# Cuaderno de problemas
# Estructuras de Datos. TADs lineales

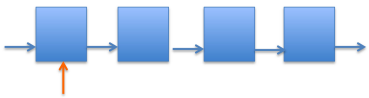
Prof. Isabel Pita

22 de febrero de 2023

# Índice

# 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.

### 1.3.1. Tipos genéricos.

- 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;
```

### 1.3.2. La clase `linked_list_ed`. Atributos y operaciones.

1. 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`).

2. 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`.

3. La clase `linked_list_ed` tiene dos atributos: un puntero al primer nodo, `prim`, y un puntero al último nodo de la lista, `ult`.

4. 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 $\sim$ . 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.

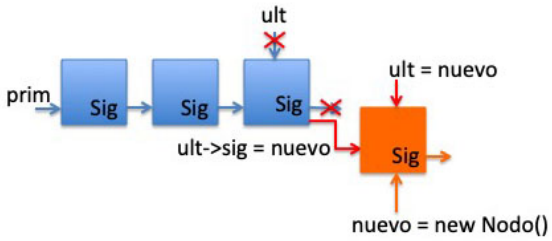
   ```
   linked_list_ed(linked_list_ed<T> const& other) {
     copia(other);
   }
   ```

   *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);
   ```

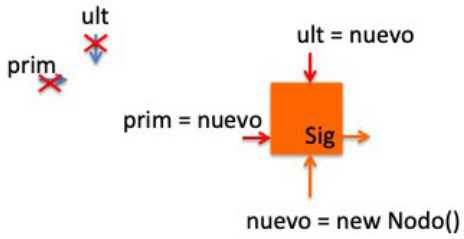
```
    }
    return *this;
}
```

*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.

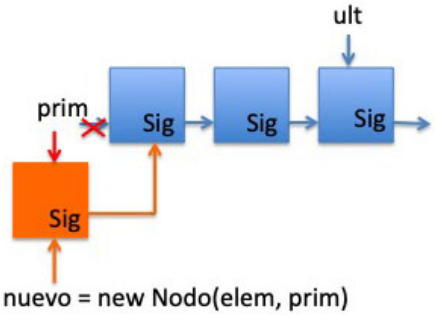



```
void push_back(T const& elem) {
  Nodo * nuevo = new Nodo(elem);
  if (ult != nullptr)
    ult->sig = nuevo;
  ult = nuevo;
  if (prim==nullptr) // lista vacia
    prim = nuevo;
}
```

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 `prim` tenía el valor `nullptr`). Si la lista no estaba vacía no hay que modificar este puntero.

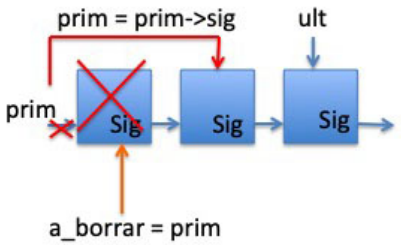



*g*) Añadir un elemento al principio de la lista: `push_front`. Se reserva memoria para un nuevo nodo con el operador `new` que llama al constructor del `Nodo` con dos parámetros: el valor del elemento y el valor que se asignará al puntero `sig`. Actualiza el valor del puntero `prim` para que quede en 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;
}
```

5. 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;
    }
  }
}
```

### 1.3.3. Propiedades genéricas.

- 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 multiplo 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 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 leido 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;});
```

### 1.3.4. Comprobar que la lista está correctamente construida.

Para comprobar que la lista está correctamente construida rotaremos todos los nodos de la lista una vez. Se declarará una función `rotar` en la clase `linked_list_ed_plus` en la que se están definiendo las nuevas operaciones. El código de esta función es:

```
void rotar() {
    if (!this->empty() && this->prim->sig != nullptr) {
        // Dos o mas elementos
        Nodo* a_mover = this->prim;
        // Desengancha el primer nodo
        this->prim= a_mover->sig;
        a_mover->sig = nullptr;
        // Lo engancha al final de la lista
        this->ult->sig = a_mover;
        // repite el proceso para todos los demas nodos de la lista
        Nodo* act = a_mover;
        while (this->ult->sig != nullptr) {
            // Desengancha el primer nodo
            a_mover = this->prim;
            this->prim = a_mover->sig;
            // Lo engancha al final de la lista
            a_mover->sig = nullptr;
            act->sig = a_mover;
            act = act->sig;
        }
    }
}
```

## 1.4. Errores frecuentes.

No se conocen

## 1.5. Implementación.

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

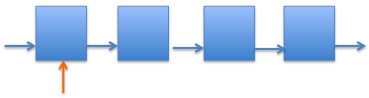
```
#include "linked_list_ed.h"

