Capítulo 7

Diccionarios[1]

If debugging is the process of removing bugs,
then programming must be the process of putting
them in.

Edsger W. Dijkstra

Resumen: En este tema se introducen los diccionarios y se presentan dos implementaciones distintas. La primera utiliza árboles de búsqueda, mientras que la segunda se basa en tablas dispersas abiertas. También estudiaremos como recorrer los datos almacenados en la tabla usando iteradores, y algunas propiedades deseables en las funciones de localización.

1. Motivación

Podemos ver un texto como una secuencia (lista) de palabras, donde cada una de ellas es un `string`. El problema de las *concordancias* consiste en contar el número de veces que aparece cada palabra en ese texto.

Implementar una función que reciba un texto como lista de las palabras (en forma de cadena) y escriba una línea por cada palabra distinta del texto donde se indique la palabra y el número de veces que aparece. La lista debe aparecer ordenada alfabéticamente.

2. Introducción

Los diccionarios (en inglés *maps*) son contenedores que almacenan colecciones de pares *(clave, valor)*, y que permiten acceder a los valores a partir de sus claves asociadas. Podemos verlas como una *extensión* de los arrays incorporados en los lenguajes de programación pero en los que los índices en vez de estar acotados por un rango de enteros determinado (en lenguajes como C/C++, Java o C# entre `0` y `n-1`, siendo `n` el tamaño del array) permiten indicar un tipo de índice distinto a los enteros y cuyo rango de valores tampoco está acotado.

De ahí es de donde surge la idea de *diccionario*, en donde hay una *entrada* para cada una de las palabras contenidas en él (de tipo cadena, y no un simple entero) y cuyo valor es, por ejemplo, una lista con las distintas definiciones.

[1] Marco Antonio Gómez y Antonio Sánchez Ruiz-Granados son los autores principales de este tema.

Es por esto que los diccionarios están parametrizados por *dos* tipos distintos: el utilizado para la clave y el utilizado para el valor.

Aunque la sintáxis concreta puede variar, conceptualmente, pues, un diccionario es como un array donde se puede utilizar como índices otra cosa distinta a enteros:

```
map<string, list<string>> v;

list<string> definiciones;
definiciones.push_back("Libro en el que se recogen...");
definiciones.push_back("Catálogo numeroso de noticias...");

v["diccionario"] = definiciones;

cout << v["diccionario"].front() << endl;
```

En este tema veremos dos implementaciones distintas, una basada en árboles binarios (los conocidos como *árboles de búsqueda*) y otra basada en tablas dispersas (*hash tables*). Ambas tienen complejidades mejores que $\mathcal{O}(n)$, pero ambas exigen ciertas condiciones a los tipos que pueden utilizarse como clave.

Como última aclaración antes de empezar, decir que a pesar de que en el tema veremos dos implementaciones distintas del TAD `map`, cada una de ellas será independiente. Es decir no nos planteamos aquí la posibilidad de que exista una clase abstracta a modo de interfaz `map` de la que hereden ambas o algún otro diseño de clases que venga a reflejar que ambas implementaciones representan el mismo TAD de origen.

2.1. Especificación

Desde un punto de vista matemático, los diccionarios son aplicaciones $t : Key \to Value$ que asocian a cada clave $c \in Key$ un determinado valor $v \in Value$. Key y $Value$ serán parámetros del tipo (que proporcionaremos en la plantilla cuando lo implementemos en C++).

Las operaciones publicas del TAD `map` son:

- $Constructora :\to map$ [Generadora]

  Construye un diccionario vacío, es decir, sin elementos.

- $insert : map, Key, Value \to map$ [Generadora]

  Añade un nuevo par (clave, valor) al diccionario. Si la clave ya existía en el diccionario inicial, se sobrescribe su valor asociado. Es decir, los diccionarios *no* permiten almacenar más de un valor por cada clave. Si se desea algo así, o bien se utiliza otro TAD distinto o bien se utiliza una lista de valores como tipo para el valor del diccionario igual que se hizo en el ejemplo del apartado 2.

- $erase : map, Key \to map$ [Modificadora]

  Elimina un par a partir de la clave proporcionada. Si el diccionario no contiene ningún par con dicha clave, no se modifica.

- $contains : map, Key \to bool$ [Observadora]

  Permite averiguar si la clave está o no en el diccionario (es decir, si tiene un valor asociado).

- $at : map, Key \rightarrow Value$ [Observadora parcial]

  Devuelve el valor asociado a la clave proporcionada, siempre que la clave exista en el diccionario.

- $empty : map \rightarrow bool$ [Observadora]

  Indica si el diccionario está vacío o no.

- $size : map \rightarrow int$ [Observadora]

  Indica si el diccionario está vacío o no.

Las operaciones generadoras presentan una peculiaridad que no ha aparecido hasta ahora: un `insert` puede *anular* el resultado de un `insert` anterior. Eso implica que hay *más de una forma* de construir el mismo diccionario.

2.2. Implementación con acceso basado en búsqueda

Usando las estructuras de datos que ya conocemos, podemos implementar las tablas como colecciones de parejas *(clave, valor)*, en las que el acceso por clave se implementa mediante una búsqueda en la estructura correspondiente. A continuación planteamos dos posibles implementaciones usando listas y árboles de búsqueda.

Una manera sencilla de implementar las tablas es mediante una lista de pares *(clave, valor)*. Dependiendo de si la lista está ordenada o no, tendríamos los siguientes costes asociados a las operaciones:

Operación	Lista desordenada	Lista ordenada basada en vectores
Constructora	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert	$\mathcal{O}(n)$	$\mathcal{O}(n)$
erase	$\mathcal{O}(n)$	$\mathcal{O}(n)$
contains	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
at	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
empty	$\mathcal{O}(1)$	$\mathcal{O}(1)$

La operación más habitual cuando se utilizan tablas es *at* (o alguna de sus versiones), que consulta el valor asociado a una clave, por lo que usar una implementación basada en una lista desordenada no parece la elección más acertada. Aún así, incluimos esta implementación en la tabla por su simplicidad y como base con la que poder comparar.

Un dato que puede llamar la atención es el coste lineal de la operación *inserta* cuando usamos listas desordenadas. Este coste se produce porque antes de insertar el nuevo par (clave, valor) es necesario comprobar si la clave ya estaba en la tabla para, en ese caso, modificar su valor asociado. La operación de inserción tiene coste $\mathcal{O}(n)$ porque la búsqueda tiene coste $\mathcal{O}(n)$.

Podemos mejorar el coste de las operaciones usando una lista ordenada basada en vectores. Con esta nueva implementación las operaciones *at* y *contains* pasan a ser logarítmicas, ya que se pueden resolver usando búsqueda binaria. Sin embargo, las operaciones *insert* y *erase* siguen siendo lineales, ya que para insertar o borrar un elemento en un vector debemos desplazar todos los que hay a la derecha.

¿Mejorarían los costes si usamos una lista enlazada ordenada? Pues en realidad no, porque el algoritmo de búsqueda binaria no se puede aplicar a listas enlazadas, ya que necesita acceder al elemento central de un intervalo en tiempo constante.

Se necesita, pues, otra estrategia distinta que permita hacer reducir la complejidad de las operaciones.

3. Árboles de búsqueda

La implementación de los diccionarios mediante lo que se conoce como árboles de búsqueda intenta conseguir las ventajas de la búsqueda binaria (y sus complejidades logarítmicas) eliminando las desventajas de las inserciones lineales. Para eso *evita* almacenar todos los elementos seguidos en memoria.

Para ser capaces de entenderla tenemos que dar primero un pequeño paso atrás y hablar de los árboles binarios ordenados. En el ejercicio 9 del tema anterior nos pedían implementar la siguiente función en los árboles binarios:

```
/**
Devuelve true si el árbol binario cumple las propiedades
de los árboles ordenados: la raiz es mayor que todos los elementos
del hijo izquierdo y menor que los del hijo derecho y tanto el
hijo izquierdo como el derecho son ordenados.
*/
template <class T>
bool bintree::esOrdenado() const;
```

Para que ésta operación tenga sentido, es evidente que el tipo `T` debe poderse ordenar (en C++ eso se traduce a que tienen implementada la comparación mediante el operador $<$). Entenderemos que un árbol está ordenado si su recorrido en inorden está ordenado en orden estrictamente mayor[2]. De lo anterior se deduce inmediatamente que la raíz del árbol es mayor que todos los elementos del hijo izquierdo y menor que todos los elementos del hijo derecho; además tanto el hijo izquierdo como el hijo derecho deben estar, a su vez, ordenados.

Con un árbol ordenado, saber si un elemento está en el árbol no requiere un recorrido por todos los nodos del mismo, sino un recorrido que recuerda a la búsqueda binaria. En efecto, durante el proceso de búsqueda se mira si la raíz contiene el elemento y en caso contrario, se busca únicamente en el hijo izquierdo o derecho dependiendo del resultado de la comparación del elemento buscado y el contenido en la raíz.

La ventaja de los árboles frente a los vectores ordenados, además, es que la inserción de un nuevo elemento *no* tiene coste lineal, pues no necesitaremos desplazar todos los elementos para hacer hueco en el vector; basta con encontrar en qué lugar del árbol debe ir el elemento para mantener el árbol ordenado y crear el nuevo nodo. Por su parte el borrado, aunque más difícil de ver de forma intuitiva, también presenta un escenario similar (no hay que desplazar todos los elementos a la izquierda para "borrar" el elemento).

Parece fácil ver que si los elementos están distribuidos uniformemente, es decir si cada nodo tiene aproximadamente el mismo número de nodos en el subárbol izquierdo y el subárbol derecho, la talla del árbol es del orden de $\log(n)$. Eso, unido al hecho de que las operaciones anteriores lo que hacen es *descender* por el árbol en busca del elemento, hace que bajo la premisa de un árbol balanceado, éstas tengan coste logarítmico (igual que en búsquedas binarias sobre vectores ordenados, pues al fin y al cabo cada vez que descendemos por uno de los lados del árbol dejamos atrás la mitad de los elementos).

