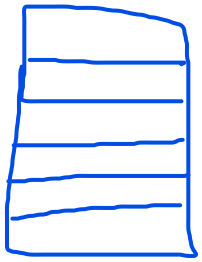
Tabla de Dispersión cerrada

Pablo R. Ramis

Universidad Nacional de Rosario, Instituto Politécnico, Dto. de Informática,
prramis@ips.edu.ar,
WWW home page: http://informatica.ips.edu.ar

Resumen Para la implementación de diccionarios existe una técnica importante y muy útil llamada ***"dispersión*** o ***"hashing"***. Utiliza tiempo constante por operación, en promedio. En el peor de los casos, requiere un tiempo proporcional al tamaño del conjunto.
La teoría y el código aca mostrado está tomado de ***Estructuras de datos y algoritmos*** de Alfred Aho, John Hopcroft y jeffrey Ullman. Addison-Wesley Iberoamericana. Ed. 1988.



1. Dispersión Cerrada

Una tabla de dispersión cerrada guarda los miembros del diccionario en la tabla de contenedores. En consecuencia, solo se puede guardar un elemento en cada contenedor, y tiene asociada ademas una *estrategia de redispersión.* Si se intenta colocar x en el contenedor $h(x)$ y esta ya tiene un elemento -situación que llamaremos *colisión*- la estrategia de redispersión elije una sucesión de localidades alternas $h_1(x)$, $h_2(x)$, ... dentro de la tabla de contenedores, en la cual es posible colocar a x. Se probará en todas esas localidades hasta encontrar una vacía. si nnguna está vacía, la tabla está llena y no es posible insertar x.

1.1. Ejemplo 1

Supongamos que B = 8 y que las claves a, b, c y d tiene valores de dispersión $h(a) = 3$, $h(b) = 0$, $h(c) = 4$, $h(d) = 3$. Se usará la estrategia de redispersión más sencilla, denominada *dispersión lineal*, en la que $h_i(x) = (h(x) + i) mod B$. Así, por ejemplo, si intentaramos insertar a en el contenedor 3, y se encontrara llena, se probaría en los contenedores 4, 5 6, 7, 0, 1 y 2, en ese orden.

En principio, se supone que la tabla está vacía, esto es, que cada contenedor guarda un valor especial *vacío*, que no es igual a ningún valor que podria intentarse insertar. Si eso no fuera viable, porque el tipo guardado no sugiere ningún valor adecuado, se puede hacer que cada contenedor tenga un campo adicional para indicarlo. Si se insertan a, b, c y d, en ese orden, en una tabla inicialmente vacía, resulta que a va al contenedor 3, b al 0, y c al 4. Al insertar d, primero se hace la prueba en $h(d)=3$ para encontrar que está llena. Luego se intenta con $h_1(d) = 4$ y ocurre lo mismo. Por último, se prueba con $h_2(d) = 5$, se encuentra un espacio vacío y d se coloca allí, como lo vemos en la figura 1

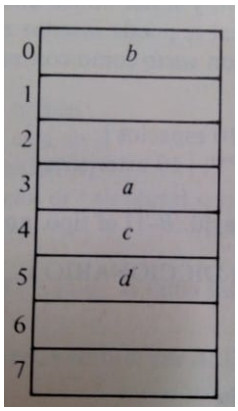


Figura 1. Tabla de dispersión parcialmente llena

La prueba de pertenencia de un elemento x al conjunto requiere examinar $h(x)$, $h_1(x)$, $h_2(x)$, ... hasta encontrar x o un contenedor vacía. Para ver por qué es posible detenerse al alcanzar un contenedor vacío, supóngase primero que las supresiones no están permitidas. Si $h_3(x)$ es el primer contenedor vacío encontrada en la serie, no es posible que x esté en los contenedores $h_4(x)$, $h_3(x)$ o más adelante en la sucesión, porque x no pudo haber sido colocada allí a menos que $h_3(x)$ hubiera estado llena en el momento de insertarla.

Sin embargo, podemos ver que si se permiten las supresiones, nunca puede existir la seguridad, si se alcanza el contenedor vacío sin encontrar x, de que x no se encuentra en alguna otra parte, y el contenedor ahora vacío estaba ocupado cuando se insertó x. Cuando deban hacerse supresiones, el enfoque más efectivo para resolver este prblema es colocar una constante llamada *suprimido* en el contenedor que contenga un elemento que se deba suprimir. Es importante que haya una diferencia entre *suprimido* y *vacío*, la constante que se encuentra en todos los contenedores que nunca se ha llenado. De esta manera, es posible permitir supresiones sin tener que buscar en la tabla completa durante la prueba *MIEMBRO*. En el momento de la inserción, es posible tratar de *suprimido* como un espacio disponible, de modo que con suerte el espacio de un elemento suprimido puede volver a utilizarse.

1.2. Ejemplo 2

Suponga que desea probar si e está en el conjunto representado en la figura 1. Si *h(e)=4*, se prueba con los contenedor 4, 5, y luego 6. El contenedor 6 está vacío y como e no se ha encontrado, la conclusión es que no está en el conjunto.

Si se suprime c, es necesario colocar la constante *suprimido* en el contenedor 4. Así, al buscar d y comenzar en *h(d)=3*, se pueden examinar 4 y 5 para encontrar a d, y no detenerse en 4 como se hubiera hecho de haber puesto *vacío* en ese contenedor.

