# Programación 2

# Estructuras Múltiples (Multiestructuras)

# Estructuras Múltiples

Con frecuencia, un problema aparentemente simple de representación de un conjunto o correspondencia conlleva un difícil problema de elección de estructuras de datos.

La elección de una estructura de datos para el conjunto simplifica ciertas operaciones, pero hace que otras lleven demasiado tiempo y al parecer no existe una estructura de datos que posibilite lograr cierta eficiencia en un conjunto de operaciones.

En tales casos, la solución suele ser el uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia, buscando acceso rápido pero sin redundancia de información.

# Ejemplos (1)

PROBLEMA: Ranking de la FIFA

Se desea mantener una escala de equipos de futbol en la que cada equipo esté situado en un único puesto. Los equipos nuevos se agregan en la base de la escala, es decir, en el puesto con numeración más alta. Un equipo puede retar a otro que esté en el puesto inmediato superior (el i al i-1, i>1), y si le gana, cambia de puesto con él.

Pensar en una representación para esta situación !!

# Ejemplos (1)

Se puede representar la situación anterior mediante un TAD cuyo modelo fundamental sea una correspondencia de nombres de equipos (cadenas de char) con puestos (enteros 1, 2, …).

Las 3 operaciones a realizar son:

- AGREGA(nombre): agrega el equipo nombrado al puesto de numeración más alta.
- RETA(nombre): es una función que devuelve el nombre del equipo del puesto i-1 si el equipo nombrado está en el puesto i, i>1.
- CAMBIA(i): intercambia los nombres de los equipos que estén en los puesto i e i-1, i>1.

# Ejemplos (1)

ALTERNATIVA 1:

Un arreglo de ESCALA, donde ESCALA[i] sea el nombre del equipo en el puesto i.

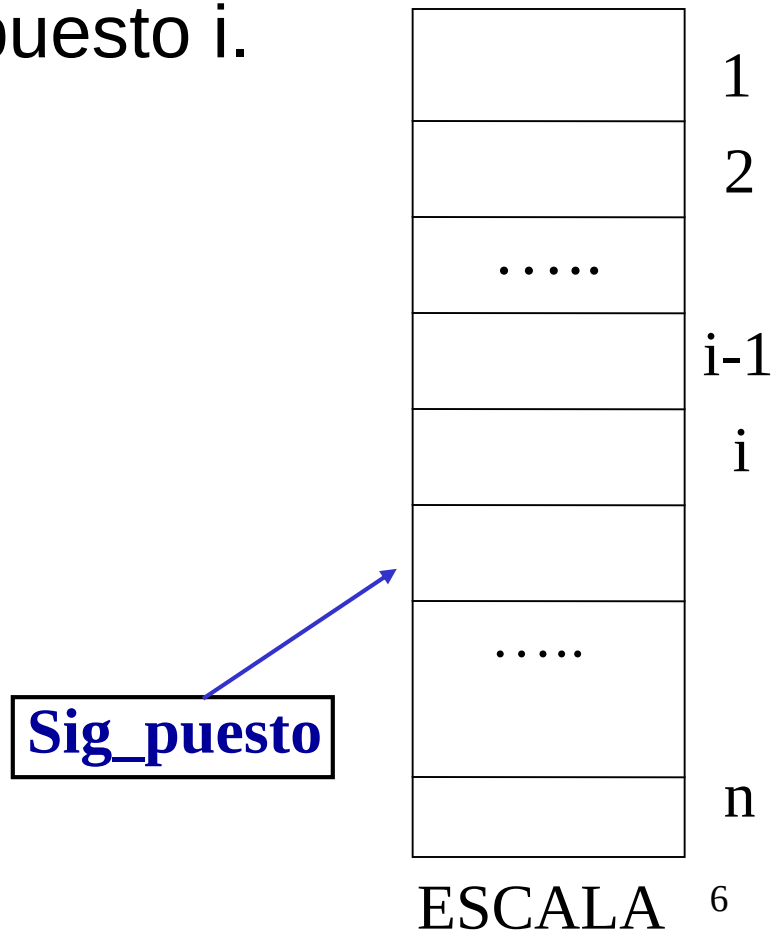
- AGREGA: Cómo sería?, Qué tiempo llevaría?
- CAMBIA: Cómo sería?, Qué tiempo llevaría?
- RETA(nom): Cómo sería?, Qué tiempo llevaría?

# Ejemplos (1)

ALTERNATIVA 1:

Un arreglo de ESCALA, donde ESCALA[i] sea el nombre del equipo en el puesto i.

