

Programación 2

**La previa:
TADs Colecciones**

El TAD Set

- **Vacio** c: construye el conjunto c vacío;
- **Insertar x c: agrega x a c, si no estaba el elemento en el conjunto;**
- **EsVacio** c: retorna true si y sólo si el conjunto c está vacío;
- **Pertenece** x c: retorna true si y sólo si x está en c;
- **Borrar** x c: elimina a x del conjunto c, si estaba;
- **Destruir** c: destruye el conjunto c, liberando su memoria;
- **Operaciones para la unión, intersección y diferencia de conjuntos, entre otras.**

Un TAD Multiset

- **Vacio** m : construye el multiset m vacío;
- **Insertar x m : agrega x a m ;**
- **EsVacio** m : retorna true si y sólo si el multiset m está vacío;
- **Ocurrencias** x m : retorna la cantidad de veces que está x en m ;
- **Borrar** x m : elimina una ocurrencia de x en m , si x está en m ;
- **Destruir** m : destruye el multiset m , liberando su memoria.

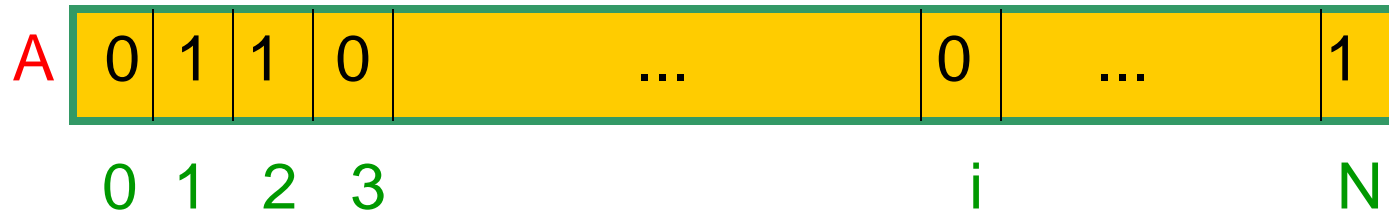
El TAD Diccionario y Cola de Prioridad

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.

Un TAD Set con las operaciones **Vacio**, **Insertar**, **EsVacio**, **Borrar** y **Pertenece** recibe el nombre de **diccionario**.

Una **cola de prioridad** es esencialmente un TAD Set (o Multiset) con las operaciones **Vacio**, **Insertar**, **EsVacio**, **Borrar_Min** y **Min** (**Borrar_Max** y **Max**).

Implementación de Diccionarios y Conjuntos con arreglos de booleanos

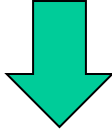


x pertenece al conjunto (diccionario) $\Leftrightarrow A[x]=1$ (true)

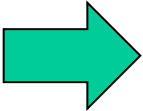
- Restricciones ?
- Eficiencia en tiempo de ejecución ?
- Uso de espacio de almacenamiento ?

