TADs de Estructuras de Datos y Algoritmos

(curso 2014-15; los exámenes de junio y septiembre incluirán copias de esta hoja)

Stack<T>

```
Stack()
void push(const T &elem)
void pop()
const T &top() const
bool empty() const
unsigned int size() const
```

Queue<T>

```
Queue()
void push_back(const T &elem)
void pop_front()
const T &front() const
bool empty() const
unsigned int size() const
```

Deque<T>

```
Deque()
void push_back(const T &e)
void push_front(const T &e)
const T &back() const
const T &front() const
void pop_back()
void pop_front()
bool empty() const
unsigned int size() const
```

Arbin<T>

```
Arbin()
Arbin(const Arbin &otro)
Arbin(const Arbin &iz, const T &e, const Arbin &dr)
bool esVacio() const
const T &raiz() const
Arbin hijoIz() const
Arbin hijoDr() const
Lista<T> preorden() const
Lista<T> inorden() const
Lista<T> postorden() const
Lista<T> niveles() const
unsigned int numNodos() const
unsigned int talla() const
unsigned int numHojas() const
```

List<T>

```
List()
List(const List<T> &otra)
void push_back(const T &e)
void push_front(const T &e)
const T &back() const
const T &front() const
void pop_back()
void pop_front()
bool empty() const
unsigned int size() const
const T &at() const
ConstIterator cbegin() const
ConstIterator cend() const
Iterator begin()
Iterator end() const
Iterator erase(const Iterator &it)
void insert(const T &elem, const Iterator &it)
```

List<T>::ConstIterator

```
void next()              // it++
const T &elem() const    // *it
```

List<T>::Iterator

```
void next()              // it++
T &elem() const          // *it
void set(const T &e) const
```

{Tree,Hash}Map<C,V>

```
TreeMap()
  ó HashMap()
TreeMap(const TreeMap &otro)
  ó HashMap(const HashMap &otro)
bool contains(const C &clave) const
const V &at(const C &clave) const
bool empty() const
V &operator[](const C &clave)
void insert(const C &clave, const V &valor)
void erase(const C &clave)
ConstIterator find(const C &clave) const
ConstIterator cbegin() const
ConstIterator cend() const
Iterator find(const C &clave)
Iterator begin()
Iterator end() const
```

{Tree,Hash}Map<C,V>::ConstIterator

```
void next()              // it++
const C &key() const
const V &value() const
```

{Tree,Hash}Map<C,V>::Iterator

```
void next()              // it++
const C &key() const
V &value() const
```

Scanned by CamScanner

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Segundo Cuatrimestre, 9 de Septiembre de 2014.

1. **(3 puntos)** Dado un árbol binario de enteros, se entiende que es un árbol genealógico correcto si cumple las siguientes reglas, producto de interpretar el entero de cada nodo como el *año de nacimiento* del individuo, el hijo izquierdo como su primer hijo de un máximo de dos, y el hijo derecho como el segundo hijo:

   - La edad del padre siempre supera en al menos 18 años las edades de cada uno de los hijos.
   - La edad del segundo hijo (si existe) es al menos dos años menos que la del primer hijo (no hay hermanos gemelos/mellizos en estos árboles).
   - Los árboles genealógicos de ambos hijos son también correctos.

   Implementa una función que reciba un árbol binario que permita averiguar si un árbol binario es o no árbol genealógico correcto y, en caso de serlo, el número de generaciones distintas que hay en la familia.

2. **(3 puntos)** Extiende la implementación de los árboles de búsqueda, añadiendo la siguiente operación:

   ```
   Iterador busca(const Clave &clave) const;
   ```

   que busque en el árbol la clave dada y devuelva un *iterador* que permita recorrer todos los elementos contenidos en el árbol desde esa clave.

   A modo de recordatorio, la definición (parcial) de la clase interna `Arbus::Iterador` vista en clase es:

   ```
   class Iterador {
   public:

      // ...

   protected:

      // Para que pueda construir objetos del
      // tipo iterador
      friend class Arbus;

      // ...

      // Puntero al nodo actual del recorrido
      // NULL si hemos llegado al final.
      Nodo *_act;
   ```

Scanned by CamScanner

```
    // Ascendientes del nodo actual
    // aún por visitar
    Pila<Nodo*> _ascendientes;
};
```

