Programación 2

La Previa:
Estructuras Múltiples
y Diseño de TADs

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

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:

Ejemplo

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

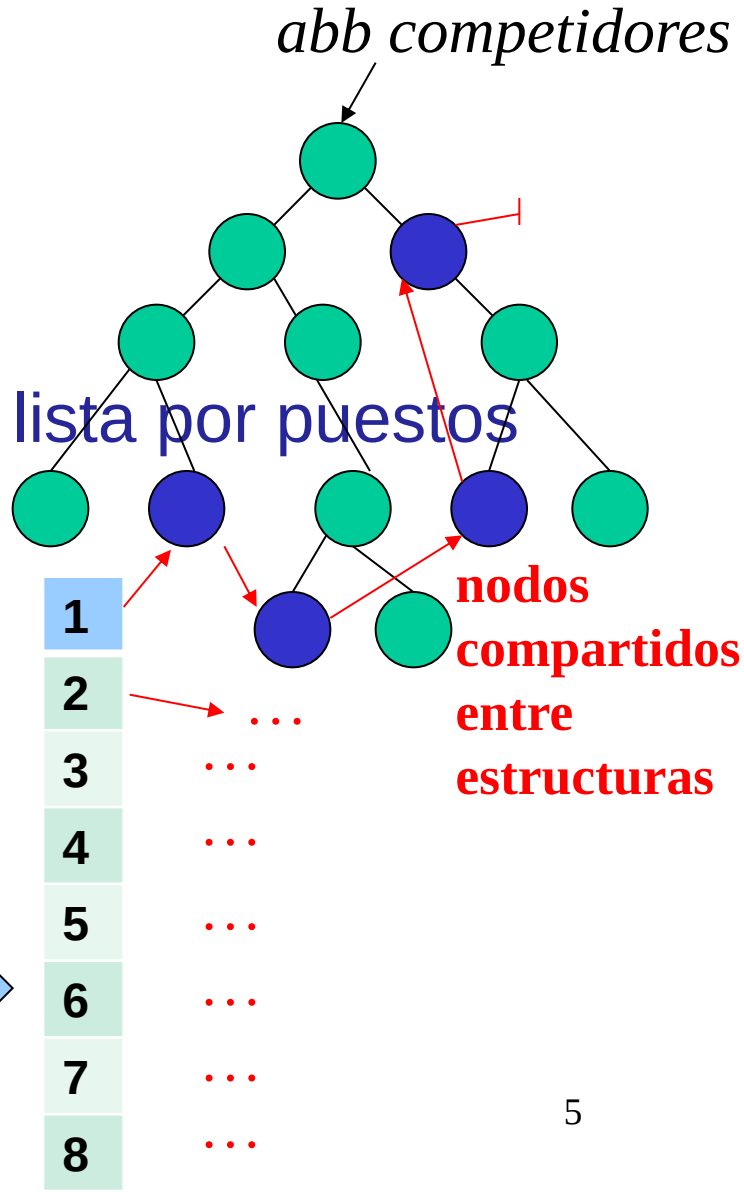
Ejemplo

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[9];
} resultados100m;
```



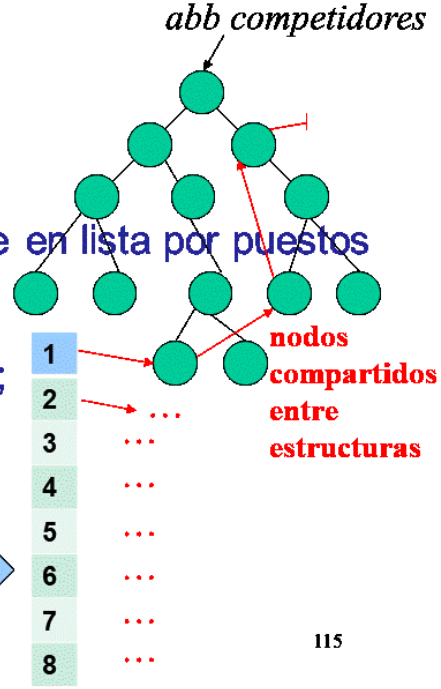
Ejemplo

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[9];
} resultados100m;
```




```
void imprimirPorPuesto (resultados100m * e, int k){
    assert(0<k && k<9);
    nodo * lista = e->puestos[k];
    while (lista!=NULL){
            cout << lista->código;
            lista = lista->sig;
    }
}
```

