Capítulo 8

Aplicaciones de tipos abstractos de datos[1]

Resumen: En este tema se estudia la resolución de problemas mediante el uso de los distintos tipos/colecciones de datos vistos en los temas anteriores. Para ello se proponen varios ejercicios resueltos con detalle.

1. Confederación hidrográfica

Una confederación hidrográfica gestiona el agua de los ríos y pantanos de una cuenca hidrográfica. Para ello debe conocer los ríos y pantanos de su cuenca, el agua embalsada en la cuenca y en cada pantano. También se permite trasvasar agua entre pantanos del mismo río o de distintos ríos dentro de su cuenca. Debemos diseñar un TAD que permita a la confederación hidrográfica gestionar sus ríos y pantanos con las siguientes operaciones:

- `crea` (constructor sin argumentos): crea una confederación hidrográfica vacía.
- `an_rio(r)`: añade el río `r` a la confederación hidrográfica. Si el río ya existe la operación se ignora. En una confederación no puede haber dos ríos con el mismo nombre.
- `an_pantano(r, p, n1, n2)`: crea un pantano de capacidad `n1` en el río `r` de la confederación. Además lo carga con `n2` hm^3 de agua (si `n2 > n1` lo llena). Si ya existe algún pantano de nombre `p` en el río `r` o no existe el río la operación se ignora.
- `embalsar(r, p, n)`: carga `n` hm^3 de agua en el pantano `p` del río `r` de la confederación. Si no cabe todo el agua el pantano se llena. Si el río o el pantano no están dados de alta en la confederación la operación se ignora.
- `embalsado_pantano(r, p)`: devuelve la cantidad de agua embalsada en el pantano `p` del río `r`. Si el pantano no existe o no existe el río devolverá el valor -1.
- `reserva_cuenca(r)`: devuelve la cantidad de agua total embalsada en la cuenca del río `r`. Si no existe el río devolverá el valor -1.
- `trasvase(r1, p1, r2, p2, n)`: trasvasa `n` hm^3 de agua del pantano `p1` del río `r1` al pantano `p2` del río `r2`, en la confederación. Si alguno de los ríos o pantanos no existe, o bien si `n` es mayor que la cantidad de agua embalsada en `p1` o no hay capacidad suficiente en `p2`, la operación se ignora.

[1] Material adaptado a partir de los apuntes de Isabel Pita Andreu.

Se pide: elegir una **representación** adecuada para el TAD utilizando las librerías de la STL de manera que la implementación de las operaciones sea **eficiente**, **implementar** todas las operaciones e **indicar y justificar sus costes**.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea, con el nombre de la operación seguido de sus argumentos (separados por espacios). Los nombres de los ríos y pantanos serán cadenas de caracteres sin espacios. La palabra `FIN` en una línea indica el final de cada caso. A continuación se proporcionan unos casos de prueba de ejemplo:

```
an_rio tajo
an_rio ebro
an_rio duero
an_pantano tajo alcantara 100 80
an_pantano tajo azutan 20 5
an_pantano ebro cereceda 40 20
an_pantano ebro flix 30 35
embalsar tajo alcantara 30
embalsado_pantano tajo alcantara
reserva_cuenca tajo
an_pantano duero cervera 100 10
trasvase tajo alcantara duero cervera 20
reserva_cuenca tajo
reserva_cuenca duero
embalsado_pantano tajo alcantara
embalsado_pantano duero cervera
FIN
an_rio r1
an_pantano r2 p1 10 10
reserva_cuenca r1
reserva_cuenca r2
embalsado_pantano r1 p1
embalsado_pantano r2 p1
FIN
```

Salida

Las únicas operaciones que producen salida son:

- `embalsado_pantano`, que imprimirá la cadena `El pantano P del R tiene X hm3`, siendo P y R los parámetros de la operación y X su salida respectivamente; o la cadena `El rio o el pantano no existe` si la salida ha sido -1.
- `reserva_cuenca`, que imprimirá `La cuenca del rio R tiene X hm3` siendo R y X el parámetro de la operación y su salida respectivamente; o la cadena `El rio no existe` si la salida ha sido -1.

Cada caso termina con una línea con tres guiones (`---`).

