TEMA 6
TABLAS

- Modelo matemático y especificación
- Implementación con acceso basado en búsqueda
- Implementación con acceso casi directo: tablas dispersas
  - Tablas dispersas abiertas
    - Funciones de localización
  - Tablas dispersas cerradas
    - Funciones de relocalización
  - Eficiencia

6.1 Modelo matemático y especificación

- Desde el punto de vista matemático, las tablas son aplicaciones que hacen corresponder a claves, $c \in C$ –C conjunto de claves–, valores, $v \in V$ –V conjunto de valores–. Podemos verlas como una generalización de los árboles de búsqueda, como colecciones de pares (clave, valor), donde el acceso se realiza por las claves y donde, a diferencia de los árboles de búsqueda, no se supone una estructura de árbol –aunque pueden ser implementadas como tales–.

  Para dar un modelo matemático de las tablas hay dos posibilidades naturales:

  – Funciones totales $T : C \to V$, suponiendo en V un elemento distinguido *NoDef*.
  – Funciones parciales $T : C \to V$.

- Las operaciones que nos interesan en las tablas son las que permiten: crear una tabla vacía, insertar una pareja (clave, valor), comprobar si la tabla está vacía, comprobar si una clave está en la tabla, consultar el valor asociado a una clave, y borrar un par (clave, valor).

- Para ciertas aplicaciones de las tablas resulta conveniente que exista un orden entre las claves, y que se disponga de una operación que extraiga la información almacenada en la tabla en forma de lista (o secuencia) de parejas de tipo (Clave, Valor), ordenada por claves.

  – Las tablas ordenadas se pueden implementar directamente como árboles de búsqueda.

- En ocasiones, también resulta útil que una inserción con clave ya presente en la tabla combine el nuevo valor con el antiguo, en lugar de limitarse a reemplazar el valor antiguo por el nuevo. Sin embargo, esto impone restricciones adicionales sobre el tipo de los valores.

Especificación de las tablas como funciones parciales

- La especificación de las tablas como funciones parciales es muy similar a la especificación de los árboles de búsqueda (basta con que los elementos tengan igualdad, no es necesario que tengan orden).

```
tad TABLA[C :: EQ, V :: ANY]
  renombra
    C.Elem a Clave
    V.Elem a Valor
  usa
    BOOL
  tipo
    Tabla[Clave, Valor]
  operaciones
    Nuevo: → Tabla[Clave, Valor]                                        /* gen */
    Inserta: (Tabla[Clave,Valor], Clave, Valor) → Tabla[Clave,Valor]    /* gen */
    consulta: (Tabla[Clave, Valor], Clave) – → Valor                    /* obs */
    borra: (Tabla[Clave, Valor], Clave) → Tabla[Clave, Valor]           /* mod */
    está: (Tabla[Clave, Valor], Clave) → Bool                           /* obs */
    esVacio: Tabla[Clave, Valor] → Bool                                 /* obs */
  ecuaciones
    ∀ t : Tabla[Clave, Valor] : ∀ i, j : Clave : ∀ x, y : Valor :
    Inserta(Inserta(t, i, x), j, y) = Inserta(t, j, y)                  si i == j
    Inserta(Inserta(t, i, x), j, y) = Inserta(Inserta(t, j, y), i,x)    si i /= j
    esVacio(Nuevo)                  = cierto
    esVacio(Inserta(t, i, x))       = falso
    está(Nuevo, j)                  = falso
    está(Inserta(t, i, x), j)       = i == j OR está(t, j)
    def consulta(t, i) si está(t, i)
    consulta(Inserta(t, i, x), j)   = x                                 si i == j
    consulta(Inserta(t, i, x), j)   =f consulta(t, j)                   si i /= j
    borra(Nuevo, j)                 = Nuevo
    borra(Inserta(t, i, x), j)      = borra(t, j)                       si i == j
    borra(Inserta(t, i, x), j)      = Inserta(borra(t, j), i, x)        si i /= j
  errores
    consulta(t, i) si NOT está(t, i)
ftad
```

6.2 Implementación con acceso basado en búsqueda

- Utilizando las estructuras de datos que ya conocemos, podemos plantear diversas implementaciones para el TAD TABLA como colecciones de parejas (Clave, Valor), donde el acceso por clave se implementa como una búsqueda:
  - Vectores de parejas (clave, valor), desordenados u ordenados por clave.

Operación	Vector desordenado	Vector ordenado
Nuevo	$O(1)$	$O(1)$
Inserta	$O(n)$ / $O(1)$ **	$O(n)$
esVacio	$O(1)$	$O(1)$
está	$O(n)$	$O(\log n)$
consulta	$O(n)$	$O(\log n)$
borra	$O(n)$	$O(n)$

  - Secuencias de parejas (clave, valor) implementadas con memoria dinámica, desordenadas u ordenadas por clave.

Operación	Secuencia desordenada	Secuencia ordenada
Nuevo	$O(1)$	$O(1)$
Inserta	$O(n)$ / $O(1)$ **	$O(n)$
esVacio	$O(1)$	$O(1)$
está	$O(n)$	$O(n)$
consulta	$O(n)$	$O(n)$
borra	$O(n)$	$O(n)$

** La complejidad de la inserción en un secuencia o vector desordenado es $O(n)$ si no permitimos claves repetidas y $O(1)$ si las permitimos, con el consiguiente desaprovechamiento de espacio y haciendo necesario que las búsquedas tengan en cuenta esta circunstancia. En el caso de las secuencias ordenadas la complejidad de la búsqueda mejora en el caso promedio, aunque no así en el caso peor.

– Arboles de búsqueda. Suponiendo árboles equilibrados (un árbol con n nodos se dice equilibrado si su talla es de orden log n) se puede conseguir:

Operación	Arbol de búsqueda
Nuevo	$O(1)$
Inserta	$O(\log n)$
esVacio	$O(1)$
está	$O(\log n)$
consulta	$O(\log n)$
borra	$O(\log n)$

➢ En este tema vamos a estudiar una nueva estructura de datos, las tablas dispersas, que en promedio permiten conseguir un coste constante para todas las operaciones sobre las tablas.

6.3 Implementación con acceso casi directo: tablas dispersas

- En este apartado estudiamos técnicas que permiten realizar todas las operaciones en tiempo $O(1)$ en promedio, aunque en el caso peor *Inserta*, *está*, *consulta* y *borra* son $O(n)$.

- La idea es conseguir que las tablas se comporten como vectores, tratando las claves como índices. Esta idea no puede aplicarse directamente cuando el conjunto de todas las claves posibles sea demasiado grande. Por ejemplo, si las claves son cadenas de caracteres con un máximo de 8 caracteres elegidos de un conjunto de 52 caracteres, habría un total de

  $$L = \Sigma\, i : 1 \leq i \leq 8 : 52^i$$

  En una aplicación práctica, la cantidad de cadenas que lleguen a usarse como claves será mucho menor, y es absolutamente impensable reservar un vector de tamaño L para implementar la tabla.

- Una idea alternativa son las *tablas dispersas* (en inglés, *hash table*). La tabla se representa como un vector de tipo

  ```
  Array[0..N–1] of Pareja[Clave, Valor]
  ```

  siendo N suficientemente grande, aunque mucho menor que el número total de claves posibles, y donde lo que se pretende es que dada una clave sea *casi directo* determinar en qué posición del vector se debe encontrar.