Implementación de Diccionarios y Conjuntos con arreglos de booleanos

Nro. curso



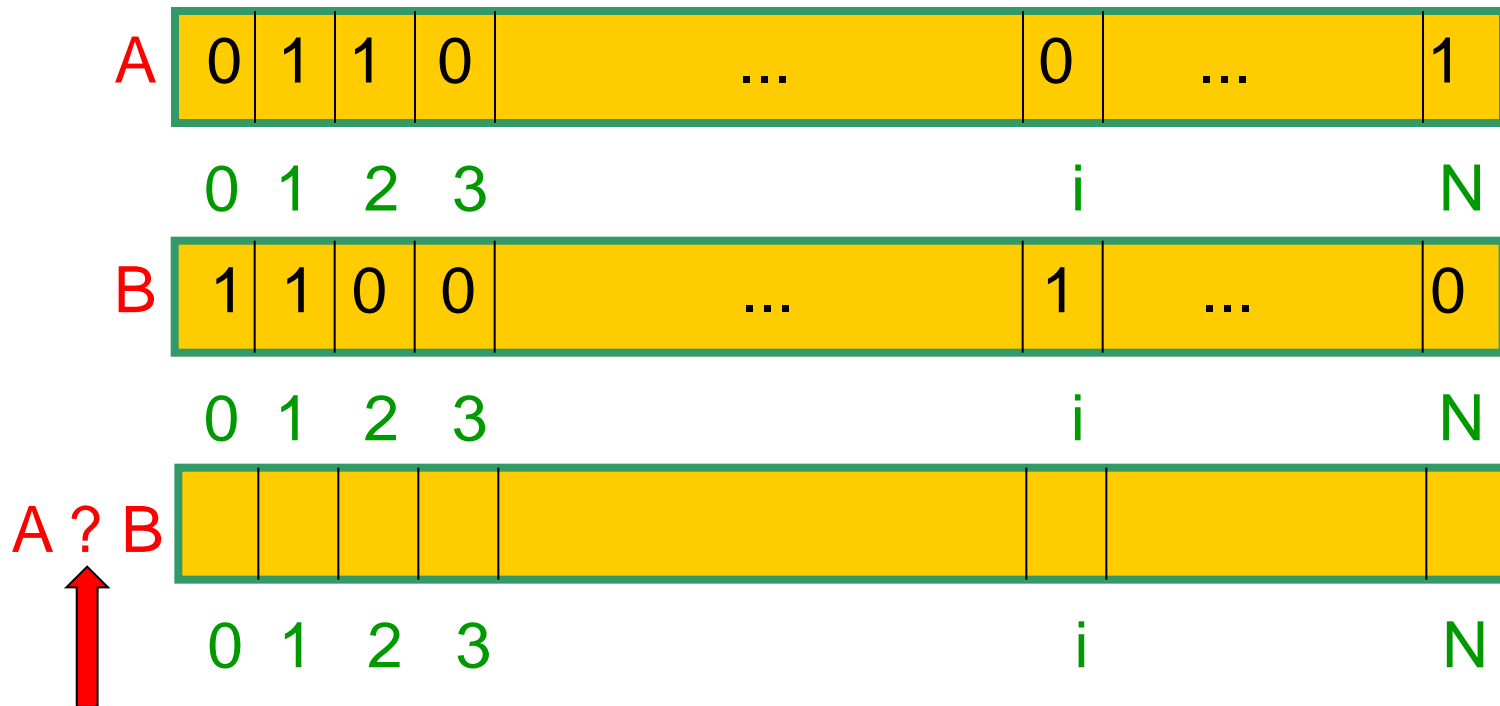
Nro estud.



0	0	0	1	0	1	0
0	0	1/0	0

Implementación de Conjuntos con arreglos de booleanos

- Para Sets, las operaciones **Union**, **InterSec** y **Diferencia** se pueden implementar en un tiempo proporcional al tamaño del conjunto universal.



Unión / Intersección / Diferencia

Especificación del TAD Conjunto de enteros en un rango $[0:N]$

```
struct RepresentacionConjunto;  
typedef RepresentacionConjunto * Conjunto;  
  
Conjunto crearConjunto (int N);  
// Devuelve el conjunto vacío, con elementos posibles en  $[0:N]$ .  
  
void insertarConjunto (int i, Conjunto &c);  
// Agrega i en c, si no estaba. En caso contrario, no tiene efecto.  
// Pre:  $0 \leq i \leq N$  (con N el valor del parámetro en la creación)  
  
void eliminarConjunto (int i, Conjunto &c);  
// Elimina i de c, si estaba. En caso contrario, no tiene efecto.  
// Pre:  $0 \leq i \leq N$  (con N el valor del parámetro en la creación)  
  
bool perteneceConjunto (int i, Conjunto c);  
// Devuelve true si y sólo si i está en c.  
// Pre:  $0 \leq i \leq N$  (con N el valor del parámetro en la creación)  
  
bool esVacioConjunto (Conjunto c);  
// Devuelve true si y sólo si c es vacío.  
  
Conjunto union/interseccion/diferencia (Conjunto c1, Conjunto c2);  
// Devuelve la unión/intersección/diferencia de c1 y c2.  
/* Pre: los valores posibles (i) en c1 y c2 cumplen:  $0 \leq i \leq N$  (con N  
    el valor del parámetro en la creación) */  
  
void destruirConjunto (Conjunto &c);  
// Libera toda la memoria ocupada por c.
```