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() {

#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;
}
```

## 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 caracter `r`, obtendremos la lista: `d 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 inserción (<<) 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.

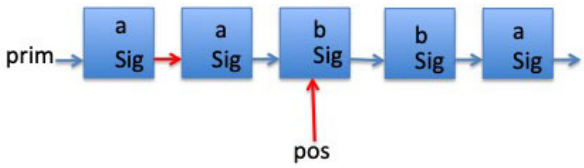
- El código de la clase `linked_list_ed` se explicó en el problema anterior *Recorrido de una lista enlazada simple*.
- Código de la clase `linked_list_ed_plus` que usaremos para implementar las nuevas operaciones de la lista enlazada.

  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.

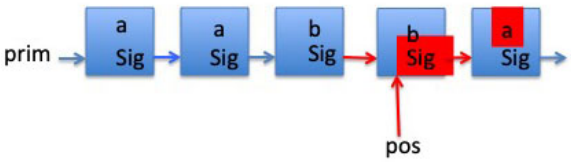
     *a*) Se busca la primera aparición del carácter que nos piden. Decidimos añadir los elementos que se desplazan detrás de la primera aparición del carácter para no tener que diferenciar el caso en que el carácter pedido esté en la primera posición de la lista, por lo tanto se marca con un puntero, `pos`, 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;
```

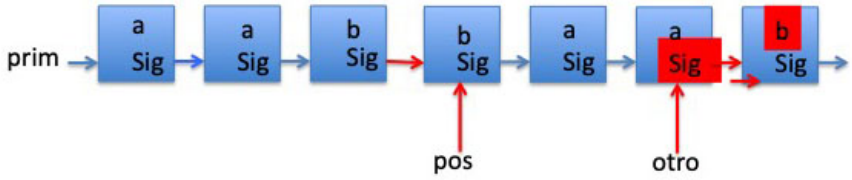
Al finalizar el bucle el puntero `pos` se encuentra apuntando al primer nodo con el carácter pedido. Si el carácter no existe apuntará al valor `nullptr`.

*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).



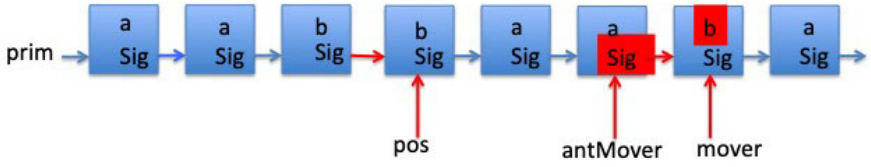

```
// saltamos todos los elementos iguales para
// anadir los elementos detras de la
// posicion de pos
while (pos->sig!=nullptr&&pos->sig->elem==ch)
  pos = pos->sig;
```

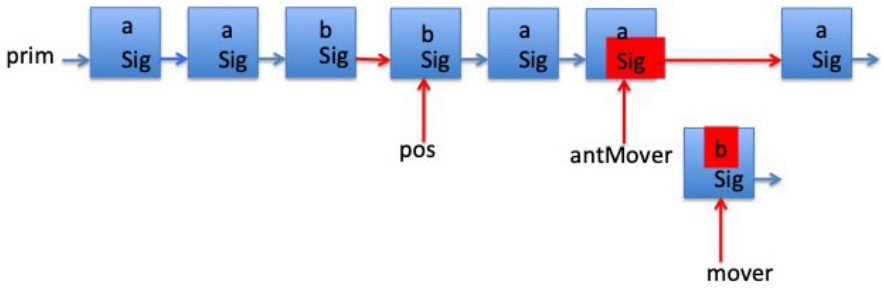
*c*) A continuación, buscamos todas las apariciones siguientes para agruparlas. En este caso debemos marcar con un puntero, `otro`, el nodo anterior a la aparición del carácter, para poder desligar el nodo de la lista. Observad como se pregunta por el elemento del nodo siguiente al apuntado por el puntero,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

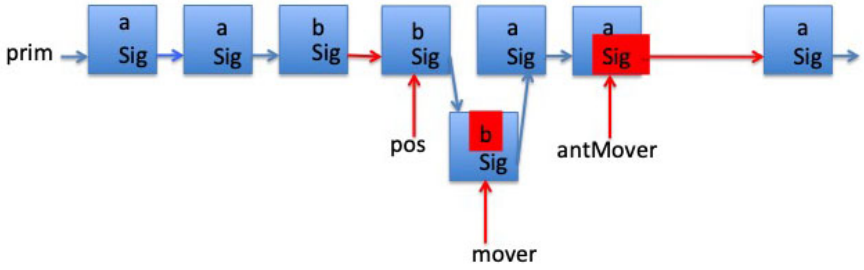



```
// Busca caracteres posteriores para agruparlos
Nodo* otro = pos->sig; // salta el nodo apuntado por pos
while (otro->sig != nullptr){
  if (otro->sig->elem == ch) {  // si contiene el elemento
    moverNodo(pos, otro);  // desplaza el nodo
    pos = pos->sig; // para que quede detras del ultimo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

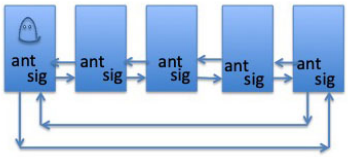
```
#include "linked_list_ed.h"