Los árboles de búsqueda consisten en utilizar esta misma idea para la implementación de los diccionarios: en vez de almacenar un único elemento en el nodo, utilizaremos dos. Por un lado el valor que utilizaremos para ordenar (que hace las veces de *clave*) y por otro la información adicional asociada a ella (el *valor*). En cada nodo guardaremos, pues, una *pareja*: la clave (por la que se ordena) y su valor asociado. Eso implica automáticamente que

[2]Eso implica que no permitimos la aparición de elementos repetidos; existen otras variaciones de estos árboles que sí lo hacen.

los árboles de búsqueda están parametrizados, al menos, *por dos tipos distintos* en vez de sólo por uno: el tipo de la clave (que debe poderse ordenar) y el tipo del valor. Añadiremos un tercer tipo (opcional) a la plantilla para poder proporcionar un comparador (objeto cuya único método implemente la operación de orden entre claves).

3.1. Implementación

La implementación del diccionario utilizando árboles de búsqueda la llamaremos directamente `map` para coincidir con el nombre del tipo análogo de la STL de C++ [3]. Se basa en tener una estructura jerárquica de nodos, igual que los árboles binarios del tema anterior. La diferencia fundamental es que en esta ocasión esos nodos guardan dos elementos y que no es necesario hacer uso de punteros inteligentes:

```
template <class Clave, class Valor, class Comparador = std::less<Clave>>
class map {
public:
   // parejas <clave, valor> mediante std::pair
   using clave_valor = std::pair<const Clave, Valor>;
protected:
   using map_t = map<Clave, Valor, Comparador>; // Alias para acortar

   /*
    Clase nodo que almacena internamente la pareja <clave, valor>
    y punteros al hijo izquierdo y al hijo derecho.
    */
   struct TreeNode;
   using Link = TreeNode *;
   struct TreeNode {
      clave_valor cv;
      Link iz, dr;
      TreeNode(clave_valor const& e, Link i = nullptr, Link d = nullptr)
                     : cv(e), iz(i), dr(d) {}
   };

   // puntero a la raíz de la estructura jerárquica de nodos
   Link raiz;

   // número de parejas <clave, valor>
   int nelems;

   // objeto función que compara elementos
   Comparador menor;
};
```

Las operaciones generales que vimos para los árboles binarios siguen siendo válidas (la liberación, copia, etc.), pero extendiéndolas cuando corresponda para que tengan en cuenta la existencia de dos valores en vez de sólo uno.

El invariante de la representación del árbol de búsqueda exige lo mismo que en el caso de los árboles binarios (añadiendo que se cumpla el invariante de la representación tanto de la clave como del valor), y que se mantenga el orden de las claves[4]:

[3] `TreeMap` es otro nombre comúnmente utilizado para estos diccionarios, por ejemplo, en Java.
[4] La definición podría haberse hecho también en base al recorrido en inorden.

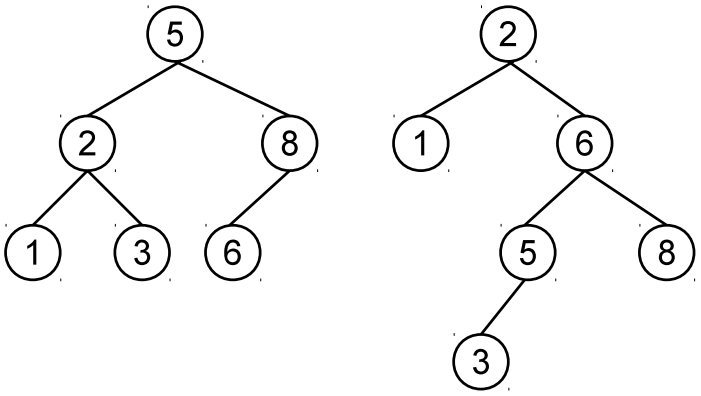


Figura 1: Dos árboles de búsqueda distintos pero equivalentes.

$$\begin{array}{l} \quad R_{map_{(C,V)}}(\mathsf{p}) \\ \Longleftrightarrow_{def} \\ \quad R_{bintree_{Pareja(C,V)}}(\mathsf{p})^{5} \wedge \\ \quad ordenado(\mathsf{p.raiz}) \end{array}$$

donde

$$\begin{array}{lll} ordenado(\mathsf{ptr}) = true & & \textbf{si} \;\; \mathsf{ptr} = \mathsf{null} \\ ordenado(\mathsf{ptr}) = \forall c \in claves(\mathsf{ptr.iz}) : c < \mathsf{ptr.cv.first} \wedge & & \\ \qquad \forall c \in claves(\mathsf{ptr.dr}) : \mathsf{ptr.cv.first} < c \wedge & & \textbf{si} \;\; \mathsf{ptr} \neq \mathsf{null} \\ \qquad ordenado(\mathsf{ptr.iz}) \wedge ordenado(\mathsf{ptr.dr}) & & \end{array}$$

$$\begin{array}{ll} claves(\mathsf{ptr}) = \emptyset & \textbf{si} \;\; \mathsf{ptr} = \mathsf{null} \\ claves(\mathsf{ptr}) = \{\mathsf{ptr.cv.first}\} \bigcup claves(\mathsf{ptr.iz}) \bigcup claves(\mathsf{ptr.dr}) & \textbf{si} \;\; \mathsf{ptr} \neq \mathsf{null} \end{array}$$

La relación de equivalencia, sin embargo, no sigue la misma idea que en los árboles binarios. Para entender por qué basta ver los dos árboles de búsqueda de la figura 1, que guardan la misma información pero tienen una estructura arbórea totalmente distinta.

En realidad dos árboles de búsqueda son equivalentes si almacenan los mismos elementos. Los árboles de búsqueda tienen la misma interfaz que los conjuntos, y su relación de equivalencia también es la misma:

$$\begin{array}{l} \quad \mathsf{a1} \equiv_{map_T} \mathsf{a2} \\ \Longleftrightarrow_{def} \\ \quad elementos(\mathsf{a1}) \equiv_{Conjunto_{Pareja(C,V)}} elementos(\mathsf{a2}) \end{array}$$

donde *elementos* devuelve un conjunto con todos los pares (clave, valor) contenidos en el árbol.

[5] Extendido para considerar claves y valores; el $Pareja(C, V)$ viene a indicarlo.

3.1.1. Implementación de la constructora y de las observadoras `empty` y `size`

El árbol de búsqueda se implementa, igual que los árboles binarios, guardando un puntero a la raíz. Por lo tanto el constructor sin parámetros para generar un árbol de búsqueda vacío lo inicializa a `nullptr`. Las operaciones `empty` y `size` se implementan por tanto de manera trivial:

```
// constructor (diccionario vacío)
map(Comparador c = Comparador()) : raiz(nullptr), nelems(0), menor(c) {}

bool empty() const {
   return raiz == nullptr;
}

int size() const {
   return nelems;
}
```

3.1.2. Implementación de `contains` y `at`

Las operaciones observadoras que permiten averiguar si una clave aparece en el árbol o acceder al valor asociado a una clave se basan en un método auxiliar que trabaja con la estructura de nodos y busca el nodo que contiene una clave dada, y que se implementa de manera trivial con recursión. La operación `contains` tomará el nombre de `count` para tener una interfaz equivalente a la operación análoga del TAD `map` de la STL de C++. En lugar de devolver un booleano devuelve un entero que será 1 en caso de encontrarse la clave y 0 en caso contrario (de ahí el nombre de `count`).

```
// Método protegido/privado
/**
Busca una clave en la estructura jerárquica de
nodos cuya raíz se pasa como parámetro, y devuelve
el nodo en el que se encuentra (o nullptr si no está).
*/
Link busca(Clave const& c, Link a) const {
   if (a == nullptr) {
      return nullptr;
   }
   else if (menor(c, a->cv.first)) {
      return busca(c, a->iz);
   }
   else if (menor(a->cv.first, c)) {
      return busca(c, a->dr);
   }
   else { // c == a->cv.first
      return a;
   }
}
```

Con ella las operaciones `count` y `at` son casi inmediatas:

```
int count(Clave const& c) const  {
   return (busca(c, raiz) != nullptr) ? 1 : 0;
}
```

```
Valor const& at(Clave const& c) const {
   Link p = busca(c, raiz);
   if (p == nullptr)
      throw std::out_of_range("La clave no se puede consultar");
   return p->cv.second;
}
```

También incluiremos (al igual que se hace en la clase `map` de la STL de C++) una versión más avanzada de la operación `at`, que se implementa mediante el operador `[]` y que además de permitir modificar el valor asociado en caso de encontrar la clave buscada, inserta un nuevo par con la clave buscada (y un valor construido por defecto) en el caso de no encontrarla.

```
Valor & operator[](Clave const& c) {
   return corchete(c, raiz);
}

Valor & corchete(Clave const& c, Link & a) {
   if (a == nullptr) {
      // se inserta la nueva clave, con un valor por defecto
      a = new TreeNode(clave_valor(c,Valor()));
      ++nelems;
      return a->cv.second;
   }
   else if (menor(c, a->cv.first)) {
      return corchete(c, a->iz);
   }
   else if (menor(a->cv.first, c)) {
      return corchete(c, a->dr);
   }
   else { // la clave ya está, se devuelve el valor asociado
      return a->cv.second;
   }
}
```

3.1.3. Implementación de la inserción

La inserción debe garantizar que:

- Si la clave ya aparecía en el árbol, el valor antiguo se sustituye por el nuevo.
- Tras la inserción, el árbol de búsqueda sigue cumpliendo el invariante de la representación, es decir, sigue estando ordenado por claves.

