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.

Ejemplo

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 !!

Ejemplo

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.

Ejemplo

ALTERNATIVA 1:

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

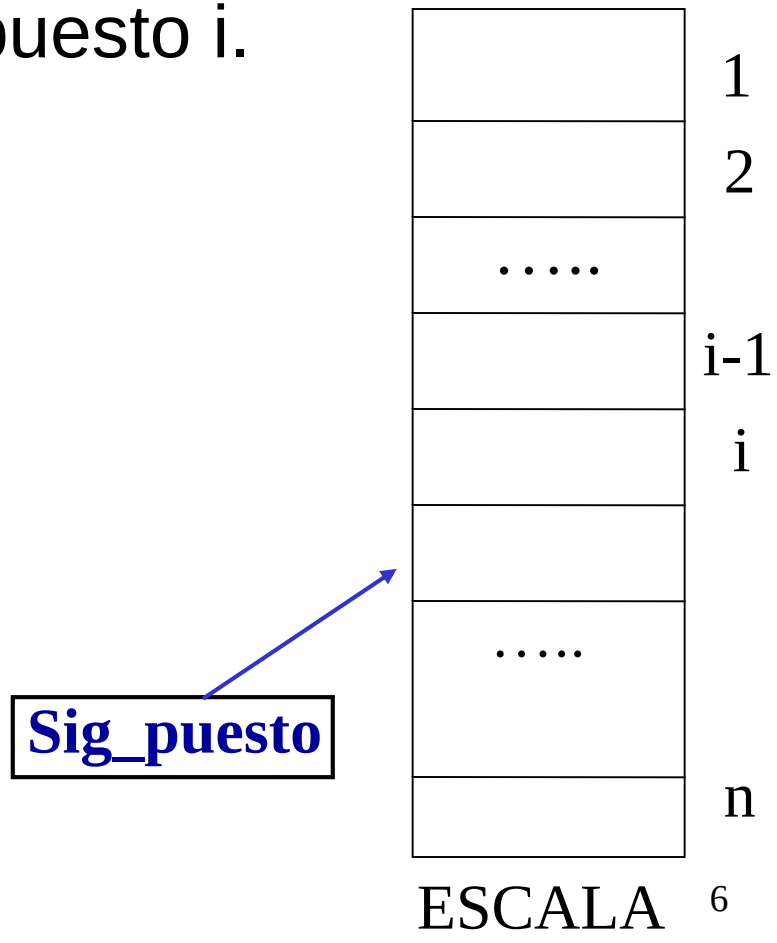
- AGREGA: Cómo sería?, Qué tiempo llevaría?
- CAMBIA: Cómo sería?, Qué tiempo llevaría?
- RETA(nom): Cómo sería?, Qué tiempo llevaría?

Ejemplo

ALTERNATIVA 1:

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

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



Ejemplo

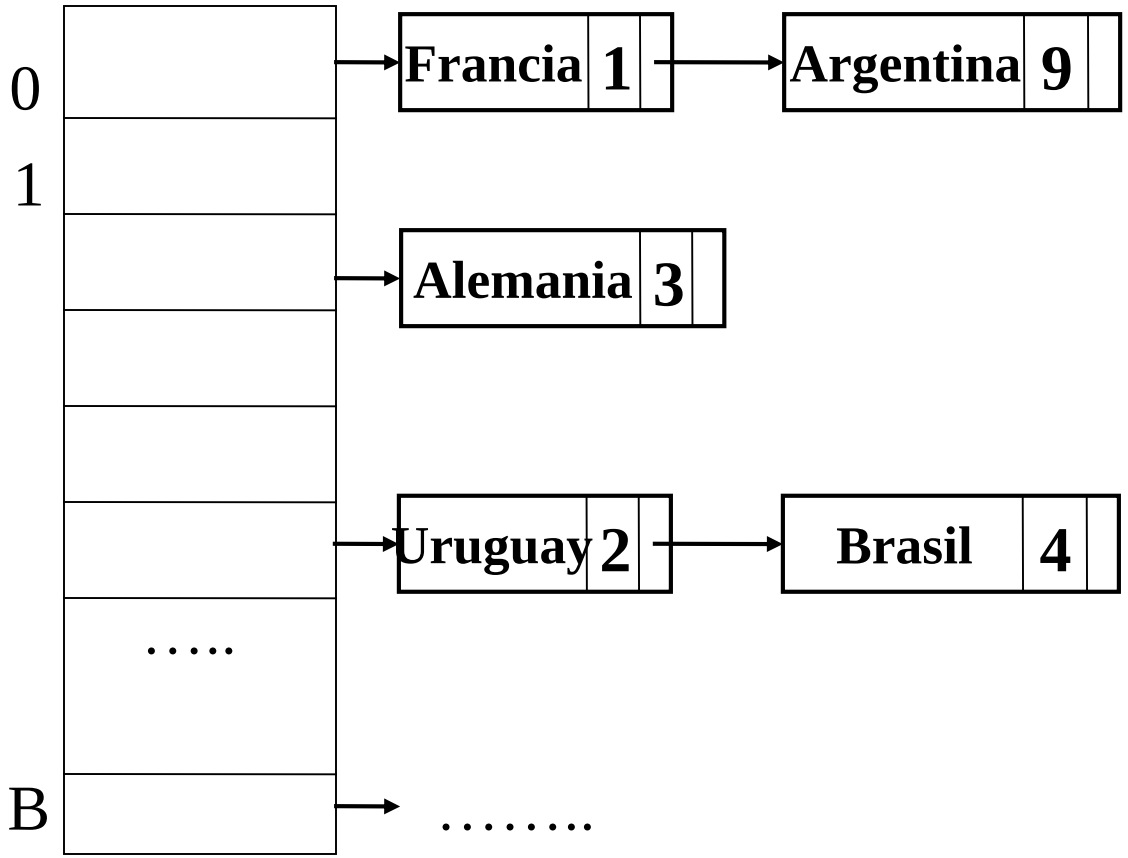
ALTERNATIVA 2: Qué otra representación podría considerarse?