3. (4 puntos) Estás trabajando en un nuevo videojuego llamado *CiudadMatic*: un simulador de ciudades, llenas de edificios de distinto tipo. Será posible construir y reparar estos edificios gastando dinero, y al final de cada turno (después de haber construido y reparado tantos edificios como se quiera), recaudar los impuestos que generen. Necesitarás implementar las siguientes operaciones:

- *CiudadMatic*: inicializa una nueva ciudad vacía, con el dinero que se pase como argumento disponible para ser gastado.
- *nuevoTipo*: añade un nuevo tipo de edificio al sistema, con un identificador proporcionado por el usuario (por ejemplo, "bar"), un coste de construcción, una cantidad de impuestos generada por turno, y una calidad de base (máximo de turnos sin reparar). No devuelve nada.
- *insertaEdificio*: dado el nombre de un edificio (por ejemplo, "El Bar de Moe"), y el identificador de su tipo, y asumiendo que se disponga del dinero necesario para construirlo, añade el edificio a la ciudad y resta su coste de construcción del dinero disponible. No devuelve nada.
- *reparaEdificio*: repara el edificio cuyo identificador se pase como argumento a "como recién construido", a un coste del 10 % del coste de construcción (descartando los decimales), independientemente de lo estropeado que estuviese. No devuelve nada.
- *finTurno*: todos los edificios construidos generan los impuestos que les corresponden por su tipo. Después, todos se estropean por un punto de calidad, y aquellos que lleguen a 0 son derribados y eliminados de la ciudad. Devuelve el dinero total disponible para el nuevo turno (impuestos generados + dinero no gastado del turno anterior).
- *listaEdificios*: dado un identificador de tipo de edificio, devuelve una lista con los edificios de ese tipo que están actuamente construidos, por orden de antigüedad (primero el más viejo).

Desarrolla en C++ una implementación de la clase *CiudadMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Scanned by CamScanner

EDA, 2014/15

8 de mayo de 2015

Examen final (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 el cliente c, siempre que éste tuviera alojamiento. En caso de no tener alojamiento, produce un error.
- *listado_hoteles():* Obtiene un lista de todos los hoteles que están dados de alta en la agencia, ordenados según el orden definido en el tipo de parámetro.
- *huespedes(h):* Permite obtener el conjunto de clientes que se alojan en un hotel dado. Dicho conjunto sera vacío si no hay clientes en el hotel.

Se pide:

(i) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas.

(ii) Implementar todas las operaciones, indicando el coste de cada una de ellas. La operación *huespedes* debe producir un lista de clientes en lugar de un conjunto.

Scanned by CamScanner

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, 11 de septiembre de 2013.

1. (3 puntos)

Especifica y deriva, o especifica, diseña y verifica, una función *iterativa* que dados dos arrays A y B de longitudes n y m respectivamente ($n \leq m$) de enteros ordenados de manera estrictamente creciente, determine si todos los elementos de A aparecen en B. La función debe tener coste **lineal** y lo debes justificar.

2. (4 puntos)

Te han contratado para implementar un sistema de gestión de barcos de pesca. Cada barco tiene una bodega de carga donde los pescadores que van en el barco van depositando las capturas que realizan, anotando siempre la especie del pez y su peso. Cuando el barco llega a puerto, cada pescador se lleva a casa lo que ha pescado de cada especie.

Las operaciones públicas del TAD *BarcoMatic* son:

- *nuevo*: Crea una nueva instancia de la estructura *BarcoMatic*, recibiendo como argumento un el peso máximo (en kilos) admitido en la bodega.
- *altaPescador*: Da de alta a un pescador, identificado por su nombre. No devuelve nada.
- *nuevaCaptura*: Registra que un pescador (que debe estar registrado) ha pescado un ejemplar de tantos kilos de una especie concreta. Las especies y los pescadores se indican mediante sus nombres, y el peso en kilos se especifica mediante un número. Si el peso de la captura, añadido a la bodega, haría que la bodega excediese su capacidad, esta operación debe fallar.
- *capturasPescador*: Recibe el nombre de un pescador, y devuelve una lista de parejas especie-kilos. Si, para una especie dada, el pescador no ha pescado nada, no la debes incluir en la lista devuelta. Puedes asumir la existencia de un TAD *Pareja<A, B>* similar al visto en clase.
- *kilosEspecie*: Recibe el nombre de una especie, y devuelve el número total de kilos de esa especie pescados, sumando las capturas de todos los pescadores.
- *kilosPescador*: Recibe el nombre de un pescador, y devuelve el número total de kilos que ha pescado, sumando todas las especies.
- *bodegaRestante*: Devuelve el número de kilos restantes en la bodega.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación (***especificando qué representa la N en cada caso concreto***) e implementación en C++ de todas las operaciones.

