se realizan por medio de la operación privada `inserta_elem` que recibe el valor del elemento y dos punteros indicando el nodo anterior y siguiente al que se quiere añadir.

```
void push_front(T const& e) {
    inserta_elem(e, fantasma, fantasma->sig);
}
```

```
void push_back(T const& e) {
    inserta_elem(e, fantasma->ant, fantasma);
}
```

- g) Eliminar el primer o último nodo de la lista: `pop_front` y `pop_back`. Esta operación es parcial, lanza una excepción cuando la lista es vacía, ya que no puede eliminar el elemento. Ambas operaciones se realizan por medio de la operación privada `borra_elem` que recibe un puntero al nodo que se quiere borrar. Observad que este TAD tiene la operación de borrar el último nodo. La operación tiene coste constante al estar la lista doblemente enlazada, por ello se ofrece como operación del TAD.

```
void pop_front() {
    if (empty())
        throw std::domain_error("eliminando el primero de una lista vacia");
    borra_elem(fantasma->sig);
}
```

```
void pop_back() {
    if (empty())
        throw std::domain_error("eliminando el ultimo de una lista vacia");
    borra_elem(fantasma->ant);
}
```

- h) Se ofrecen operaciones que devuelven el primer y último elemento de la lista. Son operaciones parciales, ya que no pueden devolver valor sobre la lista vacía. Se observa que devuelve el valor `const&` esto es debido a que el tipo es genérico, por lo que puede ser costoso realizar una copia. Por otro lado si se devuelve una referencia al objeto, este podría modificarse desde fuera de la clase. Al ser un valor `const&` se evita la copia y se prohíbe que se modifique el contenido del objeto.

```
T const& front() const {
```

```

    if (empty())
        throw std::domain_error("la lista vacia no tiene primero");
    return fantasma->sig->elem;
}

T const& back() const {
    if (empty())
        throw std::domain_error("la lista vacia no tiene ultimo");
    return fantasma->ant->elem;
}

```

- i) Comprobar si una lista es vacía: `empty`. La lista es circular y con nodo fantasma, por lo tanto es vacía cuando el puntero siguiente coincide con el primer nodo.

```

bool empty() const {
    return fantasma->sig == fantasma;
}

```

7. Se definen cuatro operaciones protegidas en las listas dobles:

- a) **libera**. Libera la memoria de todos los nodos de la lista. En este caso se debe liberar la memoria de todos los nodos, incluido el nodo fantasma. Por ello comenzamos rompiendo la circularidad de la lista, de forma que el puntero anterior del nodo fantasma tenga la dirección `nullptr` y el puntero siguiente del último nodo de la lista (que tiene la dirección del nodo fantasma) cambie su valor a `nullptr`. Una vez rota la circularidad, liberamos la memoria igual que se hace en una lista enlazada simple.

```

void libera() {
    // primero rompemos la circularidad
    fantasma->ant->sig = nullptr;
    fantasma->ant = nullptr;
    while (fantasma != nullptr) {
        Nodo * a_borrar = fantasma;
        fantasma = fantasma->sig;
        delete a_borrar;
    }
}

```

- b) **copia**. Crea una lista a partir de otra duplicando los nodos. Comienza creando una lista vacía con el nodo fantasma. A continuación recorre la lista `other` añadiendo los valores a la lista en que se copia por medio de la operación `push_back`. Observad que lo copiado es únicamente el valor, por lo que el nodo se duplica con la operación `push_back`.

```

void copia(double_linked_list_ed<T> const& other) {
    fantasma = new Nodo();
    fantasma->sig = fantasma;
    fantasma->ant = fantasma;

    // act recorre la cola que estamos copiando
    Nodo * act = other.fantasma->sig;
    while (act != other.fantasma) {
        push_back(act->elem);
        act = act->sig;
    }
}

```

- c) **inserta_elem**. Añade un nodo entre los punteros `anterior` y `siguiente` que recibe como parámetros. Para ello se reserva la memoria utilizando el constructor con tres parámetros y a continuación se enlaza la memoria reservada en la lista. Observad que al estar utilizando un nodo fantasma, el código es el mismo para el primer elemento de la lista, para el último o para uno intermedio.

```

Nodo * inserta_elem(T const& e, Nodo * anterior, Nodo * siguiente) {
    Nodo * nuevo = new Nodo(e, anterior, siguiente);
}

```

```

    anterior->sig = nuevo;
    siguiente->ant = nuevo;
    return nuevo;
}

```

- d) `borra_elem`. Elimina el puntero apuntado por el valor del parámetro. Como precondition de la función se pide que el parámetro no tenga el valor `nullptr`. La precondition se comprueba mediante la instrucción `assert`. La función modifica los punteros de la lista para desligar al nodo a borrar y a continuación devuelve la memoria al sistema.

```

void borra_elem(Nodo * n) {
    assert(n != nullptr);
    n->ant->sig = n->sig;
    n->sig->ant = n->ant;
    delete n;
}

```

- Código de la clase `double_linked_list_ed_plus` que usaremos para implementar las nuevas operaciones de la lista enlazada.

