Capítulo 8

Aplicaciones de tipos abstractos de datos[1]

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_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.

[1] 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` 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.

Todas las operaciones son totales. Se puede suponer definidos los tipos `Rio` y `Pantano`.

1.1.1. Solución

Representación:

Se propone utilizar una tabla con clave la identificación de 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 tipo *Rio* representa la identificación de los ríos (para este ejercicio es suficiente con que esta identificación sea el nombre del río) y el tipo *Pantano* representa la identificación de los pantanos (es suficiente con el nombre del pantano).

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

class ConfHidrografica {
  public:
    ConfHidrografica();
    ~ConfHidrografica();
    void an_rio(const Rio& r);
    void an_pantano(const Rio& r, const Pantano& p, int n1, int n2);
    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 transvase(const Rio& r1, const Pantano& p1, const Rio& r2,
                   const Pantano& p2, int n);
  private:
    struct InfoPantano {
       int capacidad;
       int litros_embalsados;
    };
    using InfoRio = unordered_map<Pantano,InfoPantano>;
    unordered_map<Rio, InfoRio> rios;
};
```

Implementación de las operaciones.

Operación *ConfHidrografica*, constructor.

El constructor implícito es suficiente para inicializar la tabla de ríos y pantanos, y tiene coste constante, ya que las tablas que se han estudiado, se crean con un tamaño inicial independiente del número de elementos de la tabla.

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 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.

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                  int n1, int n2){
    if (rios.count(r) && !rios[r].count(p)) {
        // crea la informacion del pantano
        InfoPantano i;
        i.capacidad = n1;
        if (n2 < n1) i.litros_embalsados = n2;
        else i.litros_embalsados = n1;
        // añade informacion al rio
        rios[r][p] = i;
    }
}
```

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 repeticiones. También se usa *insert* para así insertar el pantano solo si esté 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];
```

```
        // crea la informacion del pantano
        InfoPantano infoP;
        infoP.capacidad = n1;
        if (n2 < n1) infoP.litros_embalsados = n2;
        else infoP.litros_embalsados = n1;
        // añade informacion al rio
        infoR.insert({p, infoP});
    }
}
```

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 el código en cuanto a la no repetición de búsquedas:

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                  int n1, int n2){
    unordered_map<Rio,InfoRio>::iterator itR = rios.find(r);
    if (itR != rios.end()) {
        // crea la informacion del pantano
        InfoPantano infoP;
        infoP.capacidad = n1;
        if (n2 < n1) infoP.litros_embalsados = n2;
        else infoP.litros_embalsados = n1;
        // añade informacion al rio
        itR->second.insert({p, infoP});
    }
}
```

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].count(p)) {
        // Añade al pantano
        rios[r][p].litros_embalsados += n;
        if (rios[r][p].litros_embalsados > rios[r][p].capacidad)
            rios[r][p].litros_embalsados = rios[r][p].capacidad;
    }
}
```

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, 11 de las cuales son redundantes y se pueden evitar) gracias al método *find*.

```
void ConfHidrografica::embalsar(const Rio& r, const Pantano& p, int n){
    unordered_map<Rio,InfoRio>::iterator itR = rios.find(r);
    if (itR != rios.end()){
        InfoRio::iterator itP = itR->second.find(p);
        if (itP != itR->second.end()) {
            // Añade al pantano
```

```
            itP->second.litros_embalsados += n;
            if (itP->second.litros_embalsados > itP->second.capacidad)
                itP->second.litros_embalsados = itP->second.capacidad;
    }
}
```

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].count(p))
        n = rios[r][p].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. 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;
    unordered_map<Rio,InfoRio>::const_iterator itR = rios.find(r);
    if (itR != rios.cend()){
        InfoRio::const_iterator itP = itR->second.find(p);
        if (itP != itR->second.cend())
            n = itR->second.litros_embalsados;
    }
    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).at(p).litros_embalsados;
    } 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. Para implementarla se utiliza un iterador sobre la tabla que recorre todos los pantanos del río.

