Análisis de Algoritmos y Estructuras de Datos

Tema 4: Tipos Abstractos de Datos

Mª Teresa García Horcajadas José Fidel Argudo Argudo
Antonio García Domínguez Francisco Palomo Lozano

UCA
Universidad de Cádiz

Versión 2.1

Índice

Abstracción

Definición

Capacidad intelectual para comprender fenómenos complejos, prescindiendo de los detalles irrelevantes y resaltando los importantes.

Modos de abstraer una realidad

1. Aplicando distintos puntos de vista: Aspectos superfluos en una situación pueden ser de suma importancia en otra.
2. Con mayor o menor intensidad: Nivel de abstracción más alto cuantos más detalles se ignoren.

Abstracción

La abstracción en programación

1. Mediante la abstracción generamos un modelo conceptual o abstracto del problema a resolver.
2. Los lenguajes de programación proporcionan los medios para implementar los modelos abstractos de los problemas: abstracción operacional y abstracción de datos.

La abstracción en programación

Abstracción operacional

1. Definición de funciones para crear nuevas operaciones a partir de las propias del lenguaje.
2. Utilización de funciones sin conocer cómo están implementadas. Interesa qué hacen y se ignora cómo lo hacen: Ocultación de información
3. Extiende la máquina virtual definida por el lenguaje de programación añadiendo nuevas operaciones.

La abstracción en programación

Abstracción de datos

1. Definición de una colección de datos con las mismas características y especificación de sus operaciones y propiedades. Tipo Abstracto de Datos (TAD)
2. Utilización de los nuevos tipos y sus operaciones según la especificación. Interesa qué son y qué operaciones admiten y no cómo se almacenan ni cómo se procesan. Ocultación de información
3. Extiende la máquina virtual definida por el lenguaje de programación añadiendo nuevos tipos de datos.

Ocultación de información

1. El uso de la abstracción provoca ocultación de información.
2. Mediante la abstracción operacional, el usuario de una función sólo necesita conocer su especificación (qué hace), pero no los detalles internos de su implementación (cómo lo hace).
3. En la abstracción de datos, la ocultación de información permite separar la interfaz de un TAD (características y comportamiento visibles desde el exterior) de los detalles internos de la implementación (invisibles desde el exterior).
4. La interfaz de un TAD es un «contrato»de servicios con el usuario.
5. La ocultación de información disminuye las interdependencias entre los componentes de un sistema, es decir, reduce el acoplamiento entre ellos.

Independencia de la representación

1. La abstracción de datos se desarrolla en dos niveles:
   - Nivel conceptual o de especificación: Definición del dominio (colección de datos) y de la interfaz (operaciones y sus propiedades) del TAD.
   - Nivel de representación o implementación: Definición de la estructura de datos que soporta al TAD e implementación de las operaciones de forma que exhiban las propiedades y comportamiento especificados.
2. La especificación de un TAD es independiente de cualquier representación del mismo que se pueda diseñar.
3. El uso de un TAD se basa en su especificación y, por tanto, también es independiente de la representación subyacente.
4. La independencia de la representación implica:
   - Un mismo TAD se puede implementar con distintas representaciones.
   - Los requisitos del problema o aplicación determinarán la representación más adecuada a utilizar.

Genericidad

1. Un tipo abstracto genérico es el que define una familia de tipos abstractos con un comportamiento común, pero que se diferencian en detalles que no afectan a dicho comportamiento.
2. La abstracción de datos nos permite crear un TAD cuando surge la necesidad al resolver un problema concreto; no obstante, si es posible, conviene generalizar la definición de este TAD, con vistas a que el mismo u otro de la familia puedan servir más adelante en circustancias similares.
3. La generalización se emplea especialmente en la creación de tipos abstractos contenedores.
4. Un contenedor es un TAD genérico diseñado para almacenar una colección de elementos del mismo tipo.
5. Pilas, colas y listas son ejemplos de contenedores.

Descomposición modular

1. La descomposición de un problema en subproblemas ayuda a tratar su complejidad y conduce a dividir el programa que lo resuelve en componentes llamados módulos.
2. Una buena descomposición del programa da lugar a módulos con características funcionales independientes, lo cual aumenta la cohesión interna de los módulos y favorece su reutilización.
3. Módulo: Unidad de organización de un programa que engloba un conjunto de entidades (encapsulamiento), tales como datos, tipos, instrucciones o funciones, y que controla lo que se puede ver y utilizar desde otros módulos.
4. Ejemplos de módulos:
   - Una función encapsula un conjunto de declaraciones e instrucciones para realizar un proceso.
   - Un conjunto de tipos, datos y funciones que forman parte de un programa se pueden encapsular dentro de un módulo que se almacena en un fichero.