La implementación debe "encontrar el hueco" en el que se debe crear el nuevo nodo, y si por el camino descubre que la clave ya existía, sustituir su valor.

Lo importante de la implementación es darse cuenta de que la raíz de la estructura jerárquica *puede cambiar*. En concreto, si el árbol de búsqueda estaba vacío, en el momento de insertar el elemento se crea un nuevo nodo que pasa a ser la raíz (de ahí el paso por referencia del puntero al nodo). Teniendo esto en cuenta la implementación (con un método auxiliar recursivo) sale casi sola:

```
// solamente se inserta si la clave no existe ya en el diccionario
bool insert(clave_valor const& cv) {
```

```
      return inserta(cv, raiz);
   }

   // Método protegido
   bool inserta(clave_valor const& cv, Link & a) {
      if (a == nullptr) {
         // se inserta el nuevo par <clave, valor>
         a = new TreeNode(cv);
         ++nelems;
         return true;
      }
      else if (menor(cv.first, a->cv.first)) {
         return inserta(cv, a->iz);
      }
      else if (menor(a->cv.first, cv.first)) {
         return inserta(cv, a->dr);
      }
      else { // la clave ya está
         return false;
      }
   }
```

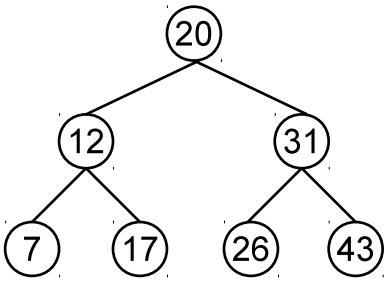
Si suponemos que el árbol está equilibrado, la complejidad del método anterior es logarítmica pues en cada llamada recursiva se divide el tamaño de los datos entre dos.

3.1.4. Implementación del borrado

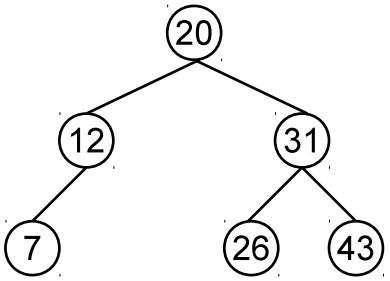
La operación de borrado es más complicada que la de la inserción, porque puede exigir reestructurar los nodos para mantener la estructura ordenada por claves. Si nos piden eliminar la clave c, se busca el nodo que la contiene y:

- Si la búsqueda fracasa (la clave no está), se termina sin modificar el árbol.
- Si la búsqueda tiene éxito, se localiza un nodo α, que es el que hay que borrar. Para hacerlo:
  - Si α es hoja, se puede eliminar directamente (actualizando el puntero del padre).
  - Si α tiene un sólo hijo, se elimina el nodo α y se coloca en su lugar el subárbol hijo cuya raíz quedará en el lugar del nodo α.
  - Si α tiene dos hijos la estrategia que utilizaremos será "subir" el elemento más pequeño del hijo derecho (que no tendrá hijo izquierdo, pues de otra forma no sería el más pequeño) a la raíz. Para eso:
    - Se busca el nodo α' más pequeño del hijo derecho.
    - Si ese nodo tiene hijo derecho, éste pasa a ocupar su lugar.
    - El nodo α' pasa a ocupar el lugar de α, de forma que su hijo izquierdo y derecho cambian a los hijos izquierdo y derecho de la raíz antigua.

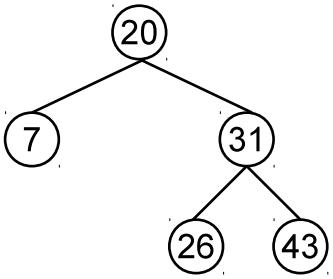
A modo de ejemplo, a continuación mostramos la evolución de un árbol de búsqueda a cuando vamos borrando sus nodos:



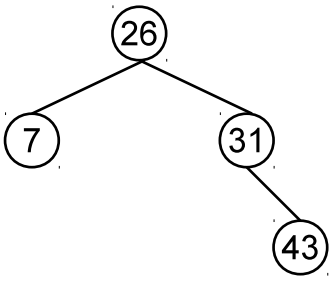
- `a.erase(17)`



- `a.erase(12)`



- `a.erase(20)`



Igual que ocurría con la inserción, la implementación utiliza métodos auxiliares; como la raíz puede cambiar éstos deben pasar el parámetro del puntero al nodo por referencia:

```
bool erase(Clave const& c) {
   return borra(c, raiz);
}
```

```
bool borra(Clave const& c, Link & a)  {
   if (a == nullptr) {
      return false;
   } else if (menor(c, a->cv.first)) {
      return borra(c, a->iz);
   }
   else if (menor(a->cv.first, c)) {
      return borra(c, a->dr);
   }
   else { // c == a->cv.first
      if (a->iz == nullptr || a->dr == nullptr) {
         Link aux = a;
         a = (a->iz == nullptr) ? a->dr : a->iz;
         --nelems;
         delete aux;
      }
      else { // tiene dos hijos
         subirMenorHD(a);
         --nelems;
      }
      return true;
   }
}

static void subirMenorHD(Link & a) {
   Link b = a->dr, padre = nullptr;
   while (b->iz != nullptr) {
      padre = b;
      b = b->iz;
   }
   if (padre != nullptr) {
      padre->iz = b->dr;
      b->dr = a->dr;
   }
   b->iz = a->iz;
   delete a;
   a = b;
}
```

3.2. Coste de las operaciones

Los costes de las operaciones de los diccionarios implementados mediante árboles de búsqueda, si suponemos árboles equilibrados, son los siguientes:

Operación	Árboles de búsqueda
Constructora	$\mathcal{O}(1)$
insert	$\mathcal{O}(\log n)$
count	$\mathcal{O}(\log n)$
at	$\mathcal{O}(\log n)$
erase	$\mathcal{O}(\log n)$
empty	$\mathcal{O}(1)$

Es importante resaltar que para que un tipo pueda utilizarse como clave en un dic-

cionario implementado como árbol de búsqueda, éste debe poseer una relación de orden, ya que los árboles de búsqueda almacenan los elementos ordenados. A cambio, conseguimos que las operaciones sean logarítmicas (si los árboles se mantienen equilibrados). Esta operación de orden puede proporcionarse como tercer parámetro al instanciar el tipo (en la declaración de la variable), o en caso de no proporcionarse se asumirá la existencia del operador de orden $<$ del tipo `Clave`.

Por otro lado, insistimos en que para que esas complejidades sean ciertas, el árbol debe estar equilibrado. Y eso sólo se garantiza si las inserciones y borrados realizados sobre el diccionario son aleatorias. Si el usuario de la clase hace algo como lo siguiente:

```
map<int, int> tablaMultiplicarDel17;
for (int i = 1; i <= 100; ++i)
   tablaMultiplicarDel17.insert({i, 17*i});
```

estará construyendo, seguramente sin darse cuenta, un árbol degenerado e incurrirá en costes lineales en las operaciones de búsqueda sobre él. Las implementaciones reales de librerías de colecciones de datos se basan en estructuras de datos más avanzadas que utilizan las ideas de los árboles de búsqueda vistos aquí, pero que incluyen lógica de *autobalanceo* en las operaciones de inserción y borrado; si se determina que el árbol corre el riesgo de desequilibrarse se reestructura.

3.3. Recorrido de los elementos mediante un iterador

En los árboles de búsqueda podríamos también añadir operaciones para el recorrido de sus elementos (preorden, por niveles, etc.). Sin embargo, dado que el único recorrido que parece tener sentido es el recorrido en inorden (pues las claves salen en orden creciente), vamos a extender la clase añadiendo la posibilidad de recorrerlo mediante un iterador.

El iterador permitirá acceder tanto a la clave como al valor del elemento visitado (devolviendo un par con ambos datos). Igual que en el caso de las listas, tendremos dos versiones del iterador, por un lado el `const_iterator` que no permitirá modificar los datos, y por otro lado el `iterator` que permitirá modificar el *valor* asociado (no se permite cambiar la clave, pues pondría en peligro el invariante de la representación: el árbol podría dejar de estar ordenado).

El recorrido, sin embargo, no es tan fácil como en el caso de las listas, porque averiguar el siguiente elemento a un nodo dado no es directo. En concreto, hay dos casos:

- Cuando el nodo visitado actualmente tiene hijo derecho, el siguiente nodo a visitar será el elemento más pequeño del hijo derecho. Éste se obtiene bajando siempre por la rama de la izquierda hasta llegar a un nodo que no tiene hijo izquierdo.

- Si el nodo visitado no tiene hijo derecho, el siguiente elemento a visitar es el *ascendiente más cercano* que aún no ha sido visitado. Dado que la estructura jerárquica mantiene punteros hacia los hijos pero no hacia los padres, necesitamos una estructura de datos auxiliar. En particular, el iterador mantendrá una *pila* con todos los ascendientes que aún quedan por recorrer. En la búsqueda del hijo más pequeño descrita anteriormente vamos descendiendo por una rama, vamos apilando todos esos descendientes para poder visitarlos después.