```
int ConfHidrografica::reserva_cuenca(const Rio& r) const {
    int n = 0;
    unordered_map<Rio,InfoRio>::const_iterator itR = rios.find(r);
    if (itR != rios.cend()){
        InfoRio::const_iterator itP = itR->second.cbegin();
        while (itP != itR->second.cend()) {
```

```
            n += itP->second.litros_embalsados;
            ++it;
        }//while
    }//if
    return n;
}
```

Coste de la operación implementada:

- El coste del método *find* es constante (en promedio).
- 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. Es importante hacer notar que se podría obtener coste constante en esta operación simplemente guardando y manteniendo un número entero en el tipo *InfoRio* con la suma de los Hm^3 embalsados en todos los pantanos de la cuenca del rio. Este dato se tendría que actualizar correspondientemente en las operaciones *embalsar* y *transvase*. No se ha incluido en la implementación para hacer notar al lector este aspecto explícitamente.

Operación *transvase*.

La operación realiza el transvase 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::transvase(const Rio& r1, const Pantano& p1,
                                 const Rio& r2, const Pantano& p2,
                                 int n){
  if (rios.count(r1) && rios.count(r2) &&
      rios[r1].count(p1) && rios[r2].count(p2) &&
      rios[r1][p1].litros_embalsados >= n &&
      rios[r2][p2].litros_embalsados + n <= rios[r2][p2].capacidad){
          rios[r1][p1].litros_embalsados -= n;
          rios[r2][p2].litros_embalsados += n;
  }
}
```

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). La siguiente versión optimiza los accesos a las tablas.

```
void ConfHidrografica::transvase(const Rio& r1, const Pantano& p1,
                                 const Rio& r2,const Pantano& p2,int n){
  unordered_map<Rio,InfoRio>::iterator itR1 = rios.find(r1);
  unordered_map<Rio,InfoRio>::iterator itR2 = rios.find(r2);
  if (itR1 != rios.end() && itR2 != rios.end()){
    InfoRio::iterator itP1 = itR1->second.find(p1);
    InfoRio::iterator itP2 = itR2->second.find(p2);
    if (itP1 != itR1->second.end() && itP1 != itR1->second.end()){
      if (itP1->second.litros_embalsados >= n &&
```

```
          itP2->second.litros_embalsados + n <= itP2->second.capacidad){
        itP1->second.litros_embalsados -= n;
        itP2->second.litros_embalsados += n;
      }
}
```

1.2. Motor de búsqueda.

Se desea crear un motor de búsqueda para documentos de texto, de forma que, tras indexar muchos documentos, sea posible solicitar todos aquellos que contengan todas las palabras de una búsqueda (por ejemplo, "elecciones rector ucm"). Implementa las siguientes operaciones en un TAD *Buscador*:

- *crea*: crea un buscador vacío, sin documentos.
- *indexa(d)*: indexa el documento d, que contendría solamente texto, de forma que resulte buscable si la consulta contiene palabras de ese texto; y devuelve un identificador para ese documento, que se usará en los resultados.
- *busca(t)*: devuelve una lista con todos los identificadores de documentos que contienen los términos que aparecen en t, una frase.
- *consulta(i)*: muestra el documento al que se le ha asignado el identificador i.

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 *busca*(), en particular, debe ser lo más eficiente posible, incluso si el número de documentos es muy elevado.

1.2.1. Solución.

Representación:

Una primera aproximación podría, para cada documento, almacenar en un *unordered_set* las palabras que aparecen en él. De esta forma podríamos ver si contiene todas las palabras de una consulta con t palabras en $\mathcal{O}(t)$ operaciones. No obstante, el coste de *busca*() se incrementaría linealmente con el número de documentos d: como para cada uno habría que ver si contiene o no todas las t palabras, el coste total estaría en $\mathcal{O}(d \cdot t)$.

Una forma mucho más eficiente de implementar *busca*() es mantener, para cada palabra (también llamada *término*), el conjunto de identificadores de documentos en los que aparece. A esto se le llama un *índice invertido*, ya que en lugar de ir de documentos a palabras, va de palabras a documentos. El coste de *busca*() en este caso sería de $\mathcal{O}(1)$ para cada una de las t palabras, calculando cada vez la intersección del conjunto de documentos devuelto. Como la intersección de dos conjuntos de d_a y d_b documentos se puede calcular en $\mathcal{O}(\min(d_a, d_b))$, el coste total de *busca*(), asumiendo que ordenemos los t términos de más raros a más frecuentes, está limitado por $\mathcal{O}(d_{min} \cdot t)$, donde d_{min} es el número de documentos que contienen el término más infrecuente (y que usaremos como conjunto de partida, sobre el que ir calculando las sucesivas intersecciones). La mejora de usar un índice invertido y ordenar los resultados por frecuencia creciente se hace más significativa cuantos más

documentos haya, y cuanto más específicas sean las consultas. Así, si buscamos "elecciones rector ucm", hay 10^6 documentos, y el número de documentos con cada término es de 40, 100 y 2000, respectivamente, podremos calcular la intersección en $40 + 40 + 40 = 120$ (o menos, según resultados intermedios) consultas en tabla si empezamos por el término más infrecuente ("elecciones"); podríamos necesitar hasta $2000 + 100 + 40 = 2140$ consultas si lo hacemos todo en orden contrario; y necesitaríamos más de 10^6 consultas si no usásemos un índice invertido.

