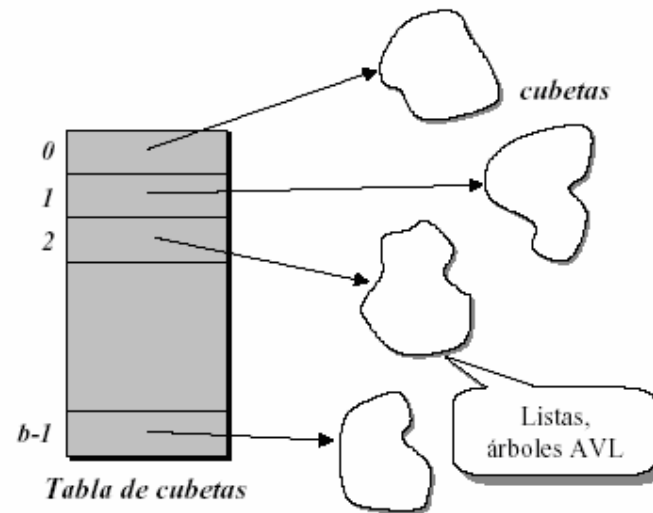


TABLAS HASH

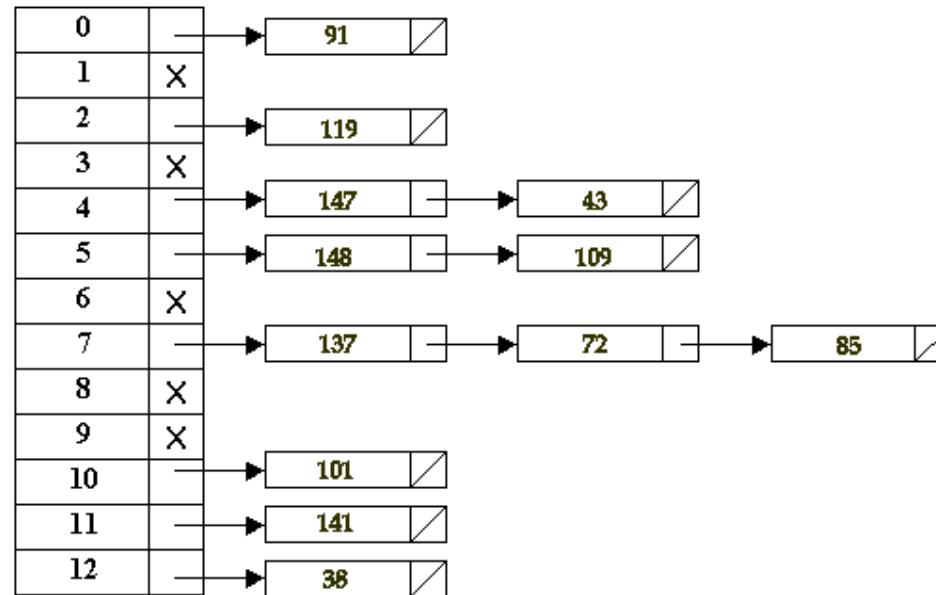
INTRODUCCIÓN

- Intuitivamente, una tabla es una colección de elementos, cada uno de los cuales tiene una clave y una información asociada.
- Esta aproximación consiste en proceder, no por comparaciones entre valores clave, sino encontrando alguna función $h(k)$, función de dispersión, que nos dé directamente la localización de la clave k en la tabla.
- La función de dispersión se aplica a lo que habitualmente se denomina como la clave del elemento (y que en ocasiones será él mismo). Se denomina valor de dispersión al valor $h(\text{clave_de_}x)$, y cubetas a las clases. Se dice que x pertenece a la cubeta de $h(\text{clave_de_}x)$.
- El objetivo será encontrar una función hash que provoque el menor número posible de colisiones (ocurrencias de sinónimos), aunque esto es solo un aspecto del problema, el otro será el de diseñar métodos de resolución de colisiones cuando éstas se produzcan.



TABLAS DE DISPERSIÓN ABIERTAS

- La manera más simple de resolver una colisión es construir, para cada localización de la tabla, una lista enlazada de registros cuyas claves caigan en esa dirección. Puede ser LIFO o FIFO.