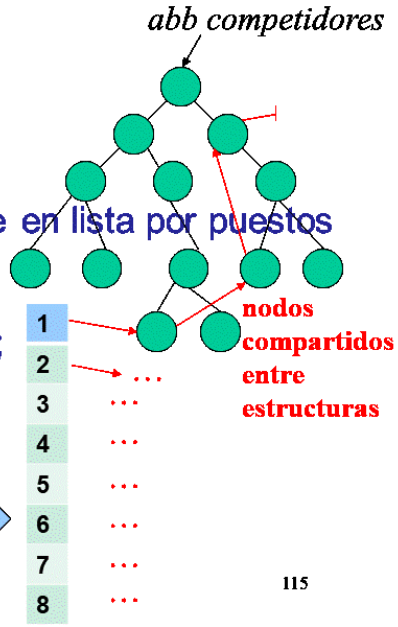
Ejemplo

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[9];
} resultados100m;
```



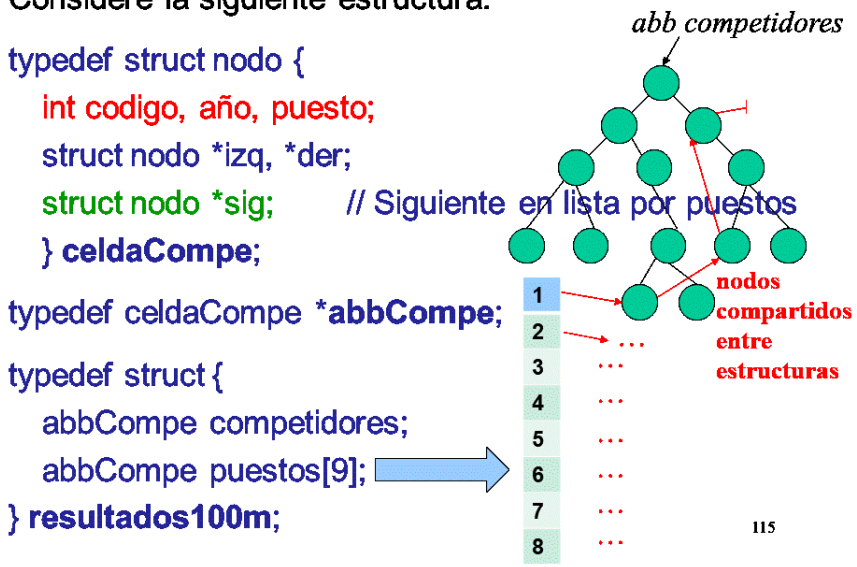

```
void imprimirCompetidores(resultados100m * e){
    imprimirCompetidoresABB(e->competidores);
}
void imprimirCompetidoresABB(nodo * abb){
    if (abb != NULL){
        imprimirCompetidoresABB(abb->izq);
        cout << abb->código;
        imprimirCompetidoresABB(abb->der);
    }
}
```

Ejemplo