Pensar aplicaciones en base a la especificación dada, donde por ejemplo los números son códigos de estudiantes en un rango $[0:N]$, cursos en cierto rango, códigos de funcionarios de una empresa en cierto rango...

En el caso de estudiantes inscritos a cursos, considere por ejemplo:

- Pasar y cargar la asistencia a un curso dado;
- en cierto curso determinar los estudiantes inscritos a cierto curso, dado que aprobaron otro curso previo;
- Saber qué estudiantes están en dos cursos a la vez, cuales estan en solo uno, qué estudiantes están activos en cierto semestre...

Hashing

Operaciones relevantes en una colección (Conjunto/Set y Diccionario)

Consideremos las operaciones en un Set:

- Insertar ($T\ x$, Set & s)
- Borrar ($T\ x$, Set & s)
- Pertenece ($T\ x$, Set s)

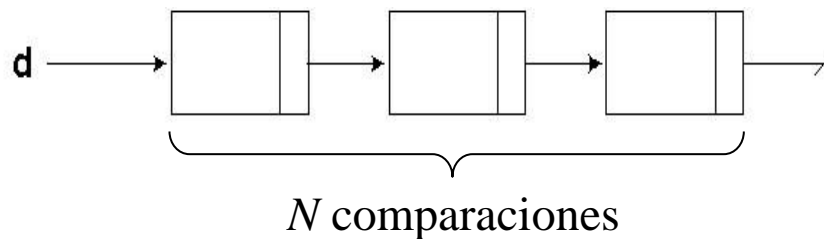
Interesa particularmente la verificación de pertenencia de un elemento.

Hashing

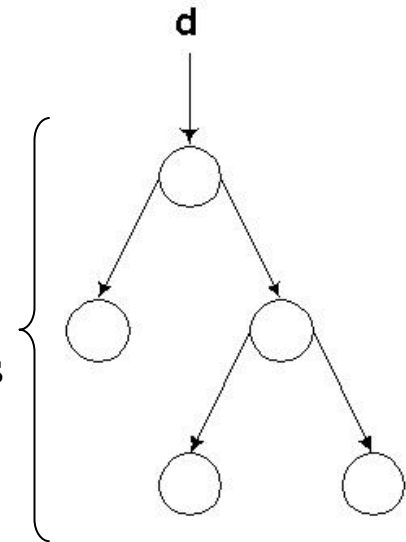
Representaciones

Algunas alternativas conocidas mencionadas:

- Listas encadenadas.
- Árboles binarios de búsqueda.



$\log(N)$
comparaciones
en promedio



Arreglo de booleanos

Sea S un set implementado con un arreglo de booleanos (1 / 0), y s un array de N elementos booleanos. Se define a $s[n]$ como verdadero si y solo si $\text{Pertenece}(n, S)$.