Resultado usando LIFO.

CLASE ABSTRACTA

```
template <class B, class H, class T>

class hashTable {
    friend class hashTableIterator<B, H, T>;
public:
    // constructores
```

```

hashTable (const unsigned int & max,
unsigned int (*f) (const H &));
hashTable (const hashTable<B, H, T> & source);
virtual bool isEmpty () const;
/* Produce: cierto si la tabla receptora está vacía. */
virtual void deleteAllValues ();
/* Modifica: la tabla receptora haciéndola vacía. */
protected:
// área de datos
const unsigned int tablesize;
vector<B *> buckets;
unsigned int (*hashfun) (const H &);
// operaciones internas
unsigned int hash (const H & key) const;
/* Necesita: una clave.
   Produce: el valor de dispersión de la clave dada. */
virtual iterator<T> * makeIterator (const unsigned int &) = 0;
/* Necesita: un número de clase o cubeta.
   Produce: un iterador para la clase dada. */
};

```

```

template <class B, class H, class T>

hashTable<B, H, T>::hashTable (const unsigned int & max,
                                unsigned int (*f)(const H &))
    : tablesize(primo(max)), buckets(primo(max)), hashfun(f)
{
    for (unsigned int i = 0; i < tablesize; i++) {
        buckets[i] = new B;
        assert(buckets[i] != 0);
    }
}

```

```

template <class B, class H, class T>

hashTable<B, H, T>::hashTable
    (const hashTable<B, H, T> & source)
    : tablesize(source.tablesize), buckets(source.buckets),
      hashfun(source.hashfun)
{
    for (unsigned int i = 0; i < tablesize; i++) {
        buckets[i] = new B(*source.buckest[i]);
        assert(buckets[i] != 0);
    }
}

```

```

template <class B, class H, class T>

bool hashTable<B, H, T>::isEmpty () const
{
    for (int i = 0; i < tablesize; i++)
        if (!buckets[i]->isEmpty())
            return false;
    // todas las cubetas están vacías
    return true;
}

```

```

template <class B, class H, class T>

void hashTable<B, H, T>::deleteAllValues()
{
    // borrar todos los elementos en cada cubeta
    for (int i = 0; i < tablesize; i++)

```

```
        buckets[i]->deleteAllValues();  
    }
```

```
template <class B, class H, class T>  
  
unsigned int hashTable<B, H, T>::hash (const H & key) const  
{  
    return (*hashfun)(key) % tablesize;  
}
```

```
template <class B, class H, class T>  
  
class hashTableIterator : public iterator<T> {  
public:  
    // constructor  
    hashTableIterator(hashTable<B, H, T> & aHashTable);  
    hashTableIterator(const hashTableIterator<B, H, T> & source);  
    // protocolo iterador  
    virtual bool init ();  
    virtual T operator () ();  
    virtual bool operator ! ();  
    virtual bool operator ++ ();  
    virtual bool operator ++ (int);  
    virtual T & operator = (const T & value);  
protected:  
    // área de datos  
    unsigned int currentIndex; // índice de la cubeta actual  
    iterator<T> * itr; // iterador de la cubeta actual  
    hashTable<B, H, T> & base;  
    // operación interna  
    bool getNextIterator ();
```

```
/* Efecto: obtiene el iterador sobre la siguiente  
cubeta no vacía.  
Produce: cierto si encuentra una cubeta no vacía. */  
};
```

```
template <class B, class H, class T>  
  
hashTableIterator<B, H, T>::hashTableIterator  
    (hashTable<B, H, T> & aHashTable)  
    : base(aHashTable), currentIndex(0), itr(0)  
{  
    init();  
}
```

```
template <class B, class H, class T>  
  
bool hashTableIterator<B, H, T>::init ()  
{  
    // obtener la primera cubeta no vacia  
    currentIndex = 0;  
    return getNextIterator();  
}
```

```
template <class B, class H, class T>  
  
T hashTableIterator<B, H, T>::operator () ()  
{  
    return (*itr)();  
}
```

```

template <class B, class H, class T>

bool hashTableIterator<B, H, T>::operator ! ()
{
    return itr != 0;
}

