TABLAS HASH

---

1. INTRODUCCIÓN.

Una aproximación a la búsqueda radicalmente diferente a las anteriores consiste en proceder, no por comparaciones entre valores clave, sino encontrando alguna función *h(k)* que nos dé directamente la localización de la clave *k* en la tabla.

La primera pregunta que podemos hacernos es si es fácil encontrar tales funciones *h*. La respuesta es, en principio, bastante pesimista, puesto que si tomamos como situacion ideal el que tal función dé siempre localizaciones distintas a claves distintas y pensamos p.ej. en una tabla de tamaño 40 en donde queremos direccionar 30 claves, nos encontramos con que hay $40^{30} = 1.15 * 10^{48}$ posibles funciones del conjunto de claves en la tabla, y sólo 40*39*11 = 40!/10! = $2.25 * 10^{41}$ de ellas no generan localizaciones duplicadas. En otras palabras, sólo 2 de cada 10 millones de tales funciones serian 'perfectas' para nuestros propósitos. Esa tarea es factible sólo en el caso de que los valores que vayan a pertenecer a la tabla hash sean conocidas a priori. Existen algoritmos para construir funciones hash perfectas que son utilizadas para organizar las palabras clave en un compilador de forma que la búsqueda de cualquiera de esas palabras clave se realice en tiempo constante.

Las funciones que evitan valores duplicados son sorprendentemente dificiles de encontrar, incluso para tablas pequeñas. Por ejemplo, la famosa "paradoja del cumpleaños" asegura que si en una reunión están presentes 23 ó más presonas, hay bastante probabilidad de que dos de ellas hayan nacido el mismo dia del mismo mes. En otras palabras, si seleccionamos una función aleatoria que aplique 23 claves a una tabla de tamaño 365 la probabilidad de que dos claves no caigan en la misma localización es de sólo 0.4927.

En consecuencia, las aplicaciones *h(k)*, a las que desde ahora llamaremos funciones hash, tienen la particularidad de que podemos esperar que $h(k_i) = h(k_j)$ para bastantes pares distintos (k_i,k_j). El objetivo será pues 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.

Esquema de una tabla hash

2. FUNCIONES HASH.

El primer problema que hemos de abordar es el cálculo de la función hash que transforma claves en localizaciones de la tabla. Más concretamente, necesitamos una función que transforme claves(normalmente enteros o cadenas de caracteres) en enteros en un rango [0..M-1], donde M es el número de registros que podemos manejar con la memoria de que dispongamos.como factores a tener en cuenta para la elección de la función *h(k)* están que minimice las colisiones y que sea relativamente rápida y fácil de calcular, aunque la situación ideal sería encontrar una función *h* que generara valores aleatorios uniformemente sobre el intervalo [0..M-1]. Las dos aproximaciones que veremos están encaminadas hacia este objetivo y ambas están basadas en generadores de números aleatorios.

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.

Cuando la elección es multiplicar *k* por sí misma y quedarse con alguno de los bits centrales, el método se denomina el cuadrado medio. Este metodo aún siendo simple y pudiendo cumplir el criterio de que los bits elegidos para marcar la localización son función de todos los bits originales de *k*, tiene como principales inconvenientes el que las claves con muchos ceros se reflejarán en valores hash también con muchos ceros, y el que el tamaño de la tabla está restringido a ser una potencia de 2.

Otro método multiplicativo, que evita las restricciones anteriores consiste en calcular h(k) = Int[M * Frac(C*k)] donde M es el tamaño de la tabla y 0 <= C <= 1, siendo importante elegir C con cuidado para evitar efectos negativos como que una clave alfabética K sea sinónima a otras claves obtenidas permutando los caracteres de k. Knuth (ver bibliografía) prueba que un valor recomendable es:

$$C = 1/2 \quad \text{con} \quad R = ½ (1 + \sqrt{5})$$

Hashing por División.