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 inserción (<<) 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

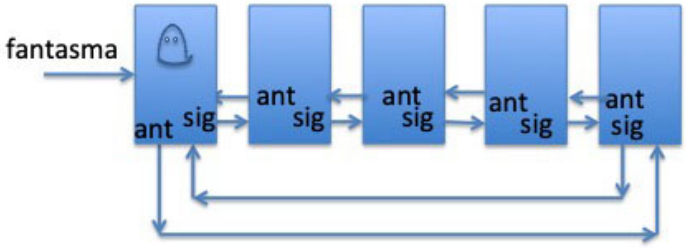
```
15:30:45 15:02:02 02:40:35 12:01:00 10:30:25
10:50:00 20:40:01 20:55:40 20:32:12
00:00:00 00:00:00 00:00:00 00:00:00 00:00:00 00:00:00
22:10:10 10:59:59 10:58:58 10:57:57 00:40:40 00:20:20
12:00:00 22:00:30 22:20:20 22:05:05 22:22:22 14:10:10 02:40:40 15:00:00 16:00:00 20:41:41
```

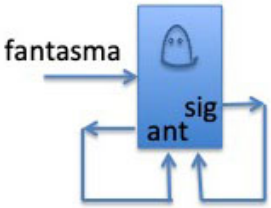
## 3.1. Objetivos del problema

- 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_liked_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.

### 3.3.1. 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`).

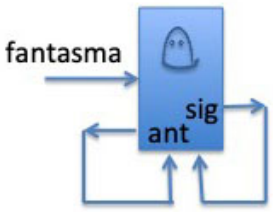
4. El `struct Nodo` consta de tres campos, el elemento que será del tipo genérico y dos punteros a una estructura de tipo `Nodo`. Uno de ellos guardará la dirección de memoria del nodo anterior en la lista y el otro guardará la dirección de memoria del nodo siguiente de la lista. Ofrece dos constructores, 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 $\sim$ . 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 prohibe 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.

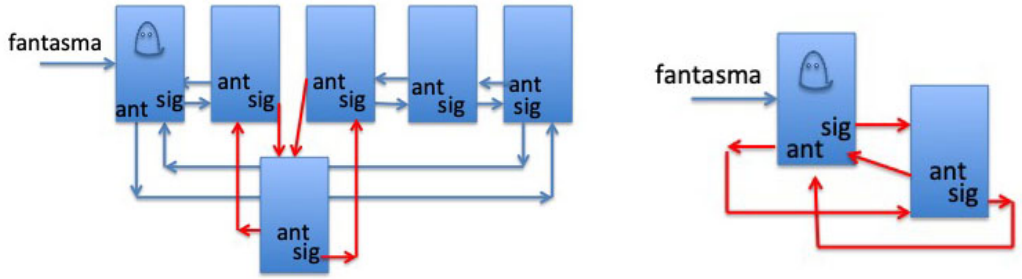
```
void libera() {
  // primero rompemos la circularidad
  fantasma->ant->sig = nullptr;
  fantasma->ant = nullptr;
  while (fantasma != nullptr) {
    Nodo * a_borrar = fantasma;
    fantasma = fantasma->sig;
    delete a_borrar;
  }
}
```

*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;
  }
}
```

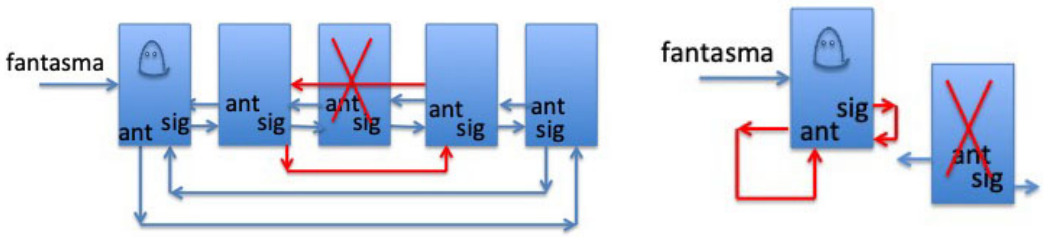
*c*) `inserta_elem`. Añade un nodo entre los punteros `anterior` y `siguiente` que recibe como parámetros. Para ello se reserva la memoria utilizando el constructor con tres parámetros y a continuación se enlaza la memoria reservada en la lista. Observad que al estar utilizando un nodo fantasma, el código es el mismo para el primer elemento de la llista, 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 precondición de la función se pide que el parámetro no tenga el valor `nullptr`. La precondición 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;
}
```




### 3.3.2. Código de la clase `double_linked_list_ed_plus`.

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).