- AGREGA(nombre)
- RETA(nombre)
- CAMBIA(i)



# Ejemplos (1)

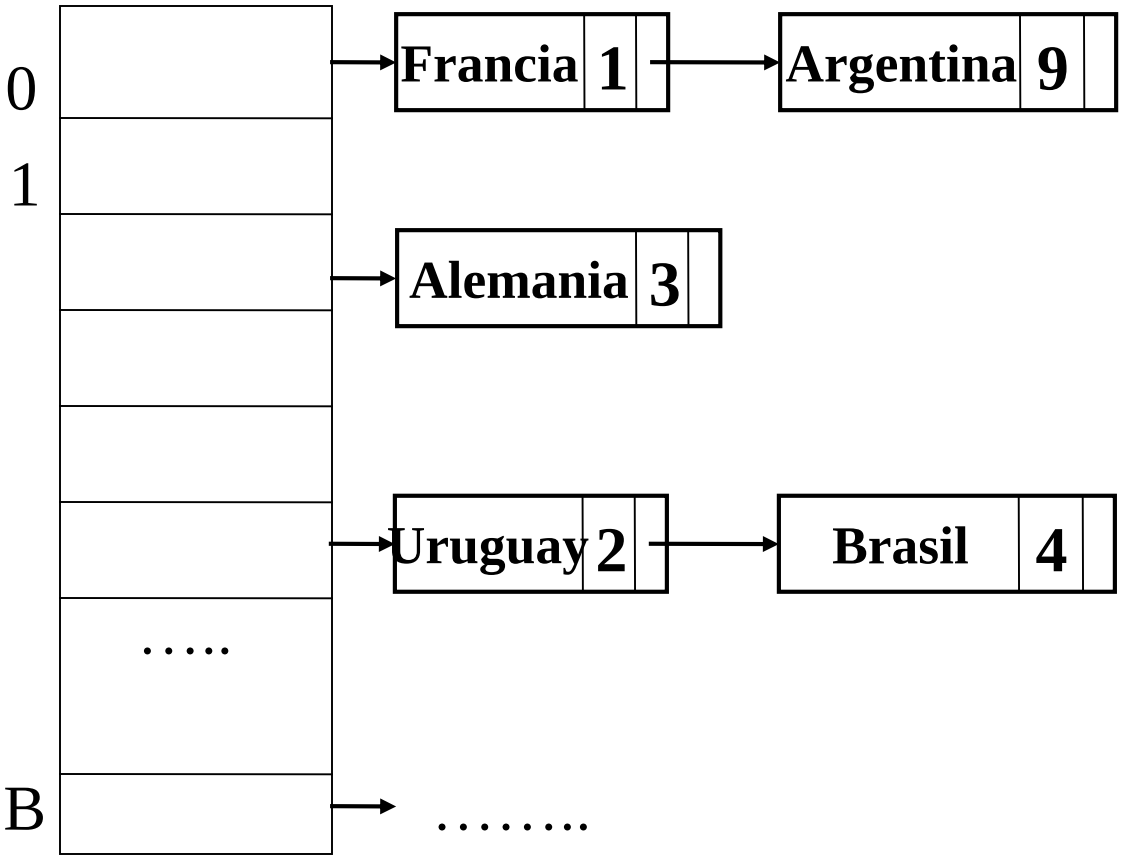
ALTERNATIVA 2: Qué otra representación podría considerarse?

=> OPEN HASING (en el supuesto que es posible mantener en número de *buckets* proporcional al número de equipos)

- AGREGA: Qué tiempo llevaría?
- CAMBIA: Qué tiempo llevaría?
- RETA(nom): Qué tiempo llevaría?

