

# Aplicaciones de tipos abstractos de datos<sup>1</sup>

---

**RESUMEN:** En este tema se estudia la resolución de problemas mediante el uso de distintos tipos de datos. Para ello se proponen varios ejercicios resueltos así como problemas a realizar por el alumno.

## 1. Problemas resueltos:

### 1.1. Confederación hidrográfica.

(Obtenido del examen final de Junio de 2010. Titulación: ITIS. Asignatura: EDI. Grupos A y B)

Una confederación hidrográfica gestiona el agua de 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.

Se pide diseñar un TAD que permita a la confederación hidrográfica gestionar la cuenca. En particular, el comportamiento de las operaciones que debe ofrecer debe ser el siguiente:

- `crea`, crea una confederación hidrográfica vacía.
- `an_río(r)` añade el río `r` a la confederación hidrográfica. 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`.

---

<sup>1</sup>Isabel Pita Andreu es la autora principal de este tema.

- `reserva_cuenca(r)` devuelve la cantidad de agua embalsada en la cuenca del río `r`.
- `transvase(r1, p1, r2, p2, n)` transvasa  $n \text{ } Hm^3$  de agua del pantano `p1` del río `r1` al pantano `p2` del río `r2`, en la confederación. Si `n` es mayor que la cantidad de agua embalsada en `p1` o no cabe en `p2` la operación se ignora.

La implementación debe ser parametrizada por la información que represente a los ríos y a los pantanos.

Todas las operaciones son totales.

### Solución

#### Representación:

Se propone utilizar una tabla con clave la información asociada a los ríos y valor todos los pantanos existentes en el río con su capacidad y litros embalsados. Los pantanos del río se almacenan en otra tabla, con clave la identificación del pantano y valor su capacidad y litros embalsados. Se utilizan tablas porque las operaciones que se van a realizar tanto sobre ríos como sobre pantanos son consultas e inserciones.

La implementación de la clase es la siguiente. El parámetro  $X$  representa la información de los ríos (para este ejercicio es suficiente con que esta información sea el nombre del río) y el parámetro  $Y$  representa la información de los pantanos (es suficiente con el nombre del pantano).

```
template <class X , class Y>
class conf_hidrografica {
public:
    conf_hidrografica();
    ~conf_hidrografica();
    void an_rio (const X& r);
    void an_pantano (const X& r, const Y& p, int n1, int n2);
    void embalsar (const X& r, const Y& p, int n);
    int embalsado_pantano(const X& r, const Y& p);
    int reserva_cuenca(const X& r);
    void transvase(const X& r1, const Y& p1, const X& r2,
                  const Y& p2,int n);
private:
    struct info_pantano {
        int capacidad;
        int litros_embalsados;
    };
    Tabla<X,Tabla<Y,info_pantano>*>*> t_rios;
};
```

#### Implementación de las operaciones.

##### Operación *conf-hidrografica*, constructor.

El constructor construye una confederación sin ningún río. La tabla está vacía. Nótese que el atributo de la clase es un puntero a la tabla.

```
template <class X , class Y>
```

```

conf_hidrografica<X,Y>::conf_hidrografica() {
    t_rios = new Tabla<X,Tabla<Y,info_pantano>*>();
}

```

El coste es constante, ya que las tablas que se han estudiado, se crean con un tamaño inicial independiente del número de elementos de la tabla.

#### **Operación *conf-hidrografica*, destructor.**

```

template <class X , class Y>
conf_hidrografica<X,Y>::~~conf_hidrografica() {
    for (typename Tabla<X,Tabla<Y,info_pantano>*>::Iterador
        it = t_rios->principio();
        it != t_rios->final();
        it.avanza()) {
        delete it.valor();
    }

    delete t_rios;
}

```

El coste es constante.

#### **Operación *an\_río*.**

Al añadir un río, si está en la cuenca no se debe modificar su valor, por lo tanto no se debe llamar a la operación inserta ya que esta modifica el valor. La tabla correspondiente al valor del río se crea vacía.

```

template <class X , class Y>
void conf_hidrografica<X,Y>::an_río (const X& r){
    // Si el río esta, la operacion de insertar modifica su valor
    // por eso no se debe llamar a la operacion
    if (!t_rios->esta(r)){
        Tabla<Y,info_pantano>* t_pan = new Tabla<Y,info_pantano>();
        t_rios->inserta(r,t_pan);
    }
}

```

Coste de la operación implementada:

- El coste de consultar una tabla es constante (operación *esta*).
- El coste de crear una tabla vacía es constante.
- El coste de insertar un elemento en la tabla es constante (operación *inserta*).

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

#### **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. Si el río no está en la confederación, o si el pantano ya está en el río la operación no tiene ningún efecto.

```
template <class X , class Y>
void conf_hidrografica<X,Y>::an_pantano (const X& r, const Y& p,
                                         int n1, int n2){
    if (t_rios->esta(r)) {
        Tabla<Y,info_pantano>* t_pan =t_rios-> consulta(r);
        if (!t_pan->esta(p)) {
            // crea la informacion del pantano
            info_pantano i;
            i.capacidad = n1;
            if (n2 < n1) i.litros_embalsados = n2;
            else i.litros_embalsados = n1;
            // añade informacion al rio
            t_pan->inserta(p,i);
            // Al ser t_pan un puntero a la tabla queda modificada
            //t_rios->inserta(r,t_pan);
        }
    }
}
```

Coste de la operación implementada:

- El coste de consultar una tabla es constante (operaciones *esta* y *consulta*).
- El coste de insertar un elemento en una tabla es constante (operación *inserta*).
- El resto de operaciones tiene coste constante.

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

### Operación *embalsar*.

La operación de embalsar añade  $n \text{ } Hm^3$  al agua embalsada en un pantano. Si se supera la capacidad, el pantano se considera lleno.

```
template <class X , class Y>
void conf_hidrografica<X,Y>::embalsar (const X& r, const Y& p,
                                         int n){
    if (t_rios->esta(r)) {
        Tabla<Y,info_pantano>* t_pan =t_rios-> consulta(r);
        if (t_pan->esta(p)) {
            // Añade al pantano
            info_pantano i = t_pan->consulta(p);
            i.litros_embalsados += n;
            if (i.litros_embalsados > i.capacidad)
                i.litros_embalsados = i.capacidad;
            // añade informacion al rio
            t_pan->inserta(p,i);
            // Al ser t_pan un puntero a la tabla queda modificada
```

```

        //t_rios->inserta(r,t_pan);
    }
}

```

Coste de la operación implementada es igual que el coste de la operación *an\_pantano*.