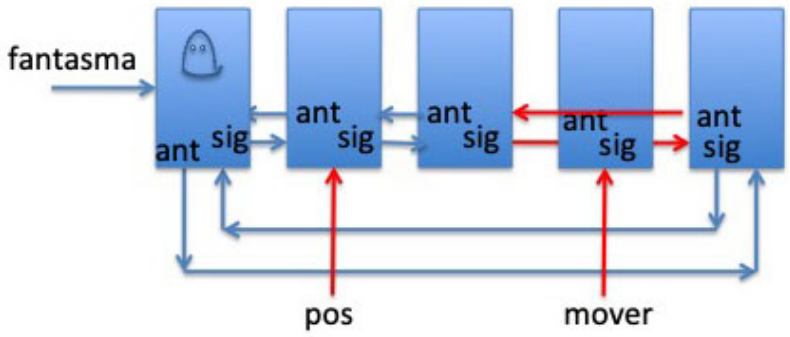
```
// Busca la primera aparicion del elemento
Nodo* pos = this->fantasma->sig;
while (pos != this->fantasma && !(pred(pos->elem)))
  pos = pos->sig;
```

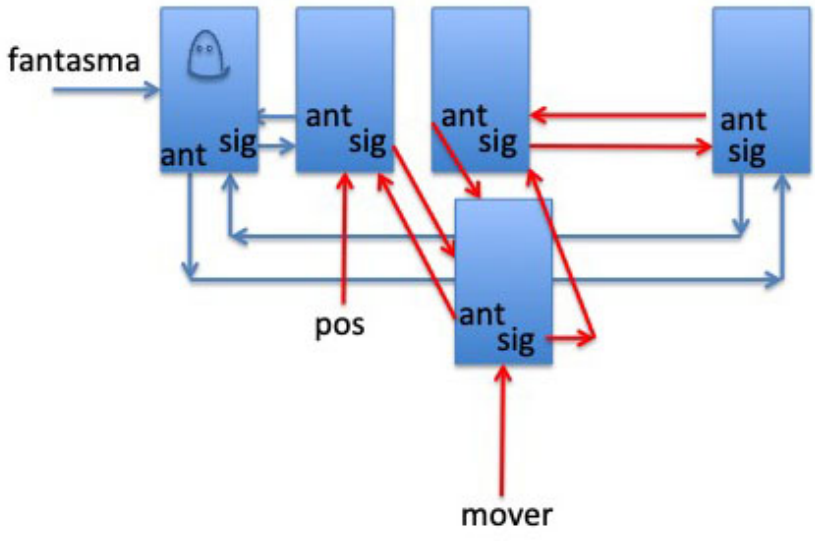
Al finalizar el bucle el puntero `pos` guarda la dirección de memoria del primer nodo con la hora pedida. Si la hora no existe tendrá el valor del puntero al nodo fantasma.

*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.

### 3.3.3. 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.

### 3.3.4. Comprobar que la lista está correctamente construida.

Para comprobar que la lista está correctamente construida rotaremos todos los nodos de la lista una vez. Se declarará una función `rotar` en la clase `linked_list_ed_plus` en la que se están definiendo las nuevas operaciones. El código de esta función es:

```
void rotar() {
    if (!this->empty() && this->fantasma->sig->sig != nullptr) {
        // Dos o mas elementos
        Nodo* a_mover = this->fantasma->sig;
        Nodo* ult = this->fantasma->ant;
        // Desengancha el primer nodo
        this->fantasma->sig= a_mover->sig;
        this->fantasma->sig->ant = this->fantasma;
        // Lo engancha al final de la lista
        a_mover->sig = this->fantasma;
        a_mover->ant = this->fantasma->ant;
        this->fantasma->ant->sig = a_mover;
        this->fantasma->ant = a_mover;
        // repite el proceso para todos los demas nodos de la lista
        Nodo* act = a_mover;
        while (ult->sig != this->fantasma) {
            a_mover = this->fantasma->sig;
            // Desengancha el primer nodo
            this->fantasma->sig= a_mover->sig;
            this->fantasma->sig->ant = this->fantasma;
            // Lo engancha al final de la lista
            a_mover->sig = this->fantasma;
            a_mover->ant = act;
            act->sig = a_mover;
            this->fantasma->ant = a_mover;
```

```
                act = act->sig;
            }
        }
    }
```

## 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() {

#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;
}
```

# 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 + 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 da $n$ 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.

En el problema se utiliza *una pila*. Utilizaremos el tipo `stack` de la librería `STL`

```
#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() {

#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;
}
```

# 5. Pilas, paréntesis equilibrados.

## Paréntesis equilibrados

Un conocido problema de aplicación del TAD `stack` es el problema de los paréntesis equilibrados: comprobar que una expresión que contiene paréntesis, corchetes y llaves abiertos y cerrados, es correcta, es decir:

1. todos los bloques que se abren se cierran.
2. ningún bloque se cierra sin haber sido abierto.
3. no se cierra un bloque si no se han cerrado todos los bloques que contiene.

### Entrada

La entrada está formada por una serie de casos de prueba, cada uno en una línea. Cada caso consiste en una secuencia de caracteres.

### Salida

Para cada caso de prueba se escribirá en una línea la palabra `SI` si la secuencia de caracteres está equilibrada y `NO` en caso contrario.

### Entrada de ejemplo

