Capítulo 10

# 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 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:

```
Dictionary<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 diccionario, cada una de ellas será independiente. Es decir no nos planteamos aquí la posibilidad de que exista una clase abstracta a modo de interfaz `Dictionary` 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 $t : C \to V$ que asocian a cada clave $c \in C$ un determinado valor $v \in V$.

Las operaciones publicas del TAD diccionario son:

- $EmptyDictionary :\to Dictionary$ [Generadora]

  Construye un diccionario vacío, es decir, sin elementos.

- $insert : Dictionary, Key, Value \to Dictionary$ [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.

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

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

- $contains : Dictionary, Key \to Boolean$ [Observadora]

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

- $at : Dictionary, Key- \to Value$ [Observadora parcial]

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

- *empty* : *Dictionary* → *Boolean* [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. Por ejemplo, los siguientes diccionarios (con enteros como claves y caracteres como valor) son equivalentes:

```
Insert(EmptyDictionary, 3, 'a')

Insert(Insert(EmptyDictionary, 3, 'b'), 3, 'a')

Insert(Insert(Insert(EmptyDictionary, 3, 'c'), 3, 'b'), 3, 'a')
```

## 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 |
|---|---|---|
| EmptyDictionary | O(1) | O(1) |
| insert | O(n) | O(n) |
| erase | O(n) | O(n) |
| contains | O(n) | O(log n) |
| at | O(n) | O(log n) |
| empty | O(1) | O(1) |

La operación más habitual cuando se utilizan tablas es *at*, 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 *insertar* 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 Arbin::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

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

que los árboles de búsqueda están parametrizados *por dos tipos distintos* en vez de sólo por uno: el tipo de la clave (que debe poderse ordenar) y el tipo del valor.

## 3.1. Implementación

La implementación del diccionario utilizando árboles de búsqueda la llamaremos `TreeMap`[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:

```
template <class Clave, class Valor>
class TreeMap {
public:

   ...

protected:

   /**
    Clase nodo que almacena internamente la pareja (clave, valor)
    y los punteros al hijo izquierdo y al hijo derecho.
    */
   class Nodo {
   public:
      Nodo() : _iz(NULL), _dr(NULL) {}
      Nodo(const Clave &clave, const Valor &valor)
         : _clave(clave), _valor(valor),
           _iz(NULL), _dr(NULL) {}
      Nodo(Nodo *iz, const Clave &clave,
           const Valor &valor, Nodo *dr)
         : _clave(clave), _valor(valor),
           _iz(iz), _dr(dr) {}

      Clave _clave;
      Valor _valor;
      Nodo *_iz;
      Nodo *_dr;
   };

   ...

private:
   /**
    Puntero a la raíz de la estructura jerárquica
    de nodos.
    */
   Nodo *_ra;
};
```

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

[3] `Map` es otro nombre comúnmenete utilizado para los diccionarios, y `TreeMap` viene a indicar un `Map` implementado con un árbol. Es la nomenclatura utilizada, por ejemplo, en Java.

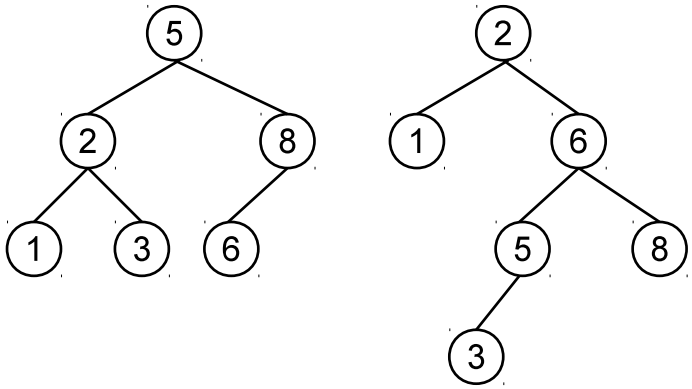



Figura 1: Dos árboles de búsqueda distintos pero equivalentes.

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]:

$$\begin{aligned} & R_{TreeMap_{(C,V)}}(\mathsf{p}) \\ \Longleftrightarrow_{def} & \\ & R_{Arbin_{Pareja(C,V)}}(\mathsf{p})^{5} \wedge \\ & ordenado(\mathsf{p}) \end{aligned}$$

donde

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

$$\begin{array}{lll} claves(\mathsf{ptr}) = \emptyset & & \textbf{si} \;\; \mathsf{ptr} = \mathsf{null} \\ claves(\mathsf{ptr}) = \{\mathsf{ptr\_.clave}\} \bigcup claves(\mathsf{ptr.\_iz}) \bigcup claves(\mathsf{ptr.\_dr}) & & \textbf{si} \;\; \mathsf{ptr} \neq \mathsf{null} \end{array}$$

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:

$$\begin{aligned} & \mathsf{a1} \equiv_{Arbin_T} \mathsf{a2} \\ \Longleftrightarrow_{def} & \\ & elementos(\mathsf{a1}) \equiv_{Conjunto_{Pareja(C,V)}} elementos(\mathsf{a2}) \end{aligned}$$

[4]La definición podría haberse hecho también en base al recorrido en inorden.

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

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

### 3.1.1. Implementación de la observadora `empty`

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 `NULL`, y la operación observadora `empty` comprueba si esa condición se sigue cumpliendo:

```
/** Constructor; operacion EmptyTreeMap */
TreeMap() : _ra(NULL) {
}

/**
 Operación observadora que devuelve si el árbol
 es vacío (no contiene elementos) o no.

 empty(EmptyTreeMap) = true
 empty(insert(c, v, arbol)) = false
 */
bool empty() const {
   return _ra == NULL;
}
```

### 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:

```
// Método protegido/privado
/**
 Busca una clave en la estructura jerárquica de
 nodos cuya raíz se pasa como parámetro, y devuelve
 el nodo en la que se encuentra (o NULL si no está).
 @param p Puntero a la raíz de la estructura de nodos
 @param clave Clave a buscar
*/
static Nodo *buscaAux(Nodo *p, const Clave &clave) {
   if (p == NULL)
      return NULL;

   if (p->_clave == clave)
      return p;

   if (clave < p->_clave)
      return buscaAux(p->_iz, clave);
   else
      return buscaAux(p->_dr, clave);
}
```

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

```
/**
 Operación observadora que devuelve el valor asociado
 a una clave dada.

 at(e, insert(c, v, arbol)) = v si e == c
 at(e, insert(c, v, arbol)) = at(e, arbol) si e != c
 error at(EmptyTreeMap)

 @param clave Clave por la que se pregunta.
*/
const Valor &at(const Clave &clave) const {
   Nodo *p = buscaAux(_ra, clave);
   if (p == NULL)
      throw EClaveErronea();

   return p->_valor;
}

/**
 Operación observadora que permite averiguar si una clave
 determinada está o no en el árbol de búsqueda.

 contains(e, EmptyTreeMap) = false
 contains(e, insert(c, v, arbol)) = true si e == c
 contains(e, insert(c, v, arbol)) = contains(e, arbol) si e != c

 @param clave Clave por la que se pregunta.
 @return true si el diccionario contiene un valor asociado
 a esa clave.
*/
bool contains(const Clave &clave) const {
   return (buscaAux(_ra, clave) != NULL) ? true : false;
}
```

### 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. Teniendo esto en cuenta la implementación (con un método auxiliar recursivo) sale casi sola:

```
/**
 Operación generadora que añade una nueva clave/valor
 a un árbol de búsqueda.
 @param clave Clave nueva.
```

```
 @param valor Valor asociado a esa clave. Si la clave
 ya se había insertado previamente, sustituimos el valor
 viejo por el nuevo.
*/
void insert(const Clave &clave, const Valor &valor) {
   _ra = insertaAux(clave, valor, _ra);
}

// Método protegido/privado
/**
 Inserta una pareja (clave, valor) en la estructura
 jerárquica que comienza en el puntero pasado como parámetro.
 Ese puntero se admite que sea NULL, por lo que se creará
 un nuevo nodo que pasará a ser la nueva raíz de esa
 estructura jerárquica. El método devuelve un puntero a la
 raíz de la estructura modificada. En condiciones normales
 coincidirá con el parámetro recibido; sólo cambiará si
 la estructura era vacía.

 @param clave Clave a insertar. Si ya aparecía en la
 estructura de nodos, se sobreescribe el valor.
 @param valor Valor a insertar.
 @param p Puntero al nodo raíz donde insertar la pareja.
 @return Nueva raíz (o p si no cambia).
*/
static Nodo *insertaAux(const Clave &clave,
                        const Valor &valor, Nodo *p) {

   if (p == NULL) {
      return new Nodo(clave, valor);
   } else if (p->_clave == clave) {
      p->_valor = valor;
      return p;
   } else if (clave < p->_clave) {
      p->_iz = insertaAux(clave, valor, p->_iz);
      return p;
   } else { // (clave > p->_clave)
      p->_dr = insertaAux(clave, valor, p->_dr);
      return p;
   }
}
```

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.

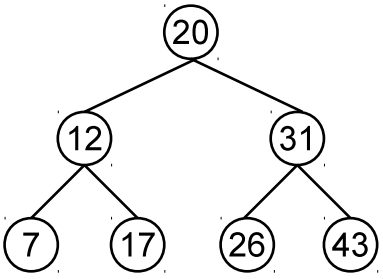
### 3.1.4. Implementación del borrado

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:

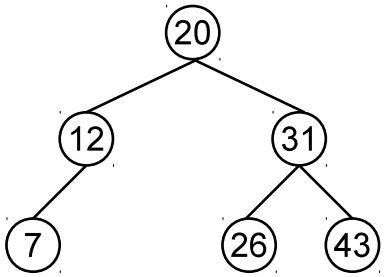
- Si la búsqueda fracasa (la clave no está), se termina sin modificar el árbol.
- Si la búsqueda tiene éxito, se localiza un nodo $\alpha$, que es el que hay que borrar. Para hacerlo:

- Si $\alpha$ es hoja, se puede eliminar directamente (actualizando el puntero del padre).
- Si $\alpha$ tiene un sólo hijo, se elimina el nodo $\alpha$ y se coloca en su lugar el subárbol hijo cuya raíz quedará en el lugar del nodo $\alpha$.
- Si $\alpha$ 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 $\alpha'$ más pequeño del hijo derecho.
  - Si ese nodo tiene hijo derecho, éste pasa a ocupar su lugar.
  - El nodo $\alpha'$ pasa a ocupar el lugar de $\alpha$, de forma que su hijo izquierdo y derecho cambian a los hijos izquierdo y derecho de la raíz antigua.

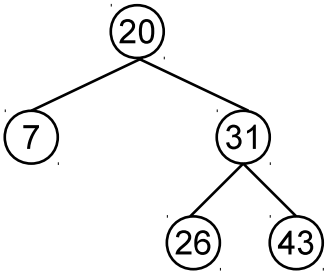
A modo de ejemplo, a continuación mostramos la evolución de un árbol de búsqueda $a$ cuando vamos borrando sus nodos:



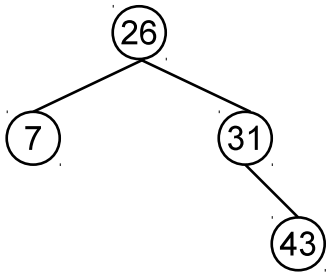

- `a.erase(17)`




- `a.erase(12)`




- `a.erase(20)`




Igual que ocurría con la inserción, la implementación utiliza métodos auxiliares; como la raíz puede cambiar éstos devuelven la nueva raíz (que puede ser igual que la vieja si no hay cambios):