3. (3 puntos)

Podemos representar la discretización de una función $f(x)$ mediante una tabla (*Tabla<int,float>*) que asocia a ciertos valores enteros de abscisa $(x_1, x_2, \ldots)$ sus correspondientes valores reales de ordenada $(f(x_1), f(x_2), \ldots)$. Suponiendo que:

- La tabla sólo contiene valores de abscisa positivos y consecutivos comenzando en $x = 1$.
- La función $f(x)$ discretizada es estrictamente creciente ($x_1 < x_2 \rightarrow f(x_1) < f(x_2)$).

Se pide implementar una función con coste $O(\log n)$ que, dada una tabla como la anterior y un valor de ordenada y, devuelva la abscisa entera x asociada a y si el punto (x,y) existe en la tabla, y -1 si no existe.

```
int buscaX(const Tabla<int,float> &f, float y)
```

Scanned by CamScanner

EDA, 2014/15

8 de mayo de 2015

Examen final (Junio 2011)

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:

- *crear:* Crea un e-reader sin ningún libro.
- *poner_libro(x,n):* Añade el libro x al e-reader. El parámetro n representa el número de páginas del libro y puede ser cualquier número positivo. Si el libro ya existe, la acción no tiene efecto.
- *abrir(x):* El usuario abre el 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 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 ningun libro abierto, se produce un error.
- *abierto():* Devuelve el último libro que se ha abierto. Si no se encuentra ningún libro abierto, se produce un error.
- *pag_libro(x):* Devuelve la página del libro x en que se quedó leyendo el usuario. Se considera que todos los libros empiezan por la página 1. Si el libro no esta dado de alta, se produce un error.
- *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 elmina y queda como último abierto el que se abrió con anterioridad.
- *esta_libro(x):* Consulta si el libro x está en el e-reader.
- *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 los contendrá a todos. Si un libro se ha abierto varias veces sólo aparecerá en la posición más reciente.
- *num_libros(x):* Consulta el número de libros que existen en el e-reader.

Se pide:

(i) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas. No se permiten utilizar vectores dinámicos ni listas enlazadas. Implementar todas las operaciones indicando el coste de cada una de ellas. El tipo de retorno de la operación *recientes* debe ser de tipo lineal, seleccionar uno y justificarlo.

(ii) Modificar la representación anterior usando 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 el qeu se obtuvo en la representación anterior, ni debe aumentar de forma significativa el gasto en memoria. Implementar todas las operaciones indicando su coste.

Scanned by CamScanner

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, 12 de junio de 2013.

1. (3 puntos)

Se tiene un vector de enteros $a[0..n-1]$, que puede ser vacío, cuyo propósito es representar los coeficientes de un polinomio en una variable: $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$. Se pide especificar, y derivar, o diseñar y verificar una función iterativa que, dado un vector de coeficientes y un valor de x, devuelva la evaluación del polinomio para ese valor. El coste ha de ser **lineal** y se ha de justificar su cálculo. La evaluación ha de seguir **obligatoriamente** la secuencia de cómputos impuesta por la llamada Regla de Horner:

$$p(x) = a_0 + a_1(\ldots a_{n-3} + (a_{n-2} + a_{n-1}x)\,x\ldots)\,x$$

Si el polinomio es vacío, se entenderá que su evaluación para cualquier x da cero.

2. (4 puntos)

Desarrolla MultaMatic, el nuevo sistema de gestión de multas de tráfico por exceso de velocidad. La red de carreteras contiene tramos vigilados en los que se coloca una cámara al principio del tramo y otra al final. Cada vez que un coche pasa frente a una cámara, se toma una foto de su matrícula y se apunta el momento en que pasó; si el tiempo trascurrido entre la foto del comienzo y la del final es demasiado breve, se le pone una multa. Para simplificar, asumiremos que los tramos no comparten cámaras ni se solapan entre sí. Las operaciones públicas del TAD son:

- *insertaTramo*: añade un nuevo tramo al sistema. Recibe un identificador de tramo, los identificadores de sus cámaras inicial y final, y el número mínimo de segundos que deben transcurrir entre las fotos de comienzo y final para *no* recibir multa. Si el tramo ya existía debe generar un error.
- *fotoEntrada*: se invoca cada vez que un coche entra en un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual (en segundos desde el 1 de enero de 1970).
- *fotoSalida*: se invoca cada vez que un coche sale de un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual. Si el coche ha ido demasiado rápido en el tramo, se le multará.
- *multasPorMatricula*: devuelve el número de multas asociadas a una matrícula.
- *multasPorTramos*: devuelve una lista con las matrículas de los coches multados en un determinado tramo. Si un coche ha sido multado varias veces, su matrícula aparecerá varias veces en la lista. Si el tramo no existe debe generar un error.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación e implementación en C++ de todas las operaciones.

