El TAD Diccionario

Cuando se usa un conjunto en el diseño de un algoritmo podría no ser necesario contar con operaciones de unión o intersección.
A menudo lo que se necesita es simplemente manipular un conjunto de objetos al que periódicamente se le agregan o quitan elementos.
También es usual que uno desee verificar si un determinado elemento forma parte o no del conjunto.

Consideremos el siguiente caso de estudio:

La Sociedad Protectora del Calamar (SPC) mantiene una base de datos en la que registra los votos más recientes de los legisladores en temas relacionadas con el calamar. Esta base de datos consiste básicamente de dos conjuntos de nombres de legisladores, llamados BuenosMuchachos (BM) y MalosMuchachos (MM). La sociedad perdona errores del pasado y tiende a olvidarse de sus amigos fácilmente. Por ejemplo: si se vota una ley para limitar la pesca del calamar en aguas territoriales, todos los legisladores que votan a favor serán insertados en BM y borrados de MM, mientras que lo contrario le ocurrirá a aquellos que voten en contra de la ley. Los legisladores que se abstienen permanecen en el conjunto en que se encontraban previo a la votación.

El programa que permite el manejo de la base de datos tiene la siguiente funcionalidad:

El sistema acepta tres comandos, cada uno representado por un carácter, seguido de un string de 10 caracteres que representa el nombre del legislador. Los comandos son los siguientes:

- F (un legislador que vota favorablemente)
- D (un legislador que vota desfavorablemente)
- ? (determina el status del legislador)

También se cuenta con un comando E para terminar el proceso.

El módulo que implementa el manejador de la base de datos es el siguiente:

```
MODULO Calamar;


#include “Diccionario.h”




        Char comando;
        char legdor [11];
        Dicc BM, MM;

int main(){
        BM = Vacio();
        MM = Vacio();
```

```
        scanf(“%c”,&comando);

        while (comando < 'E'){
                scanf(“%s”,&legdor);
                if( comando = ='F'){
                        Insertar(legdor,BM);
                        Borrar(legdor,MM);
                 }
                Else if (comando = ='D'){
                        Insertar(legdor,MM);
                        Borrar(legdor,BM);
                }else if (comando = ='?')
                        If (Pertenece(legdor,BM))
                          printf(“OK”) ;
                        else
                                printf('TodoMal') ;

                else
                        printf('Comando erroneo');

        scanf(“%c”,&comando);

        }//fin while
END Calamar.
```

Módulo de definición del TAD Diccionario

```
//.h Modulo de definición par el tad diccionario

typedef struct Dicc;

Dicc Vacio ();
/* retorna el diccionario vacío */

Dicc Insertar (T i,  Dicc D);

/* retorna un nuevo diccionario que consiste de todos los elementos de Dicc más
el elemento i, si es que i no era ya un elemento de D */

bool EsVacio (Dicc D) ;/* retorna TRUE si D es vacío */

bool Pertenece ( T i,  Dicc D);
/* retorna TRUE si i es un elemento de D */

Dicc Borrar (T i,  Dicc D);
/* Borra el elemento i del diccionario D. Si i no pertenece a D, el diccionario
retornado es D */
```

Implementación del TAD Diccionario

Existen distintas posibles implementaciones de este tipo abstracto.
Una primera posibilidad es usando listas encadenadas (ordenadas o no).
Otra posible implementación de un diccionario es por medio de un arreglo de bits (o booleanos). Este enfoque requiere que:

- los elementos del tipo base pertenezcan al subrango 1..N, para algún natural N, o
- que pueda definirse una correspondencia uno a uno entre el tipo base y el anterior subrango de naturales.

Una tercer posible implementación es usar un arreglo con tope. Este enfoque garantiza una representación correcta siempre y cuando el tamaño del diccionario no supere el largo del arreglo.

Las desventajas de este tipo de representación, al igual que cuando discutimos el TAD Lista, son:

1. el diccionario no puede crecer arbitrariamente,
2. la operación de borrado es dificultosa y
3. el espacio de almacenamiento es ineficientemente utilizado.

Finalmente, una frecuentemente usada implementación es mediante ABB. Este enfoque permite una eficiente definición de las operaciones y utilización del espacio de memoria.

Analizar implementaciones de Conjuntos con:

- listas encadenadas no ordenadas
- listas encadenadas ordenadas
- arreglos con tope
- arreglos de bits
- árboles binarios de búsqueda
- árboles balanceados, tablas de hash

Hash (tablas de dispersión)

Introducción

Entre las implementaciones conocidas del TAD Diccionario se puede encontrar: Listas, Arreglos, ABB.
Otra posibilidad para representar diccionarios es la **tabla de dispersión** o **hash**, que permite realizar operaciones de inserción, eliminación y búsqueda en un **tiempo medio constante**.