1. La nueva clase es también una clase genérica, por lo que utilizaremos la cabecera `template <typename T>`.
2. La clase hereda de forma pública de la clase `double_linked_list_ed`. Debemos también indicar que utilizaremos el tipo `Nodo` en la nueva clase con el mismo nombre, para ello utilizamos la declaración `using`. Observad que en la clase que hereda nos referimos siempre con `this` a los atributos y métodos de la clase heredada.

```

template <typename T>
class double_linked_list_ed_plus : public double_linked_list_ed<T> {
    using Nodo = typename double_linked_list_ed<T>::Nodo;
public:
    ....
};

```

3. El método que agrupa los elementos iguales a uno dado se implementa en la parte pública.
 - a) Se busca la primera aparición de la hora que nos dan. Los elementos que se desplazan se deben añadir detrás de la primera aparición de la hora para mantener el orden en que figuran en la lista original. Observad que la propiedad por la que agrupamos los elementos es genérica (ver problema sobre recorrer una lista enlazada simple).

```

// Busca la primera aparicion del elemento
Nodo* pos = this->fantasma->sig;
while (pos != this->fantasma && !(pred(pos->elem)))
    pos = pos->sig;

```


Al finalizar el bucle el puntero **pos** guarda la dirección de memoria del primer nodo con la hora pedida. Si la hora no existe tendrá el valor del puntero al nodo fantasma.

- b) A continuación, buscamos todas las apariciones siguientes para agruparlas. Marcamos con un puntero, **otro**, el nodo que debemos desplazar. Observad que en este tipo de lista podemos guardar la dirección del nodo que desplazamos.

```
// Busca elementos posteriores para agruparlos
Nodo* otro = pos->sig;
while (otro != this->fantasma){
    if (pred(otro->elem)) {
        Nodo* mover = otro;
        otro = otro->sig;
        moverNodo(pos, mover);
        pos = pos->sig;
    }
    else otro = otro->sig;
}
```

- c) La función que traslada el nodo apuntado con un puntero **mover** a continuación del nodo apuntado con un puntero **pos** es la siguiente. Primer desliga el nodo a mover dejando la lista sin este nodo y a continuación lo enlaza en la posición indicada.

```
void moverNodo(Nodo* pos, Nodo* mover) {
    // desligamos el nodo a mover
    mover->sig->ant = mover->ant;
    mover->ant->sig = mover->sig;
    //enlazamos el nodo en su posicion
    mover->sig = pos->sig;
```

```

        mover->ant = pos;
        pos->sig->ant = mover;
        pos->sig = mover;
    }

```

4. Es conveniente implementar un método que muestre el contenido de la lista, para utilizarlo posteriormente en la sobrecarga del operador <<. Este método declara un puntero al primer nodo y lo va desplazando por la lista hasta completarla.

■ Código para resolver el problema:

- Debemos declarar un objeto de clase `double_linked_list_ed_plus` para poder llamar al método de la clase. En nuestro problema la lista es de horas (`Horas`).

```
double_linked_list_ed_plus<Horas> lista;
```

- Se leen los datos de la lista y se añaden con el método `push_back`.
- Se llama al método `agrupar` con la propiedad que cumplen los elementos a agrupar (ver problema sobre recorrer una lista enlazada simple)

```
lista.agrupar([h](horas valor){return valor.get_hora() == h;});
```

- Se escribe la lista mediante la sobrecarga del operador de inserción <<. En la implementación de este método se llama a una función pública de la clase que muestre todos los elementos de la lista.

3.4. Errores frecuentes.

No se conocen.