La salida esperada para los casos de más arriba sería la siguiente:

```
El pantano alcantara del tajo tiene 100 hm3
La cuenca del rio tajo tiene 105 hm3
La cuenca del rio tajo tiene 85 hm3
La cuenca del rio duero tiene 30 hm3
```

```
El pantano alcantara del tajo tiene 80 hm3
El pantano cervera del duero tiene 30 hm3
---
La cuenca del rio r1 tiene 0 hm3
El rio no existe
El rio o el pantano no existe
El rio o el pantano no existe
---
```

Representación

Se propone utilizar una tabla con clave la identificación de los ríos y valor una estructura que incluiría: (1) otra tabla con todos los pantanos existentes en el río, cada uno con su capacidad y litros embalsados, y (2) el total de los litros embalsados en toda la cuenca del río. Se utilizan tablas porque las operaciones que se van a realizar tanto sobre ríos como sobre pantanos son consultas (búsquedas) e inserciones. Al no haber ninguna operación que requiera orden de ríos ni pantanos elegiremos el TAD `unordered_map` para las dos tablas.

```
using Rio = string;
using Pantano = string;

class ConfHidrografica {

private:
  struct InfoPantano {
    int capacidad;
    int litros_embalsados;
  };

  struct InfoRio {
    unordered_map<Pantano, InfoPantano> pantanos;
    int carga;
  };

  unordered_map<Rio, InfoRio> rios;

public:
  ConfHidrografica() { };
  void an_rio(const Rio& r);
  void an_pantano(const Rio& r, const Pantano& p, int cap, int carga);
  void embalsar(const Rio& r, const Pantano& p, int n);
  int embalsado_pantano(const Rio& r, const Pantano& p) const;
  int reserva_cuenca(const Rio& r) const;
  void trasvase(const Rio& r1, const Pantano& p1,
                const Rio& r2, const Pantano& p2, int n);
};
```

Implementación de las operaciones

Constructor

El constructor implícito es suficiente para inicializar la tabla de ríos el cual tendría coste constante.

Operación `an_rio`

Al añadir un río, si éste está en la cuenca no se debe modificar su valor (comportamiento por defecto de la operación `insert`). La tabla correspondiente al valor del río se crea vacía.

```
void ConfHidrografica::an_rio(const Rio& r) {
  // Si el rio r estaba no se inserta
  rios.insert({r, InfoRio()});
}
```

Coste de la operación implementada:

- El coste de crear una tabla vacía es constante.
- El coste de insertar un elemento en la tabla es constante (operación `insert`).

Por lo tanto, el coste de la operación es $\mathcal{O}(1)$.

Operación `an_pantano`

La operación que añade un pantano a un río, si el río está en la confederación, y si el pantano no está en el río, lo añade con la capacidad y litros embalsados que se indican. Si los litros embalsados son mayores que la capacidad se embalsa la capacidad total del pantano. También hay que incrementar el campo `carga` del río correspondientemente. Si el río no está en la confederación, o si el pantano ya está en el río la operación no tiene ningún efecto.

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                  int n1, int n2) {
  if (rios.count(r) && !rios[r].pantanos.count(p)) {
    rios[r].pantanos[p] = {cap, std::min(cap, carga)};
    rios[r].carga += std::min(cap, carga);
  }
}
```

Coste de la operación implementada:

- El coste de consultar una tabla es constante (operaciones `count` y `[]`).
- El coste de insertar un elemento en una tabla es constante (operación `[]`).
- El resto de operaciones tiene coste constante.

Por lo tanto, el coste de la operación es constante.

Aunque el coste de buscar una clave en una tabla sea constante no se puede considerar despreciable. Es por ello que se debe evitar en la medida de lo posible repetir búsquedas. El siguiente código hace uso de una variable de tipo referencia (alias) para evitar parte de dichas búsquedas redundantes. También se usa `insert` para así insertar el pantano solo si éste no existe (evitando la consulta previa con la llamada al `count`):

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                  int n1, int n2) {
  if (rios.count(r)) {
    InfoRio& infoR = rios[r]; // infoR es un alias de rios[r]
    if (!infoR.pantanos.count(p)) {
      infoR.pantanos.insert({p, {cap, std::min(cap, carga)}});
      infoR.carga += std::min(cap, carga);
    }
  }
}
```

Puesto que el operador [] inserta una clave cuando no está en la tabla y modifica el valor asociado en caso contrario, si lo utilizamos en este caso nos vemos obligados a consultar previamente si el rio está en la tabla, lo cual implica que todavía estamos repitiendo búsquedas.