# Ejemplos (1)

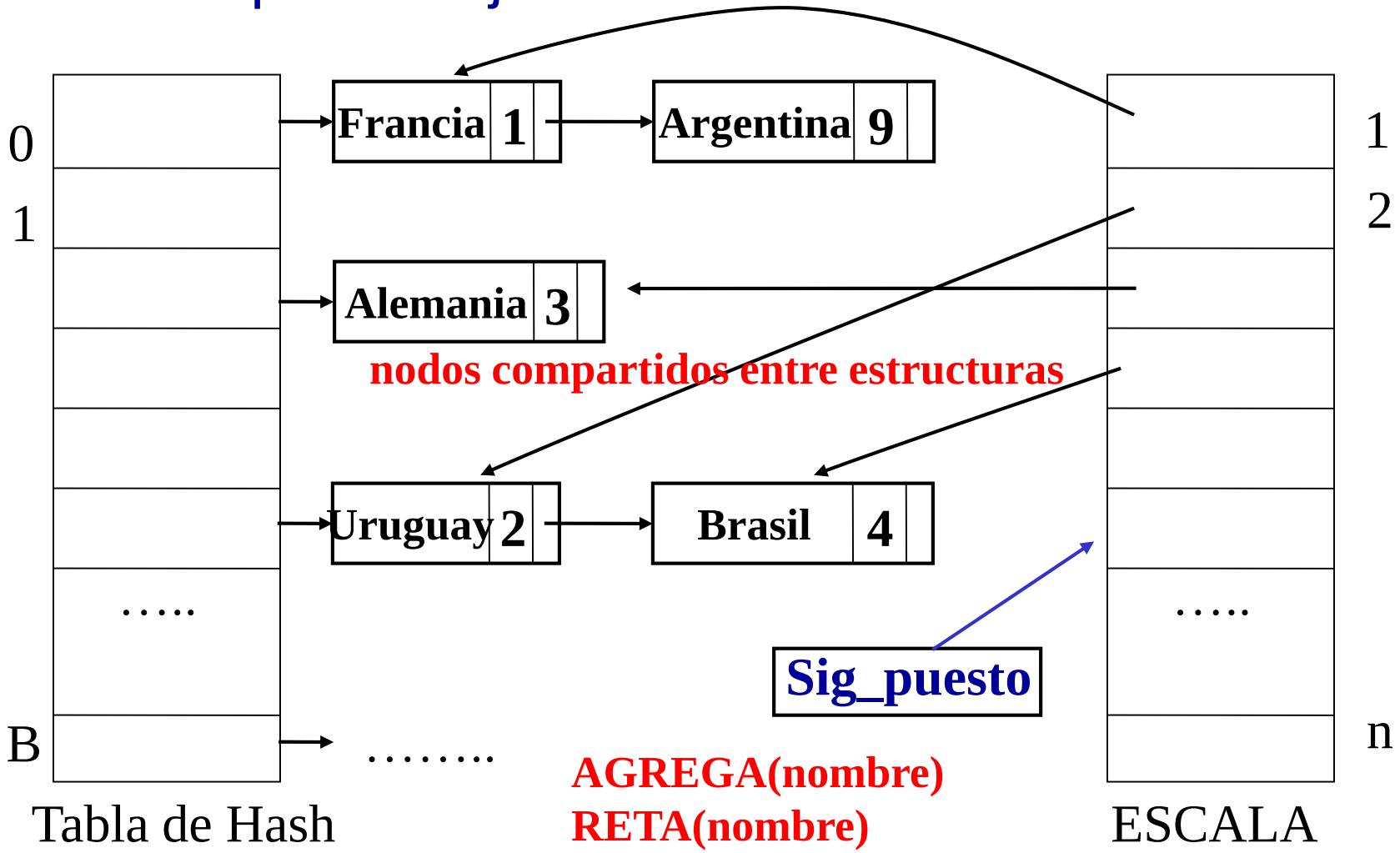
ALTERNATIVA 2: OPEN HASING para asociaciones:
nombre – puesto; con clave: nombre de equipo.




- AGREGA(nombre)
- RETA(nombre)
- CAMBIA(i)

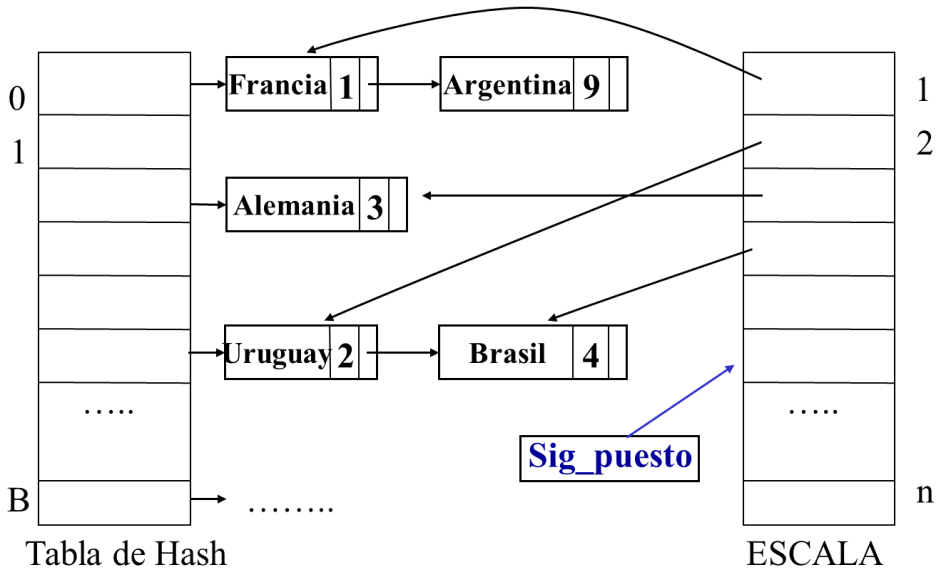
# Ejemplos (1)