El recorrido termina cuando el nodo actual no tiene hijo derecho y la pila queda vacía. En ese caso se cambia el puntero interno a `nullptr` para indicar que estamos "fuera" del recorrido.

```
// iteradores que recorren los pares de menor a mayor clave
template <class Apuntado>
class Iterador {
public:
   Apuntado & operator*() const {
      if (act == nullptr)
         throw std::out_of_range("No hay elemento a consultar");
      return act->cv;
   }

   Iterador & operator++() {  // ++ prefijo
      next();
      return *this;
   }

   bool operator==(Iterador const& that) const {
      return act == that.act;
   }
   bool operator!=(Iterador const& that) const {
      return !(this->operator==(that));
   }

protected:
   friend class map;
   Link act;
   std::stack<Link> ancestros;  // antecesores no visitados

   // construye el iterador al primero
   Iterador(Link raiz) { act = first(raiz); }

   // construye el iterador al último
   Iterador() : act(nullptr) {}

   Link first(Link ptr) {
      if (ptr == nullptr) {
         return nullptr;
      } else { // buscamos el nodo más a la izquierda
         while (ptr->iz != nullptr) {
            ancestros.push(ptr);
            ptr = ptr->iz;
         }
         return ptr;
      }
   }

   void next() {
      if (act == nullptr) {
         throw std::out_of_range("El iterador no puede avanzar");
      } else if (act->dr != nullptr) { // primero del hijo derecho
         act = first(act->dr);
      } else if (ancestros.empty()) { // hemos llegado al final
         act = nullptr;
      } else { // podemos retroceder
         act = ancestros.top();
         ancestros.pop();
```

```
        }
      }
   }; // fin de la clase Iterador

public: // de la clase map

   // iterador que no permite modificar el elemento apuntado
   using const_iterator = Iterador<clave_valor const>;

   const_iterator cbegin() const {
      return const_iterator(raiz);
   }

   const_iterator cend() const {
      return const_iterator();
   }

   // iterador que sí permite modificar el elemento apuntado (su valor)
   using iterator = Iterador<clave_valor>;

   iterator begin() {
      return iterator(raiz);
   }

   iterator end()  {
      return iterator();
   }
```

Las operaciones funcionan de forma similar a lo visto en las listas: la operación `begin` (o `cbegin`) devuelve un iterador al principio del recorrido, mientras que `end` (`cend()`) devuelve un iterador que queda *fuera* del recorrido.

3.4. Búsqueda en el diccionario con iteradores

Un problema de la implementación anterior de los diccionarios es que si un usuario quiere protegerse de los accesos indebidos al llamar al método `at` debe asegurarse primero de que la clave existe utilizando `count`. Eso fuerza a hacer dos búsquedas sobre el árbol. Si, además, ese mismo usuario quiere posteriormente modificar el valor asociado a esa tabla, incurrirá en un nuevo recorrido al llamar a `insert`.

Una solución que reduce los tres recorridos anteriores a uno consiste en añadir un método adicional de *búsqueda* de un elemento, al que llamaremos `find`, que en lugar de devolver el `bool` o el `Valor`, devuelva el *iterador* al punto donde está la clave buscada (o al final del recorrido si no está). Implementaremos la búsqueda propiamente dicha mediante la siguiente constructora en la clase interna Iterador, la cual será invocada desde el nuevo método `find`.

```
   class Iterador {
     ...
     Iterador(map_t const* m, Clave const& c) {
         act = m->raiz;
         bool encontrado = false;
         while (act != nullptr && !encontrado) {
            if (m->menor(c, act->cv.first)) {
               ancestros.push(act);
```

```
            act = act->iz;
         } else if (m->menor(act->cv.first, c)) {
            act = act->dr;
         } else
            encontrado = true;
      }
      if (!encontrado) { // vaciamos la pila
         // act == nullptr
         ancestros = std::stack<Link>();
      }
   }
   ...
}; //fin de la clase Iterador

// En la clase map
const_iterator find(Clave const& c) const {
   return const_iterator(this, c);
 }

iterator find(Clave const& c) {
   return iterator(this, c);
}
```

4. Tablas dispersas

Las tablas dispersas se basan en almacenar en un vector los *valores* y usar las *claves* como índices. De esa forma, dada una clave podemos acceder a la posición del vector que contiene su valor asociado en tiempo constante. Las tablas dispersas permiten implementar todas las operaciones en tiempo $O(1)$ en promedio, aunque en el caso peor *insert*, *count*, *at* y *erase* serán $O(n)$.

¿Cómo asociamos cada posible clave a una posición del vector? Obviamente esta idea no puede aplicarse si el conjunto de claves posible es demasiado grande. Por ejemplo, si usamos como claves cadenas de caracteres con un máximo de 8 caracteres elegidos de un conjunto de 52 caracteres posibles, habría un total de

$$L = \sum i : 1 \leq i \leq 8 : 52^i$$

claves distintas.

Es absolutamente impensable reservar un vector de tamaño L para implementar la tabla del ejemplo anterior, sobre todo si tenemos en cuenta que el conjunto de cadenas que llegará a utilizarse en la práctica será mucho menor. Necesitamos algún mecanismo que permita establecer una correspondencia entre un conjunto de claves potencialmente muy grande y un vector de valores mucho más pequeño.

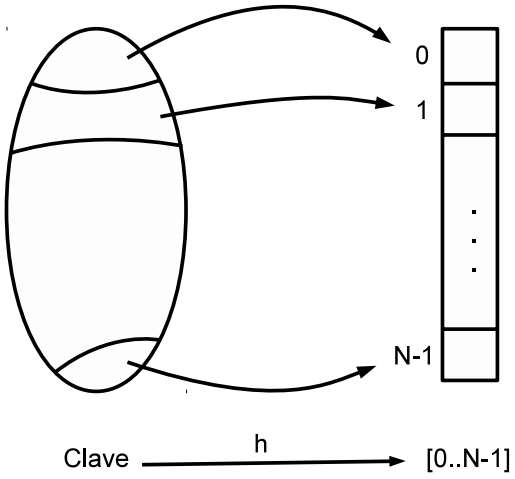
4.1. Función de localización

Lo que sí tiene sentido es reservar un vector de N posiciones para almacenar los valores (siendo N mucho más pequeño que L) y usar una *función de localización* (*hashing function*) que asocia a cada clave un índice del vector

$$h : Clave \rightarrow [0..N-1]$$

de manera que dada una clave c, $h(c)$ represente la posición del vector que debería contener su valor asociado.

Puesto que el número de claves posible, L, es mucho mayor que el número de posiciones del vector, N, la función de localización h no puede ser inyectiva. En otras palabras, existen varias claves distintas que se asocian al mismo índice dentro del vector. Gráficamente la situación es la siguiente:



Para que la búsqueda funcione de manera óptima, las funciones de localización deben tener las siguientes propiedades:

- *Eficiencia*: el coste de calcular $h(c)$ debe ser bajo.
- *Uniformidad*: el reparto de claves entre posiciones del vector debe ser lo más uniforme posible. Idealmente, para una clave c elegida al azar la probabilidad de que $h(c) = i$ debe valer $1/N$ para cada $i \in [0..N-1]$.

Supongamos que tenemos un vector de 16 posiciones ($N = 16$) y que usamos cadenas de caracteres como claves. Una posible función de localización es la siguiente:

$$\text{h(c)} = \text{ord(ult(c))} \bmod 16$$

donde $ult(c)$ devuelve el último carácter de la cadena, y *ord* devuelve el código ASCII de un carácter. Usando esta función de localización tenemos:

$$\begin{aligned}
\text{h("Fred")} &= \text{ord(`d') mod 16} &&= 100 \bmod 16 = 4 \\
\text{h("Joe")} &= \text{ord(`e') mod 16} &&= 101 \bmod 16 = 5 \\
\text{h("John")} &= \text{ord(`n') mod 16} &&= 110 \bmod 16 = 14
\end{aligned}$$

Aunque esta función de localización no es demasiado buena, la seguiremos usando en los siguientes ejemplos debido a su sencillez. Más adelante, en el apartado 6, discutiremos algunas ideas para definir mejores funciones de localización.

4.2. Colisiones

Como ya hemos explicado, en general la función de localización no puede ser inyectiva. Cuando se encuentran dos claves c y c' tales que

$$c \neq c' \wedge h(c) = h(c')$$

se dice que se ha producido una *colisión*. Se dice también que c y c' son *claves sinónimas* con respecto a h.

Es fácil encontrar claves sinónimas con respecto a la función de localización que acabamos de definir:

$$\text{h("Fred")} = \text{h("David")} = \text{h("Violet")} = \text{h(Roland")} = 4$$

En realidad, la probabilidad de que no se produzcan colisiones es mucho más baja de lo que podríamos pensar. Mediante un cálculo de probabilidades puede demostrarse la llamada "paradoja del cumpleaños", que dice que en un grupo de 23 o más personas, la probabilidad de que al menos dos de ellas cumplan años el mismo día del año es mayor que 1/2.

Debemos pensar, por tanto, qué hacer cuando se produzca una colisión. Fundamentalmente, existen dos estrategias que dan lugar a dos tipos de tablas dispersas:

- *Tablas abiertas*: cada posición del vector almacena una lista de parejas (clave, valor) con todos los pares que colisionan en dicha posición.
- *Tablas cerradas*: si al insertar un par (clave,valor) se produce una colisión, se busca otra posición del vector vacía donde almacenarlo. Para ello se van comprobando los índices del vector en algún orden determinado hasta alcanzar alguna posición vacía (técnicas de *relocalización*).

En este tema nos centraremos en el estudio de las tablas dispersas abiertas, pero animamos a los estudiantes que lo deseen a profundizar a través de la lectura de los capítulos correspondientes de los libros (Rodríguez Artalejo et al., 2011) y (Peña, 2005). En realidad, cada tipo de tabla tiene sus ventajas y sus inconvenientes: las tablas abiertas son más sencillas de entender y suelen tener mejor rendimiento, pero también ocupan más memoria que las cerradas.

5. Tablas dispersas abiertas

En las tablas abiertas, cada posición del vector contiene una *lista de colisión* que almacena los pares con claves sinónimas. Nosotros vamos a implementar estas listas de colisión como listas enlazadas en las que cada nodo almacena una clave y un valor.