3. (3 puntos)

Dada una lista de números $a_1, a_2, \ldots, a_n$ se dice que dos números producen una inversión si $a_i > a_j$ con $i < j$. Se pide un algoritmo que calcule el número de inversiones que existen en una lista de números. Se debe utilizar el método **Divide y Vencerás**. Se valorará la eficiencia del algoritmo implementado. **Nota:** Se puede conseguir un coste del orden de $O(n \log n)$.

Calcular el coste del algoritmo implementado planteando la recurrencia y resolviéndola.

Scanned by CamScanner

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Segundo Cuatrimestre, 9 de Septiembre de 2014.

1. **(3 puntos)** Dado un árbol binario de enteros, se entiende que es un árbol genealógico correcto si cumple las siguientes reglas, producto de interpretar el entero de cada nodo como el *año de nacimiento* del individuo, el hijo izquierdo como su primer hijo de un máximo de dos, y el hijo derecho como el segundo hijo:

   - La edad del padre siempre supera en al menos 18 años las edades de cada uno de los hijos.
   - La edad del segundo hijo (si existe) es al menos dos años menos que la del primer hijo (no hay hermanos gemelos/mellizos en estos árboles).
   - Los árboles genealógicos de ambos hijos son también correctos.

   Implementa una función que reciba un árbol binario que permita averiguar si un árbol binario es o no árbol genealógico correcto y, en caso de serlo, el número de generaciones distintas que hay en la familia.

2. **(3 puntos)** Extiende la implementación de los árboles de búsqueda, añadiendo la siguiente operación:

   ```
   Iterador busca(const Clave &clave) const;
   ```

   que busque en el árbol la clave dada y devuelva un *iterador* que permita recorrer todos los elementos contenidos en el árbol desde esa clave.

   A modo de recordatorio, la definición (parcial) de la clase interna `Arbus::Iterador` vista en clase es:

   ```
   class Iterador {
   public:

      // ...

   protected:

      // Para que pueda construir objetos del
      // tipo iterador
      friend class Arbus;

      // ...

      // Puntero al nodo actual del recorrido
      // NULL si hemos llegado al final.
      Nodo *_act;
   ```

Scanned by CamScanner

```
    // Ascendientes del nodo actual
    // aún por visitar
    Pila<Nodo*> _ascendientes;
};
```

3. **(4 puntos)** Estás trabajando en un nuevo videojuego llamado *CiudadMatic*: un simulador de ciudades, llenas de edificios de distinto tipo. Será posible construir y reparar estos edificios gastando dinero, y al final de cada turno (después de haber construido y reparado tantos edificios como se quiera), recaudar los impuestos que generen. Necesitarás implementar las siguientes operaciones:

- *CiudadMatic*: inicializa una nueva ciudad vacía, con el dinero que se pase como argumento disponible para ser gastado.
- *nuevoTipo*: añade un nuevo tipo de edificio al sistema, con un identificador proporcionado por el usuario (por ejemplo, "bar"), un coste de construcción, una cantidad de impuestos generada por turno, y una calidad de base (máximo de turnos sin reparar). No devuelve nada.
- *insertaEdificio*: dado el nombre de un edificio (por ejemplo, "El Bar de Moe"), y el identificador de su tipo, y asumiendo que se disponga del dinero necesario para construirlo, añade el edificio a la ciudad y resta su coste de construcción del dinero disponible. No devuelve nada.
- *reparaEdificio*: repara el edificio cuyo identificador se pase como argumento a "como recién construido", a un coste del 10 % del coste de construcción (descartando los decimales), independientemente de lo estropeado que estuviese. No devuelve nada.
- *finTurno*: todos los edificios construidos generan los impuestos que les corresponden por su tipo. Después, todos se estropean por un punto de calidad, y aquellos que lleguen a 0 son derribados y eliminados de la ciudad. Devuelve el dinero total disponible para el nuevo turno (impuestos generados + dinero no gastado del turno anterior).
- *listaEdificios*: dado un identificador de tipo de edificio, devuelve una lista con los edificios de ese tipo que están actuamente construidos, por orden de antigüedad (primero el más viejo).