```
/**
  Operación modificadora que elimina una clave del árbol.
  Si la clave no existía la operación no tiene efecto.

   erase(elem, EmptyTreeMap) = TreeMapVacio
   erase(e, insert(c, v, arbol)) =
            inserta(c, v, erase(e, arbol)) si c != e
   erase(e, insert(c, v, arbol)) = erase(e, arbol) si c == e

 @param clave Clave a eliminar.
*/
void erase(const Clave &clave) {
   _ra = borraAux(_ra, clave);
}

protected:

/**
 Elimina (si existe) la clave/valor de la estructura jerárquica
 de nodos apuntada por p. Si la clave aparecía en la propia raíz,
 ésta cambiará, por lo que se devuelve la nueva raíz. Si no cambia
 se devuelve p.

 @param p Raíz de la estructura jerárquica donde borrar la clave.
 @param clave Clave a borrar.
 @return Nueva raíz de la estructura, tras el borrado. Si la raíz
 no cambia, se devuelve el propio p.
*/
static Nodo *borraAux(Nodo *p, const Clave &clave) {

   if (p == NULL)
      return NULL;

   if (clave == p->_clave) {
      return borraRaiz(p);
   } else if (clave < p->_clave) {
      p->_iz = borraAux(p->_iz, clave);
      return p;
   } else { // clave > p->_clave
      p->_dr = borraAux(p->_dr, clave);
      return p;
   }
```

```
}


/**
 Borra la raíz de la estructura jerárquica de nodos
 y devuelve el puntero a la nueva raíz que garantiza
 que la estructura sigue siendo válida para un árbol de
 búsqueda (claves ordenadas).
 */
static Nodo *borraRaiz(Nodo *p) {

   Nodo *aux;

   // Si no hay hijo izquierdo, la raíz pasa a ser
   // el hijo derecho
   if (p->_iz == NULL) {
      aux = p->_dr;
      delete p;
      return aux;
   } else
   // Si no hay hijo derecho, la raíz pasa a ser
   // el hijo izquierdo
   if (p->_dr == NULL) {
      aux = p->_iz;
      delete p;
      return aux;
   } else {
   // Convertimos el elemento más pequeño del hijo derecho
   // en la raíz.
      return mueveMinYBorra(p);
   }
}

/**
 Método auxiliar para el borrado; recibe un puntero a la
 raíz a borrar. Busca el elemento más pequeño del hijo derecho
 que se convertirá en la raíz (que devolverá), borra la antigua
 raíz (p) y "cose" todos los punteros, de forma que ahora:

   - El mínimo pasa a ser la raíz, cuyo hijo izquierdo y
     derecho eran los hijos izquierdo y derecho de la raíz
     antigua.
   - El hijo izquierdo del padre del elemento más pequeño
     pasa a ser el antiguo hijo derecho de ese mínimo.
 */
static Nodo *mueveMinYBorra(Nodo *p) {

   // Vamos bajando hasta que encontramos el elemento
   // más pequeño (aquel que no tiene hijo izquierdo).
   // Vamos guardando también el padre (que será null
   // si el hijo derecho es directamente el elemento
   // más pequeño).
   Nodo *padre = NULL;
   Nodo *aux = p->_dr;
   while (aux->_iz != NULL) {
      padre = aux;
```

```
      aux = aux->_iz;
   }

   // aux apunta al elemento más pequeño.
   // padre apunta a su padre (si el nodo es hijo izquierdo)

   // Dos casos dependiendo de si el padre del nodo con
   // el mínimo es o no la raíz a eliminar
   // (=> padre != NULL)
   if (padre != NULL) {
      padre->_iz = aux->_dr;
      aux->_iz = p->_iz;
      aux->_dr = p->_dr;
   } else {
      aux->_iz = p->_iz;
   }

   delete p;
   return aux;
}
```

## 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 |
|---|---|
| EmptyTreeMap | O(1) |
| insert | O(log n) |
| contains | O(log n) |
| at | O(log n) |
| erase | O(log n) |
| empty | O(1) |

Es importante resaltar que para que un tipo pueda utilizarse como clave en un diccionario 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).

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:

```
TreeMap<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. Existen estructuras de datos más avanzadas que utilizan las ideas de los árboles de búsqueda vistos aquí, pero que incluyen lógica de *auto-balanceo* 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. Igual que en el caso de las listas, tendremos dos versiones del iterador, por un lado el `ConstIterator` 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 `NULL` para indicar que estamos "fuera" del recorrido.

```
/**
 Clase interna que implementa un iterador sobre
 la lista que permite recorrer el árbol pero no
 permite modificarlo.
*/
class ConstIterator {
public:
   void next() {
      if (_act == NULL) throw InvalidAccessException();

      // Si hay hijo derecho, saltamos al primero
      // en inorden del hijo derecho
      if (_act->_dr != NULL)
         _act = primeroInOrden(_act->_dr);
      else {
         // Si no, vamos al primer ascendiente
         // no visitado. Para eso consultamos
         // la pila; si ya está vacía, no quedan
         // ascendientes por visitar
         if (_ascendientes.empty())
            _act = NULL;
         else {
            _act = _ascendientes.top();
            _ascendientes.pop();
```

```
         }
      }
   }

   const Clave &key() const {
      if (_act == NULL) throw InvalidAccessException();
      return _act->_clave;
   }

   const Valor &value() const {
      if (_act == NULL) throw InvalidAccessException();
      return _act->_valor;
   }

   // ...

protected:
   // Para que pueda construir objetos del
   // tipo iterador
   friend class TreeMap;

   ConstIterator() : _act(NULL) {}
   ConstIterator(Nodo *act) {
      _act = primeroInOrden(act);
   }

   /**
    Busca el primer elemento en inorden de
    la estructura jerárquica de nodos pasada
    como parámetro; va apilando sus ascendientes
    para poder "ir hacia atrás" cuando sea necesario.
    @param p Puntero a la raíz de la subestructura.
   */
   Nodo *primeroInOrden(Nodo *p) {
      if (p == NULL)
         return NULL;

      while (p->_iz != NULL) {
         _ascendientes.push(p);
         p = p->_iz;
      }
      return p;
   }

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

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

/**
 Devuelve el iterador constante al principio del
 diccionario (clave más pequeña).
```

```
 @return iterador al principio del recorrido;
 coincidirá con cend() si el diccionario está vacío.
*/
ConstIterator cbegin() const {
   return ConstIterator(_ra);
}

/**
 @return Devuelve un iterador al final del recorrido
 (fuera de éste).
*/
ConstIterator cend() const {
   return ConstIterator(NULL);
}
```

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

Existe también la implementación del iterador no constante, `Iterator` y sus correspondientes `begin` y `end` cuya única diferencia es que el método `valor()` devuelve una referencia no constante que permite modificar el valor asociado.

### 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 `contains`. Eso fuerza a hacer dos búsquedas sobre 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 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 dos versiones distintas, una constante que no permite modificar el contenido del árbol devolviendo un `ConstIterator` y otra que sí lo permite al devolver un `Iterator`. La primera es:

```
ConstIterator find(const Clave &c) const {
   Stack<Nodo*> ascendientes;
   Nodo *p = _ra;
   while ((p != NULL) && (p->_clave != c)) {
      if (p->_clave > c) {
         ascendientes.push(p);
         p = p->_iz;
      } else
         p = p->_dr;
   }
   ConstIterator ret;
   ret._act = p;
   if (p != NULL)
      ret._ascendientes = ascendientes;
   return ret;
}
```

La versión no constante es similar.

# 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 *inserta*, *esta*, *consulta* y *borra* 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.

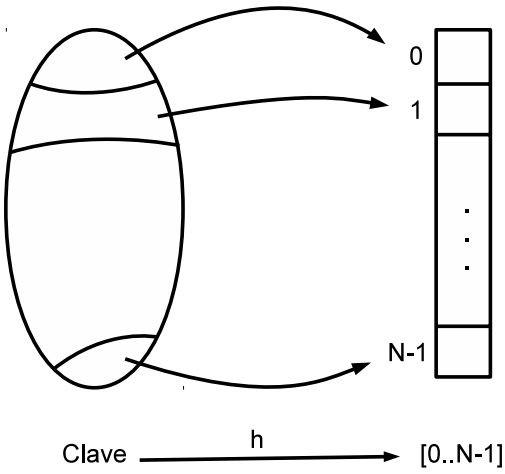
## 4.1. Función de localización

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:

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

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 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.