### **Operación *embalsado\_pantano*.**

La operación consulta los  $Hm^3$  embalsados en un pantano.

```

template <class X , class Y>
int conf_hidrografica<X,Y>::embalsado_pantano(const X& r,
                                              const Y& p) {
    int n = -1;
    if (t_rios->esta(r)) {
        Tabla<Y,info_pantano>* t_pan =t_rios-> consulta(r);
        if (t_pan->esta(p)) {
            info_pantano i = t_pan->consulta(p);
            n = i.litros_embalsados;
        }
    }
    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.

### **Operación *reserva-cuenca*.**

La operación *reserva-cuenca* obtiene la suma de los  $Hm^3$  embalsados en todos los pantanos de la cuenca.

Para implementarla se utiliza un iterador sobre la tabla que recorre todos los pantanos del río.

```

template <class X , class Y>
int conf_hidrografica<X,Y>::reserva_cuenca(const X& r) {
    int n = 0;
    if (t_rios->esta(r)) {
        Tabla<Y,info_pantano>* t_pan =t_rios->consulta(r);
        Tabla<Y,info_pantano>::Iterador it = t_pan->principio();
        while (it != t_pan->final()) {
            info_pantano i = it.valor();
            n += i.litros_embalsados;
            it.avanza();
        }
    }
    return n;
}

```

Coste de la operación implementada:

- El coste de consultar una tabla es constante (operaciones *esta* y *consulta*).

- El coste de crear un iterador es constante.
- El coste del bucle es lineal respecto al número de elementos de la tabla que almacena los pantanos de un río, ya que las operaciones que se realizan en cada vuelta del bucle tienen coste constante y el número de veces que se ejecuta el bucle coincide con el número de elementos de la tabla.

Por lo tanto el coste de la operación es lineal respecto al número de pantanos en un río.

### Operación *transvase*.

La operación realiza el transvase de un pantano a otro de  $n \text{ } Hm^3$  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.

```
template <class X , class Y>
void conf_hidrografica<X,Y>::
    transvase(const X& r1, const Y& p1,
              const X& r2, const Y& p2, int n){
    if (t_rios->esta(r1) && t_rios->esta(r2)) {
        Tabla<Y,info_pantano>* t_pan1 =t_rios->consulta(r1);
        Tabla<Y,info_pantano>* t_pan2 =t_rios->consulta(r2);
        if (t_pan1->esta(p1) && t_pan2->esta(p2)) {
            info_pantano i1 = t_pan1->consulta(p1);
            info_pantano i2 = t_pan2->consulta(p2);
            if (i1.litros_embalsados >= n &&
                i2.litros_embalsados + n <= i2.capacidad) {
                i1.litros_embalsados -= n;
                i2.litros_embalsados += n;
                // añade informacion al rio
                t_pan1->inserta(p1,i1);
                t_pan2->inserta(p2,i2);
                // Al ser t_pan un puntero a la tabla queda modificada
                //t_rios->inserta(r1,t_pan1);
                //t_rios->inserta(r2,t_pan2);
            }
        }
    }
}
```

El coste de la operación es constante ya que las operaciones que se realizan son las de consultar e insertar en una tabla (coste constante).

## 1.2. Agencia de viajes.

(Obtenido del examen extraordinario Febrero 2010)

Se desea definir un tipo abstracto de datos *Agencia* para representar una agencia hotelera. El TAD estará parametrizado respecto a la información de los clientes y a la información de los hoteles. Se deben ofrecer las siguientes operaciones:

- *crea*: crea una agencia vacía.

- *aloja(c, h)*: modifica el estado de la agencia alojando a un cliente *c* en un hotel *h*. Si *c* ya tenía antes otro alojamiento, éste queda cancelado. Si *h* no estaba dado de alta en el sistema, se le dará de alta.
- *desaloja(c)*: modifica el estado de una 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 produce un error.
- *listado – hoteles()*: obtiene una lista de todos los hoteles que están dados de alta en la agencia, ordenados según el orden definido en el tipo del parámetro.
- *huespedes(h)*: permite obtener el conjunto de clientes que se alojan en un hotel dado. Dicho conjunto será vacío si no hay clientes en el hotel.

Se pide:

- a) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas.
- b) Implementar todas las operaciones indicando el coste de cada una de ellas. La operación *huespedes* debe producir una lista de clientes en lugar de un conjunto,

### Solución.

#### Representación:

Se propone representar la agencia mediante una tabla con clave la identificación de los *clientes* y valor la identificación de los hoteles. Esta tabla permite obtener un coste constante para la operación *alojamiento*.

Sin embargo, la operación *huespedes* exige el recorrido de toda la tabla para obtener los clientes de un hotel dado. Para mejorar el coste de esta operación añadimos a la representación un árbol binario de búsqueda con clave la identificación de los hoteles. El valor asociado será una lista con todos los clientes del hotel. Con esta nueva estructura el coste de la operación *huespedes* es lineal respecto al número de clientes del hotel. Si el listado de los huespedes se quisiese ordenado habría que utilizar un árbol binario de búsqueda para almacenar los clientes en lugar de una lista.

Se selecciona un árbol binario de búsqueda para almacenar la información referente a los hoteles, para obtener la lista ordenada de los hoteles dados de alta en la agencia en tiempo lineal respecto al número de hoteles. Si se utilizase una tabla como ocurre con la información de los clientes, se podría obtener la lista de los hoteles en tiempo lineal, pero después habría que ordenarla con lo que la complejidad de la operación sería del orden de  $\mathcal{O}(n \log n)$

La definición de la clase queda como:

```
template <class C , class H>
class agencia {
public:
    agencia(){};
    void aloja(const C& c, const H& h);
    void desaloja(const C& c);
    H alojamiento(const C& c);
```