Desarrolla en C++ una implementación de la clase *CiudadMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Scanned by CamScanner

Febrero 2016 9/05/16

Problema de la agencia:

Utilizamos un treeMap para guardar la información de los hoteles, de modo que la clave será el hotel y la información asociada será una lista de los clientes que hay actualmente en el hotel.

Usamos también un HashMap para guardar clientes (clave) y el hotel en el que se hospeda (información asociada a la clave).

↳ Los hoteles están en un árbol binario ordenado.

La clase queda:

```
template <class H, class C>
class agencia {
    private:
        TreeMap <H, lista<C>> - hoteles;
        HashMap <C, H> - clientes;
    public:
        agencia();  // Crea una agencia vacía.
        void aloja (const C &c, const H &h);
        void desaloja (const C &c);
        const H alojamiento (const &c) const;
        lista <H> hoteles ();
        lista <C> huespedes (const H &h) const;
};
```

Scanned by CamScanner

Implementación de los métodos:

```
template < class H, class C >
    agencia() {};
```
→ Crea una agencia vacía.

```
template <class H, class C>
void agencia<H,C>::alojar(const C &c, const H &h) {
    //Si el cliente esta:
    if(_clientes.contains(c)) {
        //Eliminamos el cliente del hotel actual
        H hotel = clientes.at(c); → regresa el hotel donde esta el cliente.
        lista<C> &clientes = _hoteles.at(hotel);
        Lista<C>::Iterator it = l_clientes.begin();
        while(it.elem() != c) {
            it++;
        }
        l_clientes.erase(it);
        _hoteles.insert(hotel, l_clientes);
    } //Fin del if
    Lista<C> lClientes;
    if(_hoteles.contains(h)) {   //si esta el hotel es el que se va a hospedar.
        lClientes = _hoteles.at(h); → copia su lista de clientes
    }
    lClientes.insert(c);
    _hoteles.insert(h, clientes);
    _clientes.insert(c, h);
}
```

Scanned by CamScanner

Complejidad de alojar:

- Buscar cliente (contains) : constante
- Recorrer el hotel (at) : constante
- Recuperar lista clientes : log (n_hoteles)
- Recorrer la lista : n_clientes (lineal)
- Borrar un cliente (erase(it)) : constante
- Insertar hotel : log (n_hoteles)
- Buscar el hotel : log (n_hotel)
- Recorrer el hotel (_hoteles.at(n)) : log (n_hoteles.
- Insertar cliente : constante
- Insertar en el TreeMap : log
- Insertar en el HashMap : constante.

- La complejidad es Lineal en el caso peor.

// Función desalojar

```
template <class C, class H>
void agencia <C,H> :: desalojar (const C &c){
    if (_clientes.contains(c)) { → cte
        H h = _clientes.at(c); → cte
        lista <c> lClientes = _hoteles.at(h); → log
        lista <c> :: Iterator it = l_clientes.begin();
        while (it.elem() != c){
            it++;
        }
```

Lineal en el nº de clientes del hotel

Scanned by CamScanner

```
    lClientes.erase(it);  → cte
    _hoteles.insert(h, lClientes); → log
    -clientes.erase(c); → cte
  }
}
```

→ Tiene complejidad lineal en el nº de clientes

```
// Función alojamiento
template <class C, class H>
 const H alojamiento(const C &c) const {
     if(!_cliente.contains(c)) {
         throw NoHayCliente(); → El cliente no está
     }
     return _cliente.at(c);
 }

// Función de listado de hoteles
 template <class C, class H>
 Lista<H> hoteles() {
     Lista<H> lHoteles;
     TreeMap<H,C>::Iterador it = _hoteles.begin();
     while(it != _hoteles.end()) {
         lHoteles.push_back(it.Key());
         it++;
     }
     return lHoteles;
 }
```

Scanned by CamScanner

```
// Función huespedes → tiene complejidad logarítmica
template <class C, class H>
Lista<C> agencia<C,H>::huespedes(const H &h) {
    if (_hoteles.contains(h)) {
        return _hoteles.at(h);
    }
    else {
        throw NoHayHotel();
    }
}
```

Junio 2011.

Atributos del eReader:

- Nº de libros
- Lista que guarde la libro abierto recientemente, en el principio va a estar el más reciente.
- HashMap que guarde los libros (como claves) y la información asociada al libro que es: nº de pags, pag actual, y un booleano que nos diga si el libro está abierto.