Y si combinamos las dos estructuras, ¿ cuáles son los tiempos de ejecución ?



# Ejemplos (1)

```
struct nodoHash{
    char* pais;
    int puesto;
    nodoHash* sig;
};
strutct multiFIFA{
    nodoHash** ESCALA;
    int Sig_puesto;
    int cotaPaises;
    nodoHash** TablaHash;
};
typedef multiFIFA* FIFA;
```




**Asumimos: *int hash (char* nombre)*, con distribución uniforme sobre [0:-].**

**Implementar:**

**FIFA crearFIFA (int cotaPaises){ … }**

**void AGREGA(FIFA & f, char* pais){ … }**

**char* RETA(FIFA f, char* pais){ … }**

**void CAMBIA(FIFA & f, int posicion){ … }**

# Ejemplos (2)

Analicen el ejemplo de asociaciones muchos a muchos y el uso de una estructura de listas múltiples del libro de Aho, Hopcroftand y Ullman (cap 4, secc 4.12).

estudiantes - cursos

- un estudiante inscripto a más de un curso.
- un curso tiene más de un estudiante.

Pensar en una representación simple y alguna más eficiente (sobre todo en cuanto a espacio de almacenamiento)

# Ejemplos (3)

Se desea una estructura de datos para mantener información de atletas que han participado en la competencia de los 100 metros llanos en los juegos olímpicos en de los últimos 20 años. Los datos que interesan sobre cada competidor son la posición (1 a 8) y el año en que compitió. Cada competidor está identificado por un código (entero). Si un atleta participó más de una vez, aparece con la mejor posición que obtuvo.

Pensar en una representación del problema, para que los siguientes requerimientos se realicen eficientemente:

# Ejemplos (3)

- ¿ Cuáles competidores salieron alguna vez en la primera posición ?.
- ¿ Saber el año en que un competidor obtuvo el máximo puesto ?.
- Imprimir los competidores ordenados por código.
- Imprimir todos los competidores que están en el puesto k ($1 \leq k \leq 8$).
- Saber cuál es el puesto más alto obtenido por un competidor.
- Las naturales de inserción, supresión y búsqueda de competidores en una posición.

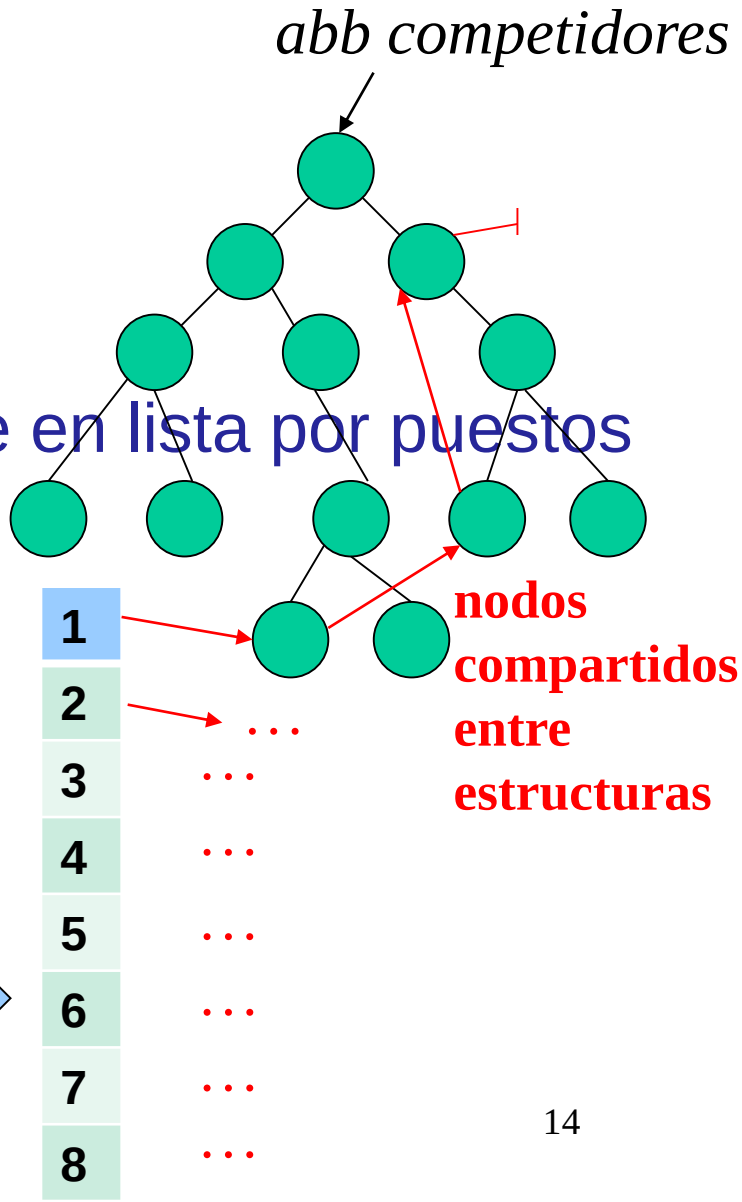
# Ejemplos (3)