3.5. Implementación.

La implementación de la clase `double_linked_list_ed` se proporciona en el campus virtual.

```

#include "double_linked_list_ed.h"
#include "Horas"

template <typename T>
class double_linked_list_ed_plus : public double_linked_list_ed<T> {
    using Nodo = typename double_linked_list_ed<T>::Nodo;
public:

    void print(std::ostream & o) const {
        Nodo* aux = this->fantasma->sig;
        if (aux != this->fantasma) {
            o << aux->elem;
            aux = aux->sig;
        }
        while (aux != this->fantasma) {
            o << ' ' << aux->elem;
            aux = aux->sig;
        }
        o << '\n';
    }

    template <typename P>
    void agrupar(P pred) {
        if (this->fantasma == this->fantasma->sig) return; // lista vacia
        // Busca la primera aparicion del elemento
        Nodo* pos = this->fantasma->sig;
        while (pos != this->fantasma && !(pred(pos->elem)))
            pos = pos->sig;
    }
}

```

```

        // Pos situado en la posicion del primer elemento
        if (pos != this->fantasma) { // encontro primer elemento
            // Busca elementos posteriores para agruparlos
            Nodo* otro = pos->sig;
            while (otro != this->fantasma){
                if (pred(otro->elem)) {
                    Nodo* mover = otro;
                    otro = otro->sig;
                    moverNodo(pos, mover);
                    pos = pos->sig;
                }
                else otro = otro->sig;
            }
        }
    }

}

private:
    // Se desplaza el nodo detras del apuntado por pos
    void moverNodo(Nodo* pos, Nodo* mover) {
        // desligamos el nodo a mover
        mover->sig->ant = mover->ant;
        mover->ant->sig = mover->sig;
        //enlazamos el nodo en su posicion
        mover->sig = pos->sig;
        mover->ant = pos;
        pos->sig->ant = mover;
        pos->sig = mover;
    }
};

template <typename T>
inline std::ostream& operator<< (std::ostream& o, double_linked_list_ed_plus<T> const& l) {
    l.print(o);
    return o;
}

class comparar {
    int h;
public:
    comparar(int hora) : h(hora){}
    bool operator() (horas valor) {
        return valor.get_hora() == h;
    }
};

bool resuelveCaso() {
    int numElem; int h;
    std::cin >> numElem;
    if (!std::cin) return false;
    std::cin >> h;
    double_linked_list_ed_plus<horas> lista;
    horas c;
    for (int i = 0; i < numElem; ++i) {
        std::cin >> c;
        lista.push_back(c);
    }
    // Usando una lambda expression
    //lista.agrupar([h](horas valor){return valor.get_hora() == h;});
    // Usando un objeto funcion
    lista.agrupar(comparar(h));
}

```

```

        std::cout << lista;
        return true;
    }

    int main() {

#ifdef DOMJUDGE
        std::ifstream in("datos.txt");
        auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

        while (resuelveCaso()) {} //Resolvemos todos los casos

#ifdef DOMJUDGE // para dejar todo como estaba al principio
        std::cin.rdbuf(cinbuf);
        system("PAUSE");
#endif

        return 0;
    }

```

4. Pilas, invertir una secuencia.

Invertir una secuencia de valores

Una de las aplicaciones más comunes de las pilas consiste en invertir una secuencia de elementos. Dada una serie de valores de entrada, debes mostrarlos invertidos.

Entrada

La entrada está formada por una serie de casos de prueba, cada uno en una línea. Cada caso consiste en una secuencia de números terminada con el valor cero que no debe procesarse.

Salida

Para cada caso de prueba se escribirá la secuencia invertida de los valores de la entrada.

Entrada de ejemplo

```
1 2 3 4 5 6 7 8 9 0
0
1 0
321 543 0
```

Salida de ejemplo

```
9 8 7 6 5 4 3 2 1
1
543 321
```

Autor: Isabel Pita

4.1. Objetivos del problema

- Conocer el TAD `stack` de la STL.
- Practicar el uso de una pila para invertir elementos.

4.2. Ideas generales.

- Un TAD adecuado para invertir una secuencia de elementos es el tipo `stack` (pila).
- Para resolver el problema utilizaremos el tipo `stack` que proporciona la librería STL de C++.
- El tipo ofrece 5 operaciones básicas (ver página www.cplusplus.com/stack):
 - Constructor sin parámetros.
 - `push`, añade un elemento en la cima de la pila.
 - `pop`, elimina el elemento de la cima de la pila.
 - `top` devuelve una referencia al elemento de la cima de la pila.
 - `empty`, indica si la pila es vacía.

todas ellas con coste constante.

- La pila tiene un comportamiento LIFO (last in first out), es decir, lo último que se añade a la pila es lo primero que se obtiene. Si añadimos los elementos de la entrada en una pila, queda en la cima de la pila el último elemento añadido. Si ahora vamos obteniendo los elementos de la pila estos se muestran en orden inverso al que entraron.