0	1	2	3	4	5
1	1	0	1	0	0

$s[0] = \text{true} \rightarrow 0 \in S$

$s[1] = \text{true} \rightarrow 1 \in S$

$s[2] = \text{false} \rightarrow 2 \notin S$

$s[3] = \text{true} \rightarrow 3 \in S$

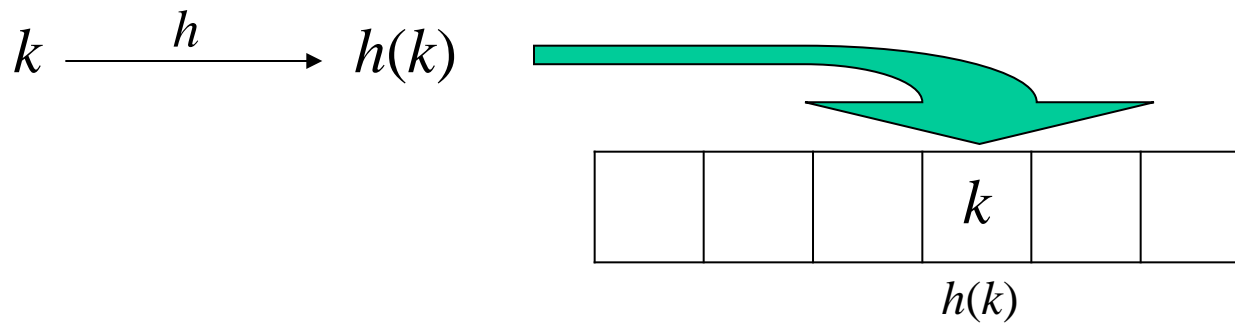
$s[4] = \text{false} \rightarrow 4 \notin S$

$s[5] = \text{false} \rightarrow 5 \notin S$

Noción de hash

Consiste en:

- Array de M elementos conocidos como “buckets” (tabla de hash).
- Función h de dispersión sobre las claves.
- Estrategia de resolución de colisiones.



Colisiones

Ninguna función de hash puede garantizar que la estructura está libre de colisiones.

Toda implementación de hash debe proveer una estrategia para su resolución.

Estrategias más comunes:

- Separate chaining (*hashing* abierto)
- Linear probing (*hashing* cerrado)
- Double hashing