```
Nodo * desenganchar NodoPuesto (resultados100m * e, int cod, int k){
    assert(0<k && k<9);
    return eliminarLogico(e->puestos[k], cod);
}
nodo * eliminarLogico (nodo * & l, int cod){
    if (l!=NULL){
        if (l->código == cod){
            nodo * ret = l;
            l =  l->sig;
            return ret;
        }
        else return eliminarLogico (l->sig, cod);
    }
    return NULL;
}
```

Considere la siguiente estructura:

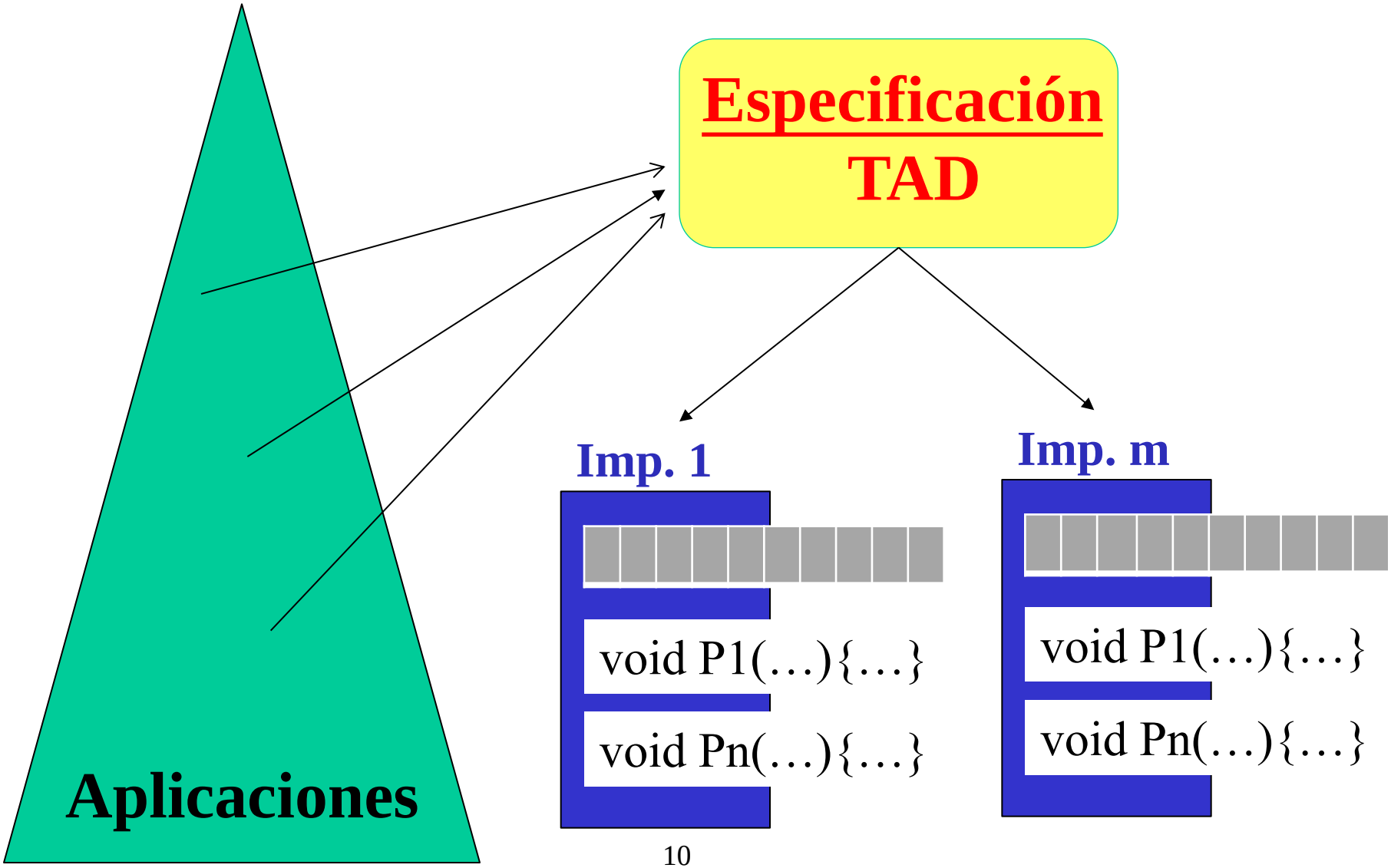
```
typedef struct nodo {
   int codigo, año, puesto;
   struct nodo *izq, *der;
   struct nodo *sig;        // Siguiente en lista por puestos
   } celdaCompe;
typedef celdaCompe *abbCompe;
typedef struct {
   abbCompe competidores;
   abbCompe puestos[9];
} resultados100m;
```



Resolver en base a lo de arriba la funcionalidad que permita actualizar el puesto y año de un competidor ya existente.

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