Encapsulamiento de TAD

1. Un TAD se puede encapsular en un tipo de módulo llamado clase, propio de los lenguajes orientados a objetos.
2. Una clase permite encapsular datos y operaciones y proporciona un mecanismo para
   (a) ocultar en su interior los detalles de implementación y
   (b) mostrar las operaciones de uso externo a través de su interfaz.

Encapsulamiento vs. ocultación de información

En el contexto de los TAD, estos dos conceptos están muy relacionados, pero no se deben confundir:

- Encapsulamiento: acción y efecto de agrupar dentro de un módulo (clase) los datos y operaciones que forman un TAD.
- Ocultación de información: acción y efecto de mantener en secreto aquellos aspectos de un TAD que no se necesitan conocer para utilizarlo en un programa.

Especificación de un TAD

¿Qué es?

La especificación comprende toda la información que necesita conocer el usuario del TAD:

1. Definición del dominio o conjunto de datos del TAD.
2. Especificación de operaciones
   - Especificación sintáctica: Indica cómo usar las operaciones, es decir, la forma en que se ha de escribir cada una, dando su nombre junto al orden y tipo de operandos y resultado.
   - Especificación semántica: Expresa el significado de las operaciones, o sea, qué hace y qué propiedades tiene cada una.

Especificación de un TAD como contrato usuario-diseñador

El usuario puede utilizar el TAD bajo las condiciones de la especificación y el diseñador está obligado a implementarlo de modo que asegure el comportamiento especificado.

Especificación de un TAD

¿Cómo se realiza?

1. El dominio se puede definir por enumeración de sus valores, mediante referencia a otros dominios conocidos, mediante definiciones recursivas,...
2. La sintaxis de las operaciones la describiremos por medio del prototipo de la función correspondiente a cada una de ellas.
3. En nuestro caso, para describir la semántica de cada operación emplearemos precondiciones y postcondiciones con la siguiente interpretación:

   Bajo el cumplimiento de las precondiciones se puede realizar la operación y, al finalizar la misma, se garantiza que se cumplirán las postcondiciones.

Especificación de un TAD

Selección de operaciones

Previamente a la especificación de operaciones debemos seleccionar las que incluiremos en el TAD. Al menos seleccionaremos un conjunto mínimo con las siguientes características:

- Escogeremos operaciones básicas o primitivas que nos permitan crear a partir de ellas otras más complejas, dentro o fuera del TAD.
- Dispondremos de un subconjunto de operaciones que permitirá generar todos los valores del dominio.
- El conjunto de operaciones deberá ser completo, es decir, hará posible implementar cualquier algoritmo para procesar los valores del dominio utilizando exclusivamente dichas operaciones.

Especificación de un TAD

Clasificación de operaciones

Constructoras	Generan valores nuevos pertenecientes al dominio.
Destructoras	Liberan las posiciones de memoria ocupadas por un valor previamente creado.
Observadoras	Devuelven el estado o contenido de un valor o de un componente de él.
Modificadoras	Cambian el estado o contenido de un valor o de un componente de él.

Implementación de un TAD

¿Qué hay que hacer?

Elegir la representación de los datos y operaciones del TAD en términos de los tipos de datos y operaciones disponibles en el lenguaje de programación. Conlleva dos tareas:

1. Diseño de una estructura de datos que represente los elementos del dominio del TAD.
2. Implementación de los algoritmos de tratamiento de dicha estructura, que realicen las operaciones del TAD tal como se ha especificado.

Tipo abstracto de datos vs. estructura de datos

El adjetivo «abstracto» expresa que un TAD es un modelo conceptual descrito por su especificación, que sólo existe en la mente del programador; mientras que una estructura de datos es la representación de esa abstracción en un lenguaje de programación.

Implementación de un TAD

Estructuras de datos

1. Se crean por agrupación de tipos de datos simples mediante los tipos estructurados del lenguaje:
   - Vectores y matrices: agregados de elementos del mismo tipo.
   - Registros o estructuras: agregados de elementos que pueden ser de distintos tipos.
   - Estructuras enlazadas mediante punteros: sus componentes, llamados nodos, son registros que incluyen punteros a las posiciones de memoria de otros nodos de la estructura.
2. Se pueden crear estructuras muy complejas componiendo las anteriores.

Implementación de un TAD

Clasificación de las estructuras de datos

1. Estructura de datos estática: Tiene un tamaño fijo (establecido en tiempo de compilación) y se almacena en posiciones contiguas de memoria. Se forma por composición de vectores/matrices y registros.
2. Estructura de datos pseudoestática: Es una estructura estática cuyo tamaño se fija en tiempo de ejecución.
3. Estructura de datos dinámica: Tiene un número variable de elementos almacenados en posiciones de memoria no contiguas y enlazados mediante punteros.