```

```

template <class B, class H, class T>

bool hashTableIterator<B, H, T>::operator ++ ()
{
    // si se puede, avanzar en el iterador actual
    if (itr && (*itr)++)
        return true;
    // en caso contrario obtener el siguiente iterador
    currentIndex++;
    return getNextIterator();
}

```

```

template <class B, class H, class T>

bool hashTableIterator<B, H, T>::getNextIterator ()
{
    if (itr != 0)
        delete itr;
    // buscar el nuevo cubo en el que iterar
    for (; currentIndex < base.tablesize; currentIndex++) {
        itr = base.makeIterator(currentIndex);
        assert(itr != 0);
        if (itr->init()) // si hay elemento actual retornar
            return true;
    }
}

```



```

        delete itr;
    }
    itr = 0; // no hay más elementos
    return false;
}

```

REPRESENTACIÓN BASADA EN LISTAS

```

// Clase hashList
// Implementación de la clase hashTable con listas.
// La clave es el valor.
template <class T>

class hashList : public hashTable<setList<T>, T, T> {
public:
    // constructores
    hashList (const unsigned int & max,
              unsigned int (*f) (const T &));
    hashList (const hashList<T> & source);
    void add (const T & value);
    /* Necesita: un valor de tipo T.
       Modifica: la tabla receptora añadiendo, si no existe,
                  el valor dado. */
    bool includes (const T & value) const;
    /* Necesita: un valor de tipo T.
       Produce: cierto si el valor dado se encuentra
                  en la tabla receptora. */
    void remove (const T & value);
    /* Necesita: un valor de tipo T.

```

*Modifica: la tabla receptora eliminando de la misma,
si existe, el valor dado. */*

```
protected:
    // operación interna
    virtual iterator<T> * makeIterator (const unsigned int & i );
};
```

```
template <class T>

hashList<T>::hashList (const unsigned int & max,
                      unsigned int (*f) (const T &))
    : hashTable<setList<T>, T, T>(max, f)
{}

```

```
template <class T>

void hashList<T>::add (const T & value)
{
    buckets[hash(value)]->add(value);
}

```

```
template <class T>

bool hashList<T>::includes (const T & value) const
{
    return buckets[hash(value)]->includes(value);
}

```

```
template <class T>
```

```
void hashList<T>::remove (const T & value)
{
    buckets[hash(value)]->remove(value);
}
```

```
template <class T>

iterator<T> * hashList<T>::makeIterator (const unsigned int & i)
{
    return new setListIterator<T>(*buckets[i]);
}
```

```
template <class T> class hashListIterator
    : public hashTableIterator<setList<T>, T, T> {
public:
    hashListIterator (hashList<T> & aHashList);
    hashListIterator (const hashListIterator<T> & source);
};
```

```
template <class T>

hashListIterator<T>::hashListIterator (hashList<T>
                                       & aHashList)
    : hashTableIterator<setList<T>, T, T>(aHashList)
{
}
```

```
template <class T>
```

```

hashListIterator<T>::hashListIterator
    (const hashListIterator<T> & source)
    : hashTableIterator<setList<T>, T, T>(source)
{
}

```

OTRAS REPRESENTACIONES

Representación Basada en Diccionarios

```

template <class H, class T>

class hashDictionary
    : public hashTable<dictionaryList<H, T>, H,
    association<H, T> *> {
public:
    // constructores
    hashDictionary (const unsigned int & max,
    unsigned int (*f) (const H &));
    hashDictionary (const unsigned int & max,
    unsigned int (*f) (const H &), const T & initial);
    hashDictionary (const hashDictionary<H, T> & source);
    // operación de acceso
    T & operator [] (const H & key);
    // test de pertenencia
    bool includesKey (const H & key) const;
    // operación de borrado
    void removeKey (const H & key);