Idea general

La estructura de una tabla de dispersión es simplemente un arreglo que contiene el producto cartesiano <clave, información> de los elementos del diccionario (en general la clave será una cadena de caracteres o un número, por ejemplo: nombre, cédula, etc.). El tamaño de la tabla es un valor que se denomina **TAMAÑO_T**, y se utiliza la convención de que los índices del arreglo se declaren en el rango 0..TAMAÑO_T-1, haciéndose corresponder cada clave de elemento con un valor del rango.

$h : Claves \rightarrow 0..(TAMAÑO\_\tilde{T} 1)$

Esta correspondencia es denominada **función de dispersión** o **función de hash**, la cual debe ser una función **simple de calcular** e idealmente debe asegurar que dos **claves distintas** caigan en **celdas distintas** del arreglo.

C1!=C2 => h(C1)!=h(C2)

Sin embargo, como existe un número finito de celdas en el arreglo y potencialmente un número infinito de claves, es prácticamente imposible asegurar que dos claves distintas caigan en celdas distintas, produciéndose **colisiones** de elementos en una misma celda. Por esto es necesario buscar una función de dispersión que **distribuya de manera uniforme (homogéneamente)** las claves de los elementos entre las celdas del arreglo.

$$P(h(cj)=K)=1/TAMAÑO\_T \text{ , } 1<=k<=TAMAÑO\_T$$

Esto es, el resultado de aplicarle la función de dispersión a una clave ($_j$) c tiene igual probabilidad de ser cualquiera de los TAMAÑO_T valores disponibles. Dicho de otra forma, cada elemento de la tabla tiene equiprobabilidad de ser ocupado por un elemento de clave $_j c$.

Vista la idea básica de la estructura tabla de dispersión, resta analizar los problemas de elección de la función de dispersión, resolución de colisiones y valor del TAMAÑO_T.

Función de dispersión

El hecho de buscar buenas funciones de dispersión es toda un área de investigación. Se verán algunas funciones sencillas, que resultan de utilidad para varios casos:

Si las claves de los elementos son **valores enteros**, una función de dispersión aceptable en casos de claves aleatorias es hallar el **módulo** entre el valor de la clave y el tamaño de la tabla.

```
clave % TAMAÑO_T
```

Esta función es útil salvo que los valores de las claves tengan algunas

particularidades que impliquen un mayor cuidado. Por ejemplo, si el tamaño de la tabla es 10 y todas las claves (o la mayoría) terminan en 0 (ej.: 0, 10, 20, 30, 40,…) el módulo es una muy mala opción, dado que todos los elementos van al mismo lugar del arreglo. Una posibilidad para evitar estas situaciones, y que por otras razones generalmente es una buena opción, es asegurarse que el tamaño de la tabla de dispersión sea un **número primo**.
Si por ejemplo las claves de los elementos son **cadenas de caracteres**, una posibilidad para la función de dispersión es sumar los valores ASCII de cada carácter de la cadena.

```
int h (const char * clave, int largo){
int disp = 0;
for(int i=0;i<largo;i++)
disp += clave[i];
return disp % TAMAÑO_T;
}
```

```
a = 97, b=98, c=99, …
- h(“a”,1) = 97
-h(“ab”,2) = 195
-h(“abc”,3) = 294
```

Obteniendo así rápidamente un número y pasando entonces a la situación anterior.
Existen problemas cuando se tienen tablas muy grandes. Por ejemplo:

```
TAMAÑO_T = 10.007
Largo máximo de c/clave = 8
```

Considerando que el máximo valor ASCII es 127

```
- 0 <= h(x) <= 1.016
```

Lo cual no es homogéneo, dado que las celdas del 1.017 al 10.006 no se usarían.
Por lo tanto las funciones de dispersión se deben estudiar para cada caso, dependiendo de la naturaleza de las entradas y del tamaño de la tabla.

