Punteros y TADs

Algoritmos y Estructuras de Datos II

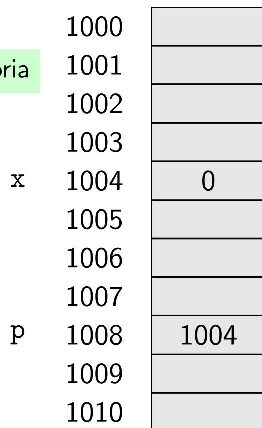
Contenidos

- ✓ Punteros.
- ✓ Estructuras.
- ✓ Memoria Dinámica.
- ✓ Tipos Abstractos de Datos.

Punteros

Son variables que almacenan direcciones de memoria

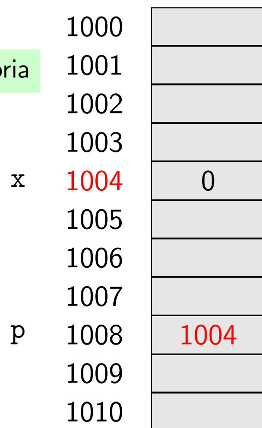
```
int x = 0;
int *p = &x;
```



Punteros

Son variables que almacenan direcciones de memoria

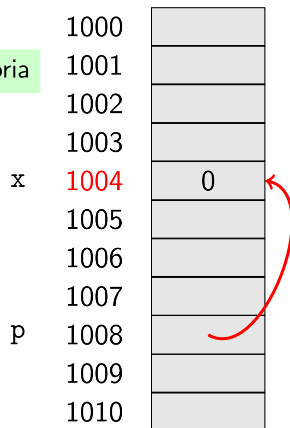
```
int x = 0;
int *p = &x;
```



Punteros

Son variables que almacenan direcciones de memoria

```
int x = 0;
int *p = &x;
```

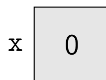


Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```

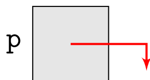
Punteros

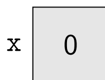
```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



Punteros

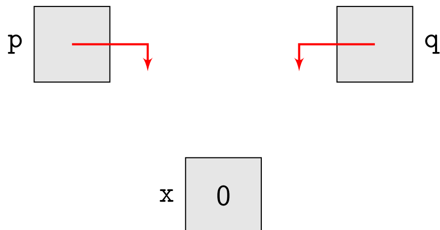
```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```





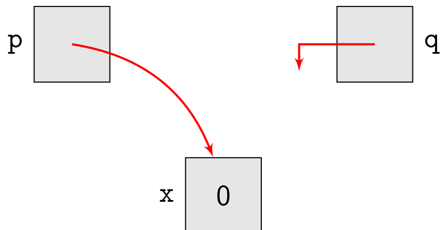
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



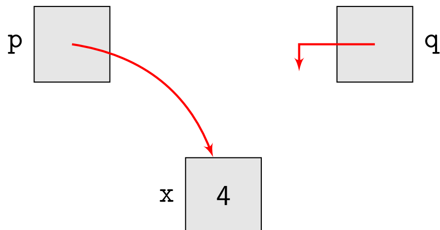
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



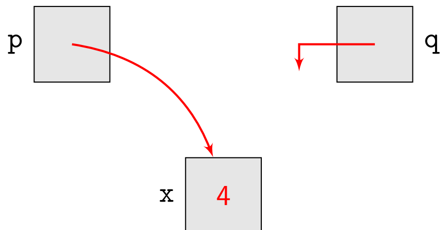
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



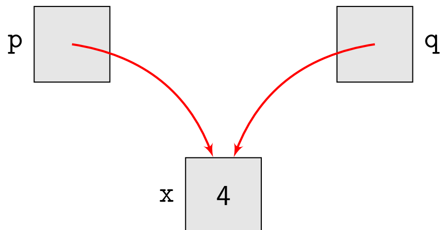
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