Llamaremos a la clase `unordered_map` para coincidir con la clase análoga de la STL de C++. En Java por ejemplo se denomina `HashMap` mientras que en C# es `Dictionary`.

La operación *insert* calculará el índice del vector asociado a la nueva clave, y añadirá un nuevo nodo a la lista correspondiente si la clave no existía, o modificará su valor asociado si ya existía. De manera simétrica, la operación de borrado calculará el índice asociado a la clave que recibe como parámetro y a continuación buscará en la lista correspondiente algún nodo que contenga dicha clave para eliminarlo.

Veamos un ejemplo. Supongamos que tenemos una tabla abierta implementada con un vector de 16 posiciones y la función de localización de siempre. Tras realizar las siguientes operaciones:

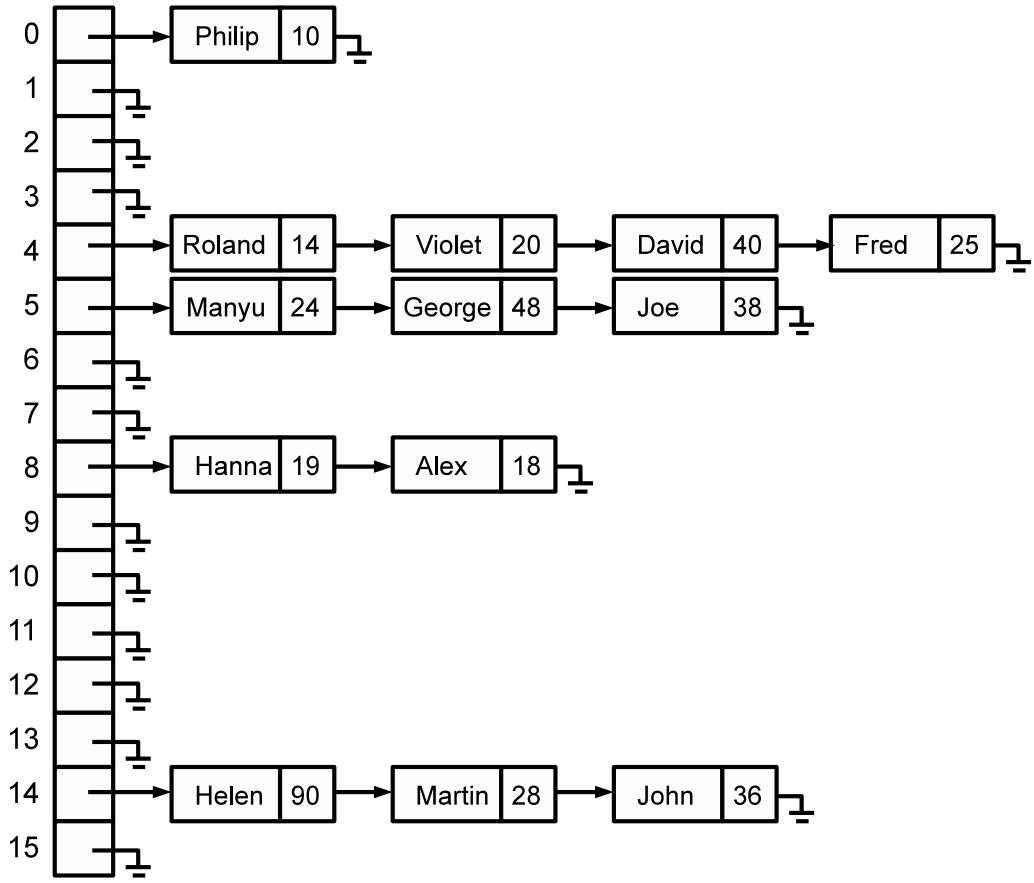


Figura 2: Ejemplo de tabla abierta. La función de localización usada en el ejemplo es claramente mejorable.

```
unordered_map<std::string, int> t;
t.insert({"Fred", 25});      t.insert({"Alex", 18});
t.insert({{"Philip", 10});    t.insert({"Joe", 38});
t.insert({"John", 36});      t.insert({"Hanna", 19});
t.insert({"David", 40});     t.insert({"Martin", 28});
t.insert({"Violet", 20});    t.insert({"George", 48});
t.insert({"Helen", 90});     t.insert({"Manyu", 24});
t.insert({"Roland", 14});
```

el resultado en memoria sería similar al que se muestra en la Figura 2.

Una característica interesante es la *tasa de ocupación* de la tabla, que se define como la relación entre el número de pares almacenados y el número de posiciones del vector. En el ejemplo anterior, la tabla contiene $n = 13$ pares en un vector con $N = 16$ posiciones, por lo que la tasa de ocupación en el estado actual es

$$\alpha = n/N = 13/16 = 0,8125$$

Es evidente que las tablas abiertas pueden llegar a almacenar más de N pares (clave, valor), pero debemos tener en cuenta que si la tasa de ocupación crece excesivamente, la velocidad de las consultas se puede degradar hasta llegar a ser lineal.

Por ejemplo, supongamos que en una tabla con un vector de $N = 16$ posiciones introducimos $n = 16000$ elementos uniformemente distribuidos en el vector. Para consultar el valor asociado a una clave, tendremos que realizar una búsqueda secuencial en una lista de 1000 elementos, una operación con coste $O(n/16) = O(n)$.

5.1. Invariante de la representación

Vamos a implementar las tablas abiertas mediante una plantilla parametrizada con los dos tipos de datos involucrados: el tipo de la *clave* y el tipo del *valor*. Tendremos otros dos parámetros opcionales para proporcionar la función hash y el operador == (en caso de no proporcionarlos se usarán las correspondientes operaciones por defecto para el tipo de las claves). De esa forma podremos crear distintos tipos de tablas de la manera habitual:

```
unordered_map<std::string, int> concordancias;
unordered_map<List<char>, Persona> dniPersonas;
unordered_map<List<char>, List<Libro>> librosPrestados;
```

La única limitación es que necesitamos definir una función de localización adecuada para el tipo de datos usado como clave. Más adelante estudiaremos cómo definir estas funciones de localización, por ahora supondremos que existe una función *hash* que realiza la conversión entre claves e índices del vector.

Como ya hemos explicado, para implementar una tabla abierta necesitamos representar un vector de listas de colisión, que vamos a implementar como listas enlazadas donde cada nodo almacena una clave y su valor asociado. La estructura `ListNode`, como no podría ser de otra manera, será interna a la clase `unordered_map` y almacenará la clave, el valor, y el puntero al siguiente nodo de la lista. La clase `unordered_map`, a su vez, contiene un vector de punteros a nodos (las listas de colisión), y el número de elementos almacenados en la tabla.

```
template <class Clave, class Valor,
          class Hash = std::hash<Clave>,
          class Pred = std::equal_to<Clave>>
class unordered_map {
public:
   // parejas <clave, valor> mediante std::pair
   using clave_valor = std::pair<const Clave, Valor>;

protected:
   using umap_t = unordered_map<Clave, Valor, Hash, Pred>;
   // Alias como shortcut

   /*
    Clase nodo que almacena internamente la pareja <clave, valor>
    y un puntero al siguiente.
    */
   struct ListNode;
   using Link = ListNode *;
   struct ListNode {
      clave_valor cv;
      Link sig;
      ListNode(clave_valor const& e, Link s = nullptr) : cv(e), sig(s) {}
   };

   // vector de listas (el tamaño se ajustará a la carga)
   std::vector<Link> array;
```

```
    static const int TAM_INICIAL = 17; // tamaño inicial de la tabla
    static const int MAX_CARGA = 75;   // máxima ocupación permitida 75%

    // número de parejas <clave, valor>
    int nelems;

    // objeto función para hacer el hash de las claves
    Hash hash;

    // objeto función para comparar claves
    Pred pred;
};
```

Pasamos a continuación a definir el *invariante de la representación*, que debe asegurar que la tabla está bien formada. Decimos que una tabla está bien formada si:

- La variable *nelems* contiene el número de elementos almacenados.
- Las listas de colisión son listas enlazadas de nodos bien formadas.
- La lista de colisión asociada al índice i del vector sólo contiene pares $(clave, valor)$ tales que $h(clave) = i$.
- Ninguna lista de colisión contiene dos pares con la misma clave.

Con estas ideas, podemos formalizar el invariante de la representación de la siguiente manera:

$$R_{Tabla_{C,V}}(\mathsf{t}) \iff_{def} ubicado(\mathsf{t.array}) \wedge \mathsf{t.nelems} = totalElems(\mathsf{t.array}, \mathsf{t.size}()) \wedge \forall i : 0 \leq i < \mathsf{t.size}() : buenaLista(\mathsf{t.array}, i)$$

$$\begin{aligned} buenaLista(\mathsf{v}, \mathsf{i}) &= R_{Lista_{Par(C,V)}}(\mathsf{v}[i]) \wedge buenaLoc(\mathsf{v}[i], i) \wedge claveUnica(\mathsf{v}[i]) \\ buenaLoc(\mathsf{l}, \mathsf{i}) &= \forall j : 0 \leq j < numElems(\mathsf{l}) : h(clave(\mathsf{l}, j)) = \mathsf{i} \\ claveUnica(\mathsf{l}) &= \forall j, k : 0 \leq j < k < numElems(\mathsf{l}) : clave(\mathsf{l}, j) \neq clave(\mathsf{l}, k) \end{aligned}$$

El predicado *ubicado* indica que se ha reservado memoria para el array, y *buenaLista* comprueba que cada una de las listas de colisión está bien formada. Consideramos que una lista está bien formada si es una lista enlazada bien formada, sólo contiene elementos cuya clave se asocia a ese índice del vector, y no contiene elementos con claves repetidas.