```

    Lista<C> hoteles();
    Lista<C> huespedes(const H& h);
private:
    Tabla<C,H> clientes;
    Arbus<H,Lista<C>> hoteles;
};

```

### Implementación de las operaciones:

El constructor es vacío, ya que los atributos de la clase se declararon estáticos.

### Operación *aloja*

```

template <class C , class H>
void agencia<C,H>::aloja(const C& c,const H& h){
    // El cliente ya tiene alojamiento en un hotel y desea cambiarlo
    if (clientes.esta(c)) {
        H hant = clientes.consulta(c);
        // Elimina al cliente del antiguo hotel.
        // El hotel siempre existe
        Lista<C> lant = hoteles.consulta(hant);
        Lista<C>::Iterador it = lant.principio();
        // El cliente siempre esta en el hotel
        while (it.elem() != c) it.avanza();
        lant.borra(it);
        hoteles.inserta(hant,lant);
        // Añade al cliente en el nuevo hotel.
        // Si el hotel no existe lo da de alta
        Lista<C> l;
        if (hoteles.esta(h)) l = hoteles.consulta(h);
        l.Cons(c);
        hoteles.inserta(h,l);
        // Cambia el hotel en la tabla de clientes
        clientes.inserta(c,h);
    }
    // El cliente no tiene alojamiento en ningun hotel
    else {
        // Añade el cliente a la tabla de clientes
        clientes.inserta(c,h);
        // Añade el cliente al hotel
        Lista<C> l;
        if (hoteles.esta(h)) l = hoteles.consulta(h);
        l.Cons(c);
        hoteles.inserta(h,l);
    }
}

```

no hace falta poner final

esta parte es igual

El coste de la operación es el siguiente:

- El coste de consultar un cliente en la tabla es constante (operaciones *esta* y *consulta*).



poner sup. árbol  
balanceado

- El coste de consultar un hotel en el árbol binario de búsqueda es, en el caso peor, lineal respecto al número de nodos del árbol. Si suponemos el árbol equilibrado el coste es logarítmico respecto al número de nodos (operación *consulta*).
- El coste de recorrer la lista de clientes es, en el caso peor, lineal respecto al número de clientes del hotel.
- El coste de borrar en la lista de clientes es constante (operación *borra*).
- El coste de insertar un hotel en el árbol binario de búsqueda si suponemos el árbol equilibrado es logarítmico respecto al número de hoteles (operación *inserta*).
- El coste de insertar un elemento en la primera posición de la lista de clientes es constante (operación *Cons*).
- El coste de insertar un cliente en la tabla de clientes es constante (operación *inserta*).

El coste de la operación, por lo tanto, es el máximo entre el logaritmo del número de hoteles dados de alta y el máximo de clientes de los hoteles.

Se podría mejorar este coste utilizando un árbol binario de búsqueda para almacenar los clientes de cada hotel. De esta forma se conseguiría un coste logarítmico al eliminar un cliente de un hotel. El coste de la operación sería el máximo entre el logaritmo del número de hoteles dados de alta y el logaritmo del número de clientes de un hotel. Se considera que el número de clientes dados de alta por una agencia en un hotel no será nunca muy elevado, se podría considerar como una constante y por lo tanto la mejora del coste sería inapreciable.

### Operación *desaloja*

```
template <class C , class H>
void agencia<C,H>::desaloja(const C& c){
    // Si el cliente no esta en la agencia
    // la operacion no tiene efecto
    if (clientes.esta(c)) {
        H h = clientes.consulta(c);
        // Elimina al cliente del hotel
        Lista<C> l = hoteles.consulta(h);
        Lista<C>::Iterador it = l.principio();
        // El cliente esta en el hotel
        while (it.elem() != c) it.avanza();
        l.borra(it);
        hoteles.inserta(h,l);
        // Elimina al cliente de la tabla de clientes
        clientes.borra(c);
    }
}
```

El coste de la operación se calcula igual que el coste de la operación *aloja*. El coste es el máximo entre el logaritmo del número de hoteles dados de alta y el máximo de clientes de los hoteles.

### Operación *alojamiento*

```
template <class C , class H>
H agencia<C,H>::alojamiento(const C& c){
    if (clientes.esta(c)) return clientes.consulta(c);
    else throw ENoExisteCliente();
}
```

El coste de la operación es el coste de consultar un cliente en la tabla (operaciones *esta* y *consulta*). Por lo tanto el coste es constante.

### Operación *listado-hoteles*

Se recorre el árbol de búsqueda utilizando el iterador, ya que el recorrido definido para este es en inorden

```
template <class C , class H>
Lista<C> agencia<C,H>::listado_hoteles(){
    Lista<H> l;
    Arbus<H,Lista<C>>::Iterador it = hoteles.principio();
    while (it != hoteles.final()) {
        l.Cons(it.clave());
        it.avanza();
    }
    return l;
}
```

El coste de la operación es el coste de realizar el recorrido en inorden de un árbol, por lo tanto es lineal respecto al número de nodos del árbol, que en este caso es el número de hoteles de la agencia.

### Operación *huespedes*