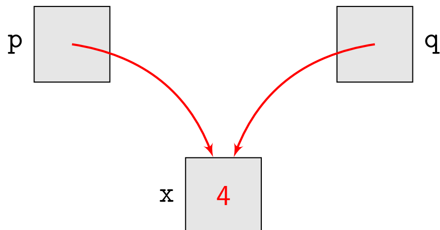
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



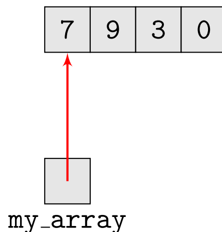
Punteros

```
int main(void) {
    int x = 0 ;
    int *p = NULL;
    int *q = NULL;

    p = &x;
    (*p) = 4;
    printf("%d\n", x);
    q = p;
    printf("%d\n", *q);

    return(0);
}
```



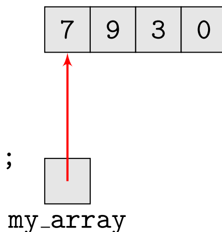
Punteros y Arreglos

```
int main(void) {
    int i = 0;
    int my_array[4] = {7,9,3,0};

    printf("%d\n", my_array[0]);

    for (i = 0; i < 4; i++) {
        printf("%d ", my_array[i]);
    }

    return(0);
}
```



Punteros y Arreglos

```
int main(void) {
    int i = 0;
    int my_array[4] = {7,9,3,0};

    printf("%d\n", *my_array);

    for (i = 0; i < 4; i++) {
        printf("%d ", *(my_array + i));
    }

    /* my_array = NULL error! */

    return(0);
}
```



Notar: `my_array` es un puntero *constante* al primer elemento del arreglo.

Punteros y Parámetros por Referencia

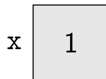
Parámetros por valor

```
int main(void) {
    int x = 1;
    pivote(x);
    printf("%d", x);

    return(0);
}

void pivote(int piv) {
    piv = 5;
}
```

Punteros y Parámetros por Referencia

Parámetros por valor

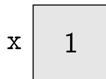
```
int main(void) {
    int x = 1;
    pivote(x);
    printf("%d", x);

    return(0);
}

void pivote(int piv) {
    piv = 5;
}
```

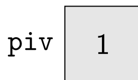


Punteros y Parámetros por Referencia

Parámetros por valor

```
int main(void) {
    int x = 1;
    pivote(x);
    printf("%d", x);

    return(0);
}

void pivote(int piv) {
    piv = 5;
}
```

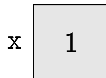


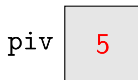


Punteros y Parámetros por Referencia

Parámetros por valor

```
int main(void) {
    int x = 1;
    pivote(x);
    printf("%d", x);

    return(0);
}

void pivote(int piv) {
    piv = 5;
}
```

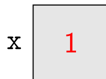




Punteros y Parámetros por Referencia

Parámetros por valor

```
int main(void) {
    int x = 1;
    pivote(x);
    printf("%d", x);

    return(0);
}

void pivote(int piv) {
    piv = 5;
}
```



Punteros y Parámetros por Referencia

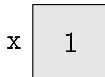
Parámetros por referencia

```
int main(void) {
    int x = 1;
    pivote(&x);
    printf("%d", x);

    return(0);
}

void pivote(int *piv) {
    (*piv) = 5;
}
```

Punteros y Parámetros por Referencia

Parámetros por referencia

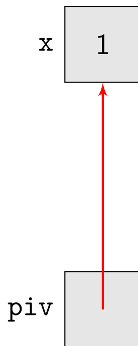
```
int main(void) {
    int x = 1;
    pivote(&x);
    printf("%d", x);

    return(0);
}

void pivote(int *piv) {
    (*piv) = 5;
}
```



Punteros y Parámetros por Referencia

Parámetros por referencia

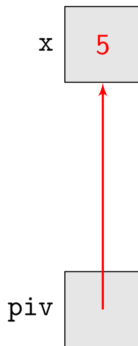
```
int main(void) {
    int x = 1;
    pivote(&x);
    printf("%d", x);

    return(0);
}

void pivote(int *piv) {
    (*piv) = 5;
}
```