```

protected:

```
virtual iterator<association<H, T> *> *  
    makeIterator (const unsigned int & i);  
};
```

template <class H, class T>

```
hashDictionary<H, T>::hashDictionary  
    (const unsigned int & max,  
     unsigned int (*f) (const H &), const T & initial)  
    : hashTable<dictionaryList<H, T>, H,  
                association<H, T> *>(max, f)  
    {  
        for (unsigned int i = 0; i < tablesize; i++)  
            buckets[i]->setInitial(initial);  
    }
```

template <class H, class T>

```
T & hashDictionary<H, T>::operator [] (const H & key)  
{  
    return buckets[hash(key)]->operator [] (key);  
}
```

template <class H, class T>

```
iterator<association <H, T> *> *  
hashDictionary<H, T>::makeIterator (const unsigned int & i)  
{  
    return new dictionaryListIterator<H, T>(*buckets[i]);  
}
```

```
}
```

```
template <class H, class T>

class hashDictionaryIterator
: public hashTableIterator<dictionaryList<H, T>, H,
  association<H, T> *> {
public:
  hashDictionaryIterator
    (hashDictionary<H, T> & aHashDictionary);
  hashDictionaryIterator
    (const hashDictionaryIterator<H, T> & source);
};
```

```
template <class H, class T>

hashDictionaryIterator<H, T>::hashDictionaryIterator
  (hashDictionary<H, T> & aHashDictionary)
: hashTableIterator<dictionaryList<H, T>,
  H, association<H, T> *>(aHashDictionary)
{
}
```

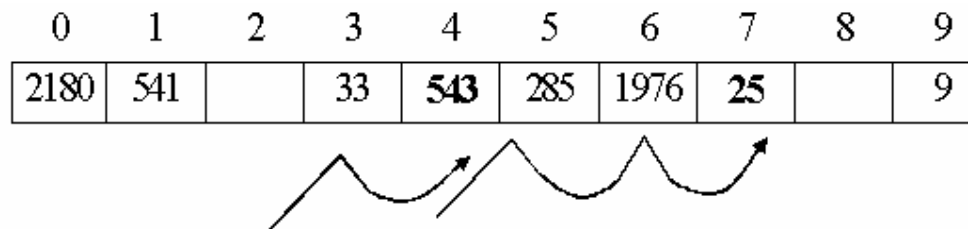
```
template <class H, class T>

hashDictionaryIterator<H, T>::hashDictionaryIterator
  (const hashDictionaryIterator<H, T> & source)
: hashDictionaryIterator<dictionaryList<H, T>,
  H, association<H, T> *>(source)
{
}
```



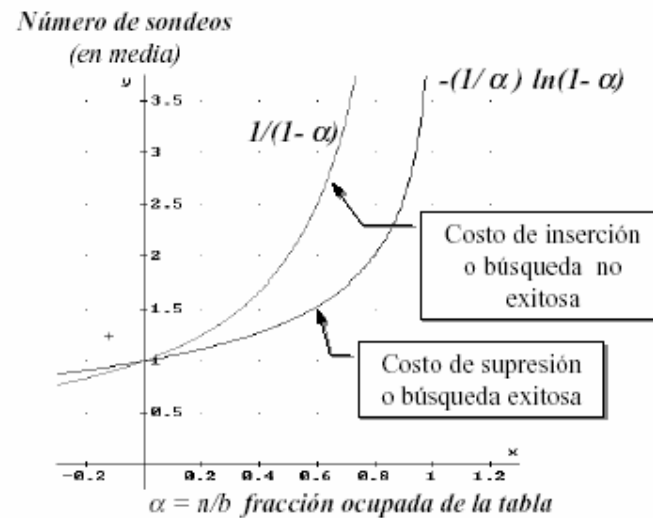
TABLAS DE DISPERSIÓN CERRADAS

- En una tabla de dispersión cerrada sólo se permite un elemento por clase, ya que los miembros de la colección se almacenan directamente en la tabla, en vez de usar ésta para almacenar las direcciones de las cubetas.
- Deberá existir una estrategia de redistribución (función de rehashing) para miembros de la colección con el mismo valor de dispersión.
- Así, si se intenta colocar x en la cubeta $h(\text{clave_de_}x)$ y ésta ya tiene un elemento (situación denominada colisión), la estrategia de redistribución elige una sucesión de cubetas alternativas $h1(\text{clave_de_}x)$, $h2(\text{clave_de_}x)$, ... hasta encontrar una libre en la que poder ubicar el elemento.