- Una característica interesante de un valor concreto de una tabla dispersa es su *tasa de ocupación*, definida como el cociente entre el número n de parejas (Clave, Valor) almacenadas en la tabla, y el número total N de posiciones del vector:

  $$\alpha = n/N$$

  En algunas implementaciones de las tablas dispersas, la tasa de ocupación puede llegar a ser mayor que 1.

Función de localización

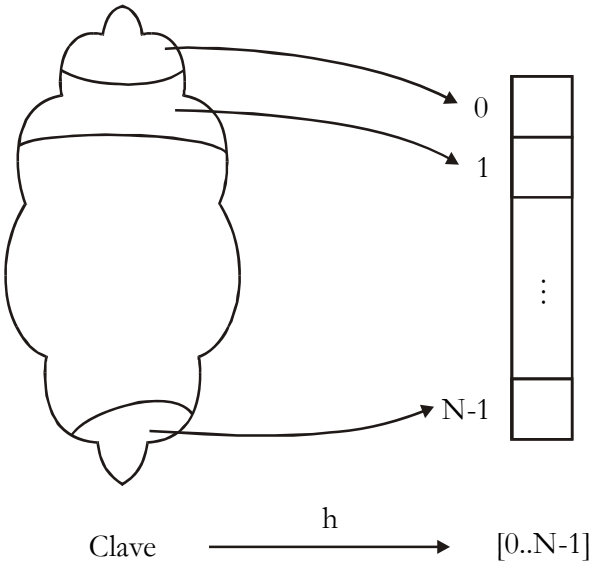
- Para operar con la tabla se necesitará interponer una función que asocie a cada clave un índice del vector:

  `h : Clave → [0..N–1]`

  Dada una clave *c*, el índice *h(c)* es la posición del vector en la que en un principio intentaremos *localizar* la clave. La función *h* que usemos para esto se llamará *función de localización* (en inglés, *hashing function*).

- Al elegir una función de localización se debe buscar
  - Eficiencia. El coste de calcular *h(c)* para una clave *c* dada debe ser bajo.
  - Uniformidad. El reparto de claves entre posiciones 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 $i \in [0..N–1]$. Una función de localización que cumpla esta condición se llama uniforme.

- Gráficamente, la situación es:



- Por ejemplo, supongamos que $N = 16$ y que las claves son cadenas de caracteres. Una posible función de localización es la definida como:

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

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

```
h("Fred") = ord('d') mod 16 = 100 mod 16 = 4
h("Joe")  = ord('e') mod 16 = 101 mod 16 = 5
h("John") = ord('n') mod 16 = 110 mod 16 = 14
```

Colisiones

- Como hemos visto, el dominio de una función de localización siempre suele tener un cardinal mucho mayor que su rango. Por lo tanto, una función de localización *h* no puede ser inyectiva.

  Cuando se encuentran claves *c, 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*.

- Por ejemplo, para la anterior función de colisión:

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

  las cuatro cadenas colisionan en el índice 4 y son sinónimas con respecto a *h*.

Tablas dispersas

- Se llaman así a las tablas implementadas de manera que:
  - La representación concreta de una tabla es un vector *t : Array [0..N–1]* ***of*** *Pareja[Clave, Valor]*
  - Se utiliza una función de localización para el paso de claves a índices.

- Las distintas implementaciones de tablas dispersas se diferencian en dos cosas:
  - El método elegido para el tratamiento de colisiones.
  - La función de localización elegida.

- Vamos a centrarnos primero en estudiar la primera de las características, suponiendo fijada una función de localización *h*, y estudiaremos la segunda característica más adelante.

  Según la técnica que se utilice para el tratamiento de las colisiones, hablamos de:

  - tablas dispersas abiertas, o
  - tablas dispersas cerradas

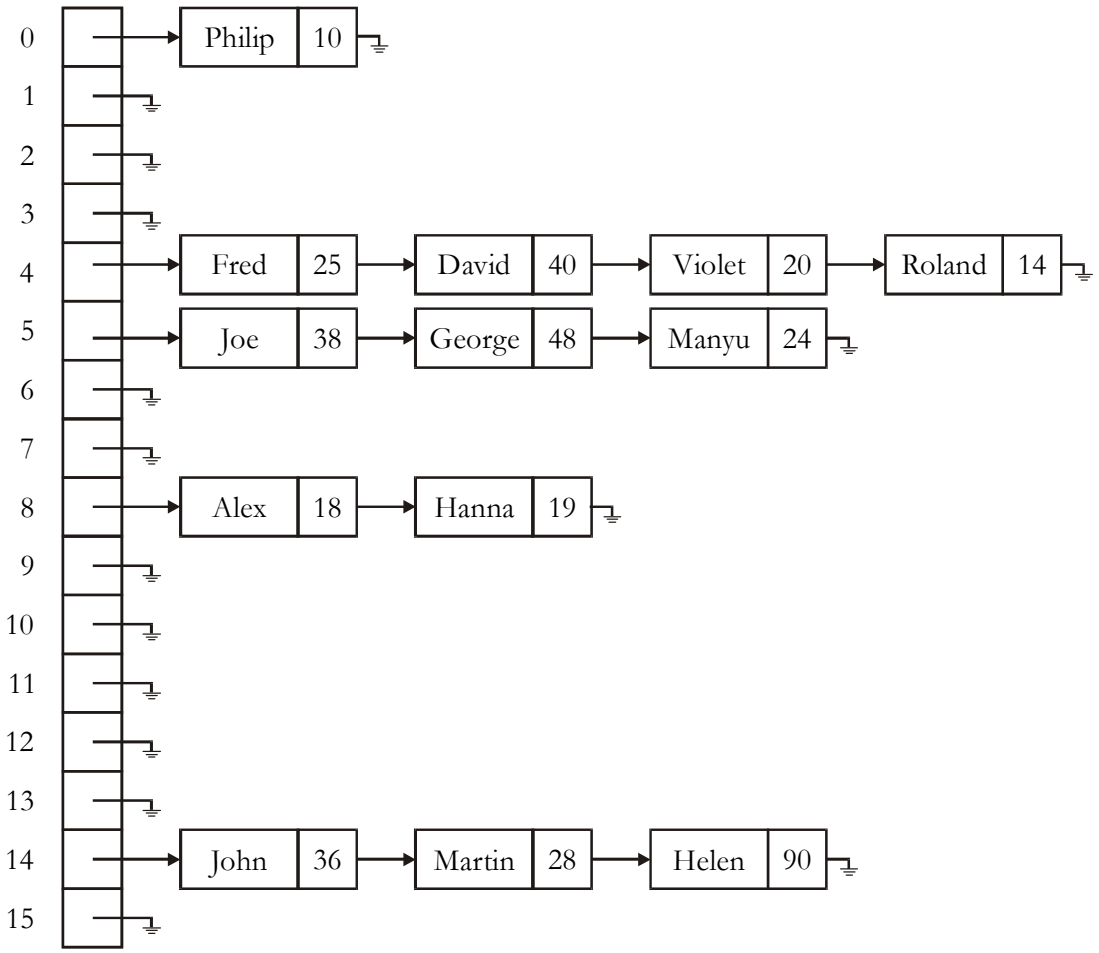
6.3.1 Tablas dispersas abiertas

- Para cada índice *i*, *t[i]* almacena una lista de parejas (Clave, Valor) con claves sinónimas *c*, tal que *h(c)* = *i*. Estas listas se llaman *agrupaciones* o, simplemente, *listas de colisiones*. Las colisiones que se producen en las inserciones se resuelven alargando las agrupaciones; en el borrado, las agrupaciones se acortan.

