CAPITULO 4
TIPOS DE ABSTRACTOS DE DATOS (TAD`S)

Todo tipo de datos simples se pueden definir como un conjunto de **valores** y un conjunto de **operaciones** sobre tales valores. Este conjunto y estas operaciones forman una estructura matemática que se implementa en un lenguaje y en el computador como un **tipo**.

Ejemplo:

Z	⟶	En C, int {-65535 .....65536}
+		+
-		-
*		*
/		/ y %

Esta misma concepción se lleva a los tipos estructurados, los cuales se que pueden definir en base a los tipos simples. De esta manera abstraemos la definición matemática de tipo a estructuras mas complejas.

Ejemplo:

Queremos definir un nuevo tipo llamado COMPLEJOS, que contenga los valores de los números complejos X + Y i con X, Y Є R. Las operaciones serían: + (suma), - (diferencia), * (producto), / (división).

En general, pocos lenguajes de programación proveen herramientas para construir TAD'S. Son los lenguajes orientados a objetos. En lenguajes como C, el diseño de programas a través de TADs es una técnica de programación.

3.1 Características de los TAD'S

- Es la manera de diseñar e implementar el ocultamiento de información.
- Nos permite concentrarnos en los objetos del problema y no en los procesos.
- Los objetos pueden ser diseñados independientes del problema.
- El problema se resuelve con la interacción de los diferentes TADs que lo componen.
- Nos olvidamos, al diseñar, de la eficiencia del espacio o del tiempo. Esto son aspectos se consideran en la implementación.

Un TAD consta de dos partes: definición de conjunto de valores, la definición de los operadores.

Para diseñar e implementar los TADs seguiremos las siguiente metodología:

1. Conceptualización del TAD, en este paso se intenta dar un modelo de los componentes (objetos) del TAD.
2. Conceptualización de las operaciones del TAD. Se define cada una de la operaciones que su usarán para manipular el TAD a través de notación funcional, así como, las pre y post condiciones. Aquí se destacan cuáles son las operaciones primitivas y cuáles operaciones se implementan en base a las primitivas.

3. Implementación de la estructura de datos. Se define los typedef correspondientes al nuevo TAD
4. Implementación de las operaciones

Ejemplo 1: Diseño del TAD RACIONAL

1. **Conceptualización del TAD**

RACIONAL: $\frac{X}{Y}$ donde X,Y Є R (reales) y Y ≠ 0

2. **Conceptualización de las operaciones**.

Veremos primero operaciones que no son primitivas:

Adición: $\frac{X1}{Y1} + \frac{X2}{Y2} = \frac{X1*Y2 + X2*Y1}{Y1 * Y2}$

Multiplicación: $\frac{X1}{Y1} * \frac{X2}{Y2} = \frac{X1 * X2}{Y1 * Y2}$

Igualdad : $\frac{X1}{Y1} = \frac{X2}{Y2} = (X1 * Y2 = X2 * Y1)$

Estas operaciones se conceptualizan con la notación funcional:

```
R_Adición: RACIONAL x RACIONAL  →  RACIONAL
/* PRE: Dados dos racionales
   POST: Devuelve la suma de los dos racionales */
R_Multiplicación: RACIONAL x RACIONAL  →  RACIONAL
/* PRE: Dados dos racionales
   POST: Devuelve la multiplicación de dos racionales */
R_Igualdad: RACIONAL x RACIONAL  →  bool
/* PRE: Dados dos racionales
   POST: Dice si dos racionales son iguales */
```

Ahora hablaremos de las primitivas del TAD.

Todo TAD contiene 4 clases de primitivas: un constructor, un destructor, los modificadores y los observadores. Veamos las primitivas del TAD RACIONAL.