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

Aunque la fórmula es aplicable a tablas de cualquier tamaño es importante elegir el valor de M con cuidado. Por ejemplo si M fuera par, todas las claves pares (resp. impares) serían aplicadas a localizaciones pares (resp. impares), lo que constituiría un sesgo muy fuerte. Una regla simple para elegir M es tomarlo como un número primo. En cualquier caso existen reglas mas sofisticadas para la elección de M (ver Knuth), basadas todas en estudios téoricos de funcionamiento de los métodos congruenciales de generación de números aleatorios.

3. RESOLUCIÓN DE COLISIONES.

El segundo aspecto importante a estudiar en el hasing es la resolución de colisiones entre sinónimos. Estudiaremos tres métodos basicos de resolución de colisiones, uno de ellos depende de la idea de mantener listas enlazadas de sinónimos, y los otros dos del cálculo de una secuencia de localizaciones en la tabla hash hasta que se encuentre que se encuentre una vacía. El análisis comparativo de los métodos se hará en base al estudio del número de localizaciones que han de examinarse hasta determinar donde situar cada nueva clave en la tabla.

Para todos los ejemplos el tamaño de la tabla será M=13 y la función hash $h_1(k)$ que utilizaremos será:

HASH = Clave Mod M

y los valores de la clave k que consideraremos son los expuestos en la siguiente tabla:

j	k_j	$h_1(k_j)$	j	k_j	$h_1(k_j)$
1	119	2	7	109	5
2	85	7	8	147	4
3	43	4	9	38	12
4	141	11	10	137	7
5	72	7	11	148	5
6	91	0	12	101	10

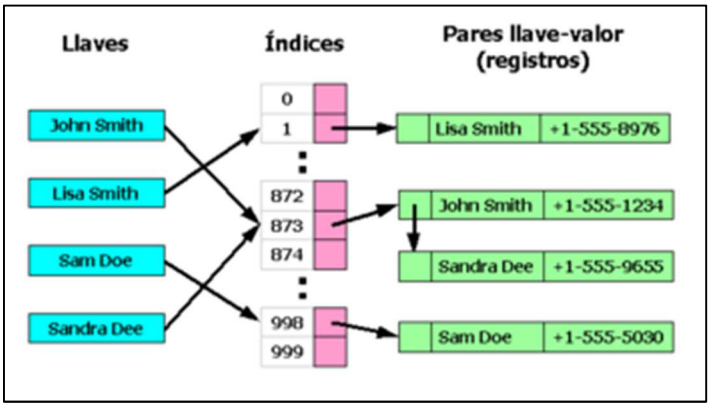
Valores ejemplo de claves k_j.

Suponiendo que k=0 no ocurre de forma natural, podemos marcar todas las localizaciones de la tabla, inicialmente vacías, dándoles el valor 0. Finalmente y puesto que las operaciones de búsqueda e inserción están muy relacionadas, se presentaran algoritmos para buscar un item insertándolo si es necesario (salvo que esta operación provoque un desbordamiento de la tabla) devolviendo la localización del item o un -1 **(NULL)** en caso de desbordamiento.

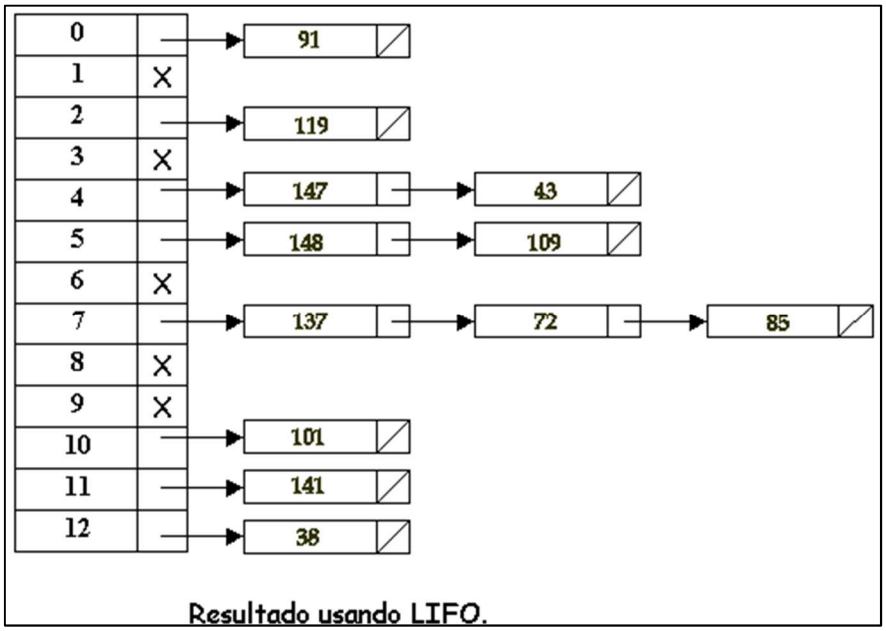
Encadenamiento separado o Hashing Abierto.