Punteros y Parámetros por Referencia

Parámetros por referencia

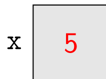
```
int main(void) {
    int x = 1;
    pivote(&x);
    printf("%d", x);

    return(0);
}

void pivote(int *piv) {
    (*piv) = 5;
}
```



Punteros y Parámetros por Referencia

Parámetros por referencia

```
int main(void) {
    int x = 1;
    pivote(&x);
    printf("%d", x);

    return(0);
}

void pivote(int *piv) {
    (*piv) = 5;
}
```



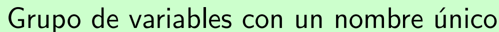
Estructuras

```
struct _point {
    int x;
    int y;
};
```

```
struct _point make_point(int x, int y) {
    struct _point p;
    p.x = x;
    p.y = y;

    return(p);
}
```

Estructuras

```
struct _point {
    int x;
    int y;
};
```




```
struct _point make_point(int x, int y) {
    struct _point p;
    p.x = x;
    p.y = y;

    return(p);
}
```

Estructuras

```
struct _point {
    int x;
    int y;
};
```

Acceso a los miembros de la estructura

```
struct _point make_point(int x, int y) {
    struct _point p;
    p.x = x;
    p.y = y;

    return(p);
}
```

Estructuras

```
struct _point {
    int x;
    int y;
};

typedef struct _point point_t;

point_t make_point(int x, int y) {
    point_t p;
    p.x = x;
    p.y = y;

    return(p);
}
```

Punteros y Estructuras

```
void add_one(point_t p) {
    p.x = p.x + 1;
    p.y = p.y + 1;
}
```

```
int main(void) {
    point_t my_point;
    my_point = make_point(0,0);
    add_one(my_point);
    printf("(%d,%d)", my_point.x, my_point.y);

    return(0);
}
```

Punteros y Estructuras

```
void add_one(point_t p) {
    p.x = p.x + 1;
    p.y = p.y + 1;
}
```

Igual que antes: Sólo cambia la copia local

```
int main(void) {
    point_t my_point;
    my_point = make_point(0,0);
    add_one(my_point);
    printf("(%d,%d)", my_point.x, my_point.y);

    return(0);
}
```

Punteros y Estructuras

```
void add_one(point_t *p) {
    (*p).x = (*p).x + 1;
    (*p).y = (*p).y + 1;
}
```

Usamos punteros y pasamos una referencia

```
int main(void) {
    point_t my_point;
    my_point = make_point(0,0);
    add_one(&my_point);
    printf("(%d,%d)", my_point.x, my_point.y);

    return(0);
}
```

Punteros y Estructuras

```
void add_one(point_t *p) {
    p->x = (p->x) + 1;
    p->y = (p->y) + 1;
}
```

Notación alternativa: `p->x == (*p).x`

```
int main(void) {
    point_t my_point;
    my_point = make_point(0,0);
    add_one(&my_point);
    printf("(%d,%d)", my_point.x, my_point.y);

    return(0);
}
```

Memoria Dinámica

Hasta ahora hemos usado memoria estática

```
int main(void) {
    char names[1000][100];
    ⋮
}
```

Reservada en tiempo de compilación

Liberada automáticamente al final de la ejecución

Memoria Dinámica

Hasta ahora hemos usado memoria estática

```
int main(void) {
    char names[1000][100];
    ⋮
}
```

¿Y si queremos reservar memoria por demanda?

Memoria Dinámica

Hasta ahora hemos usado memoria estática

```
int main(void) {
    char names[1000][100];
    ⋮
}
```

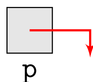
¿Y si queremos liberarla cuando ya no haga falta?