```
template <class C , class H>
Lista<C> agencia<C,H>::huespedes(const H& h){
    Lista<C> l;
    if (hoteles.esta(h)) l = hoteles.consulta(h);
    return l;
}
```

El coste de la operación es el máximo entre el coste de consultar un hotel en el árbol binario de búsqueda de los hoteles y la copia de la lista de clientes que se devuelve. Por lo tanto, es el máximo entre el logaritmo del número de hoteles (suponemos el árbol equilibrado) y el máximo número de clientes en un hotel.

## 1.3. E-reader.

(Obtenido del examen final Junio 2011. Titulación: II. Asignatura: EDI. Grupos A y C)

Se desea diseñar una aplicación para gestionar los libros guardados en un e-reader. La implementación estará parametrizada respecto a la información asociada a un libro.

El comportamiento de las operaciones es el siguiente:

1. *crear*: crea un e-reader sin ningún libro.

2. *poner-libro*( $x, n$ ): Añade un libro  $x$  al e-reader.  $n$  representa el número de páginas del libro, puede ser cualquier número positivo. Si el libro ya existe la acción no tiene efecto.
3. *abrir*( $x$ ): El usuario abre un libro  $x$  para leerlo. Si el libro  $x$  no está en el e-reader se produce un error. Si el libro ya había sido abierto anteriormente se considerará este libro como el último libro abierto.
4. *avanzar-pag*( $x$ ): 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.
5. *abierto*( $x$ ): Devuelve el último libro que se ha abierto. Si no se encuentra ningún libro abierto se produce un error.
6. *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. Si el libro no está dado de alta se produce un error.
7. *elim-libro*( $x$ ): Elimina el libro  $x$  del e-reader. 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.
8. *esta-libro*( $x$ ): Consulta si el libro  $x$  está en el e-reader.
9. *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.
10. *num-libros*( $x$ ): Consulta el número de libros que existen en el e-reader.

Se pide:

- a) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas. No se permite utilizar vectores ni memoria dinámica (listas enlazadas). Implementar todas las operaciones indicando el coste de cada una de ellas. El tipo de retorno de la operación *recientes* debe ser un tipo lineal, seleccionar uno adecuado y justificarlo.
- b) Modificar la representación anterior utilizando memoria dinámica (listas enlazadas) de forma que el coste de la operación *abrir* sea constante y el coste de *recientes* sea lineal respecto al parámetro de entrada (número de libros que se quieren obtener). El coste de las demás operaciones no debe ser mayor que con la representación del ejercicio anterior, ni debe aumentar de forma significativa el gasto en memoria. Implementar todas las operaciones indicando su coste.

### Solución.

#### Primera representación:

Se propone representar el e-reader mediante una tabla con clave la identificación de los libros y valor la información referente al total de páginas del libro, la página en que está abierto y una variable booleana que indique si está abierto.

Para poder implementar la operación *recientes* 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.

Para conseguir coste constante en la operación *num-libros* se añade una variable entera, *cantidad*, con el número de libros del e-reader.

La definición de la clase es la siguiente:

```
template <class L>
class e_reader {
public:
    e_reader();
    void poner_libro(const L& x,int n);
    void abrir(const L& x);
    void avanzar_pag();
    const L& abierto();
    int pag_libro(const L& x);
    void elim_libro(const L& x);
    bool esta_libro(const L& x);
    Lista<L> recientes(int n);
    int num_libros();
private:
    struct info_libro {
        int total_paginas;
        int pag_actual;
        bool abierto;
    };
    Tabla<L,info_libro> t_libros;
    Lista<L> sec_abiertos;
    int cantidad;
};
```

### Implementación de las operaciones:

#### Operación *e-reader*, constructora

Los atributos *t-libros* y *sec-abiertos* son estáticos. Solo es necesario inicializar la variable entera.

```
template <class L>
e_reader<L>::e_reader() {
    cantidad = 0;
}
```

El coste de la operación es constante.

#### Operación *poner-libro*.

```
template <class L>
void e_reader<L>::poner_libro(const L& x,int n){
    // si el libro ya esta o el numero de paginas
    // es negativo no se hace nada
```

```

    if (!t_libros.esta(x) && n > 0) {
        // Se crea la informacion del libro
        info_libro i;
        i.total_paginas = n;
        i.pag_actual = 1;
        i.abierto = false;
        //Se inserta en la tabla
        t_libros.inserta(x,i);
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}

```

El coste de la operación es el siguiente:

- El coste de consultar una tabla es constante (operación *esta*).
- El coste de realizar asignaciones es constante.
- El coste de insertar en una tabla es constante (operación *inserta*).

El coste de la operación es la suma de los costes de las instrucciones, por lo tanto es constante.

### Operación *abrir*.

```

template <class L>
void e_reader<L>::abrir(const L& x){
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        // El libro esta en la tabla consulta su informacion
        info_libro i = t_libros.consulta(x);
        if (i.abierto) {
            // Si esta abierto lo borra de su posicion
            Lista<L>::Iterador it = sec_abiertos.principio();
            while (it.elem() != x ) {it.avanza();}
            sec_abiertos.borra(it);
        }
        else {
            // Si no esta abierto cambia su estado a abierto
            i.abierto = true;
            t_libros.inserta(x,i);
        }
        // inserta el libro al comienzo de la secuencia
        sec_abiertos.Cons(x);
    }
}

```

El coste de la operación es el siguiente:

- El coste de consultar un libro en la tabla es constante (operaciones *esta* y *consulta*).

- El coste del bucle que recorre la lista de libros abiertos es, en el caso peor, lineal respecto a los libros abiertos del e-reader, que pueden ser todos los libros del e-reader.
- El coste de insertar un elemento en la tabla es constante (operación *inserta*).
- El coste de añadir un elemento al principio de una lista es constante (operación *Cons*).

Por lo tanto, el coste de la operación es lineal respecto al número de libros del e-reader.

#### Operación *avanzar-pag*.

```
template <class L>
void e_reader<L>::avanzar_pag() {
    // Si no hay ningun libro abierto produce error
    if (sec_abiertos.esVacia()) throw ENoabiertos();
    else {
        // Si hay libros abiertos. Obtiene el primero
        L l = sec_abiertos.primer();
        // incrementa la pagina en la informacion de la tabla
        info_libro i = t_libros.consulta(l);
        i.pag_actual++;
        // Si la pagina es mayor que la ultima vuelve a la primera
        if (i.pag_actual > i.total_paginas) i.pag_actual = 1;
        t_libros.inserta(l, i);
    }
}
```

El coste de la operación es constante ya que:

- El coste de consultar si una lista es vacía y el primero de una lista es constante (operaciones *esVacia* y *primero*).
- El coste de consultar e insertar en una tabla es constante (operaciones *consulta* y *inserta*).

#### Operación *abierto*.

```
template <class L>
const L& e_reader<L>::abierto() {
    if (sec_abiertos.esVacia()) throw ENoabiertos();
    else return sec_abiertos.primer();
}
```

El coste de la operación es constante, ya que el coste de consultar si una lista es vacía y el primero de una lista son constantes.

#### Operación *pag-libro*.