```
Hola, me llamo Jose (Pepe para los amigos).
Yo soy Francisco (o Paco]
{((()))}[]
][
```

### Salida de ejemplo

```
SI
NO
SI
NO
```

**Autor:** Isabel Pita

## 5.1. Objetivos del problema

- Conocer el TAD `stack` de la STL.
- Conocer el problema de comprobar si una expresión que contiene paréntesis, corchetes y llaves está bien construida.

## 5.2. Ideas generales.

- Se debe comprobar que todo símbolo especial: paréntesis, corchete o llave, que se abre en una expresión se cierra correctamente. Para ello hay que llevar cuenta de los símbolos que se van abriendo y los que se van cerrando.
- Si sólo tuviésemos un símbolo especial, por ejemplo el paréntesis, bastaría con llevar en una variable el número de paréntesis que se abren. Si se intenta cerrar un paréntesis cuando no queda ninguno abierto, es decir cuando la variable tiene el valor cero, se produce un error, igual que si al terminar el proceso quedan paréntesis abiertos.
- En este problema tenemos varios símbolos, y el orden en que se cierran debe corresponder con el orden en que fueron abiertos. Es decir, no basta con saber que se abrió un paréntesis sino que tenemos que saber también la posición en que se abrió para poder comprobar que todos los símbolos que se abrieron después ya están cerrados. Por ejemplo la expresión ({)} es errónea, a pesar de que hay un paréntesis abierto y otro cerrado y una llave abierta y otra cerrada. Cuando se encuentra el paréntesis cerrado se debe conocer que hay una llave que se abrió después de abrir el paréntesis y que todavía no está cerrada.
- Se debe por lo tanto guardar el orden en que se han ido abriendo los símbolos, para poder comprobar que se cierran en el mismo orden. Cómo el orden en que se cierran debe ser inverso al orden en que se abren, el TAD apropiado para guardar los valores es una *pila* de tipo `stack`.

## 5.3. Ideas detalladas.

- Guardaremos la información sobre los símbolos especiales que se van abriendo en una pila.
- La función debe leer la cadena de caracteres de la entrada y procesar cada uno de los caracteres:
  - Si el carácter leído no es un símbolo especial se pasa al siguiente carácter.
  - Si el carácter leído es un símbolo abierto se almacenará en la pila.
  - Si el carácter leído es un símbolo cerrado se debe comprobar si es el cierre del último símbolo abierto. Para ello se comprueba si tenemos algún símbolo abierto en la pila y en ese caso si coincide con la cima de la pila. Si coincide la expresión está bien formada, se desapila la cima de la pila y se pasa a procesar el siguiente carácter de la entrada. Si no coincide es que la expresión es errónea y se debe finalizar el proceso.
  - Al terminar el procesamiento de la entrada se debe comprobar que no quedaron símbolos sin cerrar, esto se comprueba observando si la pila está vacía al acabar el proceso.

## 5.4. Algunas cuestiones sobre implementación.

- La entrada se leerá en una variable de tipo `string`. Como la cadena de caracteres puede tener blancos se debe leer la entrada con la función `getline`.
- Cómo sólo se utilizan tres símbolos especiales, dados dos símbolos, uno abierto y otro cerrado, se comprobará si se corresponden con paréntesis, corchetes o llaves. Si pudiese haber muchos símbolos especiales sería necesario utilizar un diccionario (o tabla) para comprobar si el símbolo cerrado se corresponde con el símbolo abierto. Esto se verá en el tema 4 de la asignatura.

# 6. Pilas, Evaluar una expresión en notación postfija.

# Evaluar expresiones postfijas

Un conocido problema de aplicación del TAD `stack` es el problema de evaluar expresiones aritméticas en notación postfija. Existen tres formas de escribir una expresión aritmética: *notación infija*, *notación postfija* y *notación prefija*. La notación infija es la que utilizamos normalmente, en ella el operador se escribe entre los dos operandos, por ejemplo 3 + 2. Esta notación requiere el uso de paréntesis cuando se quiere modificar el orden habitual en que se evalúa la expresión, es decir de izquierda a derecha y teniendo en cuenta la prioridad de los operadores. Asi tenemos la expresión (3 + 2) * 5 cuyo resultado es 25. La notación postfija sitúa el operador a continuación de los operandos sobre los que actúa. Fue muy utilizada en las primeras calculadores científicas, debido a que las expresiones son más sencillas de evaluar. Por ejemplo, la expresión infija 3 + 2, se expresa en notación postfija como 3 2 +, mientras que la expresión (3 + 2) * 5 se expresa como 3 2 + 5 *.

En este problema vamos a desarrollar una calculadora sencilla que funcione en notación postfija. La calculadora sólo permite cinco operaciones, suma, resta, multiplicación, división y módulo, y solo admite números de una cifra.

## Entrada

La entrada comienza con un valor que indica el número de casos de prueba. Cada caso es una línea con la expresión a evaluar. Las expresiones son siempre correctas, no contienen blancos y los números sólo tienen una cifra.

## Salida

Para cada caso de prueba se escribirá en una línea el resultado de evaluar la expresión dada.

## Entrada de ejemplo