Memoria Dinámica

```
int main(void) {
    int i = 0;
    int *p = NULL;
    p = calloc(5, sizeof(int));
    if (p != NULL) {
        for (i = 0; i < 5 ; i++) {
            printf("%d ", p[i]);
        }
        free(p);
        p = NULL;
    }
    return(0);
}
```

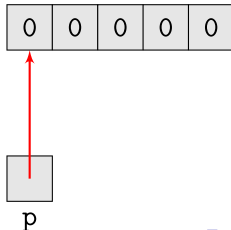
Memoria Dinámica

```
int main(void) {
    int i = 0;
    int *p = NULL;
    p = calloc(5, sizeof(int));
    if (p != NULL) {
        for (i = 0; i < 5 ; i++) {
            printf("%d ", p[i]);
        }
        free(p);
        p = NULL;
    }
    return(0);
}
```



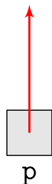
Memoria Dinámica

```
int main(void) {
    int i = 0;
    int *p = NULL;
    p = calloc(5, sizeof(int));
    if (p != NULL) {
        for (i = 0; i < 5 ; i++) {
            printf("%d ", p[i]);
        }
        free(p);
        p = NULL;
    }
    return(0);
}
```



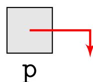
Memoria Dinámica

```
int main(void) {
    int i = 0;
    int *p = NULL;
    p = calloc(5, sizeof(int));
    if (p != NULL) {
        for (i = 0; i < 5 ; i++) {
            printf("%d ", p[i]);
        }
        free(p);
        p = NULL;
    }
    return(0);
}
```



Memoria Dinámica

```
int main(void) {
    int i = 0;
    int *p = NULL;
    p = calloc(5, sizeof(int));
    if (p != NULL) {
        for (i = 0; i < 5 ; i++) {
            printf("%d ", p[i]);
        }
        free(p);
        p = NULL;
    }
    return(0);
}
```



Memoria Dinámica

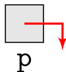
```
struct _info {
    char *name;
    int age;
};

typedef struct _info *info_t;
```

Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```

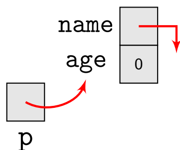
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



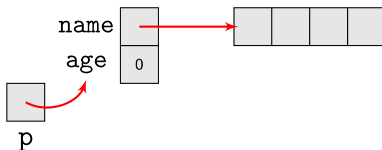
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



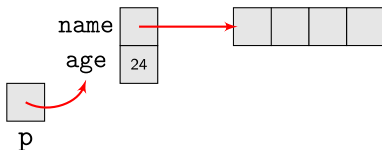
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



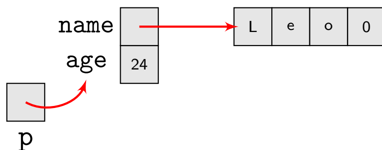
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



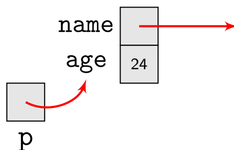
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



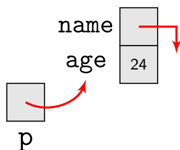
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



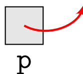
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



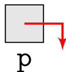
Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



Memoria Dinámica

```
int main(void) {
    info_t p = NULL;
    p = calloc(1, sizeof(struct _info));
    assert(p != NULL);
    p->name = malloc(4 * sizeof(char));
    assert(p->name != NULL);
    p->age = 24;
    p->name = strncpy(p->name, "Leo", 4);

    free(p->name);
    p->name = NULL;
    free(p);
    p = NULL;

    return(0);
}
```



Verificando el uso de la memoria

```
int main(void) {
    int *p = malloc(sizeof(int));

    return(0);
}
```

```
>$ valgrind --leak-check=full --show-reachable=yes ./mi_programa
```

```
==17387== LEAK SUMMARY:
==17387==    definitely lost: 4 bytes in 1 blocks
==17387==    indirectly lost: 0 bytes in 0 blocks
==17387==      possibly lost: 0 bytes in 0 blocks
==17387==    still reachable: 0 bytes in 0 blocks
==17387==         suppressed: 0 bytes in 0 blocks
==17387==
==17387== ERROR SUMMARY: 1 errors
```