ANÁLISIS DE DISPERSIÓN CERRADA

Número de intentos, en promedio, para llenar la tabla



```
template <class H, class T>
class hashVector {
    friend class hashVectorIterator<H, T>;
public:
    // constructores
    hashVector (unsigned int max,
```

Número de
colisiones


```

        unsigned int (*hash)(const H &),
        unsigned int (*rhash)(const H &, int));
hashVector (unsigned int max,
            unsigned int (*hash)(const H &),
            unsigned int (*rhash)(const H &, int),
            const T & init);
hashVector (const hashVector<H, T> & source);
// destructor
~hashVector ();
// test de vacío
bool isEmpty () const;
// operación de índice (adición, acceso y modificación)
T & operator [] (const H & aKey);
// operación de borrado
void remove (const H & aKey);
// test de pertenencia
bool includes (const H & aKey);
// borrado de todos los elementos
void deleteAllValues ();
protected:
// área de datos
vector<association<H,T> *> data; // tabla de cubetas
vector<bool> erase; // se borró el contenido de una
// cubeta?
unsigned int numberOfSlots; // cubetas nunca ocupadas
unsigned int numberOfElements; // elementos en la tabla
unsigned int tablesize; // dimensión de la tabla
T initial; // valor por defecto
unsigned int (*hashfun) (const H &); // función hash
unsigned int (*rhashfun) (const H &, int); // función
// de rehash

// operaciones internas
unsigned int firstEqualEmpty (const H &) const;

```

```

/* Necesita: una clave.
   Produce: la cubeta que contiene la clave dada, o la
               primera cubeta nunca ocupada que se
               encuentra en la sucesión de cubetas que se
               obtiene al aplicar la función de dispersión y,
               si es necesario, las funciones de redispersión
               a la clave dada. */
unsigned int firstEqualEraseEmpty (const H &) const;
/* Necesita: una clave.
   Produce: la cubeta que contiene la clave dada, o la
               primera cubeta nunca ocupada o en la que
               se borró el contenido, en la sucesión de cubetas
               generada al aplicar la función de dispersión
               y, si es necesario, las funciones de redispersión
               a la clave dada. */
};

```

```

template <class H, class T>

hashVector<H, T>::hashVector
    (unsigned int max, unsigned int (*hash)(const H &),
     unsigned int (*rhash)(const H &, int))
: data(primo(max), (association<H, T> *) 0),
  erase(primo(max), false), hashfun(hash),
  rhashfun(rhash), numberOfElements(0)
{
    tablesize = data.length();
    numberOfSlots = tablesize;
}

```

```

template <class H, class T>

```

```

hashVector<H, T>::hashVector
    (const hashVector<H,T> & source)
    : data(source.tablesize), erase(source.erase),
      hashfun(source.hashfun), rhashfun(source.rhashfun),
      numberOfSlots(source.numberOfSlots),
      numberOfElements(source.numberOfElements),
      tablesize(source.tablesize), initial(source.initial)
{
    for (unsigned int i = 0; i < tablesize; i++)
        if (source.data[i])
            data[i] = new association<H, T>(*source.data[i]);
        else
            data[i] = 0;
}

```

```

template <class H, class T>

hashVector<H, T>::~~hashVector ()
{
    deleteAllValues();
}

```

```

template <class H, class T>

bool hashVector<H, T>::isEmpty () const
{
    for (int i = 0; i != data.length() ; i++)
        if (data[i])
            return false;
    return true;
}

```

```
}
```

```
template <class H, class T>

unsigned int hashVector<H, T>::firstEqualEmpty
    (const H & aKey) const
{
    int pos = (*hashfun)(aKey) % tablesize;
    for (unsigned int i=0; i < tablesize && (data[pos] ||
        erase[pos]) && data[pos]->key() != aKey; i++)
        pos = (*rhashfun)(aKey, i+1) % tablesize;
    return pos;
}
```

```
template <class H, class T>

unsigned int hashVector<H, T>::firstEqualEraseEmpty
    (const H & aKey) const
{
    int pos = (*hashfun)(aKey) % tablesize;
    for (unsigned int i=0; i < tablesize && data[pos]
        && data[pos]->key() != aKey; i++)
        pos = (*rhashfun)(aKey, i+1) % tablesize;
    return pos;
}
```

```
template <class H, class T>

bool hashVector<H, T>::includes (const H & aKey)
{