La operación *insertar* 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:

```
HashMap<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*. De esa forma podremos crear distintos tipos de tablas de la manera habitual:

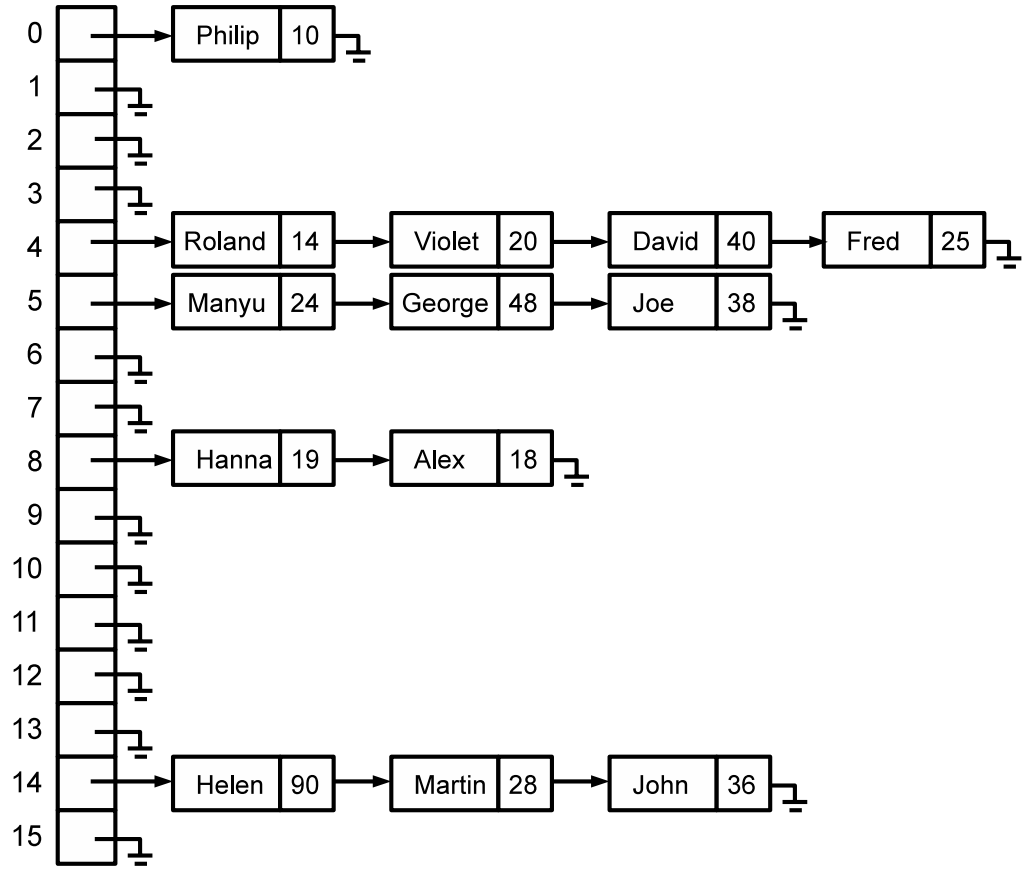



Figura 2: Ejemplo de tabla abierta. La función de localización usada en el ejemplo es claramente mejorable.

```
HashMap<std::string, int> concordancias;
HashMap<List<char>, Persona> dniPersonas;
HashMap<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 clase *Nodo*, como no podría ser de otra manera, será una clase interna de la clase *HashMap* y almacenará la clave, el valor, y el puntero al siguiente nodo de la lista. La clase *HashMap*, a su vez, contiene un array de punteros a nodos (las listas de colisión), el tamaño del array, y el número de elementos almacenados en la tabla.

```
template <class Clave, class Valor>
class HashMap {
public:
```

```
   ...

private:

   /**
    * La tabla contiene un array de punteros a nodos. Cada nodo
    * contiene una clave, un valor y el puntero al siguiente nodo.
    */
   class Nodo {
   public:
      /* Constructores. */
      Nodo(const Clave &clave, const Valor &valor) :
         _clave(clave), _valor(valor), _sig(NULL) {};

      Nodo(const Clave &clave, const Valor &valor, Nodo *sig) :
         _clave(clave), _valor(valor), _sig(sig) {};

      /* Atributos públicos. */
      Clave _clave;
      Valor _valor;
      Nodo *_sig;  // Puntero al siguiente nodo.
   };


   Nodo **_v;               // Array de punteros a Nodo
   unsigned int _tam;       // Tamaño del array
   unsigned int _numElems;  // Número de elementos en la tabla.
};
```

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 *_numElems* 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:

$$R_{Tabla_{C,V}}(\mathsf{t}) \iff_{def} \begin{array}{l} ubicado(\mathsf{t.\_v}) \wedge \mathsf{t.\_numElems} = totalElems(\mathsf{t.\_v}, \mathsf{t.\_tam}) \wedge \\ \forall i : 0 \leq i < \mathsf{t.\_tam} : buenaLista(\mathsf{t.\_v}, i) \end{array}$$

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

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.

$$\begin{array}{ll} numElems(\mathsf{p}) = 0 & \textbf{si } \mathsf{p} = \mathsf{NULL} \\ numElems(\mathsf{p}) = 1 + numElems(\mathsf{p.\_sig}) & \textbf{si } \mathsf{p} \neq \mathsf{NULL} \end{array}$$

$$\begin{array}{ll} clave(\mathsf{p}, \mathsf{i}) = \mathsf{p.\_clave} & \textbf{si } \mathsf{i} = 0 \\ clave(\mathsf{p}, \mathsf{i}) = clave(\mathsf{p.\_sig}, \mathsf{i} - 1) & \textbf{si } \mathsf{i} > 0 \end{array}$$