¿Cuál elegir?

La elección adecuada de la estructura de representación de un TAD la realizaremos considerando los requisitos de la aplicación y siguiendo criterios de eficiencia en tiempo y espacio.

Conclusiones

Programación basada en TAD

1. La resolución de un problema mediante un programa la enfocaremos como un proceso de abstracción y descomposición del problema que lleva a la construcción de un modelo formal basado en tipos abstractos de datos.
2. El concepto de TAD, por tanto, proporciona una base ideal para la descomposición modular de un programa grande.
3. Identificados los TAD que intervienen en la solución, se crean sus especificaciones y el programa se escribe con arreglo a ellas.
4. Por último, se elige una representación para cada TAD y se implementan las operaciones mediante los algoritmos más eficientes con la estructura de datos elegida.

Conclusiones

Corolario

Con una metodología de programación basada en tipos abstractos de datos se logra disminuir la complejidad inherente a la construcción de programas, separando dos tareas que se realizan de forma independiente:

1. Construir un programa a partir de unos objetos adecuados a las características particulares del problema a resolver.
2. Implementar estos objetos a partir de los elementos del lenguaje.

Especificación del TAD *racional*

Definición:

Un número racional es el conjunto de fracciones equivalentes. Cada fracción representa la razón entre dos enteros, llamados numerador y denominador, con el denominador distinto de 0. Se toma como representante canónico de un número racional a la fracción irreducible con el denominador positivo y el cero se representa mediante la fracción irreducible 0/1.

Operaciones:

racional::racional(long n = 0, long d = 1)

Precondiciones: $d \neq 0$

Postcondiciones: Crea el racional n/d en forma canónica.

long racional::num() const

Postcondiciones: Devuelve el numerador de un racional.

long racional::den() const

Postcondiciones: Devuelve el denominador de un racional.

Especificación del TAD *racional*

No modifica el valor de la variable

racional operator +(const racional& r, const racional& s)

Postcondiciones: Devuelve $r + s$.

racional operator *(const racional& r, const racional& s)

Postcondiciones: Devuelve $r * s$.

racional operator −(const racional& r)

Postcondiciones: Devuelve $-r$.

racional inv(const racional& r)

Precondiciones: $r \neq 0$.

Postcondiciones: Devuelve $1/r$.

bool operator ==(const racional& r, const racional& s)

Postcondiciones: Devuelve `true` si r y s son equivalentes y `false` en caso contrario.

bool operator <(const racional& r, const racional& s)

Postcondiciones: Devuelve `true` si $r < s$ y `false` en caso contrario.

Utilización del TAD *racional*

```
1  /* sistecu.cpp – Resolución de un sistema de 2 ecuaciones
2                    lineales con coeficientes racionales.
3  */

5  #include <iostream>
6  #include "racional.h"

8  using namespace std;

10 void LeerEcu (racional& a, racional& b, racional& c);
11 int SistEcu (racional a1, racional b1, racional c1,
12              racional a2, racional b2, racional c2,
13              racional& x, racional& y);
14 racional Det (racional a11, racional a12,
15               racional a21, racional a22);
```

Utilización del TAD *racional*

```
17 int main()
18 {
19    racional a1, b1, c1, a2, b2, c2,
20              x, y;
21    int s;

23    // Entrada de datos
24    cout << "Ecuación␣1␣(a1*x␣+␣b1*y␣=␣c1):\n";
25    LeerEcu(a1, b1, c1);
26    cout << endl;
27    cout << "Ecuación␣2␣(a2*x␣+␣b2*y␣=␣c2):\n";
28    LeerEcu(a2, b2, c2);
29    cout << endl;

31    // Cálculo de las soluciones
32    s = SistEcu(a1, b1, c1, a2, b2, c2, x, y);
```

Utilización del TAD *racional*

```
34      // Salida de datos
35      if (s == 0)
36      {
37          cout << "Solución:\n\n";
38          cout << "x = " << x.num() << " / " << x.den() << endl;
39          cout << "y = " << y.num() << " / " << y.den() << endl;
40      }
41      else if (s == 1)
42          cout << "Infinitas soluciones\n";
43      else // s == -1
44          cout << "Sistema incompatible\n";
45  }
```

Utilización del TAD *racional*

```
47 void LeerEcu (racional& a, racional& b, racional& c)
48 {
49    long num, den;

51    cout << "Coeficiente␣de␣x:\n";
52    cout << "␣␣Numerador␣=␣"; cin >> num;
53    cout << "␣␣Denominador␣=␣"; cin >> den;
54    a = racional(num, den);
55    cout << "Coeficiente␣de␣y:\n";
56    cout << "␣␣Numerador␣=␣"; cin >> num;
57    cout << "␣␣Denominador␣=␣"; cin >> den;
58    b = racional(num, den);
59    cout << "Término␣independiente:\n";
60    cout << "␣␣Numerador␣=␣"; cin >> num;
61    cout << "␣␣Denominador␣=␣"; cin >> den;
62    c = racional(num, den);
63 }
```