```

```

    int pos = firstEqualEmpty(aKey);
    return data[pos] && data[pos]->key() == aKey;
}

```

```

template <class H, class T>

T & hashVector<H, T>::operator [] (const H & aKey)
{
    int pos = firstEqualEmpty(aKey);
    // clave encontrada
    if (data[pos] && data[pos]->key() == aKey)
        return data[pos]->valueField;
    // clave no encontrada
    pos = firstEqualEraseEmpty(aKey);
    // si es necesario redimensionar la tabla
    if (numberOfSlots <= 0.1 * tablesize) {
        vector<association<H,T> *> oldData(data);
        // inicialmente en la nueva tabla no habra elementos
        tablesize = primo(2 * tablesize); // se duplica el
                                           // espacio
        data = vector<association<H,T> *> (tablesize,
                                           (association<H,T> *) 0);
        erase = vector<bool>(tablesize, false);
        numberOfSlots = tablesize;
        numberOfElements = 0;
        // insertar todos los elementos que ya estaban en
        // la tabla
        for (unsigned int i = 0; i < oldData.length(); i++)
            if (oldData[i] != 0) {
                operator [] (oldData[i]->key()) =
                           oldData[i]->value();
                delete oldData[i];
            }
    }
}

```

```

    }
}
// insertar clave y valor por defecto
data[pos] = new association<H,T>(aKey, initial);
if (erase[pos])
    erase[pos] = false;
else
    numberOfSlots--;
numberOfElements++;
return data[pos]->valueField;
}

```

```

template <class H, class T>

void hashVector<H, T>::remove (const H & aKey)
{
    int pos = firstEqualEmpty(aKey);
    if (data[pos] && data[pos]->key() == aKey) {
        erase[pos] = true;
        numberOfElements--;
        // recuperar el espacio
        delete data[pos];
        data[pos] = 0;
    }
}

```

```

template <class H, class T>

class hashVectorIterator : public iterator<T> {
public:
    // constructor

```

```

        hashVectorIterator (hashVector<H, T> & aHashVector);
        hashVectorIterator (const hashVectorIterator<H, T> &
source);
        // protocolo iterador
        virtual bool init ();
        virtual bool operator ! ();
        virtual bool operator ++ (); // prefijo
        virtual bool operator ++ (int); // postfijo
        virtual T operator () ();
        virtual T & operator = (const T & val);
        // método específico de esta subclase
        H key ();
        /* Produce: la clave del elemento actual. */
    protected:
        // área de datos
        unsigned int currentKey;
        hashVector<H, T> & theHashVector;
        // operación interna
        bool getNext ();
        /* Efecto: obtiene la siguiente cubeta que contiene
un elemento de la tabla.
Produce: cierto si encuentra una cubeta no vacía. */
};

```

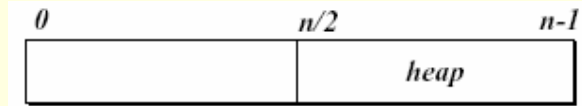
```

template <class T>

void heapSort (vector<T> & v)
    /* Necesita: un vector v.
Modifica: el vector ordenando sus componentes por
el método del montículo (heap). */
{
    unsigned int n = v.length();

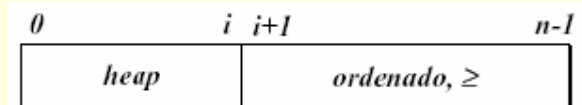
```

```
/*
```



```
*/
```

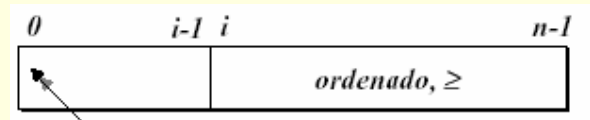
```
for (int i = n / 2 - 1; i >= 0; --i)
    reorganize(v, i, n - 1);
for (unsigned int i = n - 1; i > 0; --i) {
    /*
```



```
*/
```

```
swap(v, 0, i);
```

```
/*
```



```
*/
```

```
reorganize(v, 0, i-1);
```

```
}
```

```
}
```