```
template <class L>
int e_reader<L>::pag_libro(const L& x) {
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
```

```

        info_libro i = t_libros.consulta(x);
        return i.pag_actual;
    }
}

```

El coste de la operación es constante, ya que el coste de consultar en una tabla (operaciones *esta* y *consulta*) es constante.

### Operación *elim-libro*.

```

template <class L>
void e_reader<L>::elim_libro(const L& x){
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        info_libro i = t_libros.consulta(x);
        t_libros.borra(x);
        cantidad--;
        // Si el libro esta abierto lo borra de la lista
        if (i.abierto) {
            Lista<L>::Iterador it = sec_abiertos.principio();
            while (it.elem() != x)
                it.avanza();
            sec_abiertos.borra(it);
        }
    }
}

```

El coste de la operación es el siguiente:

- El coste de consultar y borrar en una tabla es constante (operaciones *esta* y *borra*).
- El coste de recorrer la lista para eliminar el libro si esta abierto es, en el caso peor, lineal respecto al número de libros del e-reader.
- El coste de borrar el elemento indicado por el iterador es constante (operación *borra*).

El coste de la operación, por lo tanto, es lineal respecto al número de libros del e-reader.

### Operación *esta-libro*.

```

template <class L>
bool e_reader<L>::esta_libro(const L& x){
    return t_libros.esta(x);
}

```

El coste de consultar si un libro está en el e-reader es constante, ya que el coste de consultar en una tabla es constante.

### Operación *recientes*.

```

template <class L>
Lista<L> recientes(int n){

```

```

Lista<L> l = new Lista<L>();
int cont = 0;
Lista<L>::iterador it = sec_abiertos.principio();
while (it != sec_abiertos.final() && cont < n) {
    l.Cons(it.elem());
    it.avanza();
    cont++;
}
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 y el coste de todas las operaciones que se realizan en el bucle es constante: consultar el final de un iterador *final*, añadir un elemento al principio de una lista *Cons*, consultar el elemento apuntado por un iterador *elem*, y avanzar un iterador *avanza*.

### Operación *num-libros*.

```

template <class L>
int e_reader<L>::num_libros() {
    return cantidad;
}

```

El coste de la operación es constante, ya que el coste de devolver un valor de tipo entero es constante.

### Segunda representación (alternativa a la primera):

no es el b)

Se propone a continuación otra implementación del e-reader en que se mejora el coste de algunas operaciones a costa de empeorar el coste de otras. Dependiendo del uso que se vaya a hacer del e-reader será mas apropiada una implementación o la otra.

Se define una tabla con clave la información del libro y valor el número de páginas, la página por la que se va leyendo, y un contador que indica el orden de apertura de los diversos libros. Nótese que se ha cambiado la variable booleana *abierto* de la representación anterior por este contador. De esta forma no solo tenemos información de si el libro ha sido abierto sino tambien del orden en que fueron abiertos.

La secuencia de libros abiertos se sustituye por una variable que almacena el último libro abierto. Se añade un contador de los libros que se van abriendo, que sirve para actualizar el orden en que se abre un libro dentro de la información de los libros en la tabla. Por último se mantiene la variable que almacena la cantidad de libros del e-reader.

```

template <class L>
class e_reader {
public:
    // mismas operaciones que en el caso anterior
    ...
private:
    struct info_libro {
        int total_paginas;
        int pag_actual;
        int num_abierto; // 0 indica que no se ha abierto todavia
    };
};

```



```
};
    Tabla<L,info_libro> t_libros;
    L ultimo_abierto;
    int acumulador; // Para contar orden de apertura
    int cantidad; // numero total de libros del e-reader
};
```

**Implementación de las operaciones:**. Se muestran sólo las operaciones que se modifican respecto a la representación anterior.

#### **Operación *e-reader*, constructora**

En este caso es necesario inicializar también el acumulador. Se elige el valor uno como inicialización y se incrementará su valor después de utilizarlo.

```
template <class L>
e_reader<L>::e_reader(){
    cantidad = 0;
    acumulador = 1;
}
```

El coste es constante.

#### **Operación *poner-libro***

La única modificación es la inicialización de la variable num\_abierto a cero, en lugar de la variable booleana existente en la otra representación.

```
template <class L>
void e_reader<L>::poner_libro(const L& x,int n){
    // si el libro ya esta o el numero de paginas es negativo
    // no se hace nada
    if (!t_libros.esta(x) && n > 0) {
        // Se crea a informacion del libro
        info_libro i;
        i.total_paginas = n;
        i.pag_actual = 1;
        i.num_abierto = 0;
        //Se inserta en la tabla
        t_libros.inserta(x,i);
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

La operación consulta e inserta elementos en una tabla, además de hacer algunas asignaciones, sumas y comparaciones. Por lo tanto el coste de la operación es constante.

#### **Operación *abrir***

En este caso se modifica el valor de la variable ultimo\_abierto con el libro que se está abriendo en esta operación. Se modifica también el orden en que se abrió el libro usando el valor del acumulador. No es necesario diferenciar el caso en que el libro ya había

sido abierto anteriormente. Se incrementa el valor del acumulador para utilizarlo cuando se abra otro libro.

```
template <class L>
void e_reader<L>::abrir(const L& x){
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        // El libro esta en la tabla consulta su informacion
        info_libro i = t_libros.consulta(x);
        // Pone el libro como ultimo abierto y actualiza su contador
        ultimo_abierto = x;
        i.num_abierto = acumulador;
        acumulador++;
        t_libros.inserta(x,i);
    }
}
```

La operación consulta e inserta en una tabla, y modifica el valor de algunas variables. Su coste es constante.

### **Operación *avanzar-pag***

Se comprueba que hay algún libro abierto utilizando el valor del acumulador. En este caso no podemos comprobar que la secuencia de libros abiertos es vacía como se hacia en la representación anterior.

El último libro abierto se obtiene directamente de la variable `ultimo_abierto`, en lugar de buscar el primero de la secuencia.