4.3. Algunas cuestiones sobre implementación.

- El TAD `stack` se puede implementar utilizando muchas estructuras. Las dos más sencillas son el array dinámico y la lista enlazada simple con puntero al comienzo de la lista.
- Si se emplea un array dinámico para implementar una pila (ver la implementación que se proporciona en el campus virtual) la cima se debe situar en la parte derecha del array, ya que es en esta parte en la que se pueden añadir y eliminar los elementos del array con coste constante.
- Si se emplea una lista enlazada simple con puntero al comienzo de la lista, la cima se debe situar al comienzo de la lista, ya que es donde se sitúa el puntero que nos permite añadir y eliminar elementos con coste constante.
- Para resolver problemas en los que se necesite una pila, utilizaremos el TAD `stack` de la STL que se encuentra en el espacio de nombres `std`. Para ello se debe incluir la librería.

```
#include <stack>
```

- El problema es muy sencillo, por lo que lo resolveremos en la propia función `resuelveCaso`, leyendo los datos en la pila y luego vaciándola. No se utilizarán estructuras auxiliares para guardar los datos de entrada dado que el objetivo del problema de practicar las operaciones del TAD pila se consigue sin ello.

4.4. Coste de la implementación realizada.

- El programa consta de dos bucles, en el primero de ellos se crea la pila a partir de los datos de entrada. El bucle da n vueltas siendo n el número de valores de la entrada, como el coste de cada vuelta es constante porque la operación `push` sobre las pilas tiene coste constante y la operación de lectura de los datos también, el bucle tiene un coste $\mathcal{O}(n)$, siendo n el número de valores de entrada.
- El segundo bucle de n vueltas, siendo n el número de elementos de la pila, que coincide con el número de elementos de la entrada. Las operaciones `empty`, `top` y `pop` sobre las pilas tienen coste constante, por lo tanto la condición del bucle y el cuerpo del bucle tiene coste constante. El coste del bucle es $\mathcal{O}(n)$ siendo n el número de valores de la entrada.

- Como los dos bucles son secuenciales y las operaciones que se realizan fuera de los dos bucles son todas ellas de coste constante el coste total del programa está en $\mathcal{O}(n)$, siendo n el número de valores de la entrada.

4.5. Implementación.

La implementación de la clase `double_linked_list_ed` se proporciona en el campus virtual.

```
#include <stack>

bool resuelveCaso() {
    int num;
    std::cin >> num;
    if (!std::cin) return false;
    std::stack<int> pila;
    while (num != 0) {
        pila.push(num);
        std::cin >> num;
    }
    if (!pila.empty()){
        std::cout << pila.top();
        pila.pop();
        while (!pila.empty()) {
            std::cout << ' ' << pila.top();
            pila.pop();
        }
    }
    std::cout << '\n';
    return true;
}

int main() {

#ifdef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifdef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

5. Pilas, guardar información relevante.

Épocas de recesión

El PIB (Producto Interior Bruto) es un indicador de la riqueza de un país. Normalmente en tiempo de bonanza el PIB sube, mientras que en épocas de recesión baja. Dados una serie de valores que representan el PIB de un cierto lugar a lo largo de los años, nos piden encontrar la época más larga de recesión que ha sufrido. Entendemos que una recesión finaliza cuando se supera el PIB del comienzo de la recesión.

Entrada

La entrada está formada por una serie de casos de prueba, cada uno en una línea. Cada caso consiste en una secuencia de números terminada con el valor cero que no debe procesarse.

El número de valores de una secuencia es mayor o igual que cero y menor que 100.000. Los valores de la secuencia son números enteros positivos.

Salida

Para cada caso de prueba se escribirá la longitud de la época de recesión más larga, seguida de su punto de comienzo. Para obtener el punto de comienzo se considera el primer valor de entrada como el cero.

Entrada de ejemplo

```
10
2 1 4 3 2 3 5 4 3 2
5
7 6 5 4 3
5
3 4 5 6 7
9
2 1 4 3 2 3 2 5 4
```

Salida de ejemplo