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

Respecto a la relación de equivalencia, al igual que ocurría con los árboles de búsqueda, decimos que dos tablas son equivalentes si almacenan el mismo conjunto de pares (clave, valor):

$$\begin{array}{l} \mathsf{t1} \equiv_{Arbin_T} \mathsf{t2} \\ \Longleftrightarrow_{def} \\ elementos(\mathsf{t1}) \equiv_{Conjunto_{Par(C,V)}} elementos(\mathsf{t2}) \end{array}$$

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.

```
/* Libera el array de listas enlazadas */
void libera() {

   // Liberamos las listas de nodos.
   for (int i=0; i<_tam; i++) {
      liberaNodos(_v[i]);
   }

   // Liberamos el array de punteros a nodos.
   if (_v != NULL) {
      delete[] _v;
      _v = NULL;
   }
}
```

```
/* Libera una lista enlazada de nodos  */
static void liberaNodos(Nodo *prim) {

   while (prim != NULL) {
      Nodo *aux = prim;
      prim = prim->_sig;
      delete aux;
   }
}
```

A continuación se muestran dos métodos auxiliares que permiten buscar un nodo con una cierta clave en una lista enlazada. El primer método devuelve dos punteros apuntando tanto al nodo "encontrado" como al nodo anterior a ese. Usaremos este método en las operaciones de borrado, ya que para eliminar un nodo de una lista necesitamos un puntero al nodo anterior y nuestros nodos sólo almacenan un puntero al siguiente (recordemos que estamos trabajando con listas enlazadas simples).

El segundo método de búsqueda sólo devuelve un puntero al nodo "encontrado". Usaremos este método en el resto de los casos, es decir, durante las operaciones de *insertar*, *esta* y *consulta*.

En ambos casos, si buscamos un nodo con una clave que no existe en la lista enlazada, se devolverá NULL como nodo encontrado.

```
/**
 * Busca un nodo a partir del nodo "act" que contenga la clave
 * dada. Si lo encuentra, "act" quedará apuntando a dicho nodo
 * y "ant" al nodo anterior. Si no lo encuentra "act" quedará
 * apuntando a NULL.
 *
 * @param clave clave del nodo que se busca.
 * @param act [in/out] inicialmente indica el primer nodo de la
 *            búsqueda, y al finalizar el nodo encontrado o NULL.
 * @param ant [out] puntero al nodo anterior a "act" o NULL.
 */
static void buscaNodo(const Clave &clave, Nodo* &act, Nodo* &ant) {
   ant = NULL;
   bool encontrado = false;
   while ((act != NULL) && !encontrado) {
      if (act->_clave == clave) {
         encontrado = true;
      } else {
         ant = act;
         act = act->_sig;
      }
   }
}

/**
 * Busca un nodo a partir de "prim" que contenga la clave dada.
 *
 * @param clave clave del nodo que se busca.
 * @param prim nodo a partir del cual realizar la búsqueda.
 * @return nodo encontrado o NULL.
 */
static Nodo* buscaNodo(const Clave &clave, Nodo* prim) {
   Nodo *act = prim;
```

```
   Nodo *ant = NULL;
   buscaNodo(clave, act, ant);
   return act;
}
```

## 5.3. Implementación de las operaciones públicas

El constructor del TAD tabla crea un array de punteros a nodos de un determinado tamaño e inicializa todas sus posiciones a NULL. De esa forma representamos un vector de listas de colisión con todas las listas vacías. El destructor utiliza la operación auxiliar que vimos en el apartado anterior para *liberar* toda la memoria reservada por la tabla.

```
HashMap(unsigned int tam) : _v(new Nodo*[tam]), _tam(tam), _numElems(0) {
   for (int i=0; i<_tam; ++i) {
      _v[i] = NULL;
   }
}

~HashMap() {
   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.
 *
 * @param clave clave del nuevo elemento.
 * @param valor valor del nuevo elemento.
 */
void insert(const Clave &clave, const Valor &valor) {

   // Obtenemos el índice asociado a la clave.
   unsigned int ind = ::hash(clave) % _tam;

   // Si la clave ya existía, actualizamos su valor
   Nodo *nodo = buscaNodo(clave, _v[ind]);
   if (nodo != NULL) {
      nodo->_valor = valor;
   } else {

      // Si la clave no existía, creamos un nuevo nodo y lo insertamos
      // al principio
      _v[ind] = new Nodo(clave, valor, _v[ind]);
      _numElems++;
   }
}
```

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. En este caso necesitamos usar la operación de búsqueda que devuelve el puntero al nodo encontrado y el puntero al nodo anterior a ese. Y 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.
 *
 * @param clave clave del elemento a eliminar.
 */
void erase(const Clave &clave) {

   // Obtenemos el índice asociado a la clave.
   unsigned int ind = ::hash(clave) % _tam;

   // Buscamos el nodo que contiene esa clave y el nodo anterior.
   Nodo *act = _v[ind];
   Nodo *ant = NULL;
   buscaNodo(clave, act, ant);

   if (act != NULL) {

      // Sacamos el nodo de la secuencia de nodos.
      if (ant != NULL) {
         ant->_sig = act->_sig;
      } else {
         _v[ind] = act->_sig;
      }

      // Borramos el nodo extraído.
      delete act;
      _numElems--;
   }
}
```

La operación *contains* 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 una búsqueda dentro de la lista enlazada de nodos