Para completar la formalización, definimos las funciones *numElems* que devuelve el número de elementos de una lista, *clave* que devuelve la clave del elemento i-ésimo de la lista, y *totalElems* que devuelve el número de elementos almacenados en la tabla.

$$\begin{aligned} numElems(\mathsf{p}) &= 0 && \textbf{si } \mathsf{p} = \mathsf{nullptr} \\ numElems(\mathsf{p}) &= 1 + numElems(\mathsf{p.sig}) && \textbf{si } \mathsf{p} \neq \mathsf{nullptr} \end{aligned}$$

$$\begin{aligned} clave(\mathsf{p}, \mathsf{i}) &= \mathsf{p.cv.first} && \textbf{si } \mathsf{i} = 0 \\ clave(\mathsf{p}, \mathsf{i}) &= clave(\mathsf{p.sig}, \mathsf{i} - 1) && \textbf{si } \mathsf{i} > 0 \end{aligned}$$

$$totalElems(\mathsf{v}, \mathsf{N}) = \sum i : 0 \leq i < N : numElems(\mathsf{v}[i])$$

Respecto a la relación de equivalencia, al igual que ocurría con el TAD `map`, decimos que dos objetos de tipo `unordered_map` son equivalentes si almacenan el mismo conjunto de pares (clave, valor):

$$\begin{aligned} & \mathsf{t1} \equiv_{unordered\_map_T} \mathsf{t2} \\ \Longleftrightarrow_{def} & \\ & elementos(\mathsf{t1}) \equiv_{Conjunto_{Par(C,V)}} elementos(\mathsf{t2}) \end{aligned}$$

donde *elementos* devuelve un conjunto con todos los pares (clave, valor) contenidos en la tabla.

5.2. Implementación de las operaciones auxiliares

Al igual que en temas anteriores, comenzamos definiendo algunas operaciones auxiliares que serán de utilidad para implementar las operaciones públicas del TAD. En concreto, vamos a definir métodos privados para liberar la memoria reservada por la tabla, y para buscar nodos en una lista enlazada.

Comenzamos con la operación que libera toda la memoria dinámica reservada para almacenar el vector de listas de colisión.

```
void libera() {
   for (int i = 0; i < array.size(); ++i) {
      // liberamos los nodos de la lista array[i]
      Link act = array[i];
      while (act != nullptr) {
         Link a_borrar = act;
         act = act->sig;
         delete a_borrar;
      }
      array[i] = nullptr;
   }
}
```

A continuación se muestra un método auxiliar que permite buscar un nodo con una cierta clave en una lista enlazada. A parte de devolver el puntero apuntando al nodo bucado devuelve un puntero al nodo anterior (necesario para la eliminación pues estamos trabajando con listas enlazadas simples). En ambos casos, si buscamos un nodo con una clave que no existe en la lista enlazada, se devolverá nullptr como nodo encontrado.

```
/**
 * Busca un nodo a partir del nodo "pos" que contenga la clave
 * dada. Si lo encuentra, "pos" quedará apuntando a dicho nodo
 * y "ant" al nodo anterior. Si no lo encuentra "pos" quedará
 * apuntando a nullptr.
 */
  bool localizar(Clave const& c, Link & ant, Link & pos) const {
     ant = nullptr;
     while (pos != nullptr) {
        if (pred(c, pos->cv.first))
           return true;
        else {
```

```
            ant = pos; pos = pos->sig;
         }
      }
      return false;
   }
```

5.3. Implementación de las operaciones públicas

El constructor inicializa el vector con punteros nullptr en todas sus posiciones (representando listas de colisión vacías). El destructor utiliza la operación auxiliar que vimos en el apartado anterior para *liberar* toda la memoria reservada.

```
   unordered_map(int n = TAM_INICIAL, Hash h = Hash(), Pred p = Pred()) :
                        array(n, nullptr), nelems(0), hash(h), pred(p) {}

   ~unordered_map() {
      libera();
   }
```

La siguiente operación que vamos a explicar es la que inserta un nuevo par (clave, valor) en la tabla. Esta operación debe calcular el índice del vector asociado a la clave usando la función de localización, y buscar si la lista enlazada correspondiente ya contiene algún nodo con esa clave. Si ya existía un nodo con esa clave actualiza su valor, y si no, crea un nuevo nodo y lo inserta en la lista. En nuestra implementación insertamos el nuevo nodo por delante, como el primer nodo de la lista.

```
/**
 * Inserta un nuevo par (clave, valor) en la tabla. Si ya existía un
 * elemento con esa clave, actualiza su valor.
 */
   bool insert(clave_valor const& cv) {
      int i = hash(cv.first) % array.size();
      Link ant, pos = array[i];
      if (localizar(cv.first, ant, pos)) { // la clave ya existe
         return false;
      } else {
         array[i] = new ListNode(cv, array[i]);
         ++nelems;
         return true;
      }
   }
```

La operación de borrado es similar a la anterior. Comenzamos usando la función de localización para calcular el índice del vector asociado a la clave. A continuación buscamos un nodo con esa clave en la lista, y si lo encontramos lo eliminamos. Como puede verse en el código, debemos tener especial cuidado con los punteros cuando el nodo a eliminar es el primero de la lista.

```
   /**
   * Elimina el elemento de la tabla con la clave dada. Si no existía ningún
   * elemento con dicha clave, la tabla no se modifica.
   */
    bool erase(Clave const& c) {
      int i = hash(c) % array.size();
      Link ant, pos = array[i];
```

```
      if (localizar(c, ant, pos)) {
         if (ant == nullptr)
            array[i] = pos->sig;
         else
            ant->sig = pos->sig;
         delete pos;
         --nelems;
         return true;
      } else
         return false;
   }
```

La operación *count* es muy sencilla de implementar: usamos la función de localización para calcular el índice del vector que podría contener la clave y a continuación realizamos la búsqueda dentro de la lista enlazada de nodos.

```
/**
* Operación observadora que busca la clave c y devuelve
* un 1 si la encuentra y un 0 si no lo hace.
*/
int count(Clave const& c) const  {
    int i = hash(c) % array.size();
    Link ant, pos = array[i];
    return localizar(c, ant, pos) ? 1 : 0;
}
```

La operación quizás más importante de las tablas es *at*, que devuelve el valor asociado a una clave. Como es una operación parcial, lanza una excepción si la clave no existe. De nuevo resolvemos la búsqueda en dos fases: primero calculamos el índice del vector que debería contener la clave y a continuación buscamos el nodo en la lista de colisión correspondiente.

```
/**
* Devuelve el valor asociado a la clave dada. Si la tabla no contiene
* esa clave lanza una excepción.
*/
Valor const& at(Clave const& c) const {
   int i = hash(c) % array.size();
   Link ant, pos = array[i];
   if (localizar(c, ant, pos))
      return pos->cv.second;
   else
      throw std::out_of_range("La clave no se puede consultar");
}
```

Como también hicimos en la clase `map`, incluiremos (al igual que se hace en la clase `unordered_map` de la STL de C++) una versión más avanzada de la operación `at`, que se implementa mediante el operador `[]` y que además de permitir modificar el valor asociado en caso de encontrar la clave buscada, inserta un nuevo par con la clave buscada (y un valor construido por defecto) en el caso de no encontrarla.

```
Valor & operator[](Clave const& c) {
   int i = hash(c) % array.size();
   Link ant, pos = array[i];
   if (localizar(c, ant, pos)) {
      return pos->cv.second;
```

```
    } else {
       array[i] = new ListNode(clave_valor(c,Valor()), array[i]);
       ++nelems;
       return array[i]->cv.second;
    }
}
```

Por último, terminamos con las operaciones `empty` y `size` cuyas implementaciones son triviales.

```
bool empty() const {
   return nelems == 0;
}

int size() const {
   return nelems;
}
```

5.4. Tablas dinámicas

Ya sabemos que si insertamos demasiados elementos en una tabla, el rendimiento de la búsqueda se empieza a degradar. Cuantos más elementos metemos en la tabla más colisiones se producen y, por tanto, las listas de colisiones empiezan a crecer. Si el número de elementos es muy superior al número de posiciones del vector, la mayor parte del tiempo de búsqueda se invierte en buscar la clave dentro de la listas de colisión, lo que puede degradar el coste hasta hacerlo lineal.

Una forma habitual de resolver este problema es permitir que el vector de listas de colisión pueda ampliar su tamaño automáticamente cuando el número de elementos contenidos en la tabla es demasiado grande. Al ampliar el tamaño del vector disminuye la tasa de ocupación (número de elementos / tamaño del vector), lo que favorece mantener el coste de la búsqueda constante.

Necesitaremos modificar la operación *insert* (y también el caso en el que el operador `[]` inserta) para que compruebe si la tabla ya contiene demasiados elementos y por tanto debe expandirse. Para hacerlo, calculamos la tasa de ocupación y, si es demasiado alta, ampliamos el tamaño del vector antes de insertar el nuevo elemento.

```
bool insert(clave_valor const& cv) {
   int i = hash(cv.first) % array.size();
   Link ant, pos = array[i];
   if (localizar(cv.first, ant, pos)) { // la clave ya existe
      return false;
   } else {
      if (muy_llena()) {
         amplia();
         i = hash(cv.first) % array.size();
      }
      array[i] = new ListNode(cv, array[i]);
      ++nelems;
      return true;
   }
}

// Operación privada
bool muy_llena() const {
```

```
    return 100.0 * nelems / array.size() > MAX_CARGA;
}
```

Finalmente, falta por implementar la operación auxiliar *amplia* que amplia la capacidad del vector al siguiente número primo [6]. Para implementar correctamente esta operación es importante tener en cuenta que los índices asociados a las claves pueden cambiar, ya que la función de localización depende del tamaño del vector. Eso quiere decir que debemos recalcular el índice asociado a cada nodo y colocarlo en la nueva posición del nuevo vector.