```
4 6
5 0
1 4
5 2
```

Autor: Isabel Pita

5.1. Objetivos del problema

- Conocer el TAD `stack` de la STL.
- Practicar el uso de una pila para almacenar elementos que se deben consultar en orden inverso al de entrada.

5.2. Ideas generales.

- En la asignatura de Fundamentos de Algoritmia (FAL) se han visto problemas semejantes a este, en los que es necesario guardar cierta información de la parte ya recorrida del vector para decidir si el elemento que estamos considerando cumple una cierta propiedad o no la cumple.
- Lo primero que debemos estudiar es dónde se encuentra la información necesaria para tomar la decisión sobre un determinado elemento. Si la información se encuentra a la izquierda del elemento, el vector se recorre de izquierda a derecha y si se encuentra a la derecha del elemento se recorre de derecha a izquierda. En este problema, para un determinado elemento debemos saber dónde termina la posible recesión que comience en ese punto, es decir, cuál es el primer elemento de la parte derecha que es mayor que el que estamos considerando. Con ello podemos calcular el tiempo que dura la recesión como la diferencia entre las posiciones de los dos valores. La información, por lo tanto está en la parte derecha, por lo que debemos recorrer el vector de derecha a izquierda.
- El problema en este ejercicio es que no es suficiente con guardar un valor, como ocurriría si necesitásemos el valor máximo a la derecha. En ese caso es suficiente con guardar el valor máximo encontrado en una variable y modificarla cada vez que se encuentra un nuevo máximo.
- Pero, en este ejercicio debemos encontrar para cada elemento el primer elemento a su derecha que es mayor que él, y esto depende del valor del elemento que estemos considerando. Por lo tanto debemos guardar todos aquellos elementos de la parte ya recorrida (la parte derecha) que puedan ser mayores que algún elemento de los que vamos recorriendo.
- No se pueden almacenar todos los elementos ya recorridos, porque eso supondría que para cada elemento hay que recorrer todos los elementos de la derecha para encontrar el primero que es mayor y daría lugar a un coste cuadrático en el número de elementos de la solución.
- Debemos estudiar bien los elementos a guardar.
- Supongamos que los valores de entrada son a_1, \dots, a_n , el elemento que estamos procesando es a_i , y que tenemos guardados en un TAD los elementos $a_k < a_i, < \dots$. Si $a_i < a_k$ sabemos que el valor en la posición k es el final de la recesión que empezó en a_i . Si $a_i \geq a_k$ el valor a_k forma parte todavía del periodo de recesión y deberíamos mirar el elemento anterior que tenemos guardado, a_l como posible final de la recesión. Además el elemento a_k es imposible que sea el final de la recesión para ningún valor a la izquierda de a_i (a_1, \dots, a_i) porque al ser $a_i > a_k$ el final de la recesión de cualquier valor a la izquierda será a_i , nunca a_k . Por lo tanto ya no necesitamos guardar el elemento a_k y lo deseamos. Se repite el proceso con el valor a_l comprobando si es mayor o menor que el a_i y por lo tanto si es el final de la recesión o debemos seguir comprobando los valores guardados.
- Observamos que el elemento a_i puede ser el final de una recesión que empezó anteriormente (a_1, \dots, a_{i-1}) y por lo tanto debe ser guardado. Por otro lado, cuando vamos a guardar un valor podemos desechar cualquier valor menor que él que hayamos guardado previamente, ya que el nuevo valor guardado será el fin de las recesiones que hayan comenzado antes. ¡Observad que en el TAD los elementos guardados se encuentran ordenados en orden creciente y que el último elemento añadido es siempre el elemento anterior al que estamos tratando!
- Por último debemos decidir el TAD en el que guardaremos los valores que pueden ser el final de una recesión. Se observa que el primer valor por el que se debe preguntar es por el último que se ha añadido como candidato. Por lo tanto el TAD utilizado será una pila.

4	5	6	7	8	9	10	11	12	13	14	15	
		2	4	6	3	2	8	4	20	5	7	
			↑	↑			↑		↑			
			i	k			l		m			

6
8
20

5.3. Algunas cuestiones sobre implementación.

- Para poder calcular el número de valores que forman una recesión debemos conocer el índice en que comienza y el índice en que acaba. Por lo tanto guardaremos el índice de los valores que pueden ser final de una recesión en la pila de valores.

5.4. Coste de la implementación realizada.

- El coste de la solución implementada es lineal en el número de valores de entrada.
- En este problema el estudio habitual del coste en el caso peor, consistente en multiplicar el número de vueltas que da el bucle por el coste de cada vuelta, da una cota superior demasiado elevada. El bucle da n vueltas siendo n el número de valores de entrada y en el cuerpo del bucle principal tenemos un bucle que en el caso peor podría dar n vueltas. Por lo tanto, el estudio del coste en el caso peor nos da un coste cuadrático en el caso peor, lo cual como se ha indicado antes es una cota superior muy poco ajustada.
- Podemos realizar un estudio mas fino en el cual sumemos el coste de cada una de las n vueltas del bucle principal. No podemos conocer el coste de cada vuelta del bucle, porque éste depende de los valores que haya en cada momento en la pila que estamos utilizando. Pero si conocemos el número de veces que podemos desapilar de la pila, ya que una vez que un elemento se desapila ya nunca vuelve a apilarse. Por lo tanto como mucho se pueden realizar n instrucciones `pila.pop()` en todas las vueltas del bucle principal `for`. Por lo tanto, la suma del coste de todas las vueltas del bucle `for` es del orden $\mathcal{O}(n)$ siendo n el número de valores de entrada, ya que el cuerpo del bucle `while` se ejecuta como máximo n veces y el resto de instrucciones se ejecutan una vez por cada vuelta del bucle `for`, por lo tanto también n veces. Las operaciones `empty`, `top`, y `push` de las pilas son de coste constante por lo que no añaden ningún coste adicional.

5.5. Implementación.