```
/**
 *  Operación observadora que devuelve el valor asociado
 *  a una clave dada.
 *
 *  at(e, insert(c, v, tabla)) = v si e == c
 *  at(e, insert(c, v, tabla)) = at(e, tabla) si e != c
 *  error at(EmptyHashMap)
 *
 *  @param clave Clave por la que se pregunta.
 */
bool contains(const Clave &clave) const {
   // Obtenemos el índice asociado a la clave.
   unsigned int ind = ::hash(clave) % _tam;
```

```
   // Buscamos un nodo que contenga esa clave.
   Nodo *nodo = buscaNodo(clave, _v[ind]);
   return nodo != NULL;
}
```

La operación más importante de la 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.
 *
 * @param clave clave del elemento a buscar.
 * @return valor asociado a dicha clave.
 * @throw EClaveInexistente si la clave no existe en la tabla.
 */
const Valor &at(const Clave &clave) const {

   // Obtenemos el índice asociado a la clave.
   unsigned int ind = ::hash(clave) % _tam;

   // Buscamos un nodo que contenga esa clave.
   Nodo *nodo = buscaNodo(clave, _v[ind]);
   if (nodo == NULL)
      throw EClaveInexistente();

   return nodo->_valor;
}
```

Por último, terminamos con la operación que indica si la tabla está vacía, es decir, si no contiene ningún elemento.

```
/**
 * Indica si la tabla está vacía, es decir, si no contiene ningún elemento.
 *
 * @return si la tabla está vacía.
 */
bool empty() {
   return _numElems == 0;
}
```

## 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.

Usando esta estrategia, el usuario del TAD tabla ya no necesita preocuparse del tamaño interno del vector, por lo que suele ser habitual tener un constructor que crea una tabla con un vector de tamaño predeterminado.

```
static const int TAM_INICIAL = 10;

HashMap() : _v(new Nodo*[TAM_INICIAL]), _tam(TAM_INICIAL), _numElems(0) {
   for (int i=0; i<_tam; ++i) {
      _v[i] = NULL;
   }
}
```

También necesitamos modificar la operación *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.

```
static const unsigned int MAX_OCUPACION = 80;

void insert(const Clave &clave, const Valor &valor) {

   // Si la ocupación es muy alta ampliamos la tabla
   float ocupacion = 100 * ((float) _numElems) / _tam;
   if (ocupacion > MAX_OCUPACION)
      amplia();

   // Obtenemos el índice asociado a la clave.
   unsigned int ind = ::hash(clave) % _tam;

   // Si la clave ya existía, actualizamos su valor
   Nodo *nodo = buscaNodo(clave, _v[ind]);
   if (nodo != NULL) {
      nodo->_valor = valor;
   } else {

      // Si la clave no existía, creamos un nuevo nodo y
      // lo insertamos al principio
      _v[ind] = new Nodo(clave, valor, _v[ind]);
      _numElems++;
   }
}
```

Finalmente, falta por implementar la operación auxiliar *amplia* que duplica la capacidad del vector. 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() {
   // Creamos un puntero al array actual y anotamos su tamaño.
   Nodo **vAnt = _v;
   unsigned int tamAnt = _tam;

   // Duplicamos el array en otra posición de memoria.
   _tam *= 2;
   _v = new Nodo*[_tam];
   for (int i=0; i<_tam; ++i)
      _v[i] = NULL;

   // Recorremos el array original moviendo cada nodo a la nueva
   // posición que le corresponde en el nuevo array.
   for (int i=0; i<tamAnt; ++i) {

      // IMPORTANTE: Al modificar el tamaño también se modifica
      // el índice asociado a cada nodo. Es decir, los nodos se
      // mueven a posiciones distintas en el nuevo array.

      // NOTA: por eficiencia movemos los nodos del array antiguo
      // al nuevo, no creamos nuevos nodos.

      // Recorremos la lista de nodos
      Nodo *nodo = vAnt[i];
      while (nodo != NULL) {
         Nodo *aux = nodo;
         nodo = nodo->_sig;

         // Calculamos el nuevo índice del nodo, lo desenganchamos
         // del array antiguo y lo enganchamos al nuevo.
         unsigned int ind = ::hash(aux->_clave) % _tam;
         aux->_sig = _v[ind];
         _v[ind] = aux;
      }
   }

   // Borramos el array antiguo (ya no contiene ningún nodo).
   delete[] vAnt;
}
```

## 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 *ConstIterator* e *Iterator* son clases internas a *HashMap* con una operación *next* (equivalente al operador $++$) 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, la clase *Iterator* (análogamente *ConstIterator*) necesita acceso al vector, al índice actual dentro del vector, y al nodo actual dentro de la lista de colisiones actual. A continuación mostramos el código de esta clase.