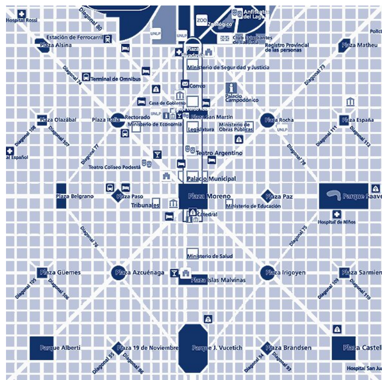
Tipos Abstractos de Datos (TADs)

- *Tipos Concretos de Datos*
  - Asociados a un lenguaje de programación.
  - En C:
    - Tipos Simples: **int**, **char**, **float**,etc.
    - Tipos Compuestos: **int ***, **int[]**, **struct**,etc.
- *Tipos Abstractos de Datos*
  - NO está asociados a un lenguaje de programación.
  - Surgen a partir de un problema que se intenta resolver.
  - Independientes de toda posible implementación.
  - Ejemplos de TADs: grafo, pila, cola, árbol binario, conjunto, etc.

Tipos Abstractos de Datos (TADs)

Ejemplo

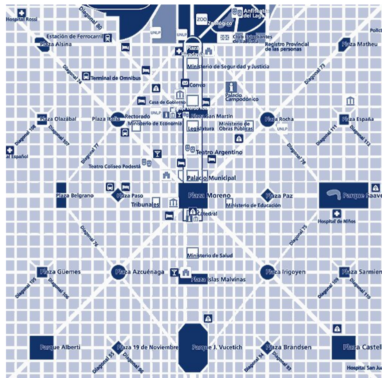
- ✓ Problema: Encontrar el camino más corto entre dos puntos de una ciudad.
- ✓ Luego de pensarlo un rato: Se me ocurre que puedo pensar un grafo.
- Nodos:
  - Los cruces de rutas.
  - Los puntos distinguidos (hospitales, escuelas, etc)
- Las aristas son los segmentos de calles entre dos nodos.



Tipos Abstractos de Datos (TADs)

Ejemplo

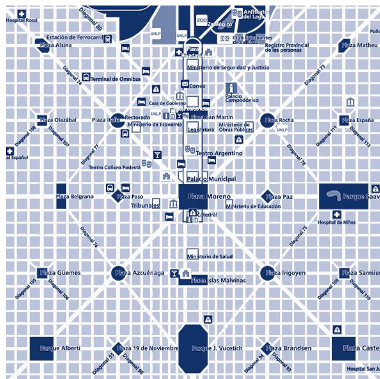
✓ Podriamos tener diferentes operaciones sobre grafo:
  - Agregar un nodo al grafo.
  - Agregar un arista al grafo.
  - Verificar si es conexo.
  - Calcular el camino más corto.



Tipos Abstractos de Datos (TADs)

Ejemplo

- ✓ Separamos el problema de la implementación.
  - Mayor entendimiento del problema.
- ✓ Reutilización: El grafo sirve para muchas otras cosas.



Ejemplo: Conjunto Finito

TAD Set [M]

Constructores

$empty : Set$
$ins : E \times Set \rightarrow Set$

Operaciones

$is_empty : Set \rightarrow Bool$
$is_full : Set \rightarrow Bool$
$belongs : E \times Set \rightarrow Bool$
$len : Set \rightarrow \mathbb{N}$
$del : E \times Set \rightarrow Set$
$join : Set \times Set \rightarrow Set$

Ejemplo: Conjunto Finito

TAD Set [M]

Ecuaciones

1. $len(s) <= M$
2. $belongs\,(x, empty) = false$
3. $belongs\,(x, ins(y, s)) = (x = y) \vee belongs\,(x, s)$
4. $belongs\,(x, s) \vee len(s) = M \Rightarrow ins(x, s) = s$
5. $len(empty) = 0$
6. $belongs\,(x, s) \Rightarrow len(ins(x, s)) = len(s)$
7. $\neg belongs\,(x, s) \;\Rightarrow len(ins(x, s)) = 1 + len(s)$
8. ... (varias más)