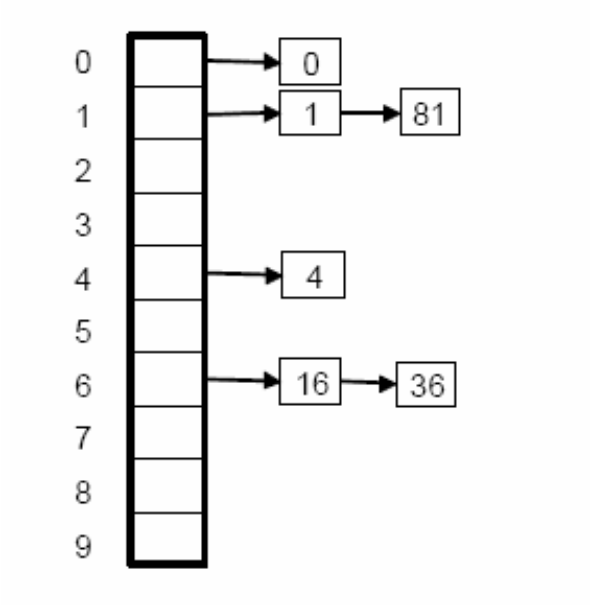
Resolución de colisiones

Otro problema es el relativo a como resolver las colisiones (cuando dos elementos caen en una misma celda del arreglo). Para resolver este problema se verán dos posibilidades: **dispersión abierta** y **dispersión cerrada**.

Dispersión Abierta

En esta estrategia se resuelven las colisiones utilizando **listas** en las que se mantienen todos los elementos que tienen la misma dispersión.
Por ejemplo, en un hash de tamaño 10 y función h(x) = x % 10.

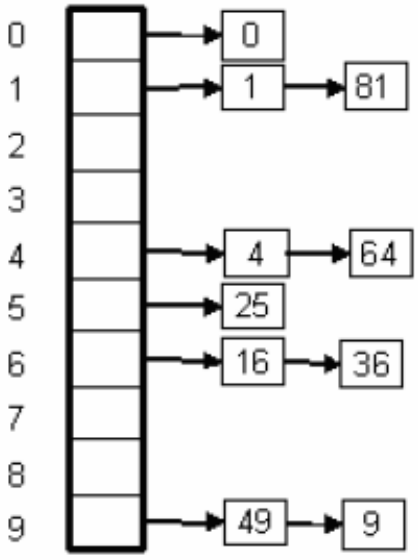
Se tiene:



Si se agrega:

25 → h(25) = 5
64 → h(64) = 4
9 → h(9) = 9
49 → h(49) = 9

Se tiene:



La operación de búsqueda implica evaluar primero la función de dispersión con la clave del elemento, de manera de conocer en cual lista se debe buscar. Luego se recorre la lista hasta llegar al elemento o al final de la misma. La operación de eliminación es similar, dado que implica una búsqueda, en cuanto a la inserción depende que tipo de inserción sobre la lista se realice y de las precondiciones.

```
bool pertenece(Hash tabla, Tipo_Clave clave){
Lista lista = tabla[h(clave)];
```

```
bool encontre = false;
while (!encontre && !vaciaLista(lista)){
Tipo_Elem e = primeroLista(lista);
encontre = comparar(darClaveElem(e),clave);
lista = restoLista(lista);
}
return encontre;
}
```

Se denomina **factor de carga** (lamda) de una tabla de dispersión, a la razón entre la cantidad de elementos de la tabla (**N**) y el tamaño de la misma.

Lamda=N/TAMAÑO_T

En el ejemplo anterior, se tiene N=10 y TAMAÑO_T=10, por lo que lamda = 1.
Si la distribución de elementos es homogénea, entonces la longitud media de cada lista es lamda.

Por lo cual una búsqueda lleva tiempo:

1) El tiempo constante (**k**) requerido para evaluar la función de dispersión.
2) El tiempo necesario para recorrer la lista (**lamda**).

-O(k+lamda) O(1)

La regla general en el caso de dispersión abierta es hacer que el **tamaño de la tabla** sea **casi tan grande como el número de elementos esperados**, de manera que lamda sea cercano a 1 , y que sea un **número primo**, de manera de obtener una buena distribución.
De esta manera se logra que las operaciones de inserción, eliminación y búsqueda tengan un **tiempo promedio** que es **constante** y "no depende" del tamaño de la entrada (cantidad de elementos).

Dispersión Cerrada

Es una alternativa al uso de una segunda estructura (listas) para la resolución de colisiones. En este caso, cuando ocurren colisiones se resuelven tratando de **buscar una celda libre alternativa** en el arreglo.
La inserción funciona de la siguiente manera: se busca colocar el elemento *x* en la celda *h(x)*, si esta celda ya tiene un elemento (colisión) se debe disponer de una estrategia
de **redispersión**, buscando colocar el elemento en una sucesión de celdas $h_0(x)$, $h_1(x)$, $h_2(x)$, … Se prueba en cada una de éstas celdas, en orden, hasta encontrar una vacía, si no
se encuentra la tabla de dispersión está llena y no es posible insertar el elemento x.
Es importante recalcar que, como todos los elementos se colocan en la tabla de dispersión, es necesario que el **tamaño** de la misma sea **mayor que en el caso de**

dispersión abierta. En general el factor de carga debería ser: $\lambda \leq 0.5$.
Formalizando:
Al insertar un elemento *x* se intenta colocarlo en una sucesión de celdas $h_0(x)$, $h_1(x)$, $h_2(x)$, … Donde $h_i(x) = (h(x) + f(i)) \% TAMAÑO\_T$
con $f(0) = 0$
La función $f(x)$ es la estrategia de redispersión o estrategia de resolución de colisiones.