```
#include <stack>

std::pair<int, int> resolver(std::vector<int> const& v) {
    std::stack<int> pila; // indice del elemento que puede terminar la recesion
    int maxi = 0; int pos = (int)v.size(); // posible solucion
    for (int i = (int)v.size()-1; i >= 0; --i) {
        // Elimina de la pila todos los elementos menores que el actual
        while (!pila.empty() && v[i] >= v[pila.top()]){
            pila.pop();
        }
        // Comprueba si el intervalo es mayor
        if (pila.empty() && maxi < v.size()-i){
            maxi = (int)v.size()-i; pos = i;
        }
        else if (!pila.empty() && maxi < pila.top()-i){
            maxi = pila.top()-i; pos = i;
        }
        pila.push(i);
    }
    return {maxi, pos};
}
```

```

bool resuelveCaso() {
    int numElem;
    std::cin >> numElem;
    if (!std::cin) return false;
    std::vector<int> v(numElem);
    for (int& n : v) std::cin >> n;
    std::pair<int, int> sol = resolver(v);
    std::cout << sol.first << ' ' << sol.second << '\n';
    return true;
}

int main() {

#ifdef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifdef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}

```

6. Iteradores. My search

Encontrar un patrón en una secuencia

Se pide implementar una función semejante a la función `search` de la librería `algorithm`.

```
template < class T >
T Mysearch(T const & ini1, T const & fin1, T const & ini2, T const & fin2);
```

Esta función busca en el rango `[ini1..fin1)` la primera aparición de la secuencia definida por `[ini2..fin2)` y devuelve un iterador al primer elemento de dicha aparición o `fin1` si la secuencia no se encuentra. La función debe ser genérica para cualquier tipo lineal que tenga definido un iterador.

La función recibe:

- `ini1`. Un iterador al principio de la secuencia en la que debemos buscar.
- `fin1`. Un iterador al final de la secuencia en la que se debe buscar.
- `ini2`. Un iterador al comienzo de la secuencia que queremos buscar.
- `fin2`. Un iterador al final de la secuencia que queremos buscar.

La función devuelve un iterador al comienzo de la primera aparición de la secuencia a buscar en la secuencia en que estamos buscando o `fin1` si no se encuentra la secuencia buscada.

Un ejemplo de llamada a esta función es:

```
std::list <int> lista1; std::list <int > lista2;
auto it = Mysearch(lista1.begin(), lista1.end(), lista2.begin(), lista2.end());
```

En este problema la función se utilizará para eliminar todas las apariciones de la segunda secuencia de entrada en la primera.

Entrada

La entrada está formada por una serie de casos. Cada caso comienza con dos valores $N1, N2$ que indican el número de valores de la primera y segunda secuencia que nos dan a continuación. En la línea siguiente nos dan los $N1$ valores de la secuencia en que debemos buscar. En la línea siguiente nos dan los $N2$ valores de la secuencia que buscamos. La entrada termina con dos ceros.

Se garantiza que $0 < N1$ y $0 < N2 < 10$. Los valores de las secuencias pueden almacenarse en una variable de tipo `int`.

Salida

Para cada caso se escribirá una línea con los valores que quedan en la primera secuencia después de eliminar todas las apariciones de la segunda secuencia.

Entrada de ejemplo