- Por ejemplo, una tabla dispersa abierta implementada sobre un vector con índices del intervalo *[0..15]* y donde se utilizada la función de localización presentada anteriormente, después de las siguientes operaciones:

```
Nuevo(t);
inserta(t, "Fred", 25); inserta(t, "Alex", 18); inserta(t, "Philip", 10);
inserta(t, "Joe", 38); inserta(t, "John", 36); inserta(t, "Hanna", 19);
inserta(t,"David", 40); inserta(t, "Martin", 28); inserta(t, "Violet", 20);
inserta(t, "George", 48); inserta(t, "Helen", 90); inserta(t, "Manyu", 24);
inserta(t, "Roland", 14);
```

resulta:



Con una tasa de ocupación $\alpha = n/N = 13/16 = 0'8125$

Métodos de localización

- Como se ha dicho más atrás, una buena función de localización debe ser uniforme y fácil de calcular eficientemente.

  Por supuesto, se debe garantizar que la función de localización aplicada sobre claves idénticas devuelva el mismo resultado.

  En general, si la clave es numérica lo más recomendable es calcular el resto módulo el tamaño N de la tabla. Es decir:

  $$h(c) =_{def} c \bmod N$$

  Aunque existen otras métodos más sofisticados que manipulan los números a nivel binario esta solución tan simple suele funcionar bien.

- Si la clave es de un tipo no numérico lo que se suele hacer es obtener un número a partir de la clave y luego aplicar una de las funciones de localización para claves numéricas. Por ejemplo, un carácter se puede localizar a partir de su ordinal.

- El tipo más habitual de clave no numérica son las cadenas de caracteres, y es por ello que se pueden encontrar algunas funciones de conversión específicas para este tipo de datos.

  Un método general es, dada una cadena c con k caracteres de la forma

  $$c = c_1\ c_2\ \ldots\ c_k$$

  obtener su representación numérica interpretando sus caracteres como dígitos en una cierta base B:

  $$\begin{aligned} num(c) &=_{def} \Sigma\ i : 0 \le i < k : B^i \cdot ord(c_{k-i}) \\ &= ord(c_k) + B \cdot ord(c_{k-1}) + \ldots + B^{k-1} \cdot ord(c_1) \end{aligned}$$

- En la elección de los métodos y de las constantes que participan en estos cálculos se suelen utilizar resultados de Teoría de números con el objetivo de obtener una mayor uniformidad en la función de localización. Así, por ejemplo, se obtienen mejores resultados si N y B son números primos.

Implementación de las tablas dispersas abiertas

Tipo representante

```
template <class TClave, class TValor>
class TNodoTabla {
  private:
    TClave _clave;
    TValor _valor;
    TNodoTabla<TClave,TValor> *_sig;
    ...
};

template <class TClave, class TValor>
class TTablaAbierta {
  ...
  private:
     static const int longMax = 10;
     TNodoTabla<TClave,TValor>* _espacio[longMax];
  ...
}
```

Implementación genérica

- A la hora de plantear una implementación genérica de las tablas se nos plantea el problema de qué le debemos exigir a los tipos que pueden actuar como claves de una tabla:
  - Deben ser comparables para poder determinar si dos claves coinciden.
  - Deben tener definida una función que permita *localizar* un valor de ese tipo dentro del vector que representa a la tabla.

- Una función de localización permite, a partir de una clave y el número *longMax* de posiciones del vector, obtener un entero en el rango *[0..longMax-1]*, con un perfil de la forma

  ```
  localiza: (Clave, Entero) → Entero
  ```

  Para llevar a cabo la localización tenemos dos alternativas:
  - incluirla directamente como una función más de las claves, o
  - repartir su implementación entre las claves y la tabla
    - las claves proporcionan una función de perfil

      ```
      num: Clave → Entero
      ```

      que, dada una clave, permite obtener un número entero.

      Esta función debe ser coherente con la igualdad definida sobre las claves, es decir, el resultado ha de ser el mismo para claves iguales.
    - las tablas proporcionan una función de perfil

      ```
      localiza: (Clave, Entero) → Entero
      ```

      que, apoyándose en la función *num* de las claves, localiza la clave dentro del vector concreto.

  Elegimos la segunda alternativa por ser más flexible, separar mejor la responsabilidad de cada tipo y ajustarse al diseño habitual de las funciones de localización:
  - Sin modificar la implementación de las claves se pueden ensayar distintas funciones de localización.
  - Las claves no necesitan conocer la longitud del vector.
  - Como hemos indicado más arriba, las funciones de localización aplicadas sobre claves no numéricas se basan en transformar dichas claves en un valor numérico y aplicar al resultado una técnica de localización.

- El problema de la anterior solución es que no se puede aplicar sobre tipos predefinidos, pues no disponen del método *num* –ni de ningún otro–.

  Para evitar este problema, podemos aprovecharnos de la sobrecarga de funciones en C++:

  – Utilizamos la función *num* en lugar del método *num*
  – Implementamos *num* para los tipos primitivos que nos interese, junto con una implementación genérica que invoca al método *num*, que será la que se aplique cuando las claves no sean de un tipo primitivo.

```
int num( int n ) {
  return n;
}

int num( char ch ) {
  return ch;
}

int num( string cad ) {
  int res = 0;
  for ( int i = 0; i < cad.length(); i++ )
    res += cad[i];
  return res;
}

template <class TClave>
int num ( const TClave& clave ) {
  return clave.num();
}
```

Interfaz de la implementación

```
#include <iostream>

using namespace std;

// Excepciones generadas por las operaciones de este TAD
// Acceso cuando se accede con una clave que no está en la tabla
class EClaveErronea { };

// El tipo TClave debe implementar
//    operator==
//    int TClave::num() const

template <class TClave, class TValor>
class TTablaAbierta;

template <class TClave, class TValor>
class TNodoTabla {
  private:
    TClave _clave;
    TValor _valor;
    TNodoTabla<TClave,TValor> *_sig;
    TNodoTabla( const TClave&, const TValor&, TNodoTabla<TClave,TValor>* );
  public:
    const TClave& clave() const;
    const TValor& valor() const;
    TNodoTabla<TClave,TValor> * sig() const;
    friend TTablaAbierta<TClave,TValor>;
};

template <class TClave, class TValor>
class TTablaAbierta {
  public:

  // Constructoras, destructora y operador de asignación
     TTablaAbierta( );
     TTablaAbierta( const TTablaAbierta<TClave,TValor>& );
     ~TTablaAbierta( );
     TTablaAbierta<TClave,TValor>& operator=( const
                                          TablaAbierta<TClave,TValor>& );
```

```
  // Operaciones de las tablas
     void inserta( const TClave&, const TValor& );
     // Pre : true
     // Post : inserta un par (clave, valor),
     //        si la clave ya está, se sustituye el valor antiguo

     void borra( const TClave& );
     // Pre : true
     // Post : elimina un par (clave, valor) a partir de una clave dada,
     //        si la clave no está, la tabla no se modifica

     // observadoras
     const TValor& consulta( const TClave& ) const throw (EClaveErronea);
     // Pre : esta( clave )
     // Post : devuelve el valor asociado con la clave dada
     // Lanza la excepción EClaveErronea si la tabla no contiene dicha clave

     bool esta( const TClave& ) const;
     // Pre : true
     // Post : devuelve true|false según si la tabla contiene o no la clave

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la tabla está o no vacía

  // Escritura
     void escribe( ostream& salida ) const;

  private:
  // Variables privadas
     static const int longMax = 10;
     TNodoTabla<TClave,TValor>* _espacio[longMax];

  // Operaciones privadas
     void libera();
     void copia( const TTablaAbierta<TClave,TValor>& );
  // función hash
     int localiza( const TClave&) const;
  // función auxiliar de búsqueda del nodo con una clave dada
     void busca ( const TClave&, TNodoTabla<TClave,TValor>* &,
                  TNodoTabla<TClave,TValor>* & ) const;

};
```