La definición de las clases queda como:

```
class MotorBusqueda {
private:
    unordered_map<int,string> docs;
    // Podría ser tb vector<string>, mientras no haya eliminaciones

    unordered_map<string,Conjunto> indice;
    int numDocs; // usado para asignar ids crecientes a los documentos

public:
    int indexa(const string& texto);
    list<int> busca(const string& consulta) const;
    string consulta(int docId) const;
};
```

Donde la implementación de *Conjunto*, que representa un conjunto de ids de documentos, es la siguiente:

```
class Conjunto {
private:
    unordered_set<int> elems;

public:
    Conjunto(): {}
    Conjunto(const Conjunto& otro): elems(otro.docs) {}

    Conjunto interseccion(const Conjunto& otro) const {
        Conjunto resultado;
        const Conjunto& min = size < otro.size ? *this : otro;
        const Conjunto& max = size >= otro.size ? *this : otro;
        unordered_set<int>::const_iterator it = min.elems.cbegin();
        for ( ; it != min.elems.cend(); ++it) {
            if (max.elems.count(*it)) {
                resultado.elems.insert(*it);
            }
        }
        return resultado;
    }

    void inserta(int elem) {
        elems.insert(elem);
    }

    list<int> comoLista() const {
        list<int> lista;
        unordered_set<int>::const_iterator it = elems.cbegin();
        for ( ; it != elems.cend(); ++it) lista.push_back(*it);
        return lista;
```

```
    }

    unsigned int size() const {
        return elems.size();
    }
};
```

Y las de Buscador quedan como sigue:

```
int indexa(const string& texto) {
    int docId = numDocs++;
    docs[docId] = texto; // Si docs fuese vector se haría push_back
    istringstream iss(texto);
    string termino;
    while (iss >> termino) {
        indice[termino].inserta(docId);
    }
    return docId;
}


list<int> busca(const string& consulta) const {
  istringstream iss(consulta);
  string termino;
  bool terminoHuerfano = false;
  list<const Conjunto*> resultados;// * para evitar copias en push_back
  while (iss >> termino && !terminoHuerfano) {
    unordered_map<string,Conjunto>::const_iterator itT = indice.find(termino);
    if (itT == indice.cend()) terminoHuerfano = true;
    else {
      const Conjunto* c = &(itT->second);
      if (resultados.empty() || c->size() < resultados.front()->size())
        resultados.push_front(c);
      else
        resultados.push_back(c);
    }//else
  }//while
  // el conjunto mas pequeño queda colocado al principio de la lista
  if (terminoHuerfano) return list<int>();
  else {
    Conjunto c = *(resultados.front());
    resultados.pop_front();
    while (!resultados.empty() && c.size() != 0) {
      c = c.interseccion(*(resultados.front()));
      resultados.pop_front();
    }
    return c.comoLista();
  }
}

string consulta(int docId) const {
    return docs.at(docId);
}
```

Las complejidades resultantes son $\mathcal{O}(|texto|)$ para *indexa*() (hace falta una inserción de $\mathcal{O}(1)$ para cada palabra del texto a indexar), y $\mathcal{O}(1)$ para *consulta*(), además del

anteriormente mencionado $\mathcal{O}(d_{min} \cdot t)$ para *busca*() con t términos.

1.3. Agencia de viajes.

(Obtenido del examen extraordinario Febrero 2010)

Se desea definir un tipo abstracto de datos *Agencia* que representa a una agencia hotelera. Dicho TAD debe 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 ordenada de todos los hoteles que están dados de alta en la agencia.
- *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.