```
template <class L>
void e_reader<L>::avanzar_pag(){
    // Si no hay ningun libro abierto produce error
    if (acumulador == 1) throw ENoabiertos();
    else {
        // Si hay libros abiertos. Obtiene el primero
        L l = ultimo_abierto;
        // incrementa la pagina en la informacion de la tabla
        info_libro i = t_libros.consulta(l);
        i.pag_actual++;
        // Si la pagina es mayor que la ultima vuelve a la primera
        if (i.pag_actual > i.total_paginas) i.pag_actual = 1;
        t_libros.inserta(l,i);
    }
}
```

La operación consulta e inserta en una tabla y modifica el valor de algunas variables. Su coste es constante.

### **Operación *abierto***

Se obtiene el valor directamente de la variable `ultimo_abierto`.

```
template <class L>
```

```
const L& e_reader<L>::abierto() {
    return ultimo_abierto;
}
```

El coste de la operación es constante.

### Operación *elim-libro*

Si el libro no era el último abierto simplemente lo borraremos de la tabla, en otro caso hay que dar valor a la variable `ultimo_abierto`. Para ello se recorre la tabla buscando el libro que tenga un valor mayor en su variable `num_abierto` que indica en que posición fue abierto. Si el único libro que se había abierto era el libro que se está eliminando, el acumulador se pone a cero para indicar que no queda ningún libro abierto.

```
template <class L>
void e_reader<L>::elim_libro(const L& x){
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        info_libro i = t_libros.consulta(x);
        t_libros.borra(x);
        cantidad--;
        // Si el libro es el ultimo abierto debe buscarse el anterior
        if (ultimo_abierto == x) {
            Tabla<L,info_libro>::Iterador it = t_libros.principio();
            int aux = 0;
            L libro_aux;
            while (it != t_libros.final()) {
                if (it.valor().num_abierto > aux) {
                    aux = it.valor().num_abierto;
                    libro_aux = it.clave();
                } // No se puede buscar el valor del acumulador
                // directamente porque se puede haber eliminado el libro
                it.avanza();
            }
            if (aux != 0) {
                ultimo_abierto = libro_aux;
                acumulador = aux + 1;
            }
            else acumulador = 0;
        }
    }
}
```

La operación realiza varias operaciones sobre la tabla de libros, todas ellas de coste constante. Se declara un iterador que recorre toda la tabla de libros. En cada vuelta del bucle las operaciones que se realizan son de acceso a los valores del iterador, por lo tanto el coste del cuerpo del bucle es constante. El coste de la operación es, por lo tanto, lineal respecto al número de libros del e-reader.

### Operación *recientes*

Para obtener los  $n$  libros que se han abierto más recientemente hay que buscarlos en la tabla. Para ello se crea un árbol binario de búsqueda en el que se van añadiendo todos

los libros de la tabla que han sido abiertos en algún momento. La clave es el orden en que fueron abiertos y el valor la información del libro. De esta forma al recorrer el árbol en inorden obtenemos los libros ordenados por el momento en que fueron abiertos.

```
template <class L>
Lista<L> recientes(int n){
    Lista<L> l = new Lista<L>();
    Arbus<int,L> aux; // clave el orden de apertura
    Tabla<L>::iterador it = t_libros.principio();
    while (it != t_libros.final()) {
        if (it.valor().num_abierto != 0) // libro abierto
            aux.inserta(it.valor().num_abierto, it.clave());
        it.avanza();
    }
    int cont = 0;
    Arbus<int,L>::Iterador ita = aux.principio();
    while (ita != aux.final() && cont < n) {
        l.Cons(ita.valor());
        ita.avanza();
        cont++;
    }
    return l;
}
```

La operación declara un iterador sobre la tabla de libros y la recorre entera. Cada elemento de la tabla se inserta en un árbol binario de búsqueda. Suponiendo el árbol equilibrado, el coste de la inserción es logarítmico respecto al número de libros abiertos del e-reader. Por último se recorre el árbol de búsqueda insertando cada elemento en una lista. Este recorrido tiene coste lineal respecto al número de libros abiertos, que es el número de nodos del árbol, ya que el coste de insertar por el principio de una lista es constante. El coste de la operación, al estar los dos recorridos en secuencia, es el máximo de los dos (el del primer bucle), esto es:  $\mathcal{O}(n_1 \log n_2)$  siendo  $n_1$  el número de libros del e-reader y  $n_2$  el número de libros abiertos del e-reader. En el caso peor en que todos los libros hayan sido abiertos el coste es  $\mathcal{O}(n \log n)$ .

### Tercera representación (apartado b. del enunciado):

Esta representación mejora los costes de la primera. Para mejorar la eficiencia de la operación *abrir* hay que evitar recorrer la lista de libros abiertos. Para ello se sustituye el TAD *Lista* por una lista doblemente enlazada con nodo cabecera, y el campo *abierto* de la tabla lo declaramos como un puntero al nodo de la lista donde se encuentra la información del libro. Al abrir un libro, en lugar de recorrer la lista para cambiar su posición, accedemos directamente al nodo que debemos cambiar a través del puntero, *abierto*, de la tabla. A continuación se realizan operaciones con punteros en la lista doblemente enlazada para modificar la posición del nodo, todas ellas con coste constante como se verá en la implementación.

La operación de eliminar un libro tiene también coste constante con esta implementación ya que no hay que realizar el recorrido de la lista.

La declaración de la clase es la siguiente:

```
template <class L>
```

```

class e_reader {
public:
    e_reader();
    void poner_libro(const L& x,int n);
    void abrir(const L& x);
    void avanzar_pag();
    const L& abierto();
    int pag_libro(const L& x);
    void elim_libro(const L& x);
    bool esta_libro(const L& x);
    Lista<L> recientes(int n);
    int num_libros();
private:
    class Nodo {
    public:
        Nodo() : _sig(NULL), _ant(NULL) {}
        Nodo(const T &elem) :
            _elem(elem), _sig(NULL), _ant(NULL) {}
        Nodo(Nodo *ant, const T &elem, Nodo *sig) :
            _elem(elem), _sig(sig), _ant(ant) {}
        T _elem;
        Nodo *_sig;
        Nodo *_ant;
    };
    struct info_libro {
        int total_paginas;
        int pag_actual;
        Nodo<L>* abierto;
    };
    Tabla<L,info_libro> t_libros;
    Nodo<L>* sec_abiertos; // Con nodo cabecera
    int cantidad;
};

```

### Operación *e-reader*, constructora.

Se crea el nodo cabecera de la lista doblemente enlazada.