Implementación de las operaciones

- Implementación de la clase de los nodos

```
template <class TClave, class TValor>
TNodoTabla<TClave,TValor>::TNodoTabla( const TClave& clave,
                                       const TValor& valor,
                                     TNodoTabla<TClave,TValor>* sig = 0 ) :
  _clave(clave), _valor(valor), _sig(sig) {
};

template <class TClave, class TValor>
const TClave& TNodoTabla<TClave,TValor>::clave() const {
  return _clave;
}

template <class TClave, class TValor>
const TValor& TNodoTabla<TClave,TValor>::valor() const {
  return _valor;
}

template <class TClave, class TValor>
TNodoTabla<TClave,TValor>* TNodoTabla<TClave,TValor>::sig() const {
  return _sig;
}
```

- La función de localización

```
template <class TClave, class TValor>
int TTablaAbierta<TClave,TValor>::localiza( const TClave& clave) const{
  return num(clave) % longMax;
};
```

- La construcción de una tabla vacía

```
template <class TClave, class TValor>
TTablaAbierta<TClave,TValor>::TTablaAbierta( ) {
  for ( int i = 0; i < longMax; i++ )
    _espacio[i] = 0;
};
```

- Las operaciones de inserción, consulta y borrado necesitan de una operación auxiliar que localice un nodo con una clave determinada dentro de una agrupación. Para que esta operación sea también útil en la eliminación, necesitamos que además del puntero al nodo con la clave buscada nos devuelva un puntero al nodo anterior.

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::busca(
  const TClave& clave, TNodoTabla<TClave,TValor>* & act,
                       TNodoTabla<TClave,TValor>* & ant ) const {
  bool encontrado = false;

  ant = 0;
  while ((! encontrado) && (act != 0) ) {
    encontrado = clave == act->clave();
    if ( ! encontrado ) {
      ant = act;
      act = act->sig();
    }
  }
};
```

Nótese que si *p* queda apuntando al primer nodo de la agrupación, entonces el valor de *ant* será 0.

- Inserción.

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::inserta( const TClave& clave,
                                            const TValor& valor ) {
  int i;
  TNodoTabla<TClave,TValor> *act, *ant;
  i = localiza( clave );
  act = _espacio[i];
  busca( clave, act, ant );
  if ( act != 0 )
    act->_valor = valor;
  else {
    act = new TNodoTabla<TClave,TValor>(clave, valor, _espacio[i]);
    _espacio[i] = act;
  }
};
```

- Borrado

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::borra( const TClave& clave ) {
  int i;
  TNodoTabla<TClave,TValor> *act, *ant;

  i = localiza(clave);
  act = _espacio[i];
  busca( clave, act, ant );
  if ( act != 0 ) {
    if ( ant != 0 )
      ant->_sig = act->sig();
    else
      _espacio[i] = act->sig();
    delete act;
  }
};
```

- Consultas

```
template <class TClave, class TValor>
const TValor& TTablaAbierta<TClave,TValor>::consulta(
  const TClave& clave ) const throw (EClaveErronea) {
  int i;
  TNodoTabla<TClave,TValor> *act, *ant;

  i = localiza(clave);
  act = _espacio[i];
  busca( clave, act, ant );
  if ( act == 0 )
    throw EClaveErronea();
  return act->valor();
};


template <class TClave, class TValor>
bool TTablaAbierta<TClave,TValor>::esta( const TClave& clave ) const {
  int i;
  TNodoTabla<TClave,TValor> *act, *ant;

  i = localiza(clave);
  act = _espacio[i];
  busca( clave, act, ant );
  return act != 0;
};
```

```
template <class TClave, class TValor>
bool TTablaAbierta<TClave,TValor>::esVacio( ) const {
  int i = 0;
  bool vacio = true;

  while ( vacio && (i < longMax) ) {
    vacio = _espacio[i] == 0;
    i++;
  }
  return vacio;
};
```

- Operación de escritura

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::escribe( ostream& salida ) const {
  TNodoTabla<TClave,TValor> *act;

  for ( int i = 0; i < longMax; i++ ) {
    act = _espacio[i];
    salida << i << ":";
    while ( act != 0 ) {
      salida << "(" << act->clave() << ", " << act->valor() << + ")";
      act = act->sig();
    }
    salida << endl;
  }
};
```

- Anulación

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::libera() {
  int i;
  TNodoTabla<TClave,TValor> *act, *aux;

  for ( i = 0; i < longMax; i++ ) {
    act = _espacio[i];
    while ( act != 0 ) {
      aux = act->sig();
      delete act;
      act = aux;
    }
  }
};
```

- Copia

```
template <class TClave, class TValor>
void TTablaAbierta<TClave,TValor>::copia(
  const TTablaAbierta<TClave,TValor>& tabla) {
  TNodoTabla<TClave,TValor> *act, *actCopia, *antCopia;