Scanned by CamScanner

```
template <class L>
class eReader {

    private:
        unsigned int _nlibros; -> Nº de libros
        lista < L > _abiertos; -> list
        struct infoLibro{
            unsigned int _npags;
            unsigned int _act;
            bool _abierto;
        };
        HashMap < L, infolibro > _libros;

    public:
        eReader(); //Crea un eReader vacio
        void poner_libro (const L &l, unsigned int n);
        void abrir (const L &l);
        void avanzar_pagina();
        const L abierto() const;
        const unsigned int pag_libro (const L &l) const;
        void elim_libro (const L &l);
        bool esta_libro (const L &l) const;
        list <L> recientes (unsigned int n);
        unsigned int num_libros();

};
```

Scanned by CamScanner

```
template <class L>
EReader<L> :: eReader() {
    _nLibros = 0;
}
```

→ Otra forma

```
EReader<L> :: EReader(): _nLibros(0) {}
```

```
template <class L>
void poner_libro (const L &l, unsigned int n) {
    if (!_libros.contains(l)) {          // cte
        struct infoLibro libro;
        libro.numPags = n;               // cte
        libro.act = 0;                   // cte
        libro.abierto = false;           // cte
        _libros.insert(l, libro);        // cte
        _nLibros++;
    }
}
```

→ Complejidad constante

```
template <class L>
void abrir (const L &l) {
    if (!_libros.contains(l)) {          // cte
        throw NoExisteLibro();
    }
    _libros.at(l).abierto = true;        // cte
    Lista<L>::Iterator it = _abiertos.begin(); // cte
    while (it != _abiertos.end(l) && !encontrado) {
        if (it.elem() == l) {
            it.erase();
            encontrado = true;
        }
        else {
            it.next();
        }
    }
}
```

→ La complejidad es lineal en función del nº de libros abiertos.

(*) NO

lineal en el nº de libros abiertos

Scanned by CamScanner

```
_abierto.push_front(l);  ← Como mucho es lineal
}
```

⊛ Op 1: struct infoLibro libro;

```
libro.numPags = _libros.at(l).numPags;
libro.act = _libros.at(l).act;
libro.lAbierto = true;
_libros.insert(l, libro);
```

```
template <class L>
void EReader<L>::avanzar_pag() {
    if(_abiertos.size() == 0) {
        throw NoHayAbiertos();
    }
    else {
        _libros[_abiertos.front()].act++;
        if((_libros[_abiertos.front()].act % _libros[_abiertos.front()].
            numPags + 1) == 0) {
            _libros[_abiertos.front()].act = 1;
        }
    }
}
```

Scanned by CamScanner

```
template <class L>
const L& abierto() const {
    if (_abiertos.empty()) {
        throw NoHayAbierto();
    }
    return (_abiertos.front());
}

template <class L>
unsigned int pag_libro(const L& l) {
    if (!_libros.contains(l)) {
        throw LibroNoExiste();
    }
    return _libros.at(l).act;
}

template <class L>
void elim_libro(const L& l) {
    bool encontrado = false;
    if (_libros.contains(l)) {
        if (_libros.at(l).labierto) {
            List<L>::Iterator it = _abiertos.begin();
            while (it != _abiertos.end() && !encontrado) {
                if (it.elem() == l) {
                    encontrado = true;
                    _abiertos.erase(it);
                }
                else {
                    it++;
                }
            }
        }
        _libros.erase(l);
    }
}
```

Complejidad constante

5.

Scanned by CamScanner

EDA 11/5/15 SCF

EX. 2011

EReader

*COMPLEJIDAD CTE

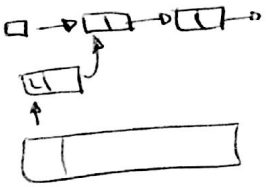
```
bool esta_libro (const L &l) {
    return _libros.contains(l); //cte
}

Lista<L> eReader::recientes (int n) {
    if(n<0)
        throw ExNroValido();
    int cont = 0;
    Lista<L> rec;
    Lista<L>::Iterador it = _abiertos.begin();
    while ((it != _abiertos.end()) && (cont < n)){
        rec.push_back(it.elem());
        it++;
        cont++;
    }
    return rec;
}

unsigned int num_libros() const {
    return _nLibros; //cte
}
```

min(n; n de libros abiertos) CLINEAL

```
template <class L>
Lista<L> _abiertos;
unsigned int n_libros;
struct info_libro {
    unsigned int numPags;
    unsigned int act;
    bool &abierto;
};
HashMap<L, info_libro> _libros;
```


Scanned by CamScanner

(PP) Modificar la clase usando listas enlazadas para que la operación "abrir" tenga complejidad constante.