El uso de iteradores con el método `find` permite optimizar más el código en cuanto a la no repetición de búsquedas:

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                  int n1, int n2) {
  auto itR = rios.find(r);
  if (itR != rios.end()) {
    auto [_,ok] = itR->second.pantanos.insert({p, {cap, std::min(cap, carga)}});
    if (!ok) itR->second.carga += std::min(cap, carga);
  }
}
```

Para evitar tener que preguntar si el pantano existe antes de insertar (con la consiguiente búsqueda redundante) hemos usado el par iterador-booleano que se obtiene como salida de la operación *insert* (en los maps de la STL) informando de si existía o no la clave en la tabla (y por tanto de si se ha insertado), y en caso de existir, el iterador apuntando al par ya existente (que en este caso no necesitamos).

Usaremos **auto** para no tener que escribir los tipos de iteradores que suelen ser largos y se deducen fácilmente por el contexto y el nombre de la variable. Por ejemplo, en el caso anterior el tipo de `itR` sería `unordered_map<Rio,InfoRio>::iterator`.

Operación `embalsar`

La operación de embalsar añade n hm^3 al agua embalsada en un pantano. Si se supera la capacidad, el pantano se considera lleno.

```
void ConfHidrografica::embalsar(const Rio& r, const Pantano& p, int n) {
  if (rios.count(r) && rios[r].pantanos.count(p)) {
    // Añade lo que quepa de n al pantano y a la cuenca de r
    int incrCarga = std::min(n, rios[r].pantanos[p].capacidad -
                                rios[r].pantanos[p].carga);
    rios[r].pantanos[p].carga += incrCarga;
    rios[r].carga += incrCarga;
  }
}
```

El coste de la operación implementada es constante. La justificación es análoga a la de la operación `an_pantano`. Sin embargo podemos optimizar mucho el código (en este caso aún más por las numerosas búsquedas que se están realizando, 8 de las cuales son redundantes y se pueden evitar) gracias al método `find`.

```
void ConfHidrografica::embalsar(const Rio& r, const Pantano& p, int n) {
  auto itR = rios.find(r);
  if (itR != rios.end()) {
    auto itP = itR->second.pantanos.find(p);
    if (itP != itR->second.pantanos.end()) {
      // Añade lo que quepa de n al pantano y a la cuenca de r
      int incrCarga = std::min(n, itP->second.capacidad -
                                  itP->second.carga);
      itP->second.carga += incrCarga;
      itR->second.carga += incrCarga;
    }
  }
}
```

Operación `embalsado_pantano`

La operación consulta los hm^3 embalsados en un pantano.

```
int ConfHidrografica::embalsado_pantano(const Rio& r, const Pantano& p) const {
  int n = -1;
  if (rios.count(r) && rios[r].pantanos.count(p))
```

```
    n = rios[r].pantanos[p].carga;
  return n;
}
```

El coste de la operación viene dado por el coste de consultar la información de un pantano en un río y por lo tanto es constante. Optimizando los accesos a las tablas usando el método `find` quedaría como sigue:

```
int ConfHidrografica::embalsado_pantano(const Rio& r, const Pantano& p) const {
    int n = -1;
    auto itR = rios.find(r);
    if (itR != rios.end()){
      auto itP = itR->second.pantanos.find(p);
      if (itP != itR->second.pantanos.end())
        n = itP->second.carga;
    }
    return n;
}
```

Una versión más concisa y también optimizada usando el método `at` y capturando su posible excepción sería la siguiente:

```
int ConfHidrografica::embalsado_pantano(const Rio& r, const Pantano& p) const {
  try {
    return rios.at(r).pantanos.at(p).carga;
  } catch (std::out_of_range& e){
    return -1;
  }
}
```

Operación `reserva_cuenca`

La operación `reserva_cuenca` obtiene la suma de los hm^3 embalsados en todos los pantanos de la cuenca de un río. Simplemente tendrá que acceder al correspondiente campo de la estructura `InfoRio` del río y chequear si el río no existe para devolver un -1.

```
int ConfHidrografica::reserva_cuenca(const Rio& r) const {
  auto itR = rios.find(r);
  if (itR != rios.end()) return itR->second.carga;
  else return -1;
}
```

El coste sería de nuevo constante como en los casos anteriores.

Operación `trasvase`

La operación realiza el trasvase de un pantano a otro de n hm^3 de agua. Si no hay suficiente agua embalsada en el pantano de origen, o si el agua a trasvasar no cabe en el pantano de destino la operación se ignora.