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. Este método se conoce normalmente con el nombre de *encadenamiento separado* y obviamente la cantidad de tiempo requerido para una búsqueda dependerá de la longitud de las listas y de las posiciones relativas de las claves en ellas. Existen variantes dependiendo del mantenimiento que hagamos de las listas de sinónimos (FIFO, LIFO, por valor Clave,

etc), aunque en la mayoría de los casos, y dado que las listas individuales no han de tener un tamaño excesivo, se suele optar por la alternativa más simple, la LIFO.



En cualquier caso, si las listas se mantienen en orden esto puede verse como una generalización del método de búsqueda secuencial en listas. La diferencia es que en lugar de mantener una sola lista con un solo nodo cabecera se mantienen M listas con M nodos cabecera de forma que se reduce el número de comparaciones de la búsqueda secuencial en un factor de M (en media) usando espacio extra para M punteros. Para nuestro ejemplo y con la alternativa LIFO, la tabla quedaría como se muestra en la siguiente figura:

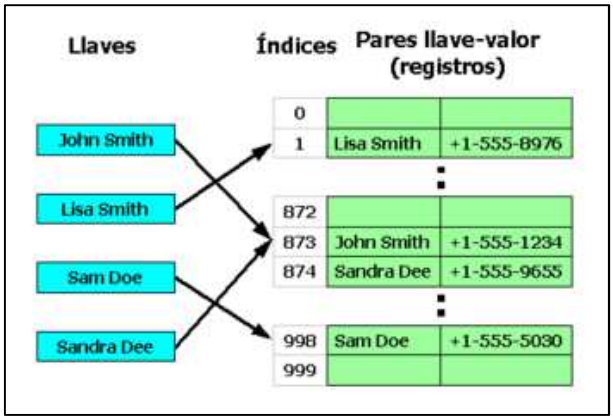


A veces y cuando el número de entradas a la tabla es relativamente moderado, no es conveniente dar a las entradas de la tabla hash el papel de cabeceras de listas, lo que nos conduciría a otro método de encadenamiento, conocido como *encadenamiento interno*. En este caso, la unión entre sinónimos está dentro de la propia tabla hash, mediante campos cursores (punteros) que son inicializados a -1 (NULL) y que irán apuntando hacia sus respectivos sinónimos.

Direccionamiento abierto o Hashing Cerrado.

Otra posibilidad consiste en utilizar un vector en el que se pone una clave en cada una de sus casillas. En este caso nos encontramos con el problema de que en el caso de que se produzca una colisión no se pueden tener ambos elementos formando parte de una

lista para esa casilla. Para solucionar ese problema se usa lo que se llama *rehashing*. El rehashing consiste en que una vez producida una colisión al insertar un elemento se utiliza una función adicional para determinar cuál será la casilla que le corresponde dentro de la tabla, a esta función la llamaremos función de rehashing ,$reh_i(k)$.