Considere la siguiente estructura:

```
typedef struct nodo {
    int codigo, año, puesto;
    struct nodo *izq, *der;
    struct nodo *sig;        // Siguiente en lista por puestos
    } celdaCompe;

typedef celdaCompe *abbCompe;

typedef struct {
    abbCompe competidores;
    abbCompe puestos[8];
} resultados100m;
```



# Ejemplos (3)

La anterior es una estrucura dual que permite ingresar por código de jugador o por puesto. El campo competidores es un árbol binario de búsqueda por código del competidor. El campo puestos es un arreglo, que en el lugar i-ésimo contiene un puntero a la lista de competidores que obtuvieron el puesto i. Esta lista está hilvanada sobre el mismo árbol.

- Ejercicio: escribir un procedimiento con el encabezado:

  **void BorrarCompetidor (resultados100m *Res, int cod)**

  Este procedimiento borra de Res al competidor con código cod.

# Ejemplos (3)

- Indique el orden de ejecución para el caso medio del procedimiento **BorrarCompetidor**.

  Modifique la estructura de forma tal que ese tiempo pase a ser O(log n).

- Ejercicio: escriba el procedimiento BorrarCompetidor para la modificación propuesta en el punto anterior.

# Ejemplos (4)

Los empleados de cierta compañía se representan en la base de datos de la compañía por su nombre (que se supone único), número de empleado y número de seguridad social. Se sabe que la cantidad de empleados puede estimarse en un valor K. Se pide:

- Sugerir una estructura de datos que permita, dada la representación de un empleado, encontrar las otras dos representaciones del mismo individuo de la forma más rápida, en promedio, y evitando redundancia de información. Qué rápida, en promedio, puede lograrse que sea cada una de estas operaciones?. Justifique.

# Ejemplos (4)

- Definir en C++ la estructura del inciso anterior y desarrollar, en C++, la operación que dado el nombre de un empleado imprime su número de empleado y número de seguridad social, si el empleado existe.
- Definir en C++ la operación que dado un empleado lo inserta en la estructura.

Considerando la estructura por usted definida, pueden imprimirse los empleados de la compañía ordenados por su nombre de forma ordenada en O(n)?. Justifique.

# Ejemplos (4)

Estructura:

La estructura está compuesta por tres tablas de hash abiertas (por ejemplo) de tamaño K, una por cada representación del empleado. Los nodos de las listas en las tablas son compartidos, entre los tres hash, para evitar redundancia de información. Cada tabla de hash tendría una función de hash (dos de ellas tienen dominio int y la otra string)

Las tres operaciones resultan O(1) promedio ya que el algoritmo de búsqueda en un hash requiere, en promedio, tiempo de ejecución constante.