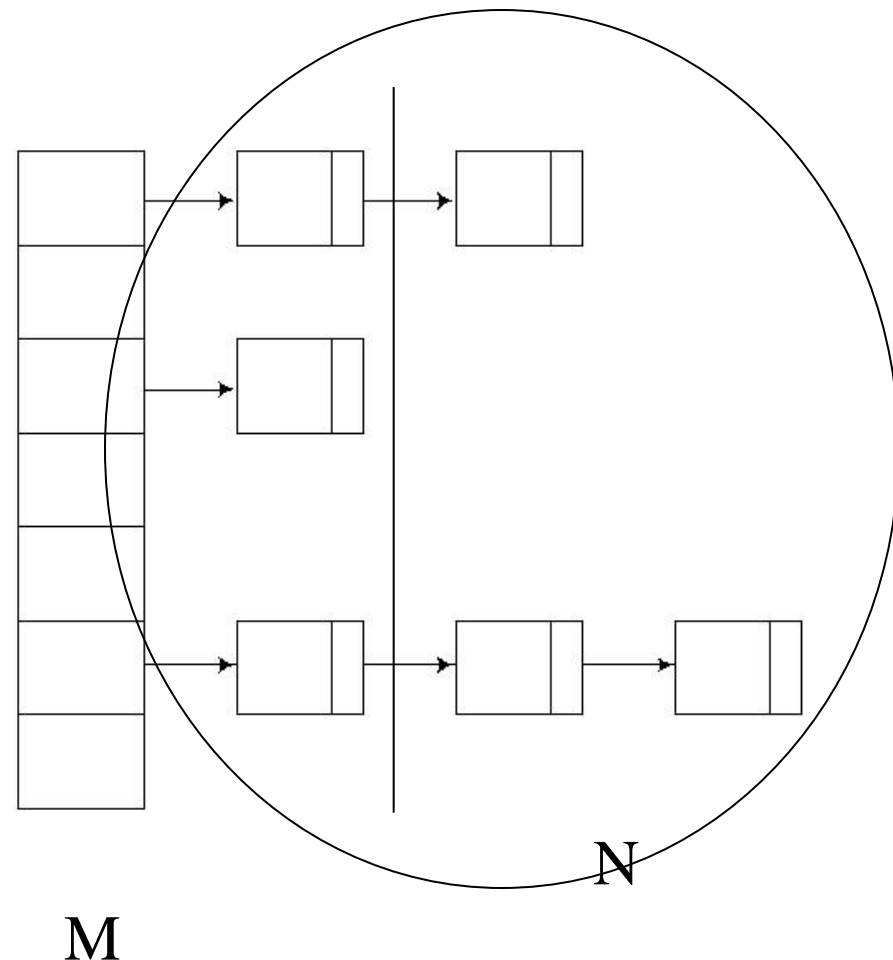
Separate chaining

Hasing Abierto

Las colisiones resultan en un nuevo nodo agregado en el bucket.

Las listas en promedio tienen largo acotado.

La verificación de pertenencia de k implica buscar en la lista del bucket $h(k)$.



Separate chaining




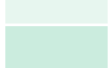
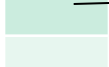


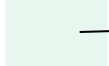


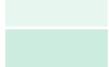


Hasing Abierto

Ejemplo:

- $M=10$
- $\text{hash: int} \rightarrow \text{int}, \text{hash}(x) = x \% M$
- 0, 5, 10, 15, 20, 25 ...

Insertar ($M=13$):

3, 0, 7, 16, 11, 26, 10, 40, 12,
131, 1308, 265, 15 ...

Tabla		
0		→ [0,26,...]
1		→ [40,131...]
2		→ [15,...]
3		→ [3,16,...]
4		→ [...]
5		→ [265,...]
6		→ [...]
7		→ [7,...]
8		→ [1308,...]
9		→ [...]
10		→ [10,...]
11		→ [11,...]
12		→ [12,...]

Tiempo promedio y peor caso

$O(1)$ promedio: Hashing utiliza tiempo constante por operación, en promedio, y no existe la exigencia de que los conjuntos sean subconjuntos de algún conjunto universal finito (como en los arreglos de booleanos). *El factor de carga y la función de hash son factores claves.*

$O(n)$ peor caso: En el peor caso este método requiere, para cada operación, un tiempo proporcional al tamaño del conjunto, como sucede con realizaciones con listas encadenadas y arreglos.

Especificación del TAD Diccionario no acotado? de enteros

```
#ifndef _DICCIONARIO_H
#define _DICCIONARIO_H

struct RepresentacionDiccionario;
typedef RepresentacionDiccionario * Diccionario;

Diccionario crearDiccionario (int cota);
// Devuelve el diccionario vacío para una cantidad estimada cota.

void insertarDiccionario (int i, Diccionario &d);
// Agrega i en d, si no estaba. En caso contrario, no tiene efecto.

void eliminarDiccionario (int i, Diccionario &d);
// Elimina i de d, si estaba. En caso contrario, no tiene efecto.

bool perteneceDiccionario (int i, Diccionario d);
// Devuelve true si y sólo si i está en d.

bool esVacioDiccionario (Diccionario d);
// Devuelve true si y sólo si d es vacío.

void destruirDiccionario (Diccionario &d);
// Libera toda la memoria ocupada por d.

#endif /* _DICCIONARIO_H */
```