Está mal la raíz

```
template <class T>
```

```
setList<setList<T> *> partes (setList<T> & c)
```

```
{
```

```
    setListIterator<T> itrC(c);
```



```

setList<setList<T> *> result;
setList<T> * vacio = new setList<T>();
// adición del conjunto vacío
result.add(vacio);
for (itrC.init(); !itrC; ++itrC) {
    setList<setList<T> *> temp(result);
    setListIterator<setList<T> *> itrT(temp);
    for (itrT.init(); !itrT; ++itrT) {
        // añadir el elemento de C a cada elemento de temp
        setList<T> * newParte = new setList<T>(*itr());
        newParte->add(itrC());
        result.add(newParte);
    }
}
return result;
}

```

FUNCIONES HASH

HASHING MULTIPLICATIVO

- Esta técnica trabaja multiplicando la clave k por sí misma o por una constante, usando después alguna porción de los bits del producto como una localización de la tabla hash.

HASHING POR DIVISIÓN

- En este caso la función se calcula simplemente como $h(k) = k \bmod M$ usando el 0 como el primer índice de la tabla hash de tamaño M .

BORRADOS Y REHASHING

- Cuando intentamos borrar un valor k de una tabla que ha sido generada por dispersión cerrada, nos encontramos con un problema. Si k precede a cualquier otro valor k en una secuencia de pruebas, no podemos eliminarlo sin más, ya que si lo hiciéramos, las pruebas siguientes para k se encontrarían el "agujero" dejado por k por lo que podríamos concluir que k no está en la tabla, hecho que puede ser falso.

- La solución es que necesitamos mirar cada localización de la tabla hash como inmersa en uno de los tres posibles estados: *vacía*, *ocupada* o *borrada*, de forma que en lo que concierne a la búsqueda, una celda borrada se trata exactamente igual que una ocupada. En el caso de las inserciones, podemos usar la primera localización vacía o borrada que se encuentre en la secuencia de pruebas para realizar la operación.

- Cuando una tabla llega a un desbordamiento o cuando su eficiencia baja demasiado debido a los borrados, el único recurso es llevarla a otra tabla de un tamaño más apropiado, no necesariamente mayor, puesto que como las localizaciones borradas no tienen que reasignarse, la nueva tabla podría ser mayor, menor o incluso del mismo tamaño que la original. Este proceso se suele denominar rehashing y es muy simple de implementar si el arca de la nueva tabla es distinta al de la primitiva, pero puede complicarse bastante si deseamos hacer un rehashing en la propia tabla.

RESOLUCIÓN DE CONFLICTOS

MEDIANTE REDISPERSIÓN

- Supongamos la siguiente figura:

Posición	Clave	Valor
0	4967000	
1		
2	8421002	
3		
⋮		
395		
396	4618396	
397	4957397	
398		
399	1286399	
400		
401		
⋮		
990	0000990	
991	0000991	
992	1200992	
993	0047993	
994		
995	9846995	
996	4618996	
997	4967997	
998		
999	0001999	

- Imaginemos que queremos introducir en la tabla un nuevo dato cuya clave es 596397. Utilizando una función de dispersión,

$$f(k) = k \% 1000$$

obtenemos $f(596397) = 397$. Por lo tanto, el dato habrá que almacenarlo en la posición 397.

- Dicha posición está ocupada por otro dato con clave distinta (4957397), por lo que el nuevo dato se deberá insertar en otra posición.

