Capítulo 1

Diseño e implementación de tablas asociativas[1]

Los programas deben ser escritos para que los lean las personas y, sólo de manera secundaria, para que los ejecuten las máquinas

Abelson and Sussman

Resumen: En este tema se introducen las tablas asociativas, centrándonos en su implementación mediante 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

Los servidores DNS traducen nombres de máquinas a direcciones IP, permitiendo acceder a los sitios web usando nombres fáciles de recordar. Por ejemplo, la dirección *www.google.com* se traduce mediante DNS a una IP *74.125.230.177* que identifica la máquina que proporciona el servicio de búsqueda.

Implementa una función que reciba como parámetros los nombres de las máquinas y sus direcciones IP asociadas, y una lista de consultas (nombres de máquinas cuya dirección IP queremos obtener); y devuelva otra lista con las direcciones IP asociadas a las consultas. La función debe tan eficiente como sea posible.

2. Introducción

La tablas son contenedores asociativos que almacenan colecciones de pares *(clave, valor)*, y que permiten acceder a los valores a partir de sus claves asociadas. Podemos verlas como una generalización de los árboles de búsqueda introducidos en el tema anterior pero, a diferencia de estos, en las tablas no se supone una estructura de árbol subyacente (aunque pueden ser implementadas como tales).

[1] Antonio Sánchez Ruiz-Granados es el autor principal de este tema.

La implementación más habitual de las tablas es mediante tablas dispersas (*hash tables*) que, como veremos, permiten recuperar la información asociada a una clave en tiempo $O(1)$ en promedio (en el caso peor la búsqueda es lineal). Por tanto, las tablas son útiles para representar colecciones de datos a las que vamos a acceder de manera frecuente mediante una *clave* o *índice*.

Existen muchas aplicaciones prácticas de las tablas en Informática, como pueden ser los sistemas de ficheros, las tablas de símbolos de los compiladores, o las tablas usadas en los sistemas de gestión de bases de datos.

2.1. Especificación

Desde un punto de vista matemático, las tablas 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 tabla son las mismas que las de los árboles de búsqueda:

- $TablaVacia :\to Tabla$ [Generadora]

  Construye una tabla vacía, es decir, sin elementos.

- $inserta : Tabla, Clave, Valor \to Tabla$ [Generadora]

  Añade un nuevo par (clave, valor) a la tabla. Si la clave ya existía en la tabla inicial, se sobrescribe su valor asociado.

- $borra : Tabla, Clave \to Tabla$ [Modificadora]

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

- $esta : Tabla, Clave \to Boolean$ [Observadora]

  Indica si la tabla contiene algún par con la clave proporcionada.

- $consulta : Tabla, Clave- \to Valor$ [Observadora parcial]

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

- $esVacia : Tabla \to Boolean$ [Observadora]

  Indica si la tabla no contiene ningún elemento.

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
TablaVacia	O(1)	O(1)
inserta	O(n)	O(n)
borra	O(n)	O(n)
esta	O(n)	O(log n)
consulta	O(n)	O(log n)
esVacia	O(1)	O(1)

La operación más habitual cuando se utilizan tablas es la 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 conste 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 $O(n)$ porque la búsqueda tiene coste $O(n)$.

Podemos mejorar el coste de las operaciones usando una lista ordenada basada en vectores. Con esta nueva implementación las operaciones *consulta* y *esta* pasan a ser logarítmicas, ya que se pueden resolver usando búsqueda binaria. Sin embargo, las operaciones *inserta* y *borra* 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.

Otra estrategia distinta consiste en implementar las tablas usando árboles de búsqueda. De hecho, puesto que ambos TADs comparten el mismo interfaz, la implementación de tablas mediante árboles de búsqueda es trivial. Los costes de las operaciones, si suponemos árboles equilibrados, son los siguientes:

Operación	Árboles de búsqueda
TablaVacia	O(1)
inserta	O(log n)
esta	O(log n)
consulta	O(log n)
borra	O(log n)
esVacia	O(1)

cuando nos piden mostrar los datos ordenados, usar árboles, else tablas porque son más eficientes aunque se saquen desordenados

Es importante resaltar que esta implementación asume que existe una relación de orden entre las claves, ya que los árboles de búsqueda almacenan los elementos ordenados. A cambio, conseguimos que las operaciones sean logarítmicas (si los árboles se mantienen equilibrados).

Durante el resto del capítulo estudiaremos otra implementación de las tablas, las *tablas dispersas*, que tienen dos ventajas sobre los árboles de búsqueda: son más eficientes en promedio, y permiten usar claves sobre las que no es necesario definir una relación de orden.

3. 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.