1.3.1. 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 $\mathcal{O}(n \log n)$

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;
using InfoHotel = unordered_set<Cliente>;

class Agencia{
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 ;
   list<Ciente> huespedes(const Hotel& h) const;
private:
   unordered_map<Cliente,Hotel> clientes;
   map<Hotel,InfoHotel> hoteles;
};
```

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) {
    unordered_map<Cliente,Hotel>::iterator 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 (suponiendo un árbol equilibrado).

El coste de la operación es por tanto logarítmico en 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 Agencia::aloja(const Cliente& c, const Hotel& h) {
    pair<unordered_map<Cliente,Hotel>::iterator, bool> ret =
                        clientes.insert({c, h});
    if (!ret.second){
        // Quitamos c del hotel en el que estaba
```

```
        hoteles[ret.first->second].erase(c);
        ret.first->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){
    unordered_map<Cliente,Hotel>::iterator 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 logarítmico en 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 ENoExisteCliente();
    }
}
```

El coste de la operación es el coste de consultar un cliente en la tabla (operación *at*). 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 éste es en inorden (lo que produce un recorrido ordenado por nombre de hotel).

```
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;
}
```

Sea H el número de hoteles dados de alta. El coste de la operación $++$ del iterador del map es logarítimo en H, y el bucle se ejecuta H veces. Esto, en principio, nos daría un coste $\mathcal{O}(H * log(H))$. 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 indica que el coste es en realidad $\mathcal{O}(H)$.

Operación *huespedes*

```
list<Cliente> Agencia::huespedes(const Hotel& h) const{
   list<Cliente> l;
   map<Hotel,InfoHotel>::const_iterator itH = hoteles.find(h);
   if (itH != hoteles.end()){
```

```
        for (auto itC = itH->second.cbegin(); itH->second.cend(); ++itC)
            l.push_back(itC->first);
    }
    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 el coste de generar 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.

1.4. 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. Suponemos que contamos con un TAD *libro* que representa la clave única para identificar 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()*: 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()*: 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, 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.
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()*: 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.

1.4.1. 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:

```
using Libro = string;

class EReader {
    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);
        bool esta_libro(const Libro& x) const;
        list<Libro> recientes(int n) const;
        int num_libros() const;
    private:
        struct InfoLibro {
            int totalPags;
            int pagActual;
            bool abierto;
        };
        unordered_map<Libro,InfoLibro> libros;
        list<Libro> secAbiertos;
        int cantidad;
};
```

Implementación de las operaciones:

Operación *e-reader*, constructora
Solo es necesario inicializar la variable entera.

```
EReader::EReader():cantidad(0){ }
```

El coste de la operación es constante.

Operación *poner_libro*.

```
void EReader::poner_libro(const Libro& x, int n){
    // si el libro ya esta o el numero de paginas
    // es negativo no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro i = {n,1,false};
        libros.insert({x, i}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

El coste de la operación es el siguiente:

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

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

Operación *abrir*.

```
void EReader::abrir(const Libro& x){
    if (!libros.count(x)) throw ENoExiste();
    else {// El libro esta en la tabla, consulta su informacion
        InfoLibro& infoL = libros[x];
        if (infoL.abierto) {// Si esta abierto lo borra de su posicion
            list<Libro>::iterator it = secAbiertos.begin();
            while ((*it) != x) ++it;
            secAbiertos.erase(it);
        }
        else { // Si no esta abierto cambia su estado a abierto
            infoL.abierto = true;
        }
        // inserta el libro al comienzo de la secuencia
        secAbiertos.push_front(x);
    }
}
```

El coste de la operación es el siguiente:

- El coste de consultar un libro en la tabla es constante (operaciones *count* y []). Se podría evitar usando *find* y el iterador en lugar de *count* y *[]*.

- 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 eliminar un elemento de la lista a través del iterador es constante (operación *erase*).
- El coste de añadir un elemento al principio de una lista es constante (operación *push_front*).

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

Operación *avanzar_pag*.

```
void EReader::avanzar_pag(){
    // Si no hay ningun libro abierto produce error
    if (secAbiertos.empty()) throw ENoabiertos();
    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.totalPags)
            infoL.pagActual = 1;
    }
}
```

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 *empty* y *front*).
- El coste de consultar y modificar el valor en una tabla es constante (operación []).

Operación *abierto*.

```
const Libro& EReader::abierto() const{
    if (secAbiertos.empty()) throw ENoabiertos();
    else return secAbiertos.front();
}
```

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*.

```
int EReader::pag_libro(const Libro& x) const{
    if (!libros.count(x)) throw ENoExiste();
    else {
        return libros[x].pagActual;
    }
}
```

El coste de la operación es constante, ya que el coste de consultar en una tabla (operaciones *count* y *at*) es constante. De nuevo se podría optimizar usando *find*.