  for ( int i = 0; i < longMax; i++ ) {
    if ( tabla._espacio[i] != 0 ) {
      act = tabla._espacio[i];
      actCopia = new TNodoTabla<TClave,TValor>(act->clave(),
                                               act->valor());
      _espacio[i] = actCopia;
      while ( act->sig() != 0 ) {
        act = act->sig();
        antCopia = actCopia;
        actCopia = new TNodoTabla<TClave,TValor>(act->clave(),
                                                 act->valor());
        antCopia->_sig = actCopia;
      }
    }
    else
      _espacio[i] = 0;
  }
};
```

6.3.2 Tablas dispersas cerradas

- La información almacenada en la tabla se representa por medio de un vector de parejas (Clave, Valor). Cualquier acceso a una tabla vía una clave dada *c* comienza calculando el llamado *índice primario*:

  $i_0 = h(c)$

  Si el índice primario produce colisión con otra clave sinónima, se ensaya un índice alternativo. Como la colisión puede reiterarse, es necesario ensayar una sucesión de índices:

  $i_1 = prueba(1, c),\ i_2 = prueba(2, c),\ \ldots,\ i_m = prueba(m, c),\ \ldots$

  llamada *sucesión de pruebas*. A esta técnica se la conoce como *relocalización*.

- Una técnica de relocalización concreta debe cumplir que para toda clave *c* se puede calcular la sucesión de pruebas:

  $i_m = prueba(m, c) \qquad 0 \le m < N$

  que es una permutación de *[0..N–1]*.
  Por lo tanto, las sucesiones de pruebas de dos claves distintas sólo difieren en el orden, pero no en los elementos.

- Por ejemplo, un método muy sencillo de relocalización, conocido como *relocalización lineal*, consiste en definir la sucesión de pruebas como:

  $i_0 = h(c)$
  $i_m = (i_{m-1} + 1) \bmod N \qquad 1 \le m < N$

  o lo que es lo mismo

  $prueba(m, c) = (h(c) + m) \bmod N$

- Resulta obvio que en una tabla dispersa cerrada es necesario distinguir las posiciones que contienen una pareja (Clave, Valor) de aquellas que no, ya sea equipando al tipo de las claves con un valor especial *claveVacía* o añadiendo un campo adicional en cada uno de los registros almacenados en el vector.

- Todos los algoritmos sobre tablas se basan en una característica que debe cumplir el invariante de la representación: cada clave *c* que aparezca en la tabla aparece necesariamente en una posición de la *ruta* de *c*.

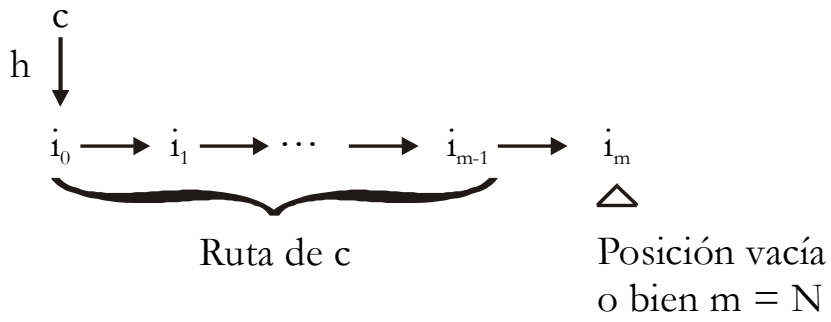
  Dados el vector *v[0..N–1]* que representa a una tabla *t* y una clave *c*, definimos la *ruta* de *c* como el segmento inicial de la sucesión de pruebas de *c* anterior a la primera posición vacía (si la hay):

  `ruta(c, t)` $=_{def}$ `[prueba(0, c), ... , prueba(m–1, c)]`

  siendo *m*

  – el menor *i* comprendido entre 0 y *N–1* tal que *v[prueba(i, c)]* está vacía, o
  – *N* si no hay ninguna posición vacía en el vector.

  Gráficamente



  Observamos que *ruta(c, t) = [ ]* cuando se cumple que *v[prueba(0, c)]* está vacía, es decir, cuando el índice primario de *c* corresponde a una posición vacía.

- En la ruta de una clave, por definición, no pueden existir posiciones vacías. ¿Qué hacer entonces cuando se elimina una clave de la tabla?