Cambiamos el campo _abierto de la estructura "info_libro" por un puntero a un nodo. Esto nos obliga a definir la lista de libros abiertos como una lista enlazada.

La nueva clase queda:

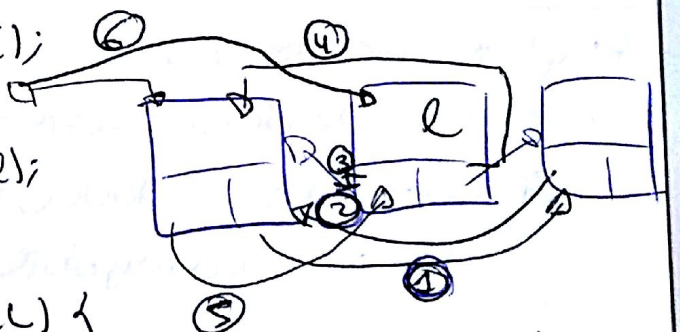
```
template <class L>
class eReader {
  private:
    class Nodo { // clase interna
      L _valor;
      Nodo *_next;
      Nodo *_prev;
      //Constructores
      Nodo(): _next(NULL), _prev(NULL){}
      Nodo<L>(const L &e): _next(NULL), _prev(NULL), _valor(e)
      Nodo (const Nodo *ant, const (&e, const Nodo *sig):
                _next(sig), _valor(e), _prev (ant) {}

struct info_libro {
        unsigned int numPags;
        unsigned int act;
        Nodo <L> *Abierto;

};
HashMap <L, info_libro> _libros;
unsigned int _num_libros;
Nodo <L> *_abiertos;
public:
```

Scanned by CamScanner

```
public:
template <class L>                    //cfe
eReader<L>::eReader(){
_numLibros = 0;
_abiertos = newNodo<L>();
```

```
template <class L>
void eReader<L>::poner_libro(const L& l,
                             unsigned int n){

    if(!_libros.contains(l)){          //cfe
      Info_libro libro;
      libro.numLibros = n;
      libro.act = 1;
      libro.eAbierto = NULL;
      _libros.insert(l, libro);
      _numLibros++;
    }
```

```
template <class L>
void eReader<L>::abrir(const L& l){
    if(!_libros.contains(l)){
      throw ExLibroNoExiste(l);
    }
  Info_libro info = _libros.at(l);
  if(info.eAbierto != NULL){
    if(info.eAbierto->prev != NULL)
①     (info.eAbierto->prev)->next = info.eAbierto->next;
    if(info.eAbierto->next != NULL)
②     (info.eAbierto->next)->prev = info.eAbierto->prev;
```



(2)

Scanned by CamScanner

```
else{                                                    //cte
  info.labierto = new    Nodo<L>(l);
}
③info.labierto->prev = NULL;
④info.labierto->next = _abiertos;)   (primero)
⑤_abiertos->prev = &info.labierto;
⑥_abiertos = info.labierto;
```

```
template <class L>
void eReader<L>::avanza_pag(){
    info_libros info;
    if (_abiertos = NULL)  //No hay lista de libros abiertos
      throw    NoHayAbiertos();
  L l =  abiertos->valor; // Libro
  info = _libros.at(l); //Informac. del libro l
  info.act++; // Avanzamos página
  if (info.act == info.numPag+1)
          info.act = 1
   _libro.insert(l, info)
}
```

```
template <class L>
const L eReader<L>::abierto() const{
   if (_abiertos == NULL){
      throw NoHayAbiertos();
   }
  return _abiertos->valor;                               //cte
}
```

Scanned by CamScanner

11/5/15

```
template <class L>
const unsigned int eReader:: pag_libro (const L& l){
    if(!_libros.contains(l))
        throw libroNoExiste();

    return _libros.at(l).act;
```

```
template <class L>
void elim_libro (const L& l){
 if (_libros.contains(l)){

   Nodo<L>* estaAbierto = libros.at(l).LAbierto;
   if (estaAbierto != NULL){
   if (estaAbierto->prev != NULL){
       estaAbierto->prev->next = estaAbierto->next;
   if (estaAbierto->next != NULL){
       estaAbierto->next->prev = estaAbierto->prev;

if (_abiertos == estaAbierto)
    _abiertos = estaAbierto->next;
    delete estaAbierto;

 libros.erase(l);
 numLibros--;
```