Una forma sencilla de implementar esta operación sería crear una nueva tabla más grande y volver a insertar todos los pares (clave, valor) contenidos en la tabla original. No vamos a utilizar esa estrategia porque implicaría volver a crear todos los nodos en memoria. En su lugar, vamos a "mover" los nodos desde el vector original a la nueva posición que les corresponde en el nuevo vector.

```
void amplia() {
    std::vector<Link> nuevo(siguiente_primo(array.size()*2), nullptr);
    for (int j = 0; j < array.size(); ++j) {
        Link act = array[j];
        while (act != nullptr) {
            Link a_mover = act;
            act = act->sig;
            int i = hash(a_mover->cv.first) % nuevo.size();
            a_mover->sig = nuevo[i];
            nuevo[i] = a_mover;
        }
    }
    swap(array, nuevo);
}
```

5.5. Recorrido usando iteradores

A veces resulta útil poder recuperar todos los pares (clave, valor) almacenados en la tabla. En este apartado vamos a extender el TAD con nuevas operaciones que permitan recorrer los elementos almacenados usando un iterador.

Como siempre, las clases *const_iterator* e *iterator* serán clases internas con la operación ++ que permite ir hasta el siguiente elemento del recorrido. Como los elementos no se almacenan en la tabla siguiendo ningún orden (de hecho, puede que el tipo de datos usado como clave ni siquiera sea ordenado), podemos recorrerlos de cualquier forma. En este caso hemos elegido recorrer la lista de colisiones de la posición 0 del vector, luego la lista de colisiones de la posición 1, etc. Durante el recorrido debemos tener en cuenta que algunas de estas listas pueden estar vacías, y por tanto puede que tengamos que saltarnos varias posiciones del vector de listas.

Para realizar ese recorrido el iterador necesita tener acceso al vector, y guardar el índice actual dentro del vector, y el puntero al nodo actual dentro de la lista de colisiones actual. A continuación mostramos el código para el caso del iterador no constante (el iterador constante es análogo).

```
// iteradores que recorren los pares de la tabla (no ordenados)
class Iterador {
public:
    clave_valor & operator*() const {
```

[6] El uso de números primos para los tamaños del vector decrementa la probabilidad de colisiones

```
         if (act == nullptr)
            throw std::out_of_range("No hay elemento a consultar");
         return act->cv;
      }

      Iterador & operator++() {  // ++ prefijo
         next();
         return *this;
      }

      bool operator==(Iterador const& that) const {
         return act == that.act;
      }

      bool operator!=(Iterador const& that) const {
         return !(this->operator==(that));
      }

   protected:
      friend class unordered_map;
      umap_t * tabla; // la tabla que se está recorriendo
      Link act;       // nodo actual
      int ind;        // índice de la lista actual

      // iterador al primer elemento o al último
      Iterador(umap_t * t, bool first = true) : tabla(t) {
         if (first) {
            ind = 0;
            while (ind < tabla->array.size() &&
                   tabla->array[ind] == nullptr) {
               ++ind;
            }
            act = (ind < tabla->array.size() ? tabla->array[ind]
                                             : nullptr);
         } else {
            act = nullptr;
            ind = tabla->array.size();
         }
      }

      void next() {
         if (act == nullptr)
            throw std::out_of_range("El iterador no puede avanzar");
         act = act->sig;
         while (act == nullptr && ++ind < tabla->array.size()) {
            act = tabla->array[ind];
         }
      }
   };

public:
   // iterador que sí permite modificar el elemento apuntado (su valor)
   using iterator = Iterador;

   iterator begin() {
      return iterator(this);
```

```
    }

    iterator end() {
       return iterator(this, false);
    }
};
```

Por ejemplo, un recorrido para imprimir todos los elementos contenidos en una tabla se podría escribir así:

```
unordered_map<string, int> edades;

... // Insertar elementos en la tabla

for (auto it = edades.begin(); it != edades.end(); ++it){
   cout << "(" << it->first << ",  " << it->second << ")\n";
   it++;
}
```

Usando un bucle *range-based-for* podría escribirse de esta otra forma (solo válido de esta forma a partir de la versión C++17):

```
  for (auto [nombre, edad] : t)
     cout << "(" << nombre << ",  " << edad << ")\n";
```

Por último, como hicimos en el TAD `map`, incluiremos la operación `find` para buscar una clave devolviendo el correspondiente iterador. De nuevo, implementaremos la búsqueda propiamente dicha en una constructora en la clase interna Iterador, la cual será invocada desde el nuevo método `find`.

```
   class Iterador {
     ...
     // iterador a una clave
     Iterador(pointer_tabla t, Clave const& c) : tabla(t) {
         ind = tabla->hash(c) % tabla->array.size();
         Link ant;
         act = tabla->array[ind];
         if (!tabla->localizar(c, ant, act)) { // iterador al final
            act = nullptr; ind = tabla->array.size();
         }
     }
     ...
   }; //fin de la clase Iterador

   // En la clase unordered_map
   iterator find(Clave const& c) {
      return iterator(this, c);
   }
```

6. Funciones de localización

Una buena función de localización debe ser uniforme y sencilla de calcular. En este apartado vamos a estudiar algunas funciones de localización habituales.

6.1. Para enteros

6.1.1. Aritmética modular y uso de números primos

El índice asociado a un número es el resto de la división entera entre otro número N prefijado, preferiblemente primo. Por ejemplo, para $N = 23$:

$$1679 \bmod 23 = 0$$
$$4567 \bmod 23 = 13$$
$$8471 \bmod 23 = 7$$
$$0435 \bmod 23 = 21$$
$$5033 \bmod 23 = 19$$

También es frecuente multiplicar resultados intermedios por un número primo grande antes de combinarlos con los restantes.

6.1.2. Mitad del cuadrado

Consiste en elevar al cuadrado la clave y coger las cifras centrales.

$$709^2 = 502681 \rightarrow 26$$
$$456^2 = 207936 \rightarrow 79$$
$$105^2 = 011025 \rightarrow 10$$
$$879^2 = 772641 \rightarrow 26$$
$$619^2 = 383161 \rightarrow 31$$

6.1.3. Truncamiento

Consiste en ignorar parte del número y utilizar los dígitos restantes como índice. Por ejemplo, para números de 7 cifras podríamos coger los dígitos segundo, cuarto y sexto para formar el índice.

$$5700931 \rightarrow 703$$
$$3498610 \rightarrow 481$$
$$0056241 \rightarrow 064$$
$$9134720 \rightarrow 142$$
$$5174829 \rightarrow 142$$

6.1.4. Plegamiento

Consiste en dividir el número en diferentes partes y realizar operaciones aritméticas con ellas, normalmente sumas o multiplicaciones. Por ejemplo, podemos dividir un número en bloques de dos cifras y después sumarlas.

$$570093 \rightarrow 57 + 00 + 93 = 150$$
$$349861 \rightarrow 34 + 98 + 61 = 193$$
$$005624 \rightarrow 00 + 56 + 24 = 80$$
$$913472 \rightarrow 91 + 34 + 72 = 197$$
$$517492 \rightarrow 51 + 74 + 92 = 217$$

6.1.5. Operaciones de bits

También es muy habitual usar los operadores de manipulación de bits para "mezclar" fragmentos de hash. Dos operadores son particularmente frecuentes: $\wedge$ (xor binario) y $\ll$ (desplazamiento de bits). Por ejemplo:

$$570093 \rightarrow (570 \ll 3) \wedge 093 = 4493$$
$$349861 \rightarrow (349 \ll 3) \wedge 861 = 2485$$
$$005624 \rightarrow (005 \ll 3) \wedge 624 = 600$$
$$913472 \rightarrow (913 \ll 3) \wedge 472 = 7504$$
$$517492 \rightarrow (517 \ll 3) \wedge 492 = 4548$$

Desarrollando el último ejemplo, $1000000101 \ll 3 = 1000000101000$, que $\wedge 111101100$ es 1000111000100.

6.2. Para cadenas

En este tema ya hemos visto una función de localización para cadenas:

$$h(c) = \text{ord}(\text{ult}(c)) \bmod N$$

El problema de esta función radica en que los códigos ASCII de los caracteres alfanuméricos están comprendidos entre los números 48 y 122, por lo que esta función no se comporta muy bien con valores de N grandes (tablas grandes).

Una mejora evidente consiste en tener en cuenta todos los caracteres de la cadena, en lugar de sólo el último, por ejemplo sumando sus códigos ASCII:

$$h(c) = (\text{ord}(c[0]) + \text{ord}(c[1]) + \ldots + \text{ord}(c[k])) \bmod N$$

Aunque esta función se comporta mejor que la anterior, la suma de los códigos ASCII de los caracteres sigue sin ser un valor muy elevado. Si trabajamos con tablas grandes, esta función tenderá a agrupar todos los datos en la parte inicial de la tabla.

La última función que vamos a plantear tiene en cuenta tanto los caracteres de la cadena como su posición. La idea es interpretar los caracteres de la cadena como dígitos de una cierta base B:

$$h(c) = (\text{ord}(c[0]) + \text{ord}(c[1]) * B + \text{ord}(c[2]) * B^2 + \ldots + \text{ord}(c[k]) * B^k) \bmod N$$

Esta función de localización se comporta bastante mejor que las anteriores con valores de N grandes. Además, por motivos de eficiencia, todas las operaciones aritméticas se realizan módulo 2^w siendo w la longitud de la palabra del ordenador. Dicho de otra forma, el ordenador ignora los desbordamientos que producen las operaciones anteriores cuando el número resultante es mayor que el que puede representar de manera natural.

Una buena elección es $B = 131$ porque para ese valor B^i tiene un ciclo máximo $mod\, 2^k$ para $8 \leq k \leq 64$.