A la hora de definir una función de rehashing existen múltiples posibilidades, la más simple consiste en utilizar una función que dependa del número de intentos realizados para encontrar una casilla libre en la que realizar la inserción, a este tipo de rehashing se le conoce como *hashing lineal*. De esta forma la función de rehashing quedaria de la siguiente forma:

$$\mathbf{reh_i(k) = (h(k)+(i-1)) \bmod M \; i=2,3,...}$$

En nuestro ejemplo, después de insertar las 7 primeras claves nos aparece la tabla A, (ver la tabla siguiente). Cuando vamos a insertar la clave 147, esta queda situada en la casilla 6, (tabla B) una vez que no se han encontrado vacías las casillas 4 y 5. Se puede observar que antes de la inserción del 147 había agrupaciones de claves en las localizaciones 4,5 y 7,8, y después de la inserción, esos dos grupos se han unido formando una agrupación primaria mayor, esto conlleva que si se trata de insertar un elemento al que le corresponde algunas de las casillas que están al principio de esa agrupación el proceso de rehashing tendrá de recorrer todas esas casillas con lo que se degradará la eficiencia de la inserción. Para solucionar este problema habrá que buscar un método de rehashing que distribuya de la forma más aleatoria posible las casillas vacías.

Despues de llevar a cabo la inserción de las claves consideradas en nuestro ejemplo, el estado de la tabla hash será el que se puede observar en la tabla (C) en la que además aparece el número de intentos que han sido necesarios para insertar cada una de las claves.

A		B		C		
i	Clave	i	Clave	i	Clave	Intentos
0	91	0	91	0	91	1
1		1		1	101	5
2	119	2	119	2	119	1
3		3		3	0	
4	43	4	43	4	43	1
5	109	5	109	5	109	1
6		6	147	6	147	3
7	85	7	85	7	85	1
8	72	8	72	8	72	2
9		9		9	137	3
10		10		10	148	6
11	141	11	141	11	141	1
12		12		12	38	1

Inserción con direccionamiento abierto.

Para intentar evitar el problema de las agrupaciones que acabamos de ver podríamos utilizar la siguiente función de rehashing:

$$\mathbf{reh_i(k) = (h(k)+(i-1)*C) \bmod M \; C>1 \text{ y primo relativo con } M}$$

pero aunque esto evitaría la formación de agrupaciones primarias, no solventaría el problema de la formación de agrupaciones secundarias (agrupaciones separadas por una distancia C). El problema básico de rehashing lineal es que para dos claves distintas que tengan el mismo valor para la función hash se irán obteniendo exactamente la misma secuencia de valores al aplicar la función de rehashing, cunado lo interenante seria que la secuencia de valores obtenida por el proceso de rehashing fuera distinta. Así, habrá que buscar una función de rehashing que cumpla las siguientes condiciones:

- Sea fácilmente calculable (con un orden de eficiencia constante),
- que evite la formación de agrupaciones,
- que genere una secuencia de valores distinta para dos claves distintas aunque tenga el mismo valor de función hash, y por último
- que garantice que todas las casillas de la tabla son visitadas.

si no cumpliera esto último se podría dar el caso de que aún quedaran casillas libres pero no podemos insertar un determinado elemento porque los valores correspondientes a esas casillas no son obtenidos durante el rehashing.

Una función de rehashing que cumple las condiciones anteriores es la función de *rehashing doble*. Esta función se define de la siguiente forma:

$$\mathbf{h_i(k) = (h_{i-1}(k)+h_0(k)) \bmod M \; i=2,3,...}$$

$$\mathbf{\text{con } h_0(k) = 1+k \bmod (M-2) \text{ y } h_1(k) = h(k).}$$

Existe la posibilidad de hacer otras elecciones de la función $h_0(k)$ siempre que la función escogida no sea constante.

Esta forma de rehashing doble es particularmente buena cuando M y M-2 son primos relativos. Hay que tener en cuenta que si M es primo entonces es seguro que M-2 es primo relativo suyo (exceptuando el caso trivial de que M=3).

El resultado de aplicar este método a nuestro ejemplo puede verse en las tablas siguientes. En la primera se incluyen los valores de h para cada clave y en la segunda pueden verse las localizaciones finales de las claves en la tabla así como las pruebas requeridas para su inserción.

k_j	$h_1(k_j)$	$h_0(k_j)$
119	2	10
85	7	9
43	4	11
141	11	10
72	7	7
91	0	4
109	5	11
147	4	5
38	12	6
137	7	6
148	5	6
101	10	3

Valores h_0 y h_1 para cada clave.

i	Clave	Intentos
0	91	1
1	72	2
2	119	1
3	101	3
4	43	1
5	109	1
6	137	3
7	85	1
8		
9	147	2
10	148	4
11	141	1
12	38	1

Localizaciones finales de las claves.