TADs en C

- Especificación de las operaciones del TAD: `set.h`
- Implementación: `set.c`

TADs en C

- Especificación de las operaciones del TAD: `set.h`
- Implementación: `set.c`

Los detalles de implementación permanecen ocultos

Especificación en C: set.h

```
#ifndef _SET_H
#define _SET_H

#include "elem.h"

typedef struct _set *set_t;

set_t set_empty(int max_size);

bool set_belongs(const set_t s, elem e);

⋮

#endif
```

Especificación en C: set.h

```
#ifndef _SET_H
#define _SET_H

#include "elem.h"

typedef struct _set *set_t;

set_t set_empty(int max_size);

bool set_belongs(const set_t s, elem e);

⋮

#endif
```

La definición de la estructura NO va acá

Implementación en C: set.c

```
struct _set {
    elem *e;
    int size;
    int max_size;
};
```

Implementación en C: set.c

```
set_t set_empty(int max_size) {
    set_t s = NULL;
    assert(max_size >= 0);
    s = calloc(1, sizeof(struct _set));
    if (s != NULL) {
        s->e = calloc(max_size, sizeof(elem));
        s->size = 0;
        s->max_size = max_size;
        if (s->e == NULL) {
            free(s);
            s = NULL;
        }
    }

    return(s);
}
```

Implementación en C: set.c

```
set_t set_empty(int max_size) {
    set_t s = NULL;
    assert(max_size >= 0);
    s = calloc(1, sizeof(struct _set));
    if (s != NULL) {
        s->e = calloc(max_size, sizeof(elem));
        s->size = 0;
        s->max_size = max_size;
        if (s->e == NULL) {
            free(s);
            s = NULL;
        }
    }

    return(s);
}
```

Si no hay memoria, devuelve NULL

Implementación en C: set.c

```
bool set_belongs(const set_t s, elem e) {
    bool found = false;
    int i = 0;
    assert(s != NULL);
    while (i < s->size && !found) {
        found = elem_equals(s->e[i], e);
        i++;
    }

    return(found);
}
```

Implementación en C: set.c

```
bool set_belongs(const set_t s, elem e) {
    bool found = false;
    int i = 0;
    assert(s != NULL);
    while (i < s->size && !found) {
        found = elem_equals(s->e[i], e);
        i++;
    }

    return(found);
}
```

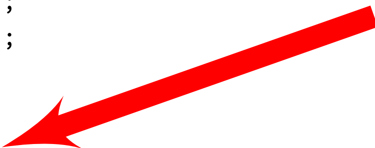
Necesitamos comparar elementos

Usando el TAD: main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "set.h"

int main(void) {
    set_t s = set_empty(3);

    set_ins(s, 1);
    set_ins(s, 1);
    set_print(s);

    s->size = 0;
    s = set_destroy(s);
    return(0);
}
```

Usando el TAD: main.c

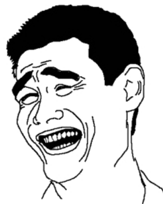
```
#include <stdio.h>
#include <stdlib.h>
#include "set.h"

int main(void) {
    set_t s = set_empty(3);

    set_ins(s, 1);
    set_ins(s, 1);
    set_print(s);

    s->size = 0;
    s = set_destroy(s);
    return(0);
}
```



Usando el TAD: main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "set.h"

int main(void) {
    set_t s = set_empty(3);

    set_ins(s, 1);
    set_ins(s, 1);
    set_print(s);

    s->size = 0;
    s = set_destroy(s);
    return(0);
}
```

NO COMPILA!



Los miembros de la estructura son privados (ocultos!)

¿Preguntas?

HAY TABLA

Si no comentan el código, hay tabla.
Si no identan el código, hay tabla.
Si no inicializan las variables, hay tabla.
Si no compilan con todos los flags, hay tabla.
Si tienen memory leaks, hay tabla.