Utilización del TAD *racional*

```
65 int SistEcu (racional a1, racional b1, racional c1,
66               racional a2, racional b2, racional c2,
67               racional& x, racional& y)
68 {
69    racional A, B, C;

71    A = Det(a1, b1, a2, b2);
72    B = Det(c1, b1, c2, b2);
73    C = Det(a1, c1, a2, c2);
74    if (!(A == 0))
75    {
76       x = B * inv(A);
77       y = C * inv(A);
78       return 0;
79    }
80    else if (B == 0 || C == 0)
81       return 1; // Infinitas soluciones
82    else
83       return -1; // Sistema incompatible
84 }
```

Utilización del TAD *racional*

```
86 racional Det (racional a11, racional a12,
87               racional a21, racional a22)
88 {
89    return a11 * a22 + -a21 * a12;
90 }
```

Implementación del TAD *racional* (racional.h)

```
1  #ifndef _RACIONAL_
2  #define _RACIONAL_
3  // Anular macro assert (verificación de precondiciones)
4  // #define NDEBUG
5  #include <cassert>

7  class racional {
8  public:
9    typedef long long entero;
10   racional(entero nu = 0, entero de = 1);
11   entero num() const {return n;}
12   entero den() const {return d;}
13   // Operadores aritméticos no miembros
14   friend racional operator +(const racional& r, const racional& s);
15   friend racional operator *(const racional& r, const racional& s);
16   friend racional operator -(const racional& r);
17   friend racional inv(const racional& r);
18 private:
19   entero n, d;
20   static entero mcd(entero, entero);
21   static entero mcm(entero, entero);
22 };
```

Implementación del TAD *racional* (racional.h)

```
24 // Operadores aritméticos no miembros
25 // Definiciones en línea
26 inline racional operator -(const racional& r)
27 { return racional(-r.n, r.d); }

29 inline racional inv(const racional& r)
30 {
31   assert(r.n != 0); // Verificar precondición
32   return racional(r.d, r.n);
33 }

35 // Operadores de comparación no miembros
36 // Definiciones en línea
37 inline bool operator ==(const racional& r, const racional& s)
38 { return (r.num() == s.num()) && (r.den() == s.den()); }

40 inline bool operator <(const racional& r, const racional& s)
41 { return (r + -s).num() < 0; }
```

Implementación del TAD *racional* (racional.h)

```
43 // Método privado
44 inline racional::entero racional::mcm(entero x, entero y)
45 // Devuelve mcm(|x|, |y|). Devuelve 0 si x o y valen 0.
46 {
47   return (x && y) ?
48          (x < 0 ? -x : x) / mcd(x,y) * (y < 0 ? -y : y) :
49           0;
50 }

52 #endif // _RACIONAL_
```

Implementación del TAD *racional* (racional.cpp)

```
1  #include "racional.h"

3  // Métodos públicos

5  // Constructor
6  racional::racional(entero nu, entero de) : n(nu), d(de)
7  {
8    assert(d != 0); // Verificar precondición
9    if (d < 0) { // Poner signo en el numerador
10     n = -n;
11     d = -d;
12   }
13   // Reducir fracción
14   entero m = mcd(n, d);
15   if (m != 1) {
16     n /= m;
17     d /= m;
18   }
19 }
```

Implementación del TAD *racional* (racional.cpp)

```
21 // Operadores aritméticos no miembros
22 racional operator +(const racional& r, const racional& s)
23 {
24   racional::entero m = racional::mcd(r.d, s.d);
25   return racional(s.d / m * r.n + r.d / m * s.n,
26                   racional::mcm(r.d, s.d));
27 }

29 racional operator *(const racional& r, const racional& s)
30 {
31   racional::entero a = racional::mcd(r.n, s.d);
32   racional::entero b = racional::mcd(r.d, s.n);
33   return racional((r.n / a) * (s.n / b),
34                   (r.d / b) * (s.d / a));
35 }
```

Implementación del TAD *racional* (racional.cpp)

```
37 // Método privado
38 racional::entero racional::mcd(entero x, entero y)
39 // Devuelve mcd(|x|, |y|). Devuelve 0 si x e y valen 0.
40 {
41   // Algoritmo de Euclides
42   if (x < 0) x = -x;
43   if (y < 0) y = -y;
44   if (y) while ((x %= y) && (y %= x));
45   return x + y; // Devolver el último divisor.
46 }
```