```

template <class L>
e_reader<L>::e_reader() {
    sec_abiertos = new Nodo<L>();
    cantidad = 0;
}

```

El coste es constante.

### Operación *poner-libro*.


El puntero de la tabla a la lista doblemente enlazada se declara a *NULL*, ya que el libro no está abierto.

```

template <class L>

```

```

void e_reader<L>::poner_libro(const L& x,int n){
    // si el libro ya esta o el numero de paginas es
    // negativo no se hace nada
    if (!t_libros.esta(x) && n > 0) {
        // Se crea la informacion del libro
        info_libro i;
        i.total_paginas = n;
        i.pag_actual = 1;
        i.abierto = NULL;  ahora es un puntero
        //Se inserta en la tabla
        t_libros.inserta(x,i);
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}

```

El coste de la operación es constante, no cambia respecto a la anterior representación.

### Operación *abrir*.

El recorrido de la lista de libros abiertos se ha sustituido por el acceso a la lista doblemente enlazada a través del puntero de la tabla.

```

template <class L>
void e_reader<L>::abrir(const L& x){
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        // El libro esta en la tabla consulta su informacion
        info_libro i = t_libros.consulta(x);
        Nodo<L>* aux;
        if (i.abierto != NULL) {
            // Si esta abierto lo elimina de su posicion, pero no lo borra
            aux = i.abierto;
            aux->ant->sig = aux->sig;
            aux->sig->ant = aux->ant;
        }
        else {
            // Si no esta abierto crea el nodo
            aux = new Nodo<L>(x);
            // Cambia su estado a abierto
            i.abierto = aux;
            t_libros.inserta(x,i);
        }
        // Añade el nodo al principio de la lista
        aux->sig = sec_abiertos->sig;
        sec_abiertos->sig->ant = aux;
        aux->ant = sec_abiertos;
        sec_abiertos->sig = aux;
    }
}

```

El coste de la operación es constante.

### **Operación *avanzar-pag.***

Se comprueba que el libro está abierto utilizando el puntero a la lista doblemente enlazada de la tabla. Se accede al último libro abierto a través del comienzo de la lista doblemente enlazada.

```
template <class L>
void e_reader<L>::avanzar_pag() {
    // Si no hay ningun libro abierto produce error
    if (sec_abiertos->sig == NULL) throw ENoabiertos();
    else {
        // Si hay libros abiertos. Obtiene el primero
        L l = sec_abiertos->sig->elem;
        // incrementa la pagina en la informacion de la tabla
        info_libro i = t_libros.consulta(l);
        i.pag_actual++;
        // Si la pagina es mayor que la ultima vuelve a la primera
        if (i.pag_actual > i.total_paginas) i.pag_actual = 1;
        t_libros.inserta(l,i);
    }
}
```

El coste es constante.

### **Operación *abierto.***

Se comprueba si el libro está abierto a través del puntero de la tabla.

```
template <class L>
const L& e_reader<L>::abierto() {
    if (sec_abiertos->sig == NULL) throw ENoabiertos();
    else return sec_abiertos->sig->elem;
}
```

El coste es constante.

### **Operación *pag-libro.***

La implementación de esta operación no se modifica respecto a las anteriores representaciones.

### **Operación *elim-libro.***

Se elimina el libro de la lista doblemente enlazada a través del puntero de la tabla.

```
template <class L>
void e_reader<L>::elim_libro(const L& x) {
    if (!t_libros.esta(x)) throw ENoExiste();
    else {
        info_libro i = t_libros.consulta(x);
        // Si el libro esta abierto lo borra de la lista
        if (i.abierto != NULL) {
```

```

        Nodo<L>* aux = i.abierto;
        aux->sig->ant = aux->ant;
        aux->ant->sig = aux->sig;
        delete aux; i.abierto = null
    }
    t_libros.borra(x);
    cantidad--;
}
}

```

El coste es constante.

### **Operación *esta-libro*.**

La implementación de esta operación no se modifica respecto a las anteriores representaciones.

### **Operación *recientes*.**

Se recorre la lista doblemente enlazada utilizando un puntero auxiliar. El resultado es del tipo TAD *Lista*.