6.3. Para clases definidas por el programador

En los ejemplos vistos hasta ahora, siempre hemos empleado tipos básicos como claves: cadenas, enteros, etc. Sin embargo, la auténtica potencia de las tablas reside en poder

utilizar cualquier clase definida por el programador, siempre que se proporcione una función de localización capaz de transformar concreciones de ese tipo en índices del vector.

En particular, para que una clase pueda usarse como tipo para las claves de una tabla hash es necesario definir dos cosas:

1. El operador de igualdad de la clase: Esto puede hacerse bien sobrecargando el `operator==` (como método de la clase o como función externa), o bien implementando una clase (*clase función*) cuyo `operator()` defina la igualdad y proporcionando esta clase en el cuarto parámetro de la instanciación del objeto tabla.

2. Implementar la función *hash* para la clase, la cual calcula el valor hash para objetos de la clase. Esto debe hacerse implementando una clase función que sobreescriba el `operator()`. De esta forma, delegamos el cálculo de la función de localización sobre el propio tipo de datos, siendo responsabilidad del programador de dicho tipo implementar una buena función de localización.

El siguiente ejemplo muestra como usar una clase `Persona` (definida por el programador) como clave en una tabla hash.

```
struct Persona{
    string nombre;
    string dni;
    bool operator==(const Persona& other) const{
        return dni == other.dni;
    }
};

struct PersonaHasher{
    std::size_t operator()(const Persona& p) const {
        return hash<string>()(p.nombre + p.dni);
    }
};

//Instanciación de un objeto
unordered_map<Persona,int,PersonaHasher> edades;
```

7. En el mundo real...

La mayor parte de los lenguajes de programación modernos incorporan algún tipo de contenedor asociativo implementado mediante tablas de dispersión. De hecho, en algunos lenguajes de *script* los "arrays" permiten utilizar cualquier tipo de dato como índice porque internamente están implementados como tablas de dispersión. En general, un array puede verse como un caso particular de tabla donde las claves son números enteros consecutivos y la función de localización es la identidad.

La librería estándar de C++ incorpora el tipo `unordered_map` análogo al definido aquí (aunque más completo y complejo).

Finalmente, en lenguajes como Java, todas las clases heredan de *Object* que define un método *hashCode()* que devuelve un valor numérico basado en la dirección de memoria del objeto. Los programadores pueden sobrescribir este comportamiento por defecto en sus clases cuando sea necesario, en particular al usarlos como claves en los contenedores `HashMap` y `HashTable`.

8. Para terminar...

Terminamos el tema con la solución a la motivación dada al principio del tema. La implementación utiliza un diccionario como variable local que va almacenando, para cada palabra encontrada en el texto, el número de veces que ha aparecido.

```
void refsCruzadas(const list<string> &texto) {

    list<string>::const_iterator it = texto.cbegin();
    map<string, int> refs;

    while (it != texto.cend()) {
        map<string,int>::iterator p = refs.find(*it);
        if (p == refs.end())
            refs.insert({*it, 1});
        else
            (*p).second++;
        ++it;
    }

    // Y ahora escribimos
    for (auto [palabra,reps] : refs) // Válido desde C++17
%       cout << palabra << " " << reps << endl;
}
```

En el código hemos utilizado un `map`. La implementación con `unordered_map` sería igual excepto que la lista de palabras no saldría ordenada.

Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (Rodríguez Artalejo et al., 2011) y de (Peña, 2005). Animamos al lector a consultar ambos libros para profundizar en el estudio de las tablas asociativas, especialmente en el caso de las tablas cerradas que en este capítulo hemos omitido.

Ejercicios

1. Extiende la implementación de los árboles de búsqueda con la siguiente operación:

   ```
   iterator erase(const iterator &it);
   ```

   que recibe un iterador y elimina la pareja (clave,valor) del diccionario, devolviendo un iterador al siguiente elemento en el recorrido.

2. Implementa una operación en los árboles de búsqueda que *balancee* el árbol. Se permite el uso de estructuras de datos auxiliares.

3. Los árboles de búsqueda y las tablas dispersas son dos tipos de contenedores asociativos que permiten almacenar pares (clave, valor) indexados por clave. Discute sus similitudes y diferencias: ¿qué requisitos impone cada uno sobre el tipo usado como clave?, ¿cuándo es más conveniente usar árboles de búsqueda y cuándo tablas dispersas?

4. Añade las siguientes operaciones a los árboles de búsqueda y analiza su complejidad:

- `consultaK`: recibe un entero k y devuelve la k-ésima clave del árbol de búsqueda, considerando que en un árbol con n elementos, $k = 0$ corresponde a la menor clave y $k = n - 1$ a la mayor.
- `recorreRango`: dadas dos claves, a y b, devuelve una lista con los valores asociados a las claves que están en el intervalo $[a..b]$.

5. ¿Qué cambios realizarías en la implementación de los árboles de búsqueda para permitir almacenar distintos valores para la misma clave? Es decir:

   - Al insertar un par (clave, valor), si la clave ya se encontrase en el árbol, en lugar de sustituir el valor antiguo por el nuevo, se asociaría el valor adicional con la clave.
   - La operación de consulta en vez de devolver un único valor, devuelve una lista con todos ellos en el mismo orden en el que fueron insertados.
   - La operación de borrado elimina todos los valores asociados con la clave dada.

   Para la implementación no debes utilizar otros TADs.

6. En la sección de motivación veíamos que un texto puede venir como una lista de palabras. Otra alternativa es que venga como una lista de *líneas*, donde cada línea es a su vez una lista de palabras (es decir, el tipo sería `list<list<string>>`). El problema de las *referencias cruzadas* consiste en crear el listado en orden alfabético de todas las palabras que aparecen en el texto, indicando, para cada una de las palabras, el número de líneas en las que aparece (si una palabra aparece varias veces en una línea, el número de línea aparecerá repetido). Implementa en C++ un método que reciba un texto y escriba la lista de palabras ordenada alfabéticamente; cada línea contendrá una palabra seguida de todas las líneas en las que ésta aparece. Analiza su complejidad si se utilizan árboles de búsqueda o tablas abiertas.

7. Implementa un TAD *Conjunto* basado en tablas dispersas con las operaciones habituales: *ConjuntoVacio*, *inserta*, *borra*, *esta*, *union*, *interseccion* y *diferencia*.

8. Propón una función de localización adecuada para la clase *Conjunto*, de manera que sea posible usar conjuntos como claves de una tabla.

9. Se define el *índice radial* de una tabla abierta como la longitud del vector por el número de elementos de la lista de colisión más larga. Extiende el TAD Tabla con un método que devuelva su índice radial.

10. Se llaman *vectores dispersos* a los vectores implementados por medio de tablas dispersas. Esta técnica es recomendable cuando el conjunto total de índices posibles es muy grande, y la gran mayoría de los índices tiene asociado un *valor por defecto* (por ejemplo, cero). Usando esta idea, podemos representar un vector disperso de números reales como una tabla *Tabla<int,float>* que sólo almacena las posiciones del vector que no contienen un 0. Implementa funciones que resuelvan la *suma* y el *producto escalar* de dos vectores dispersos de números reales.

11. (ACR270)La evaluación contínua se le ha ido de las manos al profesor. Les pide a los alumnos que no lo dejen todo para el final sino que vayan estudiando día a día, pero él no predica con el ejemplo. Ahora tiene todos los ejercicios que los alumnos han ido entregando durante todo el año en una pila de folios y le toca revisarlos. Los

ejercicios o están bien (y entonces puntúan positivamente) o están mal (y entonces restan).

Al final del día quiere tener imprimida una lista con los nombres de todos los alumnos ordenados alfabéticamente y su puntuación en la evaluación contínua (resultado de sumar todos los ejercicios que tienen bien menos los que tienen mal). Si un alumno tiene un 0 como balance no debería aparecer en la lista.

Aunque sea abusar un poco del alumnado... ¿puedes ayudar al profesor? Lo que se pide es que implementes una función que reciba dos listas de cadenas con los nombres de los alumnos. La primera lista tiene un elemento por cada ejercicio correcto entregado y contiene el nombre del alumno que lo entregó (por lo tanto un mismo alumno puede aparecer varias veces, si entregó varios ejercicios bien). De forma similar, la segunda lista contiene los ejercicios incorrectos. La función debe imprimir por pantalla el resultado final de la evaluación de los alumnos cuyo balance es distinto de 0 por orden alfabético.

12. Plantea una implementación de un TAD *Consultorio* que simule el comportamiento de un consultorio médico simplificado. Dicha implementación hará uso de los TADs *Medico* y *Paciente*, que se suponen ya conocidos. Las operaciones del TAD *Consultorio* son las siguientes:

    - *ConsultorioVacio*: crea un nuevo consultorio vacío.
    - *nuevoMedico*: da de alta un nuevo médico en el consultorio.
    - *pideConsulta*: un paciente se pone a la espera de ser atendido por un médico que ha sido dado de alta previamente en el consultorio.
    - *siguientePaciente*: consulta el paciente al que le toca el turno para ser atendido por un médico dado. El médico debe haber sido dado de alta en el consultorio y tener pacientes en espera para que la operación funcione.
    - *atiendeConsulta*: elimina el siguiente paciente de un médico. El médico debe estar dado de alta y tener pacientes en la lista de espera.
    - *tienePacientes*: indica si un médico tiene o no pacientes esperando a ser atendidos.

    Explica razonadamente los tipos abstractos de datos que vas a utilizar.

    Para cada operación indica si es generadora, observadora o modificadora, y si es total o parcial. ¿Se necesita exigir algo a los TADs *Medico* y *Paciente*?

    Razona la complejidad de la operaciones del TAD en base a la implementación elegida. Para ello considera que hay M médicos dados de alta en el consultorio, y que el médico con la lista de espera más larga tiene P pacientes esperando a ser atendidos.