Implementación de un Diccionario acotado de enteros con hashing abierto

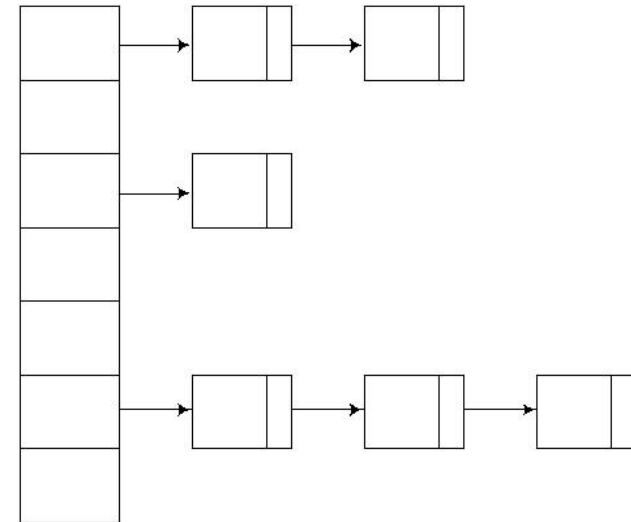
```
#include ...
#include "diccionario.h"

int hash (int i){return i;} // si todos los enteros son igualmente probables

struct nodoHash{
    int dato;
    nodoHash* sig;
};

struct RepresentacionDiccionario{
    nodoHash** tabla;
    int cantidad;
    int cota;
};

Diccionario crearDiccionario (int cota) {
    Diccionario d = new RepresentacionDiccionario();
    d->tabla = new (nodoHash*) [cota];
    for (int i=0; i<cota; i++) d->tabla[i]=NULL;
    d->cantidad = 0;
    d->cota = cota;
    return d;
}
```



Implementación de Diccionario acotado de enteros con hashing abierto

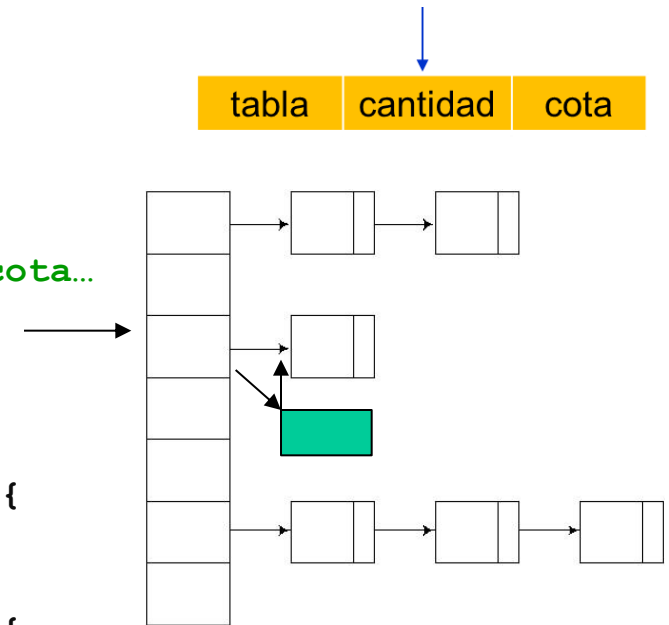
```
void insertarDiccionario (int i, Diccionario &d){
  \\ No se analiza el factor de carga
  if !peteneceDiccionario(i, d){
    nodoHash* nuevo = new nodoHash;
    nuevo->dato = i;
    nuevo->sig = d->tabla[hash(i)%(d->cota)];
    \\ la función hash podría aplicarse a i y d->cota...
    d->tabla[hash(i)%(d->cota)] = nuevo;
    d->cantidad++;
  }
}

void eliminarDiccionario (int i, Diccionario &d) {
  ...
}

bool perteneceDiccionario (int i, Diccionario d) {
  nodoHash* lista = d->tabla[hash(i)%(d->cota)];
  \\ la función hash podría aplicarse a i y d->cota...
  while (lista!=NULL && lista->dato!=i)
    lista = lista->sig;
  return lista!=NULL;
}

bool esVacioDiccionario (Diccionario d) {...}
bool esLlenoDiccionario (Diccionario d){...}

void destruirDiccionario (Diccionario &d) {...}
```



Definición de un CLON para un Diccionario acotado de enteros implementado con hashing abierto.

Asumimos que la especificación incluye la operación `clonDiccionario`

```
Diccionario clonDiccionario (Diccionario d){  
    Diccionario clon = crearDiccionario(d->cota) ;  
    for (int i=0; i<d->cota; i++){  
        nodoHash* lista = d->tabla[i];  
        while (lista!=NULL){  
            insertarDiccionario(lista->dato, clon);  
            lista = lista->sig;  
        }  
    }  
    return clon;  
}
```