```
10 2
3 5 7 9 3 5 4 6 3 5
3 5
6 3
1 2 3 1 2 3
1 2 3
6 1
1 2 3 1 2 3
3
5 3
1 2 3 4 5
4 5 6
0 0
```

Salida de ejemplo

7 9 4 6

1 2 1 2

1 2 3 4 5

Autor: Isabel Pita

6.1. Objetivos del problema

- Practicar el uso de los iteradores del TAD `list`.

6.2. Ideas generales.

- El problema nos pide buscar en una secuencia de entrada *s1* la primera aparición de otra secuencia *s2*. El algoritmo que resuelve el problema es independiente del TAD utilizado para guardar las secuencias. En este problema nos piden que resolvamos el problema cuando las secuencias están almacenadas en un TAD lista, pero el algoritmo sería el mismo si las secuencias estuvieran almacenadas en vectores, o en cualquier tipo sobre el que se definan iteradores.
- El enunciado indica que el tamaño de la secuencia a buscar (*s2*) es menor que 10. Esto nos permite utilizar un algoritmo sencillo que para cada posición de la secuencia *s1* compruebe si a partir de esa posición se encuentra la secuencia *s2*. Un algoritmo de estas características tendrá coste en el caso peor $\mathcal{O}(n * k)$ siendo *n* el número de elementos de la secuencia *s1* y *k* el número de elementos de la secuencia *s2*. Siendo $k < 10$ nos garantiza un coste lineal en el número de elementos de la secuencia, por lo que no es necesario emplear algoritmos de búsqueda de secuencias mas elaborados como el KMP.
- La función se utiliza para eliminar todas las apariciones de la secuencia *s2* en la secuencia *s1*. Para ello se utiliza un bucle que llama a la función `mySearch` y, si esta devuelve un iterador a una posición de la secuencia, elimina esa aparición de la secuencia *s2* en *s1*. El proceso se repite hasta que la función `mySearch` devuelve un iterador al final de la secuencia *s1* indicando que ya no existen más apariciones de la secuencia *s2* en *s1*.
- La función `search` de la librería `algorithm` implementa esta búsqueda.

6.3. Ideas detalladas.

- El algoritmo se implementará con un bucle externo que recorra la secuencia *s1*.
- Para cada posición *i* de la secuencia *s1*, se realizará un bucle que compruebe si los valores de las posiciones posteriores a *i* en la secuencia coinciden con los valores de *s2*.
- Para ello se utilizan dos índices, que en este problema serán iteradores. El primero de ellos recorre la lista *s1* desde la posición *i* y el segundo recorre la lista *s2* desde el comienzo.
- Si se detecta un elemento que no coincide en ambas listas el bucle termina indicando que no se ha encontrado el patrón.
- Si coinciden todos los elementos de la secuencia *s2* el bucle termina indicando que se ha encontrado una coincidencia.
- Si se termina de recorrer la secuencia *s1* sin que haya terminado de recorrerse la secuencia *s2* el bucle debe terminar indicando que no se ha encontrado la secuencia.

6.4. Algunas cuestiones sobre implementación.

- Se realiza una función genérica que permita resolver el problema sobre cualquier TAD que tenga definidos iteradores. Para ello el tipo genérico *T* representa el iterador utilizado para recorrer el TAD.

```
template <class T>
T Mysearch(T const& ini1, T const& fin1,
           T const& ini2, T const& fin2){
```

- La llamada a la función instancia el tipo genérico *T* con el iterador con el que se realiza la llamada. Por ejemplo, en el siguiente fragmento de código, `lista1` es un objeto de clase `list` cuyos elementos son de tipo `int`. El objeto `it1` es de tipo `std::list<int>::iterator`, es decir, es un iterador a una lista de enteros. El tipo `auto` es capaz de inferir este tipo a partir del método `begin()` con el que se

inicializa. La llamada a la función se realiza con el iterador `it1` y el tipo `T` queda instanciado al tipo de `it1`. Se observa en la implementación al final del problema que `lista1.end()`, `lista2.begin()`, y `lista2.end()` tienen todos ellos este mismo tipo, por lo que la llamada es correcta.