a) **constructor**: permite crear un objeto del TAD. Debemos pensar que información es necesaria al momento de crear un objeto. En el caso del racional, se necesitan el numerador y el denominador.

```
R_Crear:  int x int → RACIONAL
/* PRE: Dados dos enteros x,y con y≠0
   POST: Crea un racional con numerador x, denominador y */
```

b) **destructor**: permite liberar el espacio de memoria ocupado por un objeto del TAD. En este caso no hay un valor de retorno lo cual se indica usando la palabra `void`.

```
R_Destruir:  RACIONAL → void
/* PRE: Dado un racional
   POST: Libera el espacio ocupado por el racional */
```

c) **observadores**: permiten visualizar la información interna del TAD. Se necesita un observador para cada campo del objeto que se necesita accesar. En el caso del racional, se necesitan visualizar tanto el numerador y como el denominador.

```
R_Num:  RACIONAL → int
/* PRE: Dado un racional
   POST: Devuelve el numerador de dicho racional */

R_Den:  RACIONAL → int
/* PRE: Dado un racional
   POST: Devuelve el denominador de dicho racional */
```

d) **modificadores**: permiten modificar la información interna del TAD. Se necesita un modificador para cada campo del objeto que se necesite alterar. En el caso del racional, se necesita poder modificar tanto el numerador y como el denominador.

```
R_AsignarNum:  RACIONAL x int → RACIONAL
/* PRE: Dado un racional y un entero z
   POST: Devuelve el racional modificado con z como
         numerador */

R_AsignarDen:  RACIONAL x int → RACIONAL
/* PRE: Dado un racional y un entero z≠0
   POST: Devuelve el racional modificado con z como
         denominador */
```

3. **Implementación de la estructura del TAD**.

La implementación del TAD puede ser en cualquier lenguaje y con cualquier estructura de datos que no modifique la especificación.

Observemos que en la definición de las operaciones del TAD RACIONAL, varias de ellas devuelven un racional. En el lenguaje C, no es posible devolver una estructura por lo cual usaremos un apuntador a la estructura para poder retornar dicho apuntador.

La implementación del TAD RACIONAL en C sería:

```
typedef struct  s_rac  {
   int num;
   int den;
} STRUCTRAC;
typedef STRUCTRAC *RACIONAL;
```

Note que se declara primero la estructura (STRUCTRAC) y luego el apuntador a dicha estructura (RACIONAL) el cual es la implementación del TAD.

3. **Implementación de las operaciones del TAD**.

Las operaciones que no son primitivas se implementan en base a las primitivas. Por esta razón escribiremos primero los prototipos de las primitivas, luego, las operaciones adición, multiplicación e igualdad implementadas en base a las primitivas y finalmente, la implementación de las primitivas.

Los prototipos de las operaciones primitivas serían:

```
RACIONAL R_Crear (int x, int y);
/* PRE: y≠0
   POST: R_Crear = x/y */

void R_Destruir (RACIONAL R);
/* PRE: ninguna
   POST: Libera el espacio ocupado por el racional */

int R_Num(RACIONAL R)
/* PRE: ninguna
   POST: Devuelve el numerador del racional dado */

int R_Den(RACIONAL R)
/* PRE: ninguna
   POST: Devuelve el denominador del racional dado */

RACIONAL R_AsignarNum(RACIONAL R, int z)
/* PRE: ninguna
   POST: Devuelve el racional modificado con z como
         numerador */

RACIONAL R_AsignarDen(RACIONAL R, int z)
/* PRE: z≠0
   POST: Devuelve el racional modificado con z como
         denominador */
```

Los prototipos se colocan en el archivo Racional.h junto con los typedef de la estructura.

Ahora implementemos las operaciones adición, multiplicación e igualdad usando estos prototipos.

```
RACIONAL R_adición ( RACIONAL R1, RACIONAL R2)
/* PRE: R1 = X1/Y1 y R2=X2/Y2
   POST: R_adición = X1  + X2  = X1*Y2 + X2*Y1
                     Y1    Y2       Y1 * Y2
*/
{
   return( R_Crear(R_Num(R1)*R_Den(R2)+ R_Num(R2)*R_Den(R1),
                   R_Den(R1)*R_Den(R2) );
}

RACIONAL R_multiplicación ( RACIONAL R1, RACIONAL R2)
/* PRE: R1 = X1/Y1 y R2=X2/Y2
   POST: R_adición = X1  * X2  =  X1 * X2
                     Y1    Y2     Y1 * Y2
*/
{
   return( R_Crear(R_Num(R1)*R_Num(R2),
                   R_Den(R1)*R_Den(R2) );
}

bool R_igualdad( RACIONAL R1, RACIONAL R2)
/* PRE: R1 = X1/Y1 y R2=X2/Y2
   POST: R_igualdad=(X1*Y2 = X2*Y1 )
*/
{
   return( R_Num(R1)*R_Den(R2) == R_Num(R2)*R_Den(R1));,
}
```

La implementación de estas operaciones queda transparente de la estructura de datos del TAD. La estructura sólo es accesible por las primitivas.