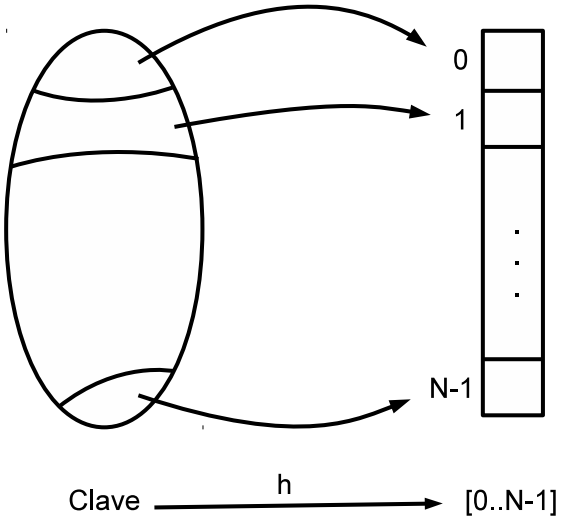
3.1. Función de localización

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 **??**, discutiremos algunas ideas para definir mejores funciones de localización.

3.2. 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 (**?**) y (**?**). 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.

4. 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:

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

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

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 $O(n/16) = O(n)$.

4.1. 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:

```
Tabla<std::string, int> concordancias;
```

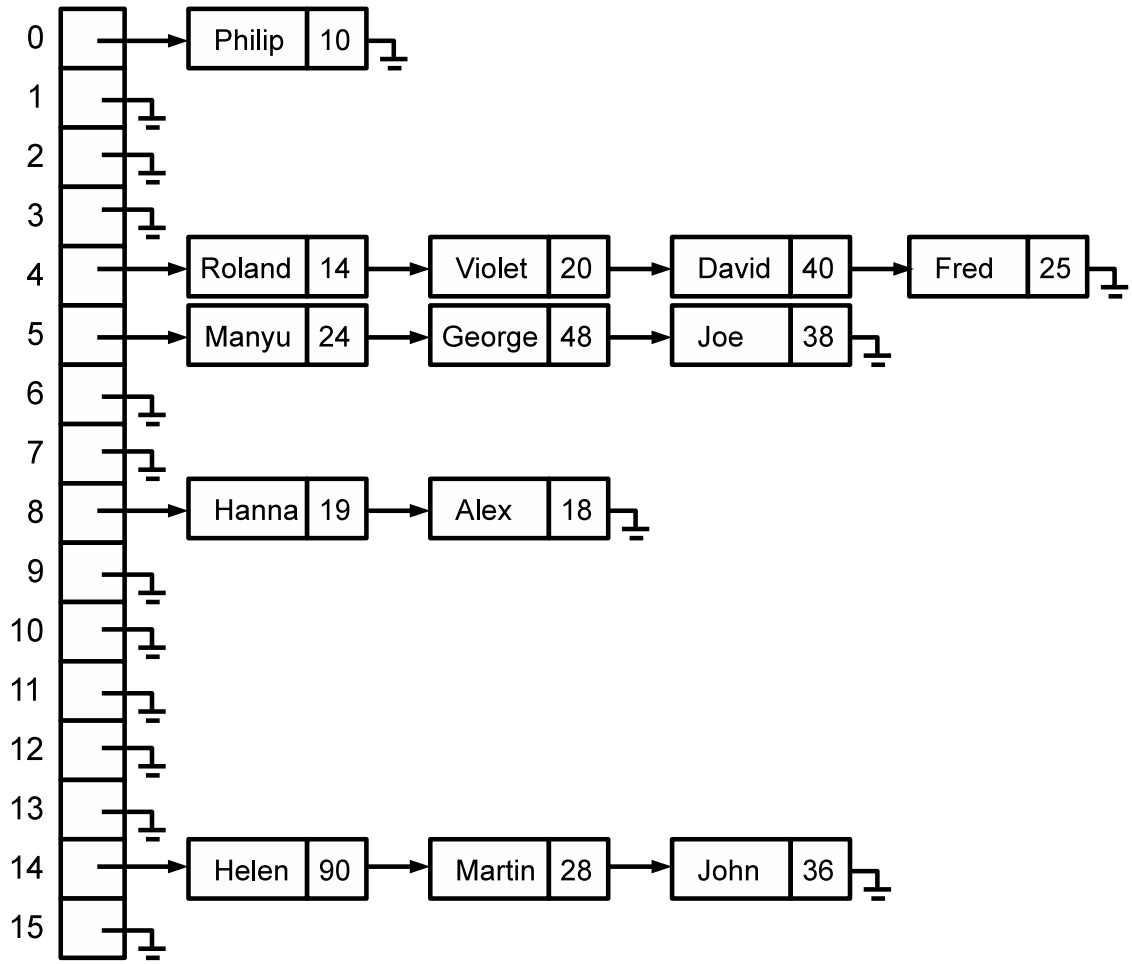


Figura 1: Ejemplo de tabla abierta.

```
Tabla<Lista<char>, Persona> dniPersonas;
Tabla<Lista<char>, Lista<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 *Tabla* y almacenará la clave, el valor, y el puntero al siguiente nodo de la lista. La clase *Tabla*, 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 C, class V>
class Tabla {
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 C &clave, const V &valor) :
            _clave(clave), _valor(valor), _sig(NULL) {};

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

      C _clave;
      V _valor;
      Nodo *_sig;
   };

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

facepalm

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 $h(valor) = i$.
- 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:

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

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

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.

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

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

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

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.

4.2. 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 C &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 C &clave, Nodo* prim) {
   Nodo *act = prim;
   Nodo *ant = NULL;
   buscaNodo(clave, act, ant);
```

```
   return act;
}
```

4.3. 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.