  Tapar el hueco, es decir, desplazar una posición hacia *atrás* todas las claves posteriores a la clave borrada en la misma sucesión de pruebas, hasta encontrar una posición vacía, perjudicaría gravemente la eficiencia del borrado.

  La solución consiste en considerar un tercer tipo de posiciones, aquellas que han contenido información que posteriormente se borró.

- Distinguimos tres clases de posiciones en el vector de parejas:
  – *vacías*: aún no se ha introducido información.
  – *ocupadas*: contienen una pareja (Clave, Valor).
  – *borradas*: han contenido información que posteriormente se borró.

- Al determinar la tasa de ocupación en una tabla dispersa cerrada hay que computar no sólo las posiciones ocupadas sino también las borradas, pues su número afecta al funcionamiento de los algoritmos.

  Es por ello que en una tabla dispersa cerrada definimos el tamaño *n* de la tabla como la suma de las posiciones ocupadas o borradas y la tasa de ocupación como el cociente entre el tamaño y la longitud del vector donde se almacena la tabla

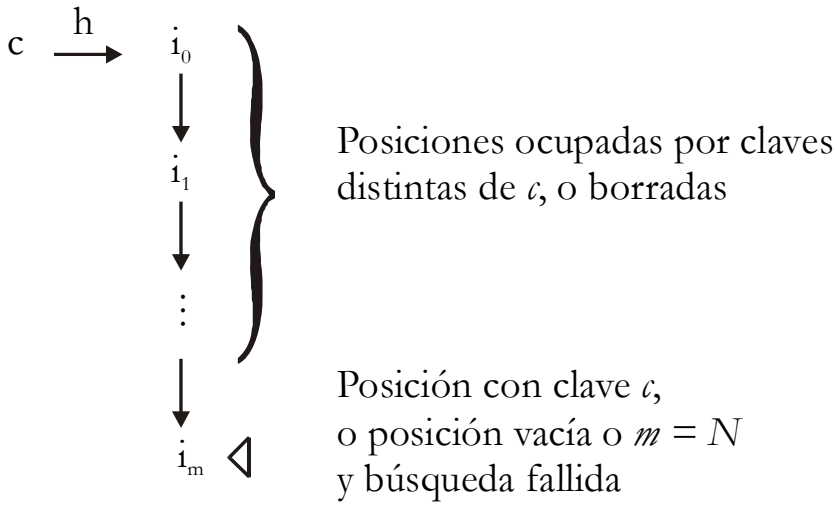
  $$\alpha = n/N$$

  Obviamente, en una tabla cerrada $0 \leq n \leq N$, y con ello $0 \leq \alpha \leq 1$.

Idea de los algoritmos

- Las operaciones de consulta y la de borrado se implementan fácilmente en términos de una operación auxiliar de búsqueda que localiza el índice donde se encuentra la clave buscada o determina que la clave no está en la tabla.

  – Partiendo del índice primario, se sigue la sucesión de pruebas hasta encontrar la clave buscada o una posición vacía. Las posiciones borradas no interrumpen la búsqueda



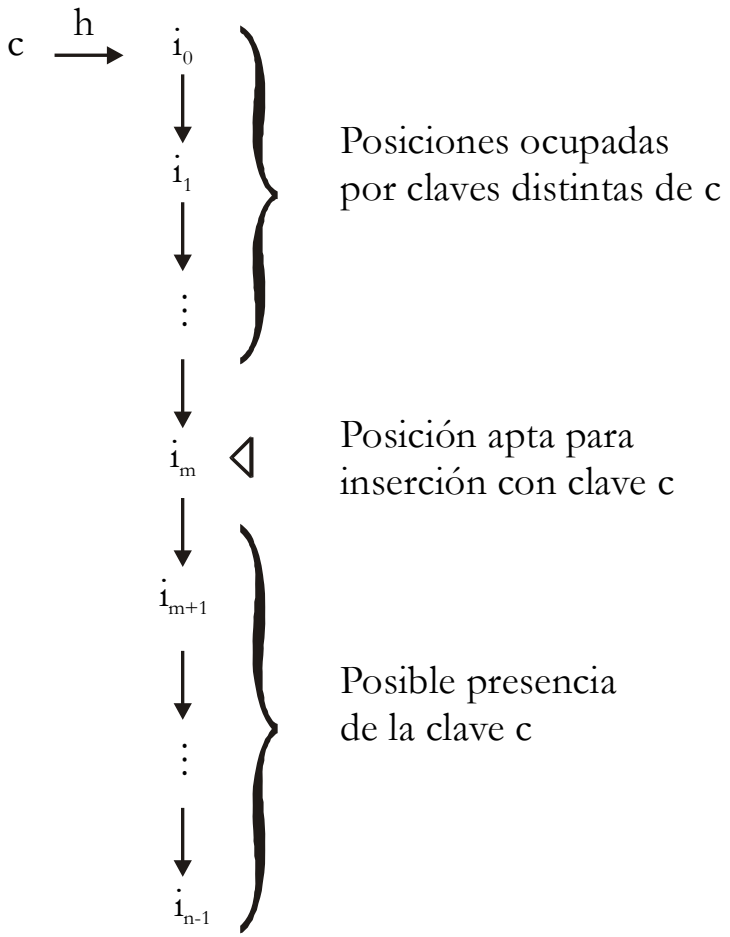
  – Para saber si una clave está en la tabla o devolver el valor asociado con una clave, se realiza una búsqueda con la clave en cuestión.

  – Para eliminar una pareja de (Clave, Valor), se realiza una búsqueda con la clave de borrado. Si la búsqueda tiene éxito, la posición correspondiente se marca como borrada.

- La inserción es un poco más compleja, ya que debemos insertar en la primera posición disponible –vacía o borrada– pero además debemos asegurarnos de que no haya claves repetidas. La idea del algoritmo:
  - Partiendo del índice primario, se sigue la sucesión de pruebas hasta agotar la tabla o encontrar una posición apta para la inserción.

    Si se agota la tabla, se produce error. Si se encuentra una posición apta, debe darse uno de los tres casos siguientes:

    - La posición está vacía. Entonces se inserta en esa posición y se incrementa *n* en 1.
    - La posición está ocupada por la clave de inserción. Entonces se cambia el valor asociado.
    - La posición está borrada. Se colocan la clave y el valor asociado. Se sigue buscando en la serie de pruebas, hasta encontrar una posición vacía, o encontrar la clave de inserción, o completar *n* pruebas. Si se encuentra la clave de inserción, hay que borrar la posición correspondiente.

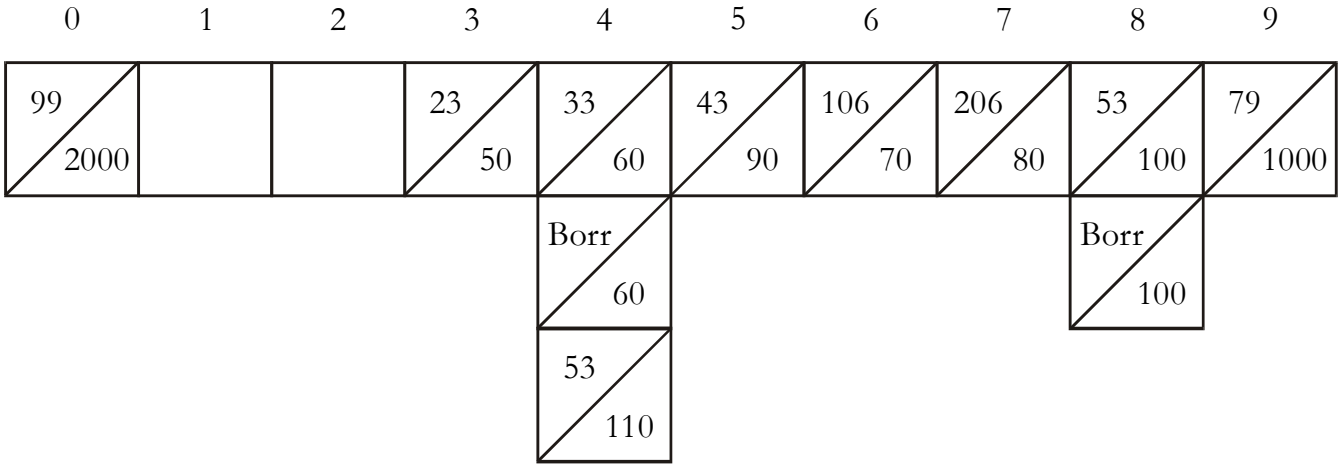


Ejemplo

- Tabla dispersa cerrada, con índices en el intervalo [0..9], usando relocalización lineal y con una función *h* de localización definida como