=> 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?

Ejemplo

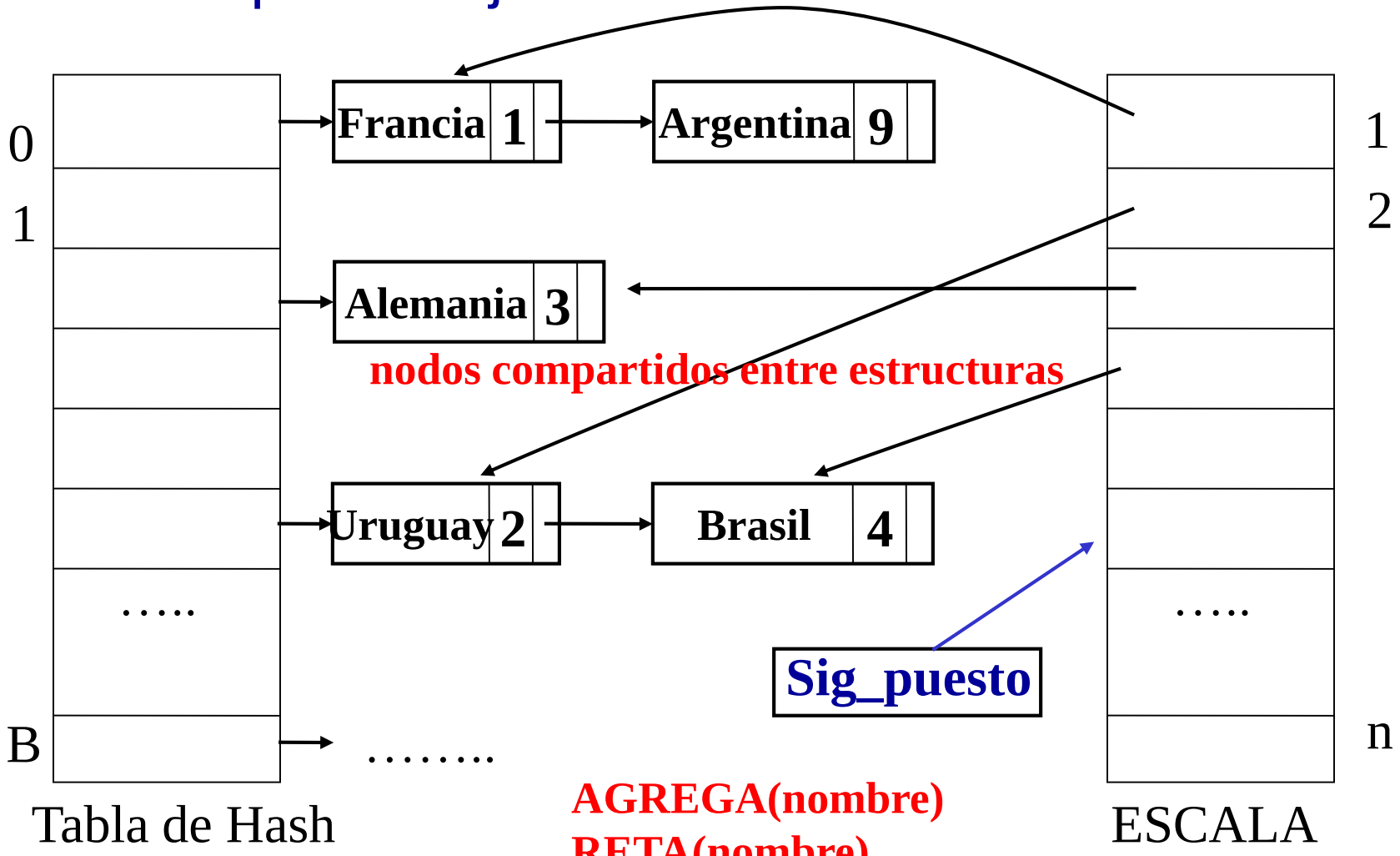
ALTERNATIVA 2: OPEN HASING para asociaciones:
nombre – puesto; con clave: nombre de equipo.