Operación *elim_libro*.

```
void EReader::elim_libro(const Libro& x){
   auto itL = libros.find(x);
   if (itL == libros.end()) throw ENoExiste();
   else {
       InfoLibro infoL = it->second;
       libros.erase(x);
       // En STL se puede hacer más eficiente con libros.erase(itL)
       cantidad--;
       // Si el libro esta abierto lo borra de la lista
       if (infoL.abierto) {
           list<Libro>::iterator it = secAbiertos.begin();
           while ((*it) != x)
               ++it;
           secAbiertos.erase(it);
       }
   }
}
```

El coste de la operación es el siguiente:

- El coste de consultar y borrar en una tabla es constante (operaciones *find* y *erase*).
- El coste de recorrer la lista para eliminar el libro si está 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 *erase*).

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

Operación *esta_libro*.

```
bool EReader::esta_libro(const Libro& x) const{
    return libros.count(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*.

```
list<Libro> recientes(int n) const{
    list<Libro> l;
    int cont = 0;
    list<Libro>::const_iterator it = secAbiertos.cbegin();
    while (it != secAbiertos.cend() && cont < n) {
        l.push_back(*it);
        ++it;
        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 *end*,

añadir un elemento al final de una lista *push_back*, consultar el elemento apuntado por un iterador, y avanzar un iterador.

Operación *num_libros*.

```
int EReader::num_libros() const{
    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):

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 más 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 también 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.

```
class EReader {
    public:
        // mismas operaciones que en el caso anterior
        ...
    private:
       struct InfoLibro {
           int totalPags;
           int pagActual;
           int numAbierto; // 0 indica que no se ha abierto todavia
       };
       unordered_map<Libro,InfoLibro> libros;
       Libro ultimoAbierto;
       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.

```
EReader::EReader(): cantidad(0), acumulador(1) { }
```

El coste es constante.

Operación *poner_libro*

La única modificación es la inicialización de la variable `numAbierto` a cero, en lugar de la variable booleana existente en la otra representación.

```
void EReader::poner_libro(const Libro& x,int n){
    // si el libro ya esta o el numero de paginas es negativo
    // no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro i = {n,1,0};
        libros.insert({x, i}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

La operación consulta y modifica 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 `ultimoAbierto` 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.

```
void EReader::abrir(const Libro& x){
    if (!libros.count(x)) throw ENoExiste();
    else {// El libro esta en la tabla, consulta su informacion
        InfoLibro& infoL = libros[x];
        // Pone el libro como ultimo abierto y actualiza su contador
        ultimoAbierto = x;
        infoL.numAbierto = acumulador;
        acumulador++;
    }
}
```

La operación consulta y modifica información 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 hacía en la representación anterior.

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

```
void EReader::avanzar_pag(){
    // Si no hay ningun libro abierto produce error
    if (acumulador == 1) throw ENoabiertos();
    else {
        // Si hay libros abiertos. Obtiene el primero
        Libro l = ultimoAbierto;
        // 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.totalPags)
            infoL.pagActual = 1;
    }
}
```

La operación consulta y modifica información en una tabla. Su coste es constante.

Operación *abierto*

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

```
const Libro& EReader::abierto() const{
   if (acumulador == 1) throw ENoabiertos();
   else return ultimoAbierto;
}
```

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 `ultimoAbierto`. Para ello se recorre la tabla buscando el libro que tenga un valor mayor en su variable `numAbierto` que indica en qué posición fue abierto. Si el único libro que se había abierto era el libro que se está eliminando, el acumulador se pone a uno para indicar que no queda ningún libro abierto.

```
void EReader::elim_libro(const Libro& x){
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    else {
        libros.erase(x);
        cantidad--;
        // Si el libro es el ultimo abierto debe buscarse el anterior
        if (ultimoAbierto == x) {
            unordered_map<Libro,InfoLibro>::const_iterator it = libros.cbegin();
            int aux = 0;
            Libro libro_aux;
            while (it != libros.cend()) {
                if (it->second.numAbierto > aux) {
                    aux = it->second.numAbierto;
                    libro_aux = it->first;
                } // No se puede buscar el valor del acumulador
                // directamente porque se puede haber eliminado el libro
                ++it;
            }
            if (aux != 0) {
                ultimoAbierto = libro_aux;
                acumulador = aux + 1;
            }
            else acumulador = 1;
        }
    }
}
```

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 añadiendo los elementos al principio de la lista resultante en lugar de al final, obtenemos los libros ordenados por el momento en que fueron abiertos de mayor a menor. A este algoritmo de ordenación que consiste en insertar los elementos a ordenar en un árbol y después recorrerlo en inorden se le llama *treesort*.