```
std::list<int> lista1;    auto it1 = lista1.begin();
while (it1 != lista1.end()) {
    it1 = Mysearch(it1, lista1.end(), lista2.begin(), lista2.end());
    if (it1 != lista1.end()) {
        for (int i = 0; i < lista2.size(); ++i) {
            it1 = lista1.erase(it1);
        }
    }
}
```

- El bucle `while` anterior se ejecuta mientras no llega al final de la lista. Recordad que el método `end()` devuelve un iterador al final de la lista, y por lo tanto esta es la forma correcta de recorrer una lista hasta el final.
- El iterador avanza con las llamadas a la función `mySearch`.
- Se utiliza el método `erase` para eliminar de la lista `lista1` cada uno de los elementos de la lista `list2`. Los elementos se eliminan de uno en uno dado que no tenemos accesible un iterador al final de la secuencia a eliminar. No puede utilizarse la aritmética de punteros sobre el iterador `it1` dado que este es un iterador sobre el tipo `list` y por lo tanto un *Bidirectional iterator*.
- El método `erase` devuelve un iterador al elemento siguiente al que se ha eliminado. Se debe asignar este iterador a `it1` para poder seguir recorriendo la lista, ya que el iterador con el que se hace la llamada queda apuntando al elemento eliminado.

6.5. Errores frecuentes.

- Solo deben eliminarse las apariciones que se encuentran en la secuencia de entrada, no las que se puedan formar al ir eliminando elementos. Por ejemplo en la secuencia 4 1 1 3 3 8, si eliminamos la secuencia 1 3, el resultado correcto es 4 1 3 8, ya que la secuencia 1 3 que queda se ha formado al eliminar la que existía en la secuencia original.

6.6. Coste de la solución

Debemos estudiar de forma separada el coste de la función `mySearch` y el coste del algoritmo que resuelve el problema:

- La función `mySearch` tiene un bucle que recorre en el caso peor todos los elementos entre `ini1` y `fin1`. En el cuerpo del bucle tenemos una comparación de dos elementos, el coste de esta comparación depende del tipo de estos elementos, es decir del tipo de los elementos de la lista. A continuación encontramos un bucle, que en el caso peor se ejecuta para todos los elementos de la lista 2. En el cuerpo del bucle tenemos una comparación de elementos cuyo coste depende del tipo de los elementos de la lista, e incrementos de iteradores que tienen coste constante. Asumiendo que la comparación de elementos sea constante el coste del bucle interno es del orden de la longitud de la lista 2. Por lo tanto el coste del bucle externo y de la función es $\mathcal{O}(n * k)$, siendo n el número de elementos de la lista 1 y k el número de elementos de la lista 2.
- En la función `resuelveCaso` analizamos el coste del bucle que recorre la lista 1 para borrar las secuencias de la lista 2. En el caso peor, la secuencia de la lista 2 se encuentra de forma consecutiva en lista 1. El bucle daría n/k vueltas. En cada vuelta del bucle se llama a la función `mySearch` que tiene coste $n * k$, y además se borran todas las apariciones de la lista, lo que tiene coste k en cada vuelta del bucle. Tendríamos por lo tanto un coste cuadrático del algoritmo. Si se realiza un análisis más fino, vemos que el coste es realmente del orden de $\mathcal{O}(n * k)$, Para ello se observa cómo avanza el iterador en cada llamada a la función `mySearch` y el coste de cada llamada. Si tenemos x llamadas a la función, y sumamos el coste de cada llamada tenemos $n_1 * k + n_2 * k + \dots + n_x * k = (n_1 + n_2 + \dots + n_x) * k$ con $n_1 + n_2 + \dots + n_x = n$, siendo n el número de elementos de la lista 1.

6.7. Implementación.

```
#include <iostream>
#include <list>

template <class T>
T Mysearch(T const& ini1, T const& fin1, T const& ini2, T const& fin2){
    auto it1 = ini1; auto it2 = ini2;
    while (it1 != fin1 && it2 != fin2) {
        if (*it1 == *ini2) { // recorre el subrango
            auto aux1 = it1; it2 = ini2;
            while (aux1 != fin1 && it2 != fin2 && *aux1 == *it2) {
                ++aux1;
                ++it2;
            }
            if (it2 != fin2) ++it1;
        }
        else ++it1;
    }
    return it1;
}

bool resuelveCaso() {
    int nElem1, nElem2; std::cin >> nElem1 >> nElem2;
    if (nElem1 == 0 && nElem2 == 0) return false;
    std::list<int> lista1;
    for (int i = 0; i < nElem1; ++i){
        int aux;
        std::cin >> aux;
        lista1.push_back(aux);
    }
    std::list<int> lista2;
    for (int i = 0; i < nElem2; ++i){
        int aux;
        std::cin >> aux;
        lista2.push_back(aux);
    }
    // Elimina todas las apariciones de la lista2 en la lista 1
    // Utilizando la funcion Mysearch
    auto it1 = lista1.begin();
    while (it1 != lista1.end()) {
        it1 = Mysearch(it1, lista1.end(), lista2.begin(), lista2.end());
        if (it1 != lista1.end()) {
            for (int i = 0; i < lista2.size(); ++i) {
                it1 = lista1.erase(it1);
            }
        }
    }
    // Escribe la lista
    auto it = lista1.begin();
    if (it != lista1.end()){
        std::cout << *it;
        ++it;
    }
    for (; it != lista1.end(); ++it)
        std::cout << ' ' << *it ;
    std::cout << '\n';

    return true;
}
```



```

int main() {

#ifdef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifdef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}

```