```
5
61+
543+-
32*86-+
84/93/*
55*3%
```

## Salida de ejemplo

```
7
-2
8
6
1
```

**Autor:** Isabel Pita

## 6.1. Objetivos del problema

- Conocer el TAD `stack` de la STL.
- Conocer el problema de evaluar una expresión en notación postfija.

## 6.2. Ideas generales.

- En notación postfija, una expresión se evalúa siempre de izquierda a derecha y cada operador actúa sobre los operadores que le preceden. En este problema se consideran sólo 5 operadores binarios: suma, resta, multiplicación, división y módulo.
- Debemos almacenar los operandos que van apareciendo en la expresión hasta que encontremos un operador. Cómo el operador se aplica sobre los últimos operandos que hayan aparecido en la expresión, el TAD adecuado para guardar los valores es una pila.

## 6.3. Ideas detalladas.

- Recorremos la expresión guardando los operandos en una pila.
- Cuando se encuentra un operador se obtienen los dos valores de la cima de la pila y se aplica el operador sobre ellos.
- El resultado de la operación se guarda en la pila, ya que el siguiente operador se aplicará sobre él
- Por ejemplo, la expresión 3 2 + 5 *. Guarda en la pila los valores 3 y 2, a continuación los saca de la pila para aplicar el operador suma. El resultado de esta suma se guarda en la pila, para que sea multiplicado por 5 a continuación.

## 6.4. Problemas relacionados.

- Transformar una expresión en notación infija a postfija.
  - Dada una expresión en notación infija, por ejemplo 7 - 5 + 5 * 3 + 4 * 2. Observamos que primero se deben realizar las dos multiplicaciones que tienen mayor prioridad que la suma y la resta. Por lo tanto tenemos que guardar la suma y resta iniciales hasta que se hayan realizado las multiplicaciones. La expresión postfija es 7 5 5 3 * + - 4 2 * +. Vemos que la suma y la resta iniciales invierten su orden en la expresión postfija. Por lo tanto los operadores que vamos encontrando y que todavía no se deben aplicar se guardan en una pila. La cuestión a resolver es cuándo debemos aplicar los operadores. Sabemos que se debe aplicar la operación 5 * 3 cuando encontramos el siguiente operador que es una suma. Como la suma tienen menor prioridad que la multiplicación debe aplicarse antes esta, así como todas las demás operaciones que haya en la pila con una prioridad mayor o igual que el operador que estamos considerando.
  - Observemos que pasa cuando la expresión tiene paréntesis, por ejemplo 3 + (5 + 2 * 3) + 4 / 2 - 1. En este caso debemos ejecutar primero la operación 5 + 2 * 3. Al resultado le sumaremos 3 y continuaremos ejecutando la división y la resta. La expresión postfija resultante debe ser: 3 5 2 3 * + + 4 2 / + 1 -. Tratamos el paréntesis abierto como una operación con prioridad máxima cuando se compara con los valores de la pila y con prioridad mínima cuando los operadores se comparan con él. De esta forma al encontrar un paréntesis abierto se añade a la pila sin desapilar ningún operador que pudiera haber en ella y sólo se elimina de la pila cuando aparece un paréntesis cerrado que obliga a desapilar toda la pila hasta el correspondiente paréntesis abierto.
  - Al acabar de tratar la expresión, si quedan operadores en la pila se deben desapilar.

### 6.5. Implementación.

```
int aplicar(char c, int a, int b) {
    switch (c) {
        case '+' : return a + b;
        case '-' : return a - b;
        case '*' : return a * b;
        case '/' : return a / b;
        case '%' : return a % b;
    }
    return 0;
}


void resuelveCaso () {
    std::string expr;
    std::cin >> expr;
    std::stack<int> pila;
    for (char c : expr) {
        if (isdigit(c)) pila.push(c-'0');
        else {
            int op2 = pila.top(); pila.pop();
            int op1 = pila.top(); pila.pop();
            pila.push(aplicar(c,op1,op2));
        }
    }
    std::cout << pila.top() << '\n';
}
```

# 7. Pilas, Problemas en que es necesario guardar varios valores.

## ¿Cuántos puntos puedo ver?

Tenemos la altura de una serie de puntos del plano en una determinada dirección. Dado un punto queremos contar cuantos puntos posteriores a él podríamos ver si nos situasemos en esa altura. Desde un punto se pueden ver todos los puntos posteriores con una altura estrictamente menor que él hasta encontrar un punto con altura mayor o igual.

### Entrada

La entrada consta de una serie de casos de prueba. Cada caso se escribe en dos líneas. En la primera se indica en número de puntos de los que se tiene la altura. En la segunda se indica la altura de cada punto. La entrada termina con un valor cero que no debe procesarse.

### Salida

Para cada caso de prueba se escribirá en una línea el número de puntos que se ven desde cada punto, empezando por el último de la entrada. Se considera que el último punto ve siempre cero puntos.

### Entrada de ejemplo