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

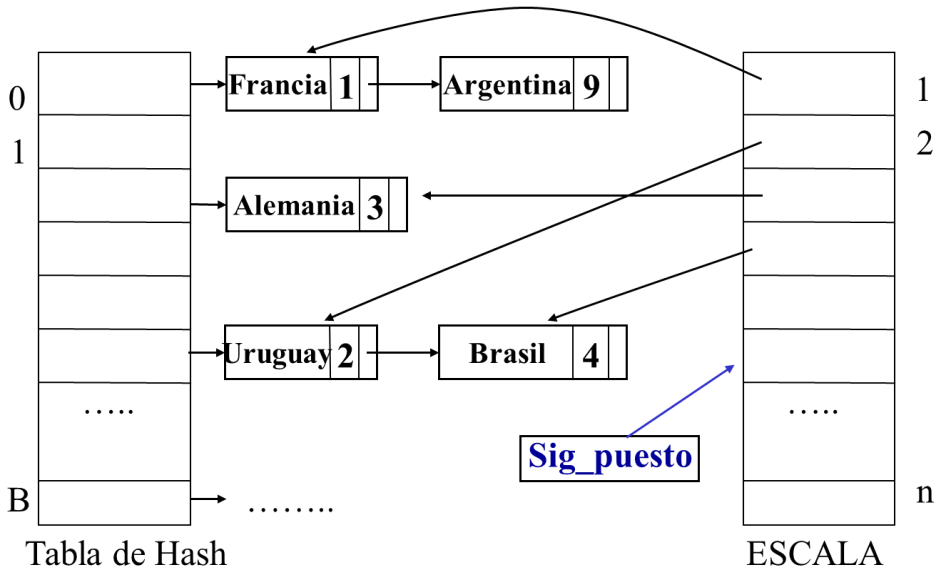
Ejemplo

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



Ejemplo

```
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){ … }

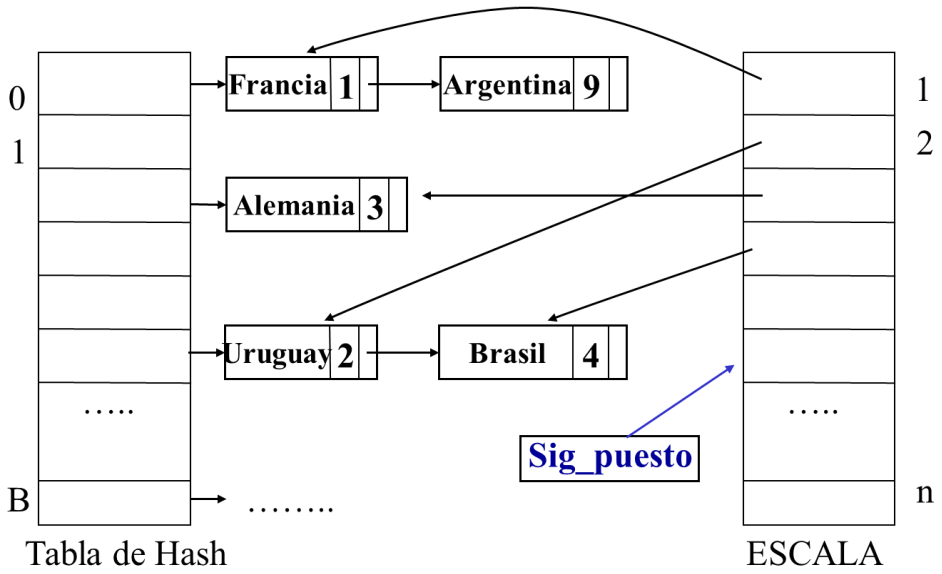
void AGREGA (FIFA & f, char* pais){ … }

char* RETA (FIFA f, char* pais){ … }

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

Ejemplo

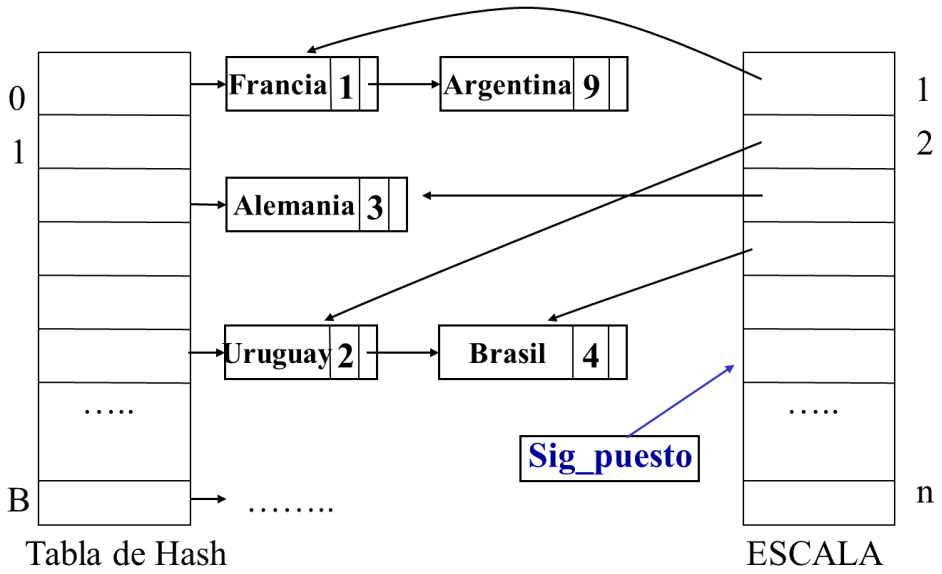
```
struct nodoHash{
        char* pais;
        int puesto;
        nodoHash* sig;
};
strutct multiFIFA{
        nodoHash** ESCALA;
        int Sig_puesto;
        int cotaPaises;
        nodoHash** TablaHash;
};
typedef multiFIFA* FIFA;
```




```
char* RETA(FIFA f, char* pais){
        int posicion = hash(pais)%(f->cotaPaises);
        nodoHash* lista = f->TablaHash[posicion];
        while (lista!=NULL && strcmp(lista->pais, pais)!=0)
                lista = lista->sig;
        if (lista!=NULL &&  lista->puesto!=1){
                return (f->ESCALA[lista->puesto - 1])->pais;
        }
        else return NULL;
}
```

Ejemplo

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




```
void CAMBIA(FIFA & f, int posicion){
        nodoHash* nodoVencedor = f->ESCALA[posicion];
        nodoHash* nodoPerdedor = f->ESCALA[posicion-1];
        nodoVencedor->puesto = posicion-1;
        nodoPerdedor->puesto = posicion;
        f->ESCALA[posicion] = nodoPerdedor;
        f->ESCALA[posicion-1] = nodoVencedor;
}
```

Ejemplo 2

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.

Ejemplo 2