En la próxima sección presentaremos el esquema de código, las declaraciones de tipos y las operaciones necesarias para la implementación del TDA DICCIONARIO, sus miembros de conjunto de tipo strings y la tabla de dispersión arbitraria *h*, como se ve en la figura 1 la cual es una posibilidad. Se usará la estrategia de dispersión lineal para redispersar colisiones. Por conveniencia, usaremos a la cadena de 10 espacios como *vacío* y la de 10 '*' como *suprimido* suponiendo que ninguna de esas cadenas serían datos para el diccionario.

El procedimiento INSERTA(x, A) primero usa *localiza* para determinar si *x* está en A, y si no, utiliza una función especial *localiza1* para encontrar una localidad en la cual se pueda insertar *x*. *localiza1* busca lugares marcadas tanto como *vacío* como *suprimido*

2. DICCIONARIO

```
1
2  const
3      vacio = "          ";{10 espacio}
4      suprimido "*********";{10 asteriscos}
5  type
6      DICCIONARIO = array[0... B-1] of string;
7
8  procedure CREAR():DICCIONARIO
9      var
10         A: DICCIONARIO;
11         i: integer;
12     begin
13         for i := 0 to B-1 do
14             A[i]:= vacio
15     return A
16     end; {ANULA}
17
18 function localiza(x: string):integer;
19     {localiza examina el DICCIONARIO desde el compatimiento
           para h(x)
20         hasta que se encuentre x; o un contenedor vacio, o se
               haya
21         revisado toda la tabla y determinado que no se
               contiene a x.
22         localiza devuelve el indice del contenedor en la que
               se
23         detiene por cualquier de esas razones.}
24 var
25     inicial, i: integer;
26         {inicial guarda h(x), i cuenta el numero de
               contenedores
27             examinados hasta el momento cuando se busca x.}
28 begin
```

```
29      inicial := h(x);
30      i := 0;
31      while (i < B) and (A[(inicial + i)mod B]<> x) and
32         (A[incial + i)mod B] <> vacio) do
33             i := i+1;
34      return ((inicial + i)mod B)
35  end; {localiza}
36
37  funcition localiza1 (x: string): integer;
38      {como localiza, pero tambien se detiene en una entrada
            con suprimido
39          y devuelve ese valor}
40
41  function MIEMBRO(x: string; var A: DICCIONARIO): boolean;
42      begin
43          if A[localiza(x)] = x then
44              return (true)
45          else
46             return (false)
47      end; {MIEMBRO}
48
49  procedure INSERTA(x: string; var A: DICCIONARIO);
50      var
51          contenedor: integer;
52      begin
53          if A[localiza(x)] = x then
54              return; {x ya esta en A}
55              contenedor := localiza1(x);
56          if(A[contenedor] = vacio) or (A[contenedor] =
                suprimido) then
57              A[contenedor] := x
58          else
59              error('INSERTA fallo, la tabla esta llena')
60      end;{INSERTA}
61
62  procedur SUPRIME(x: string; var A: DICCIONARIO);
63      var
64          contenedor: integer;
65      begin
66          contenedor := localiza(x);
67          if A[contenedor] = x then
68             A[contenedor] := suprimido
69      end; {SUPRIME}
```

Nos está faltando algo fundamental que venimos mencionando en todo el texto y código... Es nuestra función dispersión h.

No siempre es clara la posibilidad de elegir h de modo que un conjunto típico tenga sus miembros distribuidos de forma relativamente uniforme entre los contenedores.

Presentaremos una función dispersión para cadenas de caracteres la cual es muy eficiente aunque no sea perfecta. En el código, como los anteriores, es pascal, y en esta función se invoca a *ord*, donde *ord(c)* es el código entero del caracter *'c'*. Entonces, si x es el dato a guardar de tipo *array de caracteres* podemos declarar a la función h de la siguiente manera:

```
1
2  function h(x: string): 0 ...B-1;
3      var
4          i, suma: integer;
5      begin
6          suma := 0;
7          for i := 1 to 10 do
8              suma := suma + ord(x[i]);
9          h := suma mod B
10     end; {h}
```

Como vemos, se suman los valores numericos de cada letra y luego se toma el residuo de la división de la suma con B (o sea, el total de elementos que puede tener el diccionario) de ese modo, se retornara un numero entre 0 y B-1.

Nosotros, como lo implementaremos en C, no tendremos la función *ord*, tendremos que realizar el cast en su lugar.

Los prototipos podrían ser:

```
1
2
3  #include<stdio.h>
4  #include<stdlib.h>
5  #include<string.h>
6
7  #define NCasillas 25
8  #define VACIO NULL
9  static char* BORRADO = "";
10
11 typedef char **DICCIONARIO;
12
13
14 DICCIONARIO CREAR();
15 void DestruirTablaHash (DICCIONARIO);
16 void SUPRIME(char* , DICCIONARIO );
17 int h(char* );
18 int localiza(char* , DICCIONARIO );
19 int localiza1(char* , DICCIONARIO );
20 int MIEMBRO (char* , DICCIONARIO );
21 void INSERTA(char* , DICCIONARIO );
22 void SUPRIME(char* , DICCIONARIO );
```