4. BORRADOS Y REHASING.

Cuando intentamos borrar un valor *k* de una tabla que ha sido generada por direccionamiento abierto, 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 encontrarian el "agujero" dejado por *k* por lo que podríamos concluir que *k* no está en la tabla, hecho que puede ser falso. Podemos comprobarlo en nuestro ejemplo en cualquiera de las tablas. La solución es que

necesitamos mirar cada localización de la tabla hash como inmersa en uno de los tres posibles estados: *vacia, ocupada o borrada*, de forma que en lo que concierne a la busqueda, una celda borrada se trata exactamente igual que una ocupada.En caso de inserciones, podemos usar la primera localización vacía o borrada que se encuentre en la secuencia de pruebas para realizar la operación. Observemos que este problema no afecta a los borrados de las listas en el encadenamiento separado. Para la implementación de la idea anterior podría pensarse en la introducción en los algorítmos de un valor etiqueta para marcar las casillas borradas, pero esto sería solo una solución parcial ya que quedaría el problema de que si los borrados son frecuentes, las búsquedas sin éxito podrían requerir O(M) pruebas para detectar que un valor no está presente.

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.

5. EVALUACIÓN DE LOS MÉTODOS DE RESOLUCIÓN.

El aspecto más significativo de la búsqueda por hashing es que si eficiencia depende del denominado factor de almacenamiento ***Ó= n/M*** con *n* el número de items y *M* el tamaño de la tabla.

Discuteremos el número medio de pruebas para cada uno de los métodos que hemos visto de resolución de colisiones, en términos de **BE** (búsqueda con éxito) y **BF** (búsqueda sin éxito). Las demostraciones de las fórmulas resultantes pueden encontrarse en Knuth (ver bibliografía).

Encadenamiento separado.

Aunque puede resultar engañoso comparar este método con los otros dos, puesto que en este caso puede ocurrir que Ó>1, las fórmulas paroximadas son:

$$BE = 1+\alpha/2 \qquad BF = e^{-\alpha} + \alpha$$

Estas expresiones se aplican incluso cuando Ó>>1, por lo que para n>>M, la longitud media de cada lista será Ó, y deberia esperarse en media rastrear la mitad de la lista, antes de encontrar un determinado elemento.

Hasing Lineal.

Las fórmulas aproximadas son:

$$BE = 1+(1-\alpha)^{-1}/2 \qquad BF = 1+(1-\alpha)^{-2}/2$$

Como puede verse, este método, aun siendo satisfactorio para Ó pequeños, es muy pobre cuando Ó -> 1, ya que el límite de los valores medios de **BE** y **BF** son respectivamente:

$$0.33+\sqrt{(\Pi * M/8)} \quad y \quad (M+1)/2$$

En cualquier caso, el tamaño de la tabla en el hash lineal es mayor que en el encadenamiento separado, pero la cantidad de memoria total utilizada es menor al no usarse punteros.

Hasing Doble.

Las fórmulas son ahora:

$$BE=-(1/1-Ó) * \ln(1-Ó)$$

$$BF=1/(1-Ó)$$

con valores medios cuando Ó -> 1 de M y M/2, respectivamente.

Para facilitar la comprensión de las fórmulas podemos construir una tabla en la que las evaluemos para distintos valores de Ó:

α	BE	BF	BE	BF	BE	BF
0.25	1.12	1.03	1.17	1.39	1.15	1.33
0.50	1.25	1.11	1.50	2.50	1.39	2.00
0.75	1.38	1.22	2.50	8.50	1.85	4.00
0.90	1.45	1.31	5.50	50.5	2.56	10.0
	Encadenamiento Separado		Hasing Lineal		Hasing Doble	

Valores BE(α) y BF(α).