```
h(c) = c mod 10
```

Operación	Serie de pruebas	resultado
Nuevo(t);		
Inserta(t, 23, 50);	3	
Inserta(t, 33, 60);	3, 4	
Inserta(t, 106, 70);	6	
x := consulta(t, 53);	3, 4, 5	⇒ Error
Inserta(t, 206, 80);	6, 7	
Inserta(t, 43, 90);	3, 4, 5	
y := consulta(t, 33);	3, 4	⇒ 60
Inserta(t, 53, 100);	3, 4, 5, 6, 7, 8	
borra(t, 33);	3, 4	
Inserta(t, 53, 110);	3, 4, 5, 6, 7, 8	
Inserta(t, 79, 1000);	9	
Inserta(t, 99, 2000);	9, 0	
z := consulta(t, 53);	3, 4	⇒ 110
u := consulta(t, 109);	9, 0, 1	⇒ Error



Con lo que la tasa de ocupación resultante queda:

```
α = número de posiciones usadas / tamaño del vector = 8 / 10 = 0'8
```

Métodos de relocalización

- Un método de relocalización permite obtener, a partir de una clave *c* y una función de localización *h*, la sucesión de pruebas para dicha clave, es decir, una permutación $[i_0, ..., i_{N-1}]$ de $[0,...,N-1]$ donde $i_o = h(c)$.

- En general, un método de relocalización será mejor cuanto más cortas sean las rutas a las que, en promedio, da lugar. Para ello, se debe minimizar el solapamiento entre las sucesiones de pruebas de claves distintas.
  - El grado de solapamiento entre las sucesiones de pruebas a las que un método de relocalización da lugar se caracteriza por el tipo de *agrupamientos* que produce.
  - Se dice que un método de relocalización produce agrupamientos *k-arios* si el número de series de pruebas distintas, módulo permutaciones circulares, es $\Theta(N^{k-1})$. Es decir, los agrupamientos serán primarios si sólo hay $1 = N^0$ sucesión de pruebas, módulo permutaciones circulares, secundarios si hay N sucesiones diferentes, y así sucesivamente.
  - El método de relocalización ideal produciría una *relocalización uniforme* (en inglés, *uniform probing hashing*), donde cada sucesión de pruebas sería una permutación aleatoria de $[0..N-1]$, dependiente de la clave de acceso, de forma que no se produjesen agrupamientos.

- La relocalización lineal (en ingles, *Linear probing hashing*), que presentamos anteriormente, es el método más sencillo. La serie de pruebas es:

  $$\text{prueba}(m, c) =_{def} (h(c) + m) \bmod N \quad 0 \le m < N$$

  de forma que:

  $$i_0 = h(c)$$
  $$i_m = (i_{m-1} + 1) \bmod N$$

  Sin embargo, con este método, se producen *agrupamientos primarios* (k=1), es decir, hay una única sucesión de pruebas, módulo permutaciones circulares.

- La relocalización cuadrática (en inglés *Quadratic probing hashing*) genera una serie de pruebas de la forma

  $prueba(m, c) =_{def} (h(c) + m^2) \bmod N \quad 0 \le m < N$

  Con esta construcción, la sucesión de pruebas en general no es una permutación de *[0..N–1]* y, por lo tanto, no recorre toda la tabla.

  Se sabe que si *N* es un número primo de la forma *4k+3*, la siguiente sucesión cuadrática de pruebas sí es una permutación de *[0..N–1]*:

  $$\begin{aligned} prueba(0, c) &= h(c) \\ prueba(2j-1, c) &= (h(c) + j^2) \bmod N \quad 1 \le j \le (N-1)/2 \\ prueba(2j, c) &= (h(c) - j^2) \bmod N \quad 1 \le j \le (N-1)/2 \end{aligned}$$

  Este método produce *agrupamientos secundarios* (*k*=2) ya que genera $\Theta(N)$ sucesiones de pruebas distintas, módulo permutaciones circulares.

- La relocalización doble (en inglés *Double hashing*) consiste en utilizar una segunda función de localización *k* para calcular un incremento que, junto con el índice primario, determina la sucesión de pruebas:

  $prueba(m, c) = (h(c) + m \cdot k(c)) \bmod N \quad 0 \le m < N$

  Se cumple:

  $$\begin{aligned} i_0 &= h(c) \\ i_m &= (i_{m-1} + d) \bmod N \quad d = k(c) \in [1..N-1] \end{aligned}$$

  Con una buena elección de *N, h* y *k* el comportamiento de este método es muy similar en la práctica a la relocalización uniforme. Una buena elección es:

  – *N* primo, tal que *N–2* primo
  – $h(c) =_{def} num(c) \bmod N$
  – $k(c) =_{def} (num(c) \bmod N-2) + 1$

  Nótese que si *N* es primo entonces *N* y *d* son primos entre sí y, por lo tanto, la serie de pruebas recorre toda la tabla

Implementación de las tablas dispersas cerradas

Restricciones sobre el tipo de las claves

- Para marcar los distintos tipos de posiciones del vector, podemos
  - suponer que el tipo de las claves está equipado con dos valores especiales *ClaveVacía* y *ClaveBorrada*, o
  - añadir un campo adicional en cada uno de los registros almacenados en él.

  Para no complicar la implementación de las claves, optamos por la segunda solución y definimos un tipo enumerado para marcar las posiciones del vector

```
enum TPosicion {vacia, ocupada, borrada};
```

- Para implementar la relocalización tampoco necesitamos modificar el tipo de las claves pues basta con que éstas dispongan de la operación *num* que las hace *localizables*.

Tipo representante

- La estructura de datos

```
enum TPosicion {vacia, ocupada, borrada};

template <class TClave, class TValor>
class TNodoTabla {
  private:
    TClave _clave;
    TValor _valor;
    TPosicion _posicion;
  ...
};

template <class TClave, class TValor>
class TTablaCerrada {
  ...
  private:
     static const int longMax = 23; // se recomienda primo > 20
     TNodoTabla<TClave,TValor> _espacio[longMax];
  ...
};
```

Interfaz de la implementación

```
#include <iostream>
using namespace std;

// Excepciones generadas por las operaciones de este TAD
// Acceso cuando se accede con una clave que no está en la tabla
class EClaveErronea { };
// Inserción cuando el array está lleno
class EDesbordamiento { };

// El tipo TClave debe implementar
//    operator==
//    int TClave::num() const

template <class TClave, class TValor>
class TTablaCerrada;

enum TPosicion {vacia, ocupada, borrada};

template <class TClave, class TValor>
class TNodoTabla {
  private:
    TClave _clave;
    TValor _valor;
    TPosicion _posicion;
    TNodoTabla( );
  public:
    const TClave& clave() const;
    const TValor& valor() const;
    const TPosicion& posicion() const;
    friend TTablaCerrada<TClave,TValor>;
};

template <class TClave, class TValor>
class TTablaCerrada {
  public:

  // Constructoras, destructora y operador de asignación
     TTablaCerrada( );
     TTablaCerrada( const TTablaCerrada<TClave,TValor>& );
     ~TTablaCerrada( );
     TTablaCerrada<TClave,TValor>& operator=(
       const TTablaCerrada<TClave,TValor>& );
```

```
  // Operaciones de las tablas
     void inserta( const TClave&, const TValor& );
     // Pre : true
     // Post : inserta un par (clave, valor),
     //        si la clave ya está, se sustituye el valor antiguo

     void borra( const TClave& );
     // Pre : true
     // Post : elimina un par (clave, valor) a partir de una clave dada,
     //        si la clave no está, la tabla no se modifica

     // observadoras
     const TValor& consulta( const TClave& ) const throw (EClaveErronea);
     // Pre : esta( clave )
     // Post : devuelve el valor asociado con la clave dada
     // Lanza la excepción EClaveErronea si la tabla no contiene dicha clave

     bool esta( const TClave& ) const;
     // Pre : true
     // Post : devuelve true|false según si la tabla contiene o no la clave

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la tabla está o no vacía

  // Escritura
     void escribe( ostream& salida ) const;

  private:
  // Variables privadas
     static const int longMax = 23; // se recomienda primo > 20
     TNodoTabla<TClave,TValor> _espacio[longMax];

  // Operaciones privadas
     void libera();
     void copia( const TTablaCerrada<TClave,TValor>& );
  // función auxiliar de búsqueda del nodo con una clave dada
     int busca ( const TClave& ) const;
};
```

Implementación de las operaciones

- No implementamos el método de relocalización como una función aparte para mejorar la eficiencia. El código encargado de generar la sucesión de pruebas se incluirá directamente en los algoritmos que lo necesiten. Tampoco necesitamos función de localización porque la clave primaria se obtiene como el primer elemento de la sucesión de pruebas.

  Utilizaremos relocalización doble.

- Implementación de los nodos

```
template <class TClave, class TValor>
TNodoTabla<TClave,TValor>::TNodoTabla( ) :
  _posicion(vacia) {
};

template <class TClave, class TValor>
const TClave& TNodoTabla<TClave,TValor>::clave() const {
  return _clave;
}

template <class TClave, class TValor>
const TValor& TNodoTabla<TClave,TValor>::valor() const {
  return _valor;
}

template <class TClave, class TValor>
const TPosicion& TNodoTabla<TClave,TValor>::posicion() const {
  return _posicion;
}
```

- La construcción de una tabla vacía

```
template <class TClave, class TValor>
TTablaCerrada<TClave,TValor>::TTablaCerrada( ) {
  for ( int i = 0; i < longMax; i++ )
    _espacio[i]._posicion = vacia;
};
```

- Función que detecta si una tabla está vacía.

```
template <class TClave, class TValor>
bool TTablaCerrada<TClave,TValor>::esVacio( ) const {
  int i = 0;
  bool vacio = true;

  while ( vacio && (i < longMax) ) {
    vacio = _espacio[i].posicion() != ocupada;
    i++;
  }
  return vacio;
};
```

- Las operaciones *está, consulta* y *borra* se programan con ayuda de una función auxiliar privada de búsqueda.