```
void ConfHidrografica::trasvase(const Rio& r1, const Pantano& p1,
                                const Rio& r2, const Pantano& p2, int n){
  auto itR1 = rios.find(r1);
  auto itR2 = rios.find(r2);
  if (itR1 != rios.end() && itR2 != rios.end()) {
    auto itP1 = itR1->second.pantanos.find(p1);
    auto itP2 = itR2->second.pantanos.find(p2);
    if (itP1 != itR1->second.pantanos.end() && itP1 != itR1->second.pantanos.end()) {
      if (itP1->second.carga >= n && itP2->second.carga + n <= itP2->second.capacidad) {
        itP1->second.carga -= n; itP2->second.carga += n;
        itR1->second.carga -= n; itR2->second.carga += n;
      }
    }
```

```
    }
}
```

El coste de la operación es de nuevo constante.

2. Agencia de viajes

Nos piden implementar un sistema muy simplificado para gestionar las reservas de una agencia de viajes. Para ello, debemos definir e implementar un TAD `Agencia` con las siguientes operaciones:

- `constructora` (sin argumentos): crea una agencia vacía.
- `aloja(c,h)`: modifica el estado de la agencia alojando a un cliente `c` en el hotel `h`. Si `c` ya tenía antes otro alojamiento, éste queda cancelado (un cliente solo puede tener una reserva en la agencia). Si `h` no estaba dado de alta en el sistema, se le dará de alta.
- `desaloja(c)`: modifica el estado de la agencia desalojando a un cliente `c` del hotel que éste ocupase. Si `c` no tenía alojamiento, el estado de la agencia no se altera.
- `alojamiento(c)`: permite consultar el hotel donde se aloja un cliente `c`, siempre que éste tuviera alojamiento. En caso de no tener alojamiento se produce una excepción de tipo `std::domain_error` con el mensaje `Cliente no encontrado`.
- `listado_hoteles()`: obtiene una lista ordenada (por orden alfabético) de todos los hoteles dados de alta en la agencia.
- `huespedes(h)`: obtiene un listado (no necesariamente ordenado) de los clientes que se alojan en el hotel `h`. Si el hotel no está dado de alta en el sistema se produce una excepción de tipo `std::domain_error` con el mensaje `Hotel no encontrado`.

Se pide: elegir una **representación** adecuada para el TAD utilizando las librerías de la STL de manera que la implementación de las operaciones sea **eficiente**, **implementar** todas las operaciones e **indicar y justificar sus costes**.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea, con el nombre de la operación seguido de sus argumentos (separados por espacios). Los nombres de los hoteles y clientes serán cadenas de caracteres sin espacios. La palabra `FIN` en una línea indica el final de cada caso. A continuación se proporcionan unos casos de prueba de ejemplo:

```
aloja Pepe nh
aloja Juan nh
aloja Rosa nh
aloja Maria ibis
aloja Sonia ibis
aloja Carmen melia
aloja Juan melia
alojamiento Sonia
alojamiento Juan
alojamiento Daniel
listado_hoteles
huespedes nh
aloja Juan nh
desaloja Rosa
huespedes nh
FIN
aloja c h
desaloja c
alojamiento c
listado_hoteles
```

```
huespedes h
huespedes h2
FIN
```

Salida

Las únicas operaciones que producen salida son:

- `alojamiento`, que imprimirá la cadena `C tiene reserva en el hotel H`, siendo C y H el parámetro y la salida de la operación respectivamente;
- `listado_hoteles`, que imprimirá `Listado de hoteles:` seguido del listado de hoteles (separados por espacios); y
- `huespedes`, que imprimirá `Reservas del hotel H:` seguido del listado de huéspedes del hotel `H` (separados por espacios) que deberá imprimirse en orden (y al no venir ordenado deberá ordenarse fuera del TAD).

A parte de eso, si una operación produce excepción, entonces se escribirá una línea con el mensaje `ERROR:`, seguido del error que devuelve la operación, y no se escribirá nada más para esa operación. Cada caso termina con una línea con tres guiones (`---`).

La salida esperada para los casos de más arriba sería la siguiente:

```
Sonia tiene reserva en el hotel ibis
Juan tiene reserva en el hotel melia
ERROR: Cliente no encontrado
Listado de hoteles: ibis melia nh
Reservas del hotel nh: Pepe Rosa
Reservas del hotel nh: Juan Pepe
---
ERROR: Cliente no encontrado
Listado de hoteles: h
Reservas del hotel h:
ERROR: Hotel no encontrado
---
```

Representación

Se propone representar la agencia mediante una tabla de tipo `unordered_map` con clave la identificación de los *clientes* y valor la identificación de los hoteles. Esta tabla permite implementar las operaciones `aloja`, `desaloja` y `alojamiento` con coste constante (promedio).

Sin embargo, solo con esa tabla no dispondríamos de suficiente información para poder implementar la operación `listado_hoteles`. Para ello, podríamos mantener también un listado de los hoteles registrados en el sistema. Lo más adecuado sería usar un `set`, que nos permitiría obtener el listado ordenado en complejidad lineal en el nº de hoteles y además consultar si un hotel está ya o no registrado (para evitar repeticiones) en complejidad logarítmica. Por otro lado, la operación *huespedes* exigiría un recorrido de toda la tabla de clientes para obtener los clientes de un hotel dado. Podemos mejorar fácilmente el coste de esta operación si en lugar de almacenar los hoteles en un `set` lo hacemos en un `map`, en el que guardamos, asociado a cada hotel, un listado de los clientes con reserva en ese hotel. Para este listado, puesto que hay que andar buscando clientes para evitar repeticiones y eliminar, lo más adecuado será usar un `unordered_set` (un set implementado mediante una tabla hash).

El tipo *Cliente* representa la información de los clientes (para este ejercicio es suficiente con que esta información sea el nombre del cliente) y el tipo *Hotel* representa la información de los hoteles (es suficiente con el nombre del hotel). Así, la definición de la clase queda como se muestra a continuación:

```
using Hotel = string;
using Cliente = string;

class Agencia {

private:
  using InfoHotel = unordered_set<Cliente>;

  unordered_map<Cliente, Hotel> clientes;
  map<Hotel, InfoHotel> hoteles;

public:
  Agencia(){ };
  void aloja(const Cliente& c, const Hotel& h);
  void desaloja(const Cliente& c);
  const Hotel& alojamiento(const Cliente& c) const;
  list<Hotel> listado_hoteles() const ;
  vector<Cliente> huespedes(const Hotel& h) const;
};
```

Implementación de las operaciones

El constructor implícito es suficiente para inicializar las tablas de clientes y hoteles.

Operación `aloja`

```
void Agencia::aloja(const Cliente& c, const Hotel& h) {
  auto it = clientes.find(c);
  if (it != clientes.end()){
    hoteles[it->second].erase(c);
  }
  clientes[c] = h; // Inserta o actualiza si ya estaba
  hoteles[h].insert(c);
}
```

El coste de la operación es el siguiente:

- Los costes de las operaciones `find`, `insert` y `erase` sobre el `unordered_map` son constantes en promedio.
- El coste de la operación [] sobre el `map` es logarítmico en su tamaño.

El coste de la operación es por tanto $\mathcal{O}(log(H))$ siendo H el número de hoteles dados de alta.

Podemos eso sí optimizar el código en cuanto a evitar búsquedas redundantes gracias al par iterador-booleano que se obtiene como salida de la operación `insert` (en los maps de la STL) informando de si existía o no la clave en la tabla (y por tanto de si se ha insertado), y en caso de existir, el iterador apuntando al par ya existente.

```
void aloja(const Cliente& c, const Hotel& h) {
  auto [itC, insertado] = clientes.insert({c, h});
  if (!insertado){ // c ya estaba registrado
    // Quitamos c del hotel en el que estaba
    hoteles[itC->second].erase(c);
    itC->second = h; // Se actualiza el hotel del cliente c
  }
  hoteles[h].insert(c); // Ponemos c como cliente del hotel h
}
```

Operación `desaloja`

```
void Agencia::desaloja(const Cliente& c) {
  auto it = clientes.find(c);
  if (it != clientes.end()){
    hoteles[it->second].erase(c);
    clientes.erase(c);
  }
}
```

El coste de la operación se calcula igual que el coste de la operación `aloja` y sería también $\mathcal{O}(log(H))$ siendo H el número de hoteles dados de alta.

Operación `alojamiento`

```
const Hotel& Agencia::alojamiento(const Cliente& c) const {
  try {
    return clientes.at(c);
  } catch (std::out_of_range& e){
    throw std::domain_error("Cliente no encontrado");
  }
}
```

El coste de la operación es el coste de consultar un cliente en la tabla de clientes (operación `at`) que sería de nuevo constante (promedio).

Operación `listado_hoteles`

Se recorre el con el iterador el map de hoteles produciendo un recorrido ordenado que vamos almacenando en una lista.