```
/**
   *   Clase interna que implementa un iterador sobre
   *   la árbol de búsqueda que permite recorrer la lista e incluso
   *   alterar el valor de sus elementos.
   */
class Iterator {
public:
   void next() {
      if (_act == NULL) throw InvalidAccessException();

      // Buscamos el siguiente nodo de la lista de nodos.
      _act = _act->_sig;

      // Si hemos llegado al final de la lista de nodos, seguimos
      // buscando por el vector _v.
      while ((_act == NULL) && (_ind < _tabla->_tam - 1)) {
         ++_ind;
         _act = _tabla->_v[_ind];
      }
   }

   const Clave &key() const {
      if (_act == NULL) throw InvalidAccessException();
      return _act->_clave;
   }

   Valor &value() const {
      if (_act == NULL) throw InvalidAccessException();
      return _act->_valor;
   }

   bool operator==(const Iterator &other) const {
      return _act == other._act;
   }

   bool operator!=(const Iterator &other) const {
      return !(this->operator==(other));
   }

   Iterator &operator++() {
      next();
      return *this;
   }

   Iterator operator++(int) {
      Iterator ret(*this);
      operator++();
      return ret;
   }

protected:
   // Para que pueda construir objetos del tipo iterador
```

```
   friend class HashMap;

   Iterator(const HashMap* tabla, Nodo* act, unsigned int ind)
   : _tabla(tabla), _act(act), _ind(ind) { }

   const HashMap *_tabla;  ///< Tabla que se está recorriendo
   Nodo* _act;             ///< Puntero al nodo actual del recorrido
   unsigned int _ind;      ///< Índice actual en el vector _v
};
```

También debemos añadir dos nuevas operaciones públicas al TAD tabla que devuelvan iteradores apuntando al principio y al final del recorrido.

```
/**
   *   Devuelve el iterador al principio de la lista.
   *   @return iterador al principio de la lista;
   *   coincidirá con final() si la lista está vacía.
   */
Iterator begin() {
   unsigned int ind = 0;
   Nodo *act = _v[0];
   while (ind < _tam-1 && act == NULL) {
      ind++;
      act = _v[ind];
   }
   return Iterator(this, act, ind);
}

/**
   *   @return Devuelve un iterador al final del recorrido
   *   (fuera de éste).
   */
Iterator end() const {
   return Iterator(this, NULL, 0);
}
```

Con las modificaciones anteriores, es sencillo imprimir todos los elementos contenidos en una tabla. Como en este caso no necesitamos modificar valores, usaremos *ConstIterator* en lugar de un *Iterator* estándar:

```
HashMap<string, int> t;

... // Insertar elementos en la tabla

HashMap<string,int>::ConstIterator it = t.cbegin();
while (it != t.cend()) {
   cout << "(" << it.key() << ",  " << it.value() << ")" << endl;
   it++;
}
```

## 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.

Una manera sencilla de hacerlo es "obligar" a todas las clases usadas como clave, a proporcionar un método público

```
unsigned int hash();
```

que devuelve un entero asociado al objeto concreto sobre el que se invoca. De esa 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.

Para que esta metodología funcione correctamente, la definición de la plantilla *HashMap* debe proporciona la implementación de las funciones de localización para los tipos básicos y, además, una función de localización especial para los tipos definidos por el programador:

```
unsigned int hash(unsigned int c) { ... };
unsigned int hash(float c) { ... };
unsigned int hash(std::string &c) { ... };

template<class C>
unsigned int hash(const C &c) {
   return c.hash();
}
```

Cuando la clase *HashMap* usa la función *hash* pasándole una clave, el mecanismo de sobrecarga de operadores de C++ invocará a la función adecuada. Y si la clave no es de un tipo básico, se terminará invocando al método *hash* de la clase que se está usando como clave.

Por ejemplo, supongamos que hemos creado una clase pareja capaz de contener dos valores cualesquiera y queremos usarla como clave de una tabla. Una posible implementación de esta clase sería la siguiente:

```
template<class A, class B>
class Pareja {
public:
   Pareja() {};
   Pareja(A prim, B seg) : _prim(prim), _seg(seg) {};

   A primero() const { return _prim; };
   B segundo() const { return _seg; };

   unsigned int hash() const {
      return ::hash(_prim) * 1021 + ::hash(_seg);
   };

   bool operator==(const Pareja& otra) const {
      return _prim == otra._prim && _seg == otra._seg;
   }

   A _prim;
   B _seg;
};
```

En general, al implementar la función de localización de una clase debemos tratar de ser coherentes con el operador de igualdad, es decir si $a == b$ entonces $a.hash() == b.hash()$.

## 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.

Aunque la librería estándar de C++ no incorpora tablas dispersas, sí que existen extensiones de uso común que las proporcionan. Estas implementaciones no oficiales tienen la misma interfaz que la clase *std::map* y permiten acceder a los valores almacenados utilizando la sintaxis de los arrays.

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.

## 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. En el código hemos utilizado un `TreeMap`; la implementación con tablas es similar.

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

   List<string>::ConstIterator it(texto.cbegin());
   TreeMap<string, int> refs;

   while (it != texto.cend()) {
      TreeMap<string,int>::Iterator p = refs.find(it.elem());
      if (p == refs.end())
         refs.insert(it.elem(), 1);
      else
         p.value()++;
      ++it;
   }

   // Y ahora escribimos
   TreeMap<string, int>::ConstIterator ita = refs.cbegin();
   while (ita != refs.cend()) {
      cout << ita.key() << " " << ita.value() << endl;
      ++ita;
   }
}
```

La diferencia fundamental es que en la implementación con árboles de búsqueda la lista de palabras saldrá 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.