```
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

K es una constante aquí. Podría ser una variable, componente de Estructura

Ejemplo 2

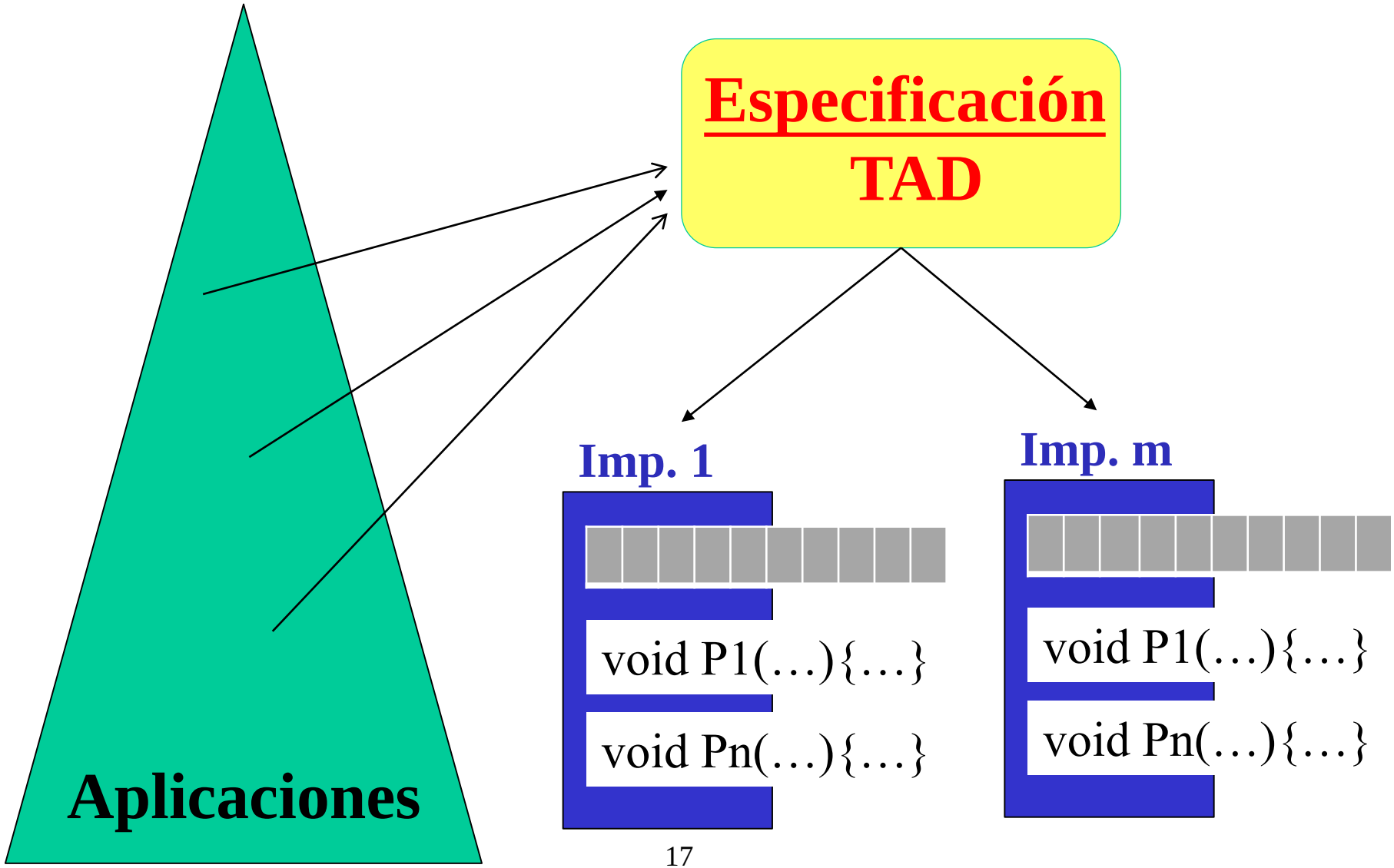
Supongamos definida la siguiente función de hash:

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

```
void ImprimirDatosEmpleado (Estructura* e, char *nom)
{  nodoEmpleado *listNom;
   listNom = e->HashNombre[hNomEmp(nom)%K];
   while (listNom != NULL && strcmp(listNom->nombre,nom)!=0)
        listNom = listNom->sigHash_nombre;
   if (listNom != NULL) {
            cout << listNom->nro_seg_social;
            cout << listNom->nro_empleado;
    }
}
```

Introducción al Diseño de Tipos Abstractos de Datos

Sobre TADs


Especificación
TAD
Aplicaciones
Imp. 1
void P1(…){…}
void Pn(…){…}
Imp. m
void P1(…){…}
void Pn(…){…}

Especificación de TADs

Tipo (opaco – puntero a una representación elegida)

Operaciones

CONSTRUCTORAS

SELECTORAS / DESTRUCTORAS

PREDICADOS

Para cada operación se especifica su (eventual) precondición y su poscondición, pero NO su implementación.

Especificación de TADs

TADs acotados vs no acotados

- Especificaciones diferentes
- ¿Qué relación hay con estructuras de datos estáticas y dinámicas?

Especificación mínima vs extensiones

- Operaciones adicionales de un TAD:
  - En el mismo módulo o a través de una extensión
  - Implementar accediendo a la representación del TAD o usando las operaciones mínimas

Especificación de TADs

Especificación procedural vs funcional

- Pro y contras
- Criterio de uso de la memoria en las implementaciones

Políticas asociadas a las precondiciones

- Cuándo poner y cuándo no; rol de las precondiciones
- Impactos vinculados con precondiciones (uso, implementación)

¿Por qué varias implementaciones de un TAD?

- Eficiencia: tiempo o espacio
- Claridad, facilidad de uso, reuso, tiempo de desarrollo, facilidad en el mantenimiento
- Criterios anteriores en general contrapuestos:
  - Eficiencia vs otros aspectos
  - Tiempo vs espacio

Ejemplo

Especifique un TAD T de elementos de un tipo genérico que permita almacenar a lo sumo K elementos donde se respeta la política LIFO (el último en entrar es el primero en salir).

Ejemplos de uso del TAD T:

- *Delete/Undelete* en un *file system* o en un procesador de texto;
- registro de archivos recientemente accedidos, modificados en un editor;
- Ctrl z (deshacer).

Especificación del TAD T

(variante de Pila)

```
#ifndef _T_H
#define _T_H

struct RepresentacionT;
typedef RepresentacionT * T;

// CONSTRUCTORAS
T crear (int K);
/* Devuelve la pila vacía, que podrá contener hasta K elementos. */

void apilar (int i, T &p);
/* Inserta i en p. Si estaba llena p antes de la inserción (tenía K
   elementos), elimina el valor ingresado primero (el más antiguo).
*/
```

T = Pila/Cola (LIFO/FIFO)
Acotada o no acotada?

Especificación del TAD T

```
// SELECTORAS
int tope (T p);
/* Devuelve el tope de p (el último ingresado).
   Precondicion: !esVacia(p). */

void desapilar (T &p);
/* Remueve el tope de p (el último ingresado).
   Precondicion: !esVacia(p). */

// PREDICADOS
bool esVacia (T p);
/* Devuelve 'true' si p es vacia, 'false' en otro caso. */

bool esLlena (T p);
/* Devuelve 'true' si p tiene K elementos, donde K es el valor del
   parámetro pasado en crear, 'false' en otro caso. */

// DESTRUCTORA
void destruir (T &p);
/* Libera toda la memoria ocupada por p. */

#endif /* _T_H */
```

¿Sería adecuado agregar una operación para eliminar el elemento más antiguo?

Conclusiones

Algunas ventajas del uso de TADs

Modularidad

Adecuados para sistemas no triviales

Separación entre especificación e implementación. Esto hace al sistema:

- más legible
- más fácil de mantener
- más fácil de verificar y probar que es correcto. Robustez.
- más fácil de reusar
- más extensible
- lo independiza en cierta manera de las distintas implementaciones (complejidad tiempo-espacio)
- Rápida prototipación de sistemas

Todo concluye al fin, nada puede escapar
Todo tiene un final, todo termina…

Gracias... totales!!

Suerte en esta última etapa y
lo mejor para Ustedes en la carrera !!