```
list<Hotel> Agencia::listado_hoteles() const {
  list<Hotel> l;
  for (auto it = hoteles.cbegin(); it != hoteles.cend(); ++it)
    l.push_back(it->first);
  return l;
}
```

El coste de la operación `++` del iterador del map es logarítimo en H, y el bucle se ejecuta H veces. Esto podría hacernos pensar que el coste es $\mathcal{O}(H * log(H))$. Sin embargo, si analizamos el comportamiento del operador `++` a lo largo de todo el recorrido del árbol se puede demostrar que nunca se pasa por un nodo más de dos veces, lo que garantiza que el coste es en realidad $\mathcal{O}(H)$.

Operación `huespedes`

```
vector<Cliente> Agencia::huespedes(const Hotel& h) const {
  auto itH = hoteles.find(h);
  if (itH == hoteles.end()) throw std::domain_error("Hotel no encontrado");
  else {
    vector<Cliente> l(itH->second.size());
    for (Cliente const& c : itH->second)
      l.push_back(c);
    return l;
  }
}
```

En este caso usamos un `vector` para la salida con el objetivo de facilitar su ordenación desde fuera antes de mostrarse por pantalla (ya que la salida esperada indica que el listado debe ir en orden). El coste de la operación es el máximo entre el coste de consultar un hotel en el map de

hoteles y el coste de recorrer el map de los clientes del hotel. Más formalmente, el coste sería $\mathcal{O}(max(log(H), MaxC))$ siendo H el nº de hoteles registrados y $MaxC$ el máximo número de clientes en un hotel.

3. eReader

Se desea diseñar una aplicación para gestionar los libros guardados en un eReader. Para ello debemos implementar un TAD con las siguientes operaciones:

- `crear` (constructor sin argumentos): crea un eReader sin ningún libro.
- `poner_libro(x,n)`: Añade un libro x al eReader. n representa el número de páginas del libro (cualquier número positivo). Si el libro ya existe la acción no tiene efecto.
- `abrir(x)`: El usuario abre un libro x para leerlo. Si el libro x no está en el eReader se produce un error con el mensaje `Libro no encontrado`. Si el libro ya había sido abierto anteriormente se considerará este libro como el último libro abierto.
- `avanzar_pag()`: Pasa una página del último libro que se ha abierto. La página posterior a la última es la primera. Si no existe ningún libro abierto se produce un error con el mensaje `Ningun libro abierto`.
- `abierto()`: Devuelve el último libro que se ha abierto. Si no se encuentra ningún libro abierto se produce un error con el mensaje `Ningun libro abierto`.
- `pag_libro(x)`: devuelve la página, del libro x, en la que se quedó leyendo el usuario. Se considera que todos los libros empiezan en la página 1, y ese será el resultado en caso de no haberse abierto nunca el libro. Si el libro no está dado de alta se produce un error con el mensaje `Libro no encontrado`.
- `elim_libro(x)`: Elimina el libro x del eReader. Si el libro no existe la acción no tiene efecto. Si el libro es el último abierto se elimina y queda como último abierto el que se abrió con anterioridad.
- `recientes(n)`: Obtiene una lista con los n últimos libros que fueron abiertos, ordenada según el orden en que se abrieron los libros, del más reciente al más antiguo. Si el número de libros que fueron abiertos es menor que el solicitado, la lista contendrá todos ellos. Si un libro se ha abierto varias veces solo aparecerá en la posición más reciente.

Se pide: elegir una **representación** adecuada para el TAD utilizando las librerías de la STL de manera que la implementación de las operaciones sea **eficiente**, **implementar** todas las operaciones e **indicar y justificar sus costes**.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD. Los errores se representan mediante excepciones del tipo `std::domain_error` con el mensaje correspondiente.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea, con el nombre de la operación seguido de sus argumentos (separados por espacios). Los libros serán cadenas de caracteres sin espacios. La palabra `FIN` en una línea indica el final de cada caso. A continuación se proporcionan unos casos de prueba de ejemplo:

```
poner_libro a 2
poner_libro b 5
poner_libro c 3
poner_libro d 10
abierto
abrir c
abrir b
abrir a
avanzar_pag
avanzar_pag
```

```
pag_libro a
pag_libro d
abrir b
recientes 5
elim_libro b
recientes 2
FIN
poner_libro lib 1
abrir a
avanzar_pag
abierto
pag_libro a
elim_libro a
recientes 5
FIN
```

Salida

Las únicas operaciones que producen salida son:

- `abierto`, que imprimirá la cadena `Leyendo X`, siendo X la salida de la operación;
- `pag_libro`, que imprimirá la cadena `Libro X por pagina N`, siendo X y N el parámetro y la salida de la operación respectivamente; y
- `recientes`, que imprimirá `Ultimos libros abiertos:` seguido de la lista de los últimos libros abiertos separados por espacio.

A parte de eso, si una operación produce excepción, entonces se escribirá una línea con el mensaje `ERROR:` , seguido del error que devuelve la operación, y no se escribirá nada más para esa operación. Cada caso termina con una línea con tres guiones (`---`).

La salida esperada para los casos de más arriba sería la siguiente:

```
ERROR: Ningun libro abierto
Libro a por pagina 1
Libro d por pagina 1
Ultimos libros abiertos: b a c
Ultimos libros abiertos: a c
---
ERROR: Libro no encontrado
ERROR: Ningun libro abierto
ERROR: Ningun libro abierto
ERROR: Libro no encontrado
Ultimos libros abiertos:
---
```

Representación - 1ª versión

Se propone representar el eReader mediante una tabla con clave la identificación de los libros y valor un struct con la información necesaria del libro (nº de páginas, la página en que está abierto y un booleano que indique si está abierto). Con eso tendríamos suficiente información para poder implementar todas las operaciones excepto `recientes` y `elim_libro` (pues no se sabría cuál fue el libro abierto con anterioridad si el libro a eliminar es el abierto actualmente).

Para subsanarlo se añade a la representación una lista con los libros que se han abierto en el orden en que se abren. Si un libro ya abierto se vuelve a abrir se coloca en primer lugar de esta lista y se elimina su anterior aparición.

La definición de la clase es la siguiente:

```
using Libro = string;

class EReader {

private:
  struct InfoLibro {
    int numPags;
    int pagActual;
    bool abierto;
  };

  unordered_map<Libro,InfoLibro> libros;
  list<Libro> secAbiertos;

public:
  EReader() { };
  void poner_libro(const Libro& x, int n);
  void abrir(const Libro& x);
  void avanzar_pag();
  const Libro& abierto() const;
  int pag_libro(const Libro& x) const;
  void elim_libro(const Libro& x);
  list<Libro> recientes(int n) const;
};
```

Implementación de las operaciones (para 1ª versión)

Constructora

El constructor por defecto valdría. Proporcionamos un constructor vacío para hacer explícito que no es necesario hacer nada (más que inicializar los atributos con sus valores por defecto).

Operación `poner_libro`

```
void EReader::poner_libro(const Libro& x, int n){
  // si el libro ya esta no se hace nada
  InfoLibro infoL = {n, 1, false};
  libros.insert({x, infoL}); // Se inserta en la tabla
}
```

El coste de la operación viene dado por el coste de la operación `insert` que es constante (promedio).

Operación `abrir`

```
void EReader::abrir(const Libro& x) {
  auto itL = libros.find(x);
  if (itL == libros.end()) throw std::domain_error("Libro no encontrado");
  else { // El libro esta en la tabla. Consulta su informacion
    InfoLibro& infoL = itL->second;
    if (infoL.abierto) {
      // Si esta abierto lo elimina de su posicion
      secAbiertos.remove(x); // Ojo, coste lineal
    }
    // Añade el libro al principio de la lista y lo pone como abierto
    infoL.abierto = true;
    secAbiertos.push_front(x);
  }
```

```
}
```

Es importante hacer notar que la operación `remove` del TAD `list` implementa una búsqueda lineal para buscar el elemento, y en caso de encontrarlo borrarlo. Su coste es por tanto lineal en el nº de elementos de la lista. El resto de operaciones invocadas es constante (promedio). Por tanto el coste sería $\mathcal{O}(A)$ siendo A el nº de libros que se han abierto (que evidentemente podría llegar a ser el total de libros en el eReader).

Operación `avanzar_pag`

```
void EReader::avanzar_pag() {
  // Si no hay ningun libro abierto produce error
  if (secAbiertos.empty()) throw std::domain_error("Ningun libro abierto");
  else { // Si hay libros abiertos obtiene el primero
    Libro l = secAbiertos.front();
    // incrementa la pagina en la informacion de la tabla
    InfoLibro& infoL = libros[l];
    infoL.pagActual++;
    // Si la pagina es mayor que la ultima vuelve a la primera
    if (infoL.pagActual > infoL.numPags) infoL.pagActual = 1;
  }
}
```

El coste de la operación es constante (promedio).

Operación `abierto`

```
const Libro& EReader::abierto() const {
  if (secAbiertos.empty()) throw std::domain_error("Ningun libro abierto");
  else return secAbiertos.front();
}
```

El coste de la operación es de nuevo constante (promedio).

Operación `pag_libro`

```
int EReader::pag_libro(const Libro& x) const {
  auto itL = libros.find(x);
  if (itL == libros.end()) throw std::domain_error("Libro no encontrado");
  else return itL->second.pagActual;
}
```

El coste de la operación es de nuevo constante.

Operación `elim_libro`