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

~Tabla() {
   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 inserta(const C &clave, const V &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 de la lista. es más eficiente que al final
      _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 borra(const C &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 *esta* 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

```
/**
 * Comprueba si la tabla contiene algún elemento con la clave dada.
 *
 * @param clave clave a buscar.
 * @return si existe algún elemento con esa clave.
 */
bool esta(const C &clave) {
   // 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 *consulta*, 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 V &consulta(const C &clave) {

   // 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 esVacia() {
   return _numElems == 0;
}
```

4.4. 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

factor de carga = número_elem / tam (> 80%)

con un vector de tamaño predeterminado.

```
static const int TAM_INICIAL = 10;

Tabla() : _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 inserta(const C &clave, const V &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. Nosotros 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;
}
```

4.5. 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, la clase *Iterador* es una clase interna a *Tabla* con una operación *avanza* 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 ser vacías, y por tanto puede que tengamos que avanzar varias posiciones en el vector.

Para realizar ese recorrido, la clase *Iterador* 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 el conjunto de pares
```

```
 * (clave, valor).
 */
class Iterador {
public:
   void avanza() {
      if (_act == NULL) throw EAccesoInvalido();

      // 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 C& clave() const {
      if (_act == NULL) throw EAccesoInvalido();
      return _act->_clave;
   }

   const V& valor() const {
      if (_act == NULL) throw EAccesoInvalido();
      return _act->_valor;
   }

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

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

private:
   // Para que pueda construir objetos del tipo iterador
   friend class Tabla;

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


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

};
```

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

```
Iterador principio() {

   unsigned int ind = 0;
```

```
   Nodo* act = _v[ind];

   while ((act == NULL) && (ind < _tam - 1)) {
      ++ind;
      act = _v[ind];
   }

   return Iterador(this, act, ind);
}

Iterador final() const {
   return Iterador(this, NULL, _tam);
}
```

Con las modificaciones anteriores, es sencillo imprimir todos los elementos contenidos en una tabla:

```
Tabla<string, int> t;

... // Insertar elementos en la tabla

Tabla<string,int>::Iterador it = t.principio();
while (it != t.final()) {
   cout << "(" << it.clave() << ",  " << it.valor() << ")" << endl;
   it.avanza();
}
```

5. 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.

5.1. Para enteros

Aritmética modular

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

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

Mitad del cuadrado

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

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

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.

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

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.

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

5.2. 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 $8 \leq k \leq 64$.

5.3. 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 *Tabla* 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 *Tabla* 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 $a.hash() == b.hash()$.

6. 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.

7. Para terminar...

Terminamos el tema con la solución a la motivación planteada al principio del tema. La implementación utiliza una tabla para resolver las consultas rápidamente.

```
Lista<std::string> direccionesIP(Lista<std::string> &nombres,
                        Lista<std::string> &ips,
                        Lista<std::string> &consultas) {

   // Insertar pares (nombre, ip) en una tabla
   Tabla<std::string,std::string> dns;
   Lista<std::string>::Iterador itNombres = nombres.principio();
   Lista<std::string>::Iterador itIPs = ips.principio();
   while (itNombres != nombres.final()) {
      dns.inserta(itNombres.elem(), itIPs.elem());
      itNombres.avanza();
      itIPs.avanza();
   }

   // Recuperar IPs asociadas a las consultas
   Lista<std::string> res;
   Lista<std::string>::Iterador itConsultas = consultas.principio();
   while (itConsultas != consultas.final()) {
      res.ponDr(dns.consulta(itConsultas.elem()));
      itConsultas.avanza();
   }
```

```
    return res;
}
```

En un servidor DNS real el registro de los pares (nombre, IP) se haría al iniciar el sistema y después se resolverían las consultas una a una según se fuera produciendo.

Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (**?**) y de (**?**). 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. 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?

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

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

4. 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.

5. 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.

6. Resuelve de nuevo el *problema de las concordancias* (ver motivación del capítulo **??**) utilizando, en lugar de un árbol de búsqueda, una tabla. Analiza el coste de ejecución del algoritmo que obtengas.

7. Resuelve de nuevo el *problema de las referencias cruzadas* (ver ejercicio **??** del capítulo **??**) usando una tabla. Analiza el coste temporal del algoritmo que obtengas.

8. Especifica un TAD *Consultorio* que simule el comportamiento de un consultorio médico simplificado. Dicha especificación 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 dadi 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.

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*?

9. Plantea una implementación del TAD *Consultorio* de ejercicio anterior explicando razonadamente los tipos abstractos de datos que vas a utilizar. 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.