La elección del mejor método hash para una aplicación particular puede no ser fácil. Los distintos métodos dan unas características de eficiencia similares. Generalmente, lo mejor es usar el encadenamiento separado para reducir los tiempos de búsqueda cuando el número de registros a procesar no se conoce de antemano y el hash doble para buscar claves cuyo número pueda, de alguna manera, predecirse de antemano.

En comparación con otras técnicas de búsqueda, el hashing tiene ventajas y desventajas. En general, para valores grandes de *n* (y razonables valores de Ó) un buen esquema de hashing requiere normalmente menos pruebas (del orden 1.5 - 2) que cualquier otro método de búsqueda, incluyendo la búsqueda en árboles binarios. Por otra parte, en el caso peor, puede comportarse muy mal al requerir *O(n)* pruebas. También puede considerarse como una ventaja el hecho de que debemos tener alguna estimación a priori de número máximo de items que vamos a colocar en la tabla aunque si no disponemos de tal estimación siempre nos quedaría la opción de usar el metodo de encadenamiento separado en donde el desbordamiento de la tabla no constituye ningún problema.

Otro problema relativo es que en una tabla hash no tenemos ninguna de las ventajas que tenemos cuando manejamos relaciones ordenadas, y así p.e. no podemos procesar los items en la tabla secuencialmente, ni concluir tras una búsqueda sin éxito nada sobre los items que tienen un valor cercano al que buscamos, pero en cualquier caso el mayor problema que tener el hashing cerrado es el de los borrados dentro de la tabla.

6. IMPLEMENTACIÓN DE LAS TABLAS HASH.

Implementación de Hasing Abierto.

En este apartado vamos a realizar una implementación simple del hasing abierto que nos servirá como ejemplo ilustrativo de su funcionamiento. Para ello supondremos un tipo de dato *char* * para el cual diseñaremos una función hash simple consistente en la suma de los codigos ASCII que componen dicha cadena.

Una posible implementación utilizando el tipo de dato abstracto lista sería la siguiente:

```
 #define NCASILLAS 100 /*Ejemplo de número de entradas en la tabla.*/

typedef tLista *TablaHash;
```

Para la cual podemos diseñar las siguientes funciones de creación y destrución:

```
TablaHash CrearTablaHash ()
{
   tLista *t;
   register i;

   t=(tLista *)malloc(NCASILLAS*sizeof(tLista));
   if (t==NULL)
      error("Memoria insuficiente.");

   for (i=0;i<NCASILLAS;i++)
      t[i]=crear();

   return t;
}



void DestruirTablaHash (TablaHash t)
{
   register i;

   for (i=0;i<NCASILLAS;i++)
      destruir(t[i]);

   free(t);
}
```

Como fue mencionado anteriormente la función hash que será usada es:

```
int Hash (char *cad)
{
   int valor;
   unsigned char *c;

   for (c=cad,valor=O;*c;c++)
      valor+=(int)(*c);

   return(valor%NCASILLAS);
}
```

Y funciones del tipo *MiembroHash*, *InsertarHash*, *BorrarHash* pueden ser programadas:

```
int MiembroHash (char *cad,TablaHash t)
{
   tPosicion p;
   int enc;
   int pos=Hash(cad);

   p=primero(t[pos]);
   enc=0;
   while (p!=fin(t[pos]) && !enc) {
      if (strcmp(cad,elemento(p,t[pos]))==0)
         enc=1;
      else
         p=siguiente(p,t[pos]);
   }

   return enc;
}

void InsertarHash (char *cad,TablaHash t)
{
   int pos;

   if (MiembroHash(cad,t))
      return;

   pos=Hash(cad);
   insertar(cad,primero(t[pos]),t[pos]);
}

void BorrarHash (char *cad,TablaHash t)
{
   tPosicion p;
   int pos=Hash(cad);

   p=primero(t[pos]);
   while (p!=fin(t[pos]) && !strcmp(cad,elemento(p,t[pos])))
      p=siguiente(p,t[pos]));

   if (p!=fin(t[pos]))
      borrar(p,t[pos]);
}
```

Como se puede observar esta implementación es bastante simple de forma que puede sufrir bastantes mejoras. Se propone como ejercicio el realizar esta labor dotando al tipo de dato de posibilidades como:

- Determinación del tamaño de la tabla en el momento de creación.
- Modificación de la función hash utilizada, mediante el uso de un puntero a función.
- Construcción de una función que pasa una tabla hash de un tamaño determinado a otra tabla con un tamaño superior o inferior.
- Construcción de un iterador a través de todos los elementos de la tabla.
- etc...