Veamos la implementación de las primitivas:

```
RACIONAL R_Crear (int x, int y)
/* PRE: y≠0
   POST: R_Crear = x
                   y */
{
     RACIONAL R;

     if ((R=(RACIONAL)malloc(sizeof(STRUCTRAC)))!=NULL)}
     {
          R->num = x;
          R->den = y;
     }
     return(R);
}
```

```
void R_Destruir (RACIONAL R)
/* PRE: ninguna
   POST: Libera el espacio ocupado por el racional */
{
     free(R);
}

int R_Num(RACIONAL R)
/* PRE: ninguna
   POST: Devuelve el numerador del racional dado */
{
     return(R->num);
}

int R_Den(RACIONAL R)
/* PRE: ninguna
   POST: Devuelve el denominador del racional dado */
{
     return(R->den);
}


RACIONAL R_AsignarNum(RACIONAL R, int z)
/* PRE: ninguna
   POST: Devuelve el racional modificado con z como
         numerador */
{
     R->num = z;
     return(R);
}


RACIONAL R_AsignarDen(RACIONAL R, int z)
/* PRE: z≠0
   POST: Devuelve el racional modificado con z como
    denominador */
{
     R->den = z;
     return(R);
}
```

Note que las primitivas accesan los campos del TAD a través del apuntador, por eso es necesario asignar memoria con el constructor y liberarla en el destructor.

Ejemplo 2: Diseño del TAD CONJUNTO

1. **Conceptualización del TAD**

CONJUNTO : $\{e_1, e_2 \ldots , e_n\}$ donde $e_i \in$ int y n es la cardinalidad del conjunto

2. **Conceptualización de las operaciones**.

a) **constructor**:

```
C_Crear:  void → CONJUNTO
/* PRE: ninguna
   POST: Crea un conjunto vacío */
```

b) **destructor**:

```
C_Destruir:  CONJUNTO → void
/* PRE: Dado un conjunto
   POST: Libera el espacio ocupado por el conjunto */
```

c) **Analizadoras**:

```
C_Vacio:  CONJUNTO → bool
/* PRE: Dado un conjunto
   POST: Dice si el conjunto es vacío */

C_Pertenece:  CONJUNTO x int → bool
/* PRE: Dado un conjunto y un elemento
   POST: Dice si el elemento está en el conjunto */

C_Elemento:  CONJUNTO x int → int
/* PRE: Dado un conjunto no vacío y un entero i
   POST: Devuelve el elemento i del conjunto */

C_Cardinalidad:  CONJUNTO → int
/* PRE: Dado un conjunto
   POST: Devuelve la cardinalidad del conjunto */
```

d) **modificadoras**:

```
C_InsertElem:  CONJUNTO x int → CONJUNTO
/* PRE: Dado un conjunto y un elemento
   POST: Devuelve el conjunto con el elemento insertado
         Si ya existe no lo inserta */

C_ElimElem:  CONJUNTO x int → CONJUNTO
/* PRE: Dado un conjunto y un elemento
   POST: Devuelve el conjunto sin el elemento dado.
         Si el elemento no pertenece al conjunto no hace
         nada */
```

e) **otras operaciones**:

```
C_Unión: CONJUNTO x CONJUNTO  →  CONJUNTO
/* PRE: Dados dos conjuntos
   POST: Devuelve la unión de ellos */
```

```
C_Intersección: CONJUNTO x CONJUNTO  →  CONJUNTO
/* PRE: Dados dos conjuntos
   POST: Devuelve la intersección de ellos */

C_Iguales: CONJUNTO x CONJUNTO  →  bool
/* PRE: Dados dos conjuntos
   POST: Dice si los dos conjuntos son iguales */
```

3. **Implementación de la estructura del TAD**.

```
#define MAX 100
typedef struct  s_cjto  {
   int card;
   int elementos[MAX];
} STRUCTCJTO;
typedef STRUCTCJTO *CONJUNTO;
```

3. **Implementación de las operaciones del TAD**.