```
template <class TClave, class TValor>
int TTablaCerrada<TClave,TValor>::busca( const TClave& clave ) const {
  int i, n, d, m;
  bool encontrado;

  n = num(clave);
  i = n % longMax;
  d = (n % (longMax-2)) + 1;
  m = 0; // contabiliza el número de pruebas por si la tabla está llena
  encontrado = false;
  while ( (! encontrado) && (_espacio[i].posicion() != vacia) &&
          (m < longMax) ) {
    if ( _espacio[i]._posicion == ocupada )
      encontrado = clave == _espacio[i].clave();
    if ( ! encontrado ) {
      m++;
      i = (i + d) % longMax;
    }
  }
  if ( ! encontrado )
    i = longMax;
  return i;
};
```

Y ahora resulta sencillo implementar *consulta*, *está* y *borra*

```
template <class TClave, class TValor>
const TValor& TTablaCerrada<TClave,TValor>::consulta(
  const TClave& clave ) const throw (EClaveErronea) {
  int i;

  i = busca(clave);
  if ( i == longMax )
    throw EClaveErronea();
  return _espacio[i].valor();
};

template <class TClave, class TValor>
bool TTablaCerrada<TClave,TValor>::esta( const TClave& clave ) const {
  int i;

  i = busca(clave);
  return i != longMax;
};

template <class TClave, class TValor>
void TTablaCerrada<TClave,TValor>::borra( const TClave& clave ) {
  int i;

  i = busca(clave);
  if ( i != longMax )
    _espacio[i]._posicion = borrada;
};
```

- La inserción

```
template <class TClave, class TValor>
void TTablaCerrada<TClave,TValor>::inserta( const TClave& clave,
                                            const TValor& valor ) {
  int i, n, d, m;
  bool encontrado;

  n = num(clave);
  i = n % longMax;
  d = (n % (longMax-2)) + 1;
  m = 0; //contabiliza el número de pruebas por si la tabla está llena
  encontrado = false;
```

```
  while ( ( ! encontrado ) && (_espacio[i].posicion() == ocupada) &&
          (m < longMax) ) {
    if ( _espacio[i]._posicion == ocupada )
      encontrado = clave == _espacio[i].clave();
    if ( ! encontrado ) {
      m++;
      i = (i + d) % longMax;
    }
  }
  if ( m == longMax )
    throw EDesbordamiento();
  else if ( encontrado )
    _espacio[i]._valor = valor;
  else if ( _espacio[i].posicion() == vacia ) {
    _espacio[i]._clave = clave;
    _espacio[i]._valor = valor;
    _espacio[i]._posicion = ocupada;
  }
  else if ( _espacio[i].posicion() == borrada ) {
    _espacio[i]._clave = clave;
    _espacio[i]._valor = valor;
    _espacio[i]._posicion = ocupada;
    m++;
    i = (i + d) % longMax;
    while ( ( ! encontrado ) && (_espacio[i].posicion() != vacia) &&
            (m < longMax) ) {
      if ( _espacio[i].posicion() == ocupada )
        encontrado = clave == _espacio[i].clave();
      if ( ! encontrado ) {
        m++;
        i = (i + d) % longMax;
      }
    }
    if ( encontrado )
      _espacio[i]._posicion = borrada;
  }
};
```

- Escritura

```
template <class TClave, class TValor>
void TTablaCerrada<TClave,TValor>::escribe( ostream& salida ) const {

  for ( int i = 0; i < longMax; i++ ) {
    salida << i << ":";
    if ( _espacio[i].posicion() == ocupada )
      salida << "(" << _espacio[i].clave() << ", "
             << _espacio[i].valor()<< ")";
    else if ( _espacio[i].posicion() == borrada )
      salida << "###";
    salida << endl;
  }
};
```

- Anulación

```
template <class TClave, class TValor>
void TTablaCerrada<TClave,TValor>::libera() {
};
```

- Copia

```
template <class TClave, class TValor>
void TTablaCerrada<TClave,TValor>::copia(
  const TTablaCerrada<TClave,TValor>& tabla) {
  for ( int i = 0; i < longMax; i++ ) {
    _espacio[i]._posicion = tabla._espacio[i]._posicion;
    if ( _espacio[i]._posicion == ocupada ) {
      _espacio[i]._clave = tabla._espacio[i].clave();
      _espacio[i]._valor = tabla._espacio[i].valor();
    }
  }
};
```

6.3.3 Eficiencia

- Los resultados sobre eficiencia vienen expresados en términos de la tasa de ocupación $\alpha = n/N$, donde n es el número de parejas en las tablas abiertas y el número de posiciones usadas en las tablas cerradas.

  En el caso peor, una búsqueda o inserción puede requerir tiempo $O(n)$. Las evaluaciones experimentales, sin embargo, muestran para las tablas dispersas un rendimiento superior al de los árboles de búsqueda equilibrados.

- Los análisis probabilísticos permiten estimar la complejidad en promedio esperada para los distintos modelos de tabla dispersa, suponiendo que la función de localización es *uniforme*; es decir, para cada $i \in [0..N-1]$, la probabilidad de que una clave aleatoria c cumpla $h(c) = i$ debe ser $1/N$.

  Se estiman dos valores diferentes

  – C_n el número promedio de accesos a posiciones de una tabla de tamaño n para localizar una clave presente en la tabla.
  – C'_n el número promedio de accesos a posiciones de una tabla de tamaño n para insertar con una nueva clave no presente en la tabla.

  Se cumple entonces:

  (a) Para tablas abiertas

  $$C_n \approx 1 + \alpha/2 \qquad C'_n \approx \alpha$$

  (b) Para tablas cerradas con relocalización lineal

  $$C_n \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad C'_n \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

  (c) Para tablas cerradas con relocalización cuadrática o doble:

  $$C_n \approx -\left(\frac{1}{\alpha}\right)\ln(1-\alpha) \qquad C'_n \approx \frac{1}{1-\alpha}$$

  Suponiendo $\alpha = 0'8$, las fórmulas aproximadas anteriores quedan:

  (a) $C_n \approx 1'4$ $\quad C'_n \approx 0'8$

  (b) $C_n \approx 3$ $\quad C'_n \approx 13$

  (c) $C_n \approx 2'0118$ $\quad C'_n \approx 5$

- Para una tasa de ocupación fijada, las tablas abiertas son más eficientes que las cerradas. Como contrapartida, las tablas abiertas consumen más memoria.

- En ambos tipos de tablas la eficiencia puede degradarse si la tasa de ocupación aumenta en exceso. Algunas posibilidades para prevenir esta circunstancia son:
  - En las tablas cerradas, se puede incluir una operación que "limpie" la tabla, vaciando las posiciones borradas y reestructurando las ocupadas, de forma que todas las informaciones con claves sinónimas queden en posiciones consecutivas de su serie de pruebas, a partir de la posición primaria que corresponda.
  - En ambos tipos de tablas, se puede incluir una operación que duplique el tamaño del vector. Para ello, es necesario cambiar la función de localización *h* a otra con rango doble del anterior, y reorganizar el vector representante de la tabla.

  Ambas operaciones son costosas, pero debido a la disminución que producen en la tasa de ocupación, son operaciones beneficiosas desde el punto de vista de un análisis amortizado del tiempo requerido por una serie de operaciones de acceso a la misma tabla.

  Para poder aplicar estas técnicas es conveniente que las tablas almacenen explícitamente su tamaño *n* para que así resulte sencillo detectar cuándo la tasa de ocupación sobrepasa el límite prefijado.