```
5
10 3 1 2 6
4
1 2 3 4
4
4 3 2 1
10
4 2 10 6 4 2 5 12 3 5
0
```

### Salida de ejemplo

```
0 0 0 2 4
0 0 0 0
0 1 2 3
0 0 2 0 0 1 3 4 0 1
```

**Autor:** Isabel Pita

## 7.1. Objetivos del problema

- Conocer el TAD `stack` de la STL.
- Resolver problemas en los que es necesario guardar varios valores de los ya recorridos, cuando el acceso a los elementos guardados se hace en orden inverso al que fueron guardados.

## 7.2. Ideas generales.

- El problema se puede resolver fácilmente con complejidad cuadrática en el número de elementos. Se recorre el vector de izquierda a derecha y para cada elemento se busca el primer elemento a su derecha que sea mayor que él. En el caso peor, el vector está ordenado de forma decreciente y para cada elemento debemos recorrer el resto del vector a su derecha.
- Para mejorar el coste de la implementación, estudiamos si es posible guardar información que evite recorrer el vector para cada elemento. Para cada punto nos interesa saber el primer punto a su derecha mayor o igual que él. Por lo tanto, para obtener información relevante para un punto debemos haber recorrido los elementos de su derecha, esto nos indica que el vector lo recorreremos de derecha a izquierda.
- Debemos guardar aquellas alturas que puedan ser el final de las vistas para algún punto a su izquierda. Cómo el final de las vistas es el punto a la derecha más cercano, el último valor guardado debe ser el primero en consultarse, ya que es el más cercano. Por lo tanto utilizaremos una pila para guardar los valores.
- Veamos ahora que valores debe guardar la pila. Todo valor es candidato a ser el final de vistas de los elementos a su izquierda, por lo que debe ser añadido a la pila, sin embargo observamos que los valores ya añadidos a la pila que sean menores que el valor que estamos tratando no podrán ser final de vistas para ningún punto de la izquierda, ya que son menores que el valor que se va a añadir a la pila. Por lo tanto, antes de añadir un valor a la pila se deben eliminar de esta los valores que sean menores que él.

## 7.3. Ideas detalladas.

- Se recorren los valores de la entrada de derecha a izquierda.
- Para cada valor se comprueba si existe algún elemento en la pila que sea mayor que él. Si el elemento de la cima de la pila es más pequeño se elimina de la pila y se comprueba el siguiente valor. Si la pila se queda vacía es que se pueden ver todos los puntos a la derecha del elemento que estamos considerando. Si se encuentra un valor más grande, este valor es el final de las vistas.
- Después de calcular el número de puntos que se ven, se debe añadir ese punto a la pila, ya que podría ser el final de vistas para puntos a su izquierda.
- Para poder calcular el número de puntos que se pueden ver es necesario guardar en la pila además de la altura del punto su posición.

## 7.4. Coste de la solución.

La solución tiene coste lineal en el número de elementos de la entrada. Observamos que cada elemento del vector se trata una única vez. El problema radica en que cuando tratamos un elemento, debemos desapilar todos los valores de la pila que sean menores que él. Sin embargo, dado que los valores de la pila, una vez desapilados no vuelven a entrar nunca en la pila, el número total de veces que se apilan y desapilan elementos en la pila (considerando todas las iteraciones del algoritmo) está acotado por el número de elementos de entrada. Por lo tanto el número de vueltas del bucle `while` interno es como mucho $n$ en todas las iteraciones del bucle for, siendo $n$ el número de valores de la entrada.

Cómo el bucle `for` da $n$ vueltas y las vueltas totales del bucle `while` están acotadas, la complejidad es de $\mathcal{O}(n)$ siendo $n$ el número de elementos de la entrada.

## 7.5. Implementación.