# Ejemplos (4)

```
struct nodoEmpleado
{  char             *nombre;
   int              nro_seg_social;
   int              nro_empleado;
   nodoEmpleado     *sigHash_nombre;
   nodoEmpleado     *sigHash_nro_empleado;
   nodoEmpleado     *sigHash_nro_seg_social;
};

struct Estructura
{  nodoEmpleado*    HashNombre[K];
   nodoEmpleado*    HashNroEmpleado[K];
   nodoEmpleado*    HashNroSegSocial[K];
};
```

**nodos compartidos entre estructuras**

# Ejemplos (4)

Para definir el procedimiento supondremos definida las siguiente función:

```
int hNomEmp (char *)
```

función de hash para la tabla por nombre de empleado que retorna un entero entre 0 y K-1

```
void ImprimirDatosEmpleado (Estructura e, char *nom)
{  nodoEmpleado *listNom;
   listNom = e.HashNombre [hNomEmp(nom)];
   while (listNom != NULL && listNom->nombre != nom)
       listNom = listNom->sigHash_nombre;
   if (listNom != NULL) {
            cout << listNom->nro_seg_social;
            cout << listNom->nro_empleado;
    }
}
```

# Ejemplos (5)

Una institución desea mantener información de los alumnos de una carrera universitaria. Para cada alumno interesa su código de alumno de tipo alfanumérico de 10 caracteres (que se supone único), nombre completo (que se supone único), la colección de materias que aprobó, junto con las notas de aprobación obtenidas, y el conjunto de los compañeros con los cuales realizó algún proyecto a lo largo de su carrera. Cada materia está identificada por un código, que se supone único. Estos códigos son números posibles entre 0 y un entero positivo L. Cada alumno puede tener a lo sumo K compañeros de proyectos diferentes a lo largo de toda su carrera. La relación “compañero de proyecto”, entre dos alumnos, es simétrica (necesariamente). Se sabe que el número de alumnos está acotado por un valor M y se pude asumir que L = O(log M).

Se quieren satisfacer los siguientes requerimientos:

# Ejemplos (5)

*Nombre* **nombre** (*Estructura* e, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema y el código *cod* de un alumno, devuelve el nombre del alumno correspondiente. Si el código de alumno no existe la operación retorna “NN” como nombre. Esta operación se debe realizar en O(1) caso promedio.

*CodigoAlumno* **codigo** (*Estructura* e, *Nombre* nom)

en el cual dada la estructura que representa los datos del problema y el nombre *nom* de un alumno, devuelve el código de alumno correspondiente. Si el nombre del alumno no existe la operación retorna “-1” como código. Esta operación se debe realizar en O(1) caso promedio.

# Ejemplos (5)

void **listado** (*Estructura* e)

en el cual dada la estructura que representa los datos del problema, imprime los nombres de los alumnos ordenados a lfabéticamente. Esta operación se debe realizar en O(n) peor caso, siendo *n* la cantidad actual de alumnos.

void **asignar_compañero** (*Estructura* & e, *Nombre* nom, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema, el nombre *nom* de un alumno y el código *cod* de otro alumno, los asigna como compañeros de proyecto, siempre que esto sea posible. La operación no tiene efecto si *cod* o *nom* no están registrados como alumnos en la estructura, o si se excede el máximo de compañeros permitidos para *nom* o para el alumno con código *cod*. Asumimos que ambos alumnos no son ya compañeros de proyecto. Esta operación se debe realizar en O(1) caso promedio.

# Ejemplos (5)

void **cargar_materia** (*Estructura* & e, *CodigoAlumno* cod, *CodigoMateria* cm, *Calificación* cal)

en el cual dada la estructura que representa los datos del problema, el código *cod* de un alumno, el código *cm* de una materia y una calificación *cal*, agrega a la materia de código *cm*, y con calificación *cal*, a la colección de materias aprobadas por el alumno, siempre que esto esa posible. Si el alumno tiene ya la materia aprobada, actualiza la calificación con *cal*. Esta operación se debe realizar en O(1) caso promedio.

void **agregar_alumno** (*Estructura* & e, *Nombre* nom, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema, el nombre *nom* de un alumno y su código de alumno *cod*, agrega al alumno en la estructura, con ninguna metria cursada y un conjunto vacío de compañeros de proyectos.

# Ejemplos (5)

En caso de encontrar algún alumno con el mismo código *cod* o con el mismo nombre *nom* devuelve un mensaje de error sin modificar la estructura. Esta operación se debe realizar en O(log n) caso promedio, siendo *n* la cantidad actual de alumnos.

Se pide:

a) Diseñar estructuras de datos apropiadas para implementar eficientemente dichas operaciones, describiendo como se obtienen las cotas de tiempo pedidas.

b) Dar una declaración en C/C++ de los tipos de datos descriptos en la parte a).

c) Escribir en C/C++ el procedimiento *agregar_alumno*. A los efectos de definir este procedimiento hay que utilizar solamente las estructuras definidas en la parte a).