De las primitivas:

```
CONJUNTO C_Crear ()
/* PRE: ninguna
   POST: C_Crear = {} conjunto vacio */
{
     CONJUNTO C;

     if ((C=(CONJUNTO)malloc(sizeof(STRUCTCJTO)))!=NULL)}
     {
          C->card = 0;
          for (int i=0; i<MAX; i++)
          {  C->elementos[i] = 0; }
     }
     return(C);
}

void C_Destruir (CONJUNTO C)
/* PRE: ninguna
   POST: Libera el espacio ocupado por el conjunto */
{
     free(C);
}
```

```
bool C_Vacio(CONJUNTO C)
/* PRE: Conjunto Creado
   POST: Dice si el conjunto está vacío */
{
     return(C->card == 0);
}

bool C_Pertenece(CONJUNTO C, int e)
/* PRE: Conjunto Creado
   POST: Dice si e pertenece a C*/
{
     for (int i=0; i<C->card; i++)
     {
        if (C->elementos[i] == e)
          return(TRUE);
     }
     return(FALSE);
}

int C_Elemento(CONJUNTO C,int i)
/* PRE: Conjunto Creado y no vacío, 0<=i<C_Cardinalidad(C)
   POST: Devuelve el elemento i del conjunto*/
{
     return(C->elemento[i]));
}

int C_Cardinalidad(CONJUNTO C)
/* PRE: Conjunto Creado
   POST: Devuelve la cardinalidad del conjunto dado */
{
     return(C->card);
}

CONJUNTO C_InsertElem(CONJUNTO C, int e)
/* PRE: Conjunto creado
   POST: Devuelve el conjunto con el elemento e insertado*/
{
     if (!C_Pertenece(C,e))
     {
        C->elementos[C->card] = e;
        C->card++;
     }
     return(C);
}
```

```
CONJUNTO C_ElimElem(CONJUNTO C, int e)
/* PRE: Conjunto creado
   POST: Devuelve el conjunto eliminando el elemento e
         Si no existe no hace nada*/
{
     int i=0;

     while (i<C->Card)
     {
        if (C->elementos[i] == e)
        {
           C->card--; /* Encontre el elemento decremento
                         la cardinalidad */
           break;
        }
        i++;
     }
     /* Elimina el elemento moviendo todos a la posición
        anterior */
     while (i<C->Card)
        C->elemento[i] = C->elemento[i+1];
     return(C);
}
```

```
CONJUNTO C_unión( CONJUNTO C1, CONJUNTO C2)
/* PRE: Dados dos conjuntos creados
   POST: C_unión = C1 ∪ C2 , C1,C2 quedan destruidos */
{
   CONJUNTO U;
   int e;

   U= C_Crear();

   while (!C_Vacio(C1))
   {
      e = C_elemento(C1);
      U = C_InsertElem(U,e);
      C1 = C_ElimElem(C1,e);
   }

   while (!C_Vacio(C2))
   {
      e = C_elemento(C2);
      U = C_InsertElem(U,e);
      C2 = C_ElimElem(C2,e);
   }

   return(U);
}
```

```
CONJUNTO C_intersección(CONJUNTO C1, CONJUNTO C2)
/* PRE: Dados dos conjuntos creados
   POST: C_intersección = C1 ∩ C2 , C1 queda destruido */
{
   CONJUNTO I;
   int e;

   I= C_Crear();

   while (!C_Vacio(C1))
   {
      e = C_elemento(C1);
      if (C_Pertenece (C2,e))
         I = C_InsertElem(I,e);
      C1 = C_ElimElem(C1,e);
   }

   return(I);
}

bool C_Iguales(CONJUNTO C1, CONJUNTO C2)
/* PRE: Dados dos conjuntos creados
   POST: C_Iguales=(C1 =C2 ), C1 queda destruido
*/
{
   if (C1->card != C2->card)
      return(FALSE);

   while (!C_Vacio(C1))
   {
      e = C_elemento(C1);
      if (!C_Pertenece(C2,e))
         return(FALSE);
      C1 = C_ElimElem(C1,e);
   }

   return(TRUE);,
}
```