```
list<Libro> recientes(int n) const{
    list<Libro> l;
    map<int,Libro> aux; // clave el orden de apertura
    Hashmap<Libro>::const_iterator it = libros.cbegin();
    while (it != libros.cend()) {
        if (it->second.numAbierto != 0) // libro abierto
            aux.insert({it->second.numAbierto, it->first});
        ++it;
    }
    int cont = 0;
    map<int,Libro>::const_iterator ita = aux.cbegin();
    while (ita != aux.cend() && cont < n) {
        l.push_front(ita->second);
        ++ita;
        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 $\mathcal{O}(n_1 \log n_1)$.

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 guardamos junto a la información de cada libro un nuevo campo con un iterador a su posición en la lista de libros abiertos. Así, al abrir un libro, en lugar de recorrer la lista para cambiar su posición, lo borramos en tiempo constante mediante el operador `erase` de la clase `list`. La operación de eliminar un libro tendría también coste constante (también se borraría el libro de la lista de abiertos mediante el método `erase` usando el iterador).

La declaración de la clase sería la siguiente:

```
class EReader {
    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);
        bool esta_libro(const Libro& x) const;
        list<Libro> recientes(int n) const;
        int num_libros() const;
    private:
        struct InfoLibro {
            int totalPags;
            int pagActual;
            list<Libro>::iterator posSecAbiertos;
        };
        unordered_map<Libro,InfoLibro> libros;
        list<Libro> secAbiertos;
        int cantidad;
};
```

Operación *poner_libro*.

```
void EReader::poner_libro(const Libro& x, int n){
    // si el libro ya esta o el numero de paginas es
    // negativo no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro infoL = {n,1,secAbiertos.end()};
        libros.insert({x, infoL}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

El iterador a la lista de abiertos se inicializa a `secAbiertos.end()` para indicar que el libro no se encuentra en la lista. El coste de la operación es constante, no cambia respecto a la anterior representación.

Operación *abrir*.

```
void EReader::abrir(const Libro& x){
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    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);
        }
        // Añade el libro al principio de la lista y guarda el iterador
        infoL.posSecAbiertos = secAbiertos.insert(secAbiertos.begin(),x);
    }
```

```
}
```

El recorrido de la lista de libros abiertos se ha sustituido por la llamada al método `erase` cuyo coste es constante. El coste de la operación es por tanto constante.

Operación *elim_libro*.

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

De nuevo la búsqueda para encontrar el libro en la lista ya no es necesaria gracias al iterador. El coste es por tanto constante.

La implementación del resto de operaciones es la misma que en la primera representación.

Comparación de los costes obtenidos con ambas representaciones

	Primera repr.	Segunda repr.	Tercera repr.
Constructora	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
poner_libro	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
abrir	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
avanzar_pag	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
abierto	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
pag_libro	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
elim_libro	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
esta_libro	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
recientes	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
num_libros	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

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.

b) 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.

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.

5. MultaMatic (EDA - Junio 2013)

   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.

6. BarcoMatic (EDA - Septiembre 2013)

   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.

7. Complejidad Instantánea (adaptado del repositorio de la UVA, problema 586)

   Es posible calcular la complejidad de algunos programas de forma muy fácil, siempre y cuando no haya condicionales. El lenguaje "VeryBasic", que no tiene condicionales, es ideal para esto. La gramática de este lenguaje es la siguiente:

```
<Program> ::= "BEGIN" <Statementlist> "END"
<Statementlist> ::= <Statement> | <Statement> <Statementlist>
<Statement> ::= <LOOP-Statement> | <OP-Statement>
<LOOP-Statement> ::= <LOOP-Header> <Statementlist> "END"
<LOOP-Header> ::= "LOOP" <number> | "LOOP n"
<OP-Statement> ::= "OP" <number>
```

   Así, el siguiente programa tiene complejidad $n^2 + 1997$, porque `OP` tiene el coste que indica el número que le sigue, y estar dentro de un bucle con n repeticiones multiplica el coste de lo que tiene dentro por n:

```
BEGIN
   OP 1997
   LOOP n
      LOOP n
         OP 1
      END
   END
END
```

   Escribe un programa que, dado un programa "VeryBasic", devuelva la complejidad de este programa. Probablemente debas implementar una clase *Polinomio* que te ayude a modularizar esta tarea (puedes probarlo en el juez de la UVA[2]).

[2] http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=527