```
bool resuelveCaso () {
    int numElem;
    std::cin >> numElem;
    if (numElem == 0) return false;
    std::vector<int> v(numElem);
    for (int & i : v) std::cin >> i;
    std::stack<std::pair<int,int>> p;
    p.push({v[v.size()-1],v.size()-1});
    std::cout << '0';
    for (int i = (int)v.size()-2; i >= 0; --i) {
        while (!p.empty() && p.top().first < v[i]) p.pop();
        if (p.empty()) std::cout << ' ' << v.size() - i - 1;
        else std::cout << ' ' << p.top().second - i - 1;
        p.push({v[i],i});
    }
    std::cout << '\n';
    return true;
}
```

# 8. Colas, Clasificar valores.

## Colas de impresión

En este problema vamos a utilizar dos colas para clasificar los trabajos enviados a una impresora. Los trabajos se clasifican según el número de páginas que se envían. Por una lado guardaremos los trabajos de menos de 100 páginas y por otro los trabajos con 100 páginas o mas.

La impresora procesa los trabajos de un grupo hasta que se superan N páginas, después pasa a procesar los trabajos del otro grupo. Siempre empieza a procesar primero los trabajos con menos páginas y siempre se procesa al menos un trabajo de cada cola, aunque supere las N páginas. Si se superan las páginas para imprimir cuando está procesando un trabajo acaba de procesarlo.

### Entrada

La entrada está formada por una serie de casos de prueba, cada uno en una línea. Cada caso comienza con un línea en que se indica el número de trabajos enviados a la impresora, y el número de páginas que se imprimen cada vez. En la línea siguiente se indica el número de páginas de cada trabajo. La entrada de datos termina con un cero que no debe procesarse.

### Salida

Para cada caso de prueba se escribirá en una línea el orden en que se imprimen los trabajos. Los trabajos se identifican por su posición en la entrada, siendo la posición del primer trabajo la uno.

### Entrada de ejemplo

```
10 200
150 50 50 210 50 60 100 90 70 50
5 300
70 100 100 50 70
5 20
40 60 10 15 10
1 100
200
0
```

### Salida de ejemplo

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

**Autor:** Isabel Pita

## 8.1. Objetivos del problema

- Conocer el TAD `queue` de la STL.
- Uso de colas para clasificar valores sin perder el orden de entrada.

## 8.2. Ideas generales.

- Los valores se añaden a una cola u otra dependiendo del valor que tengan.
- La cola mantiene el orden de llegada en cada grupo.

## 8.3. Ideas detalladas.

- Los valores de entrada se procesan según se van leyendo de la entrada.
- Si el número de páginas leido es menor que 100 el valor se añade a la primera cola, si es mayor se añade a la segunda cola.
- Para poder identificar el trabajo cuando se imprima debemos guardar además del número de páginas la posición que ocupa en la entrada.
- Las colas se procesan empezando por la que tiene los trabajos con menos de 100 páginas. Imprimiremos trabajos de esta cola hasta que el número de páginas imprimidas supere el valor dado en la entrada.
- A continuación pasaremos a imprimir los trabajos de la segunda cola.
- Para diseñar el algoritmo utilizaremos el esquema de la mezcla ordenada. En un primer bucle imprimiremos trabajos mientras ninguna de las dos colas sea vacía. A continuación imprimiremos aquella cola que no se haya acabado de imprimir en el primer bucle. Para ello se implementan dos bucles secuenciales, en el primero se termina de imprimir la primera cola si no es vacía y en el segundo se termina de imprimir la segunda cola si no es vacía.

## 8.4. Algunas cuestiones sobre implementación.

- Se declara un vector de colas con dos posiciones. Esto nos permite tratar las dos colas con el mismo código, llevando en una variable el número de cola que estamos tratando. Para cambiar de una cola a otra utilizaremos el módulo: `numCola = (numCola + 1) % 2;`. Esta forma de implementarlo nos permite también pasar más facilmente a tener un mayor número de colas.

## 8.5. Implementación.

```
bool resuelveCaso() {
    // leer los datos de la entrada
    int numTrabajos,maxPaginas;
    std::cin >> numTrabajos;
    if (numTrabajos == 0) return false;
    std::cin >> maxPaginas;
    // cada cola guarda el numero de paginas del trabajo y
    // el numero del trabajo en la entrada
    std::vector<std::queue<std::pair<int,int>>> v(2);
    for (int i = 0; i < numTrabajos; ++i) {
        int p; std::cin >> p;
        if (p < 100) {
            v[0].push({p,i+1});
        }
        else {
            v[1].push({p,i+1});
        }
    }
```

```
    int numCola = 0; bool primero = true;
    while (!v[0].empty() && !v[1].empty()) {
        int paginas = 0;
        while (!v[numCola].empty() && paginas < maxPaginas) {
            paginas += v[numCola].front().first;
            if (!primero) std::cout << ' '; else primero = false;
            std::cout << v[numCola].front().second;
            v[numCola].pop();
        }
        numCola = (numCola+1)%2;
    }
    while (!v[0].empty()) {
        if (!primero) std::cout << ' '; else primero = false;
        std::cout << v[0].front().second;
        v[0].pop();
    }
    while (!v[1].empty()) {
        if (!primero) std::cout << ' '; else primero = false;
        std::cout << v[1].front().second;
        v[1].pop();
    }
    std::cout << '\n';
    return true;
}
```

# 9. 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 aparicion 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

## 9.1. Objetivos del problema

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

## 9.2. 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.

## 9.3. 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.

## 9.4. 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.

## 9.5. 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.

## 9.6. 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 $n_1*k+n_2*k+\ldots n_x*k = (n_1+n_2+\ldots n_x)*k$ con $n_1+n_2+\ldots n_x = n$, siendo $n$ el número de elementos de la lista 1.

## 9.7. 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() {

#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;
}
```