```

template <class L>
Lista<L> recientes(int n) {
    Lista<L> l = new Lista<L>();
    int cont = 0;
    Nodo<L>* aux = sec_abiertos->sig;
    while (aux != NULL && cont < n) {
        l.Cons(aux->elem);
        aux = aux->sig;
        cont++;
    }
    return l;
}

```

El coste es el mínimo entre el parámetro de entrada y el número de libros abiertos del e-reader.

### **Operación *num-libros*.**

La implementación de esta operación no se modifica respecto a las anteriores representaciones.

## **Comparación de los costes obtenidos con ambas representaciones .**



	Primera representación	Segunda representación	Con lista enlazada
Constructora	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>poner-libro</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>abrir</i>	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>avanzar-pag</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>abierto</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>pag-libro</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>elim-libro</i>	$\mathcal{O}(n)$ libros abiertos	$\mathcal{O}(n)$ libros totales	$\mathcal{O}(1)$
<i>esta-libro</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<i>recientes</i>	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
<i>num-libros</i>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Donde  $n$  representa el número de libros del e-reader.

## 2. Problemas propuestos:

1. Librería. (Obtenido del examen final Septiembre 2011. Titulación: II. Asignatura: EDI. Grupos A y C)

Se desea diseñar una aplicación para gestionar un sistema de venta de libros por Internet. El TAD será paramétrico respecto a la información asociada a un libro.

El comportamiento de las operaciones es el siguiente:

- *crear*: crea un sistema sin ningún libro.
- *an\_libro(x,n)*: Añade  $n$  ejemplares de un libro  $x$  al sistema. Si  $n$  toma el valor cero significa que el libro está en el sistema, aunque no se tienen ejemplares disponibles. Se pueden añadir mas ejemplares de un libro que ya esté en el sistema.
- *comprar(x)*: Un usuario compra un libro  $x$ . Si no existen ejemplares disponibles del libro  $x$  se produce un error.
- *esta-libro(x)*: Indica si un libro  $x$  se ha añadido al sistema. El resultado será cierto aunque no haya ejemplares disponibles del libro.
- *elim-libro(x)*: Elimina el libro  $x$  del sistema. Si el libro no existe la operación no tiene efecto.
- *num-ejemplares(x)*: Devuelve el número de ejemplares de un libro  $x$  que hay disponibles en el sistema. Si el libro no está dado de alta se produce un error.
- *top10()*: Obtiene una lista con los 10 libros que más se han vendido. Si hay mas de 10 libros distintos con un máximo número de ventas la aplicación obtiene 10 de ellos, sin que se especifique cuales. Si no se han vendido 10 libros distintos se listarán todos ellos.
- *num-libros()*: Obtiene el número de libros distintos que existen en el sistema.

Se pide:

- a) Obtener una representación eficiente del tipo. No se permite utilizar directamente vectores ni memoria dinámica (listas enlazadas con nodos y punteros). Implementar todas las operaciones indicando el coste de cada una de ellas. El tipo de retorno de la operación *top10* debe ser un tipo lineal, seleccionar uno

adecuado y justificarlo. Las operaciones de consulta sobre los TAD devuelven una copia de la estructura.

- a) Generalizar la operación `top10` anterior con una operación `topN(n)` que obtenga una lista con los  $n$  libros que más se han vendido, ordenada según el número de ejemplares vendidos, de los más vendidos a los menos vendidos. Si se ha vendido el mismo número de ejemplares de varios libros, se mostrará primero el que se haya vendido más recientemente. Si el número de libros vendidos es menor que el solicitado, la lista contendrá todos ellos.

Proponer los cambios necesarios en la implementación del TAD anterior de forma que el coste de la operación `topN` sea, en el caso peor, lineal respecto al máximo número de libros cuyas unidades vendidas sean las mismas y no empeore el coste de ninguna de las otras operaciones. Para esta implementación se puede utilizar memoria dinámica, (listas enlazadas de nodos con punteros). Tampoco debe aumentar de forma significativa el gasto en memoria.

(Existen representaciones que proporcionan coste constante para la operación `topN`, sin embargo en el ejercicio será suficiente con proporcionar el coste pedido.)

2. Restaurante. (Obtenido del examen final Septiembre 2006. Titulación: II. Asignatura: EDI. Grupos A, B y C)

El conocido restaurante Salmonelis necesita una base de datos para gestionar mejor sus afamados platos. Cada plato dispondrá de un código único de tipo plato y tendrá asociado un código de tipo y un precio. Ejemplos de códigos de tipo son: entrante, carne, pescado, sopa, etc.

Las operaciones que debe ofrecer el TAD son las siguientes:

- *crear*, crea una base de datos vacía
- *anadir*( $p, t, n$ ), añade un plato  $p$  con su tipo  $t$  y precio  $n$ . Produce error si el plato ya está en la base de datos.
- *modif-tipo*( $p, t$ ), modifica el tipo de un plato  $p$ . Produce error si el plato no está en la base de datos.
- *modif-precio*( $p, n$ ), modifica el precio de un plato  $p$ . Produce error si el plato no está en la base de datos.
- *tipo*( $p$ ), devuelve el tipo de un plato  $p$ . Produce error si el plato no está en la base de datos.
- *precio*( $p$ ), devuelve el precio de un plato  $p$ . Produce error si el plato no está en la base de datos.
- *platos-tipo*( $t$ ), devuelve una lista de platos, de un tipo  $t$  con sus precios, ordenada por precios crecientes. Si no existe ningún plato del tipo pedido devuelve la lista vacía.

Se pide obtener una representación eficiente del tipo utilizando tipos conocidos. Implementar todas las operaciones indicando el coste de cada una de ellas.

3. Clínica. (Obtenido del examen final Septiembre 2004. Titulación: II. Asignatura: EDI.)

Tras evaluar su funcionamiento durante el último año, la dirección de la Clínica *Casi Me Muero* ha decidido renovar el sistema informático de su consultorio médico para realizar (al menos) las siguientes operaciones:

- *crear*, genera un consultorio vacío, sin ninguna información,
- *alta-médico(m)*, da de alta a un nuevo médico  $m$  que antes no figuraba en el consultorio,
- *pedir-vez(p,m)* hace que un paciente  $p$  pida la vez para ser atendido por un médico  $m$ ,
- *atender-consulta(m)*, atiende al paciente que le toque en la consulta de un médico  $m$ ,
- *cancelar-cita(m)* permite que el último paciente en la consulta de un médico  $m$ , debido al cansancio de la espera, cancele la cita,
- *pedir-vez-enchufe(p,m)* hace que un paciente  $p$ , haciendo uso de algún *enchufe* que aquí no nos interesa, pida la vez para colocarse el primero en la consulta de un médico,
- *baja-médico(m)* da de baja a un médico  $m$ , borrando todas sus citas,
- *num-citas(p)*, devuelve el número de citas que un mismo paciente tiene en todo el consultorio.

Se pide obtener una representación eficiente del tipo utilizando tipos conocidos. Implementar todas las operaciones indicando el coste de cada una de ellas.

4. Campeonato de atletismo. (Obtenido del examen final de Junio de 2009. Titulación: ITIS. Asignatura: EDI. Grupos A y B)

El TAD *Campeonato* se utiliza para manejar la información relativa a unas pruebas de atletismo.

El comportamiento de las operaciones que debe ofrecer el TAD es el siguiente:

- *crea*. Constructora que crea un TAD vacío.
- *an\_prueba(p)*. Añade una prueba  $p$  al campeonato. Si la prueba ya está en el TAD éste no se modifica.
- *an\_atleta(a,p)*. Añade un atleta a una prueba del campeonato. Si no está la prueba en el campeonato la operación produce error. El orden de los atletas en una prueba es importante.
- *obtener-sig-atleta(p)*. Obtiene el siguiente atleta a participar en una prueba. El orden viene dado por el orden en que se incorporaron al sistema. Produce un error si la prueba no está dada de alta o no hay ningún atleta en la prueba.

Se pide obtener una representación para el TAD e implementar todas las operaciones. Para cada operación calcular el coste de la implementación realizada.