- El método más simple consiste en realizar una búsqueda lineal de la siguiente posición libre en el vector, es decir, definir

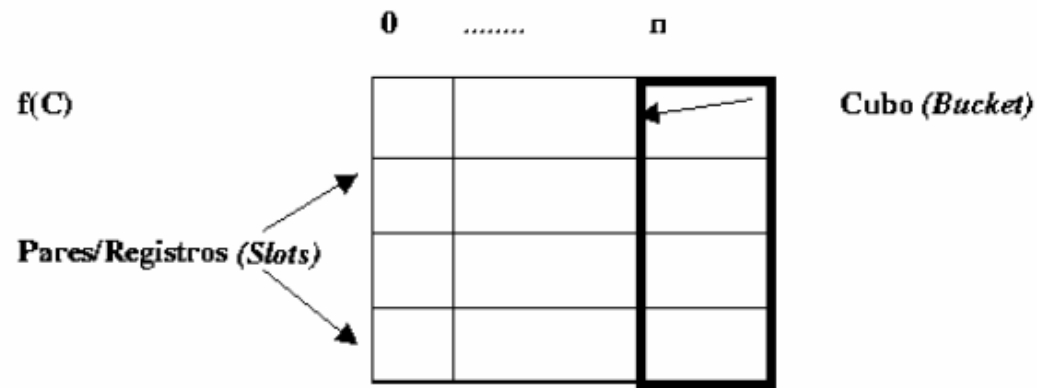
$$h(k) = (k+1) \% 1000$$

- Hay que exigir a la función h que produzca índices uniformemente distribuidos en el intervalo $0, \dots, n-1$ (siendo n el número de elementos del vector).

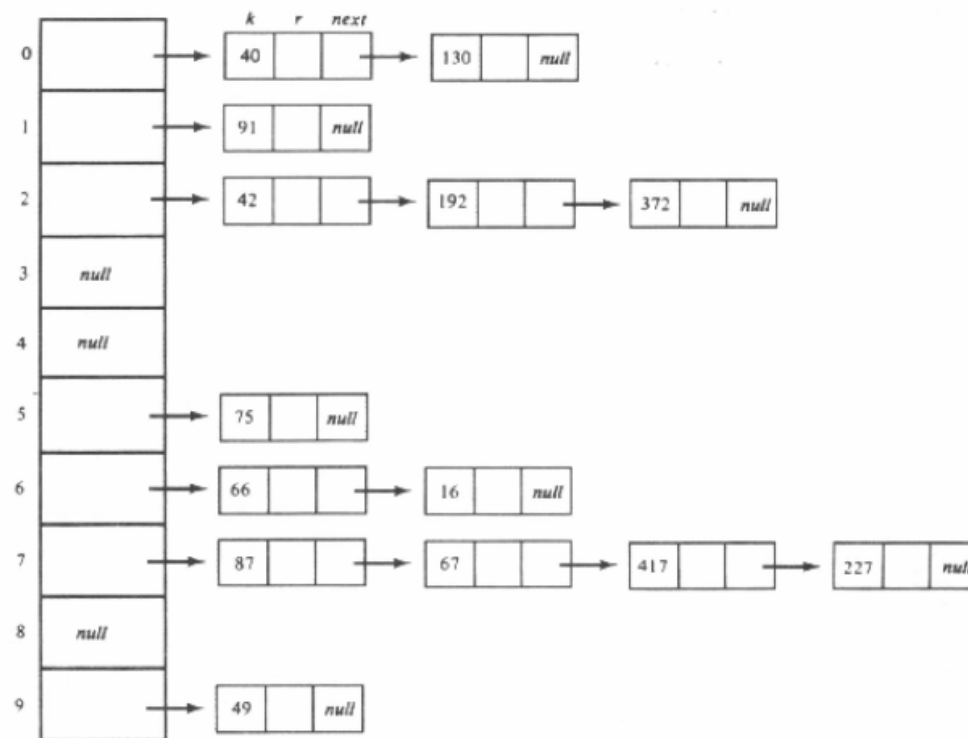
MEDIANTE ENCADENAMIENTO

- Este método consiste en permitir múltiples claves de registros en una misma posición. En este caso existen dos posibilidades:

1. Que cada dirección calculada contenga espacios para múltiples pares, en vez de un único par. Cada una de estas posiciones se denomina **cubo**.



2. Otra solución, es usar la dirección calculada, no como posición real del registro, sino como el índice en un vector de listas enlazadas. Cada posición del vector accede a una lista de pares que comparten la misma dirección.



- Para buscar un registro dado, aplicamos primero la función hash a su clave y luego buscamos la cadena para el registro en la línea. De este modo, si la lista es muy grande, se pierde la eficiencia del método.