Scanned by CamScanner

```
template <class L>
bool EReader<L>::este_libro(const L& l){
    return _libros.contains(l);
}
```

```
template <class L>
List<L> EReader<L>::recientes(unsigned int n)
    List<L> eRecientes;
    if (_abiertos){                                    // LINEAL
        Nodo *act = _abiertos;
        while(act != NULL && n>0){
            eRecientes.push_back(act->valor);
            act = act->next;
            n--;
    }
    return eRecientes;
```

```
template <class L>
unsigned int EReader<L>::num_libros(){
    return numLibros;
}
```

Scanned by CamScanner

25/04/15

Ejemplo Dragones:

En la raiz hay tesoro, en la nodo interno hay 'Dragones' o 'Via libre' y en las hojas hay distintas puertas. Elegir la puerta cuyo recorrido hasta la raiz tenga el min nº de dragones. A igualdad de dragones, la puerta que esta más a la izq. Si hay un

0 → Tesoro
1 → Dragón
2 → Via libre
>3 → Puertas.

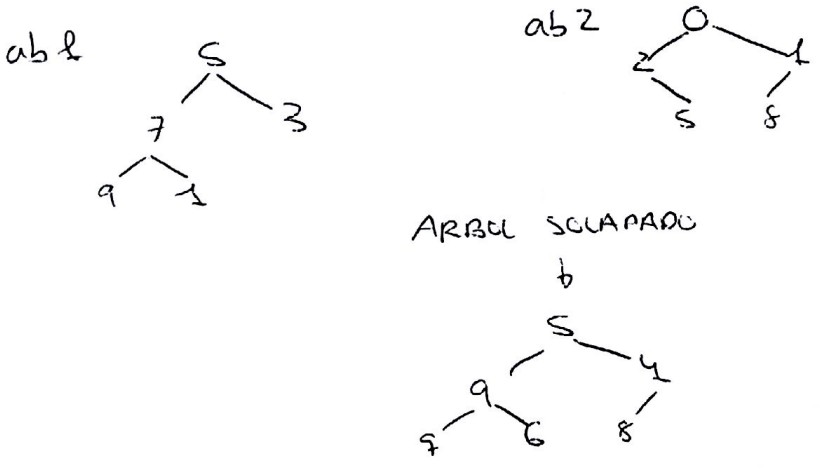
VARIABLES:
- NDragonesMin (int)
- NDragonesAct (int)
- PuertaMejor (int)

```
void elegirPuerta(const AB<int> &mapa, int &puerta){
        ElijePuertaAux(mapa, INT_MAX, 0, puerta);
}

void elegirPuertaAux(const AB<int> &mapa, int &NDragonesMin, int
&NDragonesActuales, int &puerta){
    if(!mapa.esVacio()){
        if(mapa.raiz() >= 3){ // Es hoja
            if(NDragonesMin > NDragonesAct){
                NDragonesMin = NDragonesAct;
                puerta = mapa.raiz();
            }
        }
        if(mapa.raiz() == 1){ // Es dragon
            NDragonesAct++;
        }
        ElijePuertaAux(mapa.hijoIzq(), NDragonesMin, NDragonesAct, puerta);
        ElijePuertaAux(mapa.hijoDer(), NDragonesMin, NDragonesAct,
                       puerta);
}
```


Scanned by CamScanner

```
if (mapa.raiz() == 1)
        NDragonesAct--;
{
```

Ejercicio 23. Solapar arboles




```
template <class E>
AB<E> solapaAB (const AB<E> &ab1, const AB<E> &ab2) {
    if (ab1.esVacio() && ab2.esVacio()) {
        return AB<E>(); //Devolvemos un vacio que creamos
                        con el constructor.

    }
    if(ab1.esVacio()){ //ab1 es vacio y ab2 no
        return AB<E>(ab2.raiz(), ab2.hijoIzq(), ab2.hijoDer());
    }
    else if (ab2.esVacio()){
        return AB<E>(ab1.raiz(), ab1.hijoIzq(), ab1.hijoDer());

    }
    else {
        return AB<E>(ab1.raiz() + ab2.raiz(),
                solapaAB(ab1.hijoIzq(), ab2.hijoIzq()),
                solapaAB(ab1.hijoDer(), ab2.hijoDer()));
    }
}
```

Scanned by CamScanner