Implementación de Hasing Cerrado.

En este apartado vamos a realizar una implementación simple del hashing cerrado. Para ello supondremos un tipo de dato*char* * al igual que en el apartado anterior, para el cual diseñaremos la misma función hash.

Una posible implementación de la estructura a conseguir es la siguiente:

```
#define NCASILLAS 100
#define VACIO NULL
static char * BORRADO='''';

typedef char **TablaHash;
```

Para la cual podemos diseñar las siguientes funciones de creación y destrución:

```
TablaHash CrearTablaHash ()
{
   TablaHash t;
   register i;

   t=(TablaHash)malloc(NCASILLAS*sizeof(char *));
   if (t==NULL)
      error("Memoria Insuficiente.");

   for (i=0;i<NCASILLAS;i++)
      t[i]=VACIO;

   return t;
}

void DestruirTablaHash (TablaHash t)
{
   register i;

   for (i=O;i<NCASILLAS;i++)
      if (t[i]!=VACIO && t[i]!=BORRADO)
         free(t[i]);

   free t;
}
```

La función hash que será usada es igual a la que ya hemos usado para la implementación del Hasing Abierto. Y funciones del tipo *MiembroHash*, *InsertarHash*, *BorrarHash* pueden ser programadas tal como sigue, teniendo en cuenta que en esta implementación haremos uso de un rehashing lineal.

```
int Hash (char *cad)
{
   int valor;
   unsigned char *c;

   for (c=cad, valor=0; *c; c++)
         valor += (int)*c;

   return (valor%NCASILLAS);
}

int Localizar (char *x,TablaHash t)

/* Devuelve el sitio donde esta x  o donde deberia de estar. */
/* No tiene en cuenta los borrados.                          */

{
```

```
   int ini,i,aux;

   ini=Hash(x);

   for (i=O;i<NCASILLAS;i++) {
      aux=(ini+i)%NCASILLAS;
      if (t[aux]==VACIO)
         return aux;
      if (!strcmp(t[aux],x))
         return aux;
   }
   return ini;
}

int Localizar1 (char *x,TablaHash t)

/* Devuelve el sitio donde podriamos poner x  */

{
   int ini,i,aux;

   ini=Hash(x);

   for (i=O;i<NCASILLAS;i++) {
      aux=(ini+i)%NCASILLAS;
      if (t[aux]==VACIO || t[aux]==BORRADO)
         return aux;
      if (!strcmp(t[aux],x))
         return aux;
   }

   return ini;
}

int MiembroHash (char *cad,TablaHash t)
{
   int pos=Localizar(cad,t);

   if (t[pos]==VACIO)
      return 0;
   else
      return(!strcomp(t[pos],cad));
}

void InsertarHash (char *cad,TablaHash t)
{
   int pos;

   if (!cad)
      error("Cadena inexistente.");

   if (!MiembroHash(cad,t)) {
       pos=Localizar1(cad,t);
       if (t[pos]==VACIO || t[pos]==BORRADO) {
          t[pos]=(char *)malloc((strlen(cad)+1)*sizeof(char));
          strcpy(t[pos],cad);
       } else {
          error("Tabla Llena. \n");
       }
   }
}

void BorrarHash (char *cad,TablaHash t)
{
   int pos = Localizar(cad,t);

   if (t[pos]!=VACIO && t[pos]!=BORRADO) {
```

```
      if (!strcmp(t[pos],cad)) {
         free(t[pos]);
         t[pos]=BORRADO;
      }
   }
}
```