El TAD Tabla (Mapping, Table)

Una tabla es una función (parcial) de elementos de un tipo, llamado el tipo dominio, a elementos de otro (posiblemente el mismo) tipo, llamado el tipo recorrido.

Que una tabla T asocia el elemento r del recorrido al elemento d del dominio lo denotaremos T(d) = r.

Existen funciones, como sqr(i) = i2, que pueden ser fácilmente implementadas como un procedimiento de C que implementa la expresión que computa la función. Sin embargo, para muchas funciones no existe otra forma de describir T(d) más que almacenar para cada d el valor T(d).

Ejemplo: implementar una función que asocie a cada estudiante el conjunto de asignaturas de la carrera que tiene aprobadas.

Veamos cuáles son las operaciones que definen a este tipo abstracto.

Dado un elemento d del dominio nos interesa recuperar el elemento T(d) o saber si T(d) está definido (si d pertenece al dominio de T).

También nos interesa dar de alta elementos en el dominio de T y almacenar sus correspondientes valores. Alternativamente, también nos interesa modificar el valor T(d) para un determinado d. Finalmente, necesitamos una operación que nos permita construír una tabla vacía, es decir, la tabla cuyo dominio es vacío.

Módulo de definición del TAD Tabla

```
//.h módulo de definición del TAD tabla
#ifndef TABLA_H
#define TABLA_H


      #include Dominio
      #include Rango
```

```
typedef struct TablaNodo *Tabla;
```

Tabla CreoTabla ();
/* retorna la tabla vacía */

Tabla Insertar (Dom d, Rango r, Tabla T);
/* Define T(d) = r, independientemente de que T(d) estuviera ya definido */

void Recuperar(Dom d, Tabla T, Rango& resp, bool& def);
/* retorna def=TRUE y r = T(d) si T está definida para d, sino def=FALSE */

#endif TABLA_H

Implementación del TAD Tabla

Existen muchas posibles implementaciones de tablas con dominios finitos. Notar que las implementaciones de conjuntos se adecuan a Tablas.

Una de las implementaciones más utilizadas es por medio de tablas de Hash.

En general, cualquier tabla con dominio finito puede ser representada como una lista de pares (d_1, r_1), (d_2, r_2), ..., (d_k, r_k) donde los d_i son los elementos corrientes del dominio y r_i es el valor que la tabla asocia con d_i, para i = 1..k.

Nuevamente, si el dominio de la tabla admite un orden total, también es una alternativa de implementación razonable usar ABBs (o árboles balanceados).

Una posible Implementación de Tabla

```
#include “Tabla.h”

struct TablaNodo{
      Dom x;
      Rango fx;
      Tabla sig;
}
```

Tabla CreoTabla (){

```
 Tabla t;
 Dom i;
//Implemento utilizando celda Dummy
t= new TablaNodo;
return t ;
```

```
 }

Tabla Insertar( Dom d, Rango r,  Tabla t){

          if(t->sig==NULL){
            t->sig=new TablaNodo;
            t->x=DominioCrear(d);
            t->fx=RangoCrear(r);
            t->sig=NULL;
           return t;

          }else{
            / /se assume que los elementos de tipo Dom tiene alguan propiedad de
          //orden.En caso de no ser así se puede agregar pro ejemplo los elementos
          //al principio de la lista.
             Tabla aux=t;
             while(t->sig!=NULL && !DominioEsMenor(t->sig->x,d)){
                t=t->sig;
             }
             Tabla nuevo= new TablaNodo;
             nuevo->x=DominioCrear(d);
             nuevo->fx=RangoCrear(r);
             nuevo->sig=t->sig;
              t->sig=nuevo;

             return aux;
          }


}

void Recuperar( Dom d,  Tabla T,  Rango & resp, bool& def)
{

  bool termine=false;

  def=false;

    t=t->sig;

    while(!termne && t!=NULL ){

          if  (DominioEsIgual(t->x,d)){

                  def=tremine=true;

          }   else{

               If(  DominioEsMenor(d,t->x)) termine=true;

          }
```

```
        }


    }
```