```
void EReader::elim_libro(const Libro& x) {
  auto itL = libros.find(x);
  if (itL != libros.end()) {
    InfoLibro infoL = itL->second;
    // Si el libro esta abierto lo elimina de la lista
    if (infoL.abierto) {
      secAbiertos.remove(x); // Ojo que es lineal
      libros.erase(x);
    }
  }
}
```

Como pasaba en la operación `abrir`, la complejidad queda dominada por la operación `remove` del TAD `list`, que es lineal en el nº de elementos de la lista. El resto de operaciones invocadas es constante (promedio). Por tanto el coste sería $\mathcal{O}(A)$ siendo A el nº de libros que se han abierto (que podría llegar a ser el total de libros en el eReader).

Operación `recientes`

```
list<Libro> recientes(int n) const {
  list<Libro> l;
  for (auto it = secAbiertos.begin(); it != secAbiertos.end() && n > 0; n--, ++it)
    l.push_back(*it);
  return l;
}
```

El coste de la operación es lineal respecto al valor del parámetro de entrada, ya que este es el número de veces que como máximo se ejecuta el bucle.

Representación - Versión más eficiente

Podemos evitar las búsquedas lineales (provocadas por las llamadas a `list::remove`) si, asociado a cada libro, nos guardamos un iterador a la aparición de ese libro en la secuencia de libros abiertos (que sería el iterador `end` si el libro no se ha abierto). De esa forma, cuando haya que borrar la aparición del libro en la lista podremos hacer uso de la operación `list::erase` usando el iterador con complejidad constante. Con esto conseguiremos costes constantes en las operaciones `abrir` y `elim_libro` (que antes eran lineales).

La nueva representación (parte privada de la clase) quedaría así:

```
class EReader {

private:

  struct InfoLibro {
    int numPags;
    int pagActual;
    list<Libro>::iterator posSecAbiertos;
  };

  unordered_map<Libro,InfoLibro> libros;
  list<Libro> secAbiertos;

public:
  ...
};
```

Implementación de las operaciones (para versión eficiente)

Mostramos a continuación la implementación de las operaciones `poner_libro`, `abrir` y `elim_libro`. El resto de las operaciones no cambian.

Operación `poner_libro`

```
void EReader::poner_libro(const Libro& x, int n){
  // si el libro ya esta no se hace nada
  // Se crea la informacion del libro
  InfoLibro infoL = {n,1,secAbiertos.end()}; // Iterador end indica no abierto
  libros.insert({x, infoL}); // Se inserta en la tabla
}
```

El único cambio respecto a la versión anterior es la inicialización del iterador a la lista de abiertos, que se inicializa a `secAbiertos.end()` para indicar que el libro no se encuentra en la lista. El coste de la operación seguiría siendo constante.

Operación `abrir`

```
void EReader::abrir(const Libro& x) {
  auto itL = libros.find(x);
  if (itL == libros.end()) throw std::domain_error("Libro no encontrado");
  else { // El libro esta en la tabla. Consulta su informacion
    InfoLibro& infoL = itL->second;
    if (infoL.posSecAbiertos != secAbiertos.end()) {
      // Si esta abierto lo elimina de su posicion
      secAbiertos.erase(infoL.posSecAbiertos); // Coste constante!
    }
    // Añade el libro al principio de la lista y guarda el iterador
    infoL.posSecAbiertos = secAbiertos.insert(secAbiertos.begin(), x);
  }
}
```

La llamada a `list::remove` para borrar, en su caso, la aparición previa del libro en la secuencia de libros abiertos, que internamente hacía una búsqueda lineal, se ha sustituido por la llamada al método `erase`, cuyo coste es constante gracias al iterador. El coste de la operación queda por tanto constante.

Operación `elim_libro`

```
void EReader::elim_libro(const Libro& x) {
  auto itL = libros.find(x);
  if (itL != libros.end()) {
    InfoLibro infoL = itL->second;
    // Si el libro esta abierto lo elimina de la lista
    if (infoL.posSecAbiertos != secAbiertos.end()) {
      secAbiertos.erase(infoL.posSecAbiertos); // Coste constante
      libros.erase(x); // Mejor libros.erase(itL); (solo versión STL)
    }
  }
}
```

De nuevo, la búsqueda lineal que hacía la llamada a `list::remove` para encontrar y eliminar el libro de la lista ya no es necesaria y se sustituye por la llamada a `list::erase`. El coste es por tanto constante.