Unidad 4 – Tipos Abstractos de Datos

Introducción

El problema fundamental que se presenta en el desarrollo de programas está vinculado a la alta complejidad de los mismos. En la mayoría de los casos, dicha complejidad escapa a la posibilidad de una comprensión directa e inmediata de la totalidad del problema a resolver.

Surge entonces la necesidad de una primera fase de análisis, previa a la implementación de un problema en un lenguaje de programación, cuyo objetivo fundamental es lograr una correcta comprensión del problema a resolver. El análisis es el primer paso en la construcción de programas que permite obtener una descripción lo más precisa posible (aunque no necesariamente formal) de dicho problema.

Una manera de tratar con la complejidad de los problemas es descomponer el problema en problemas más pequeños, a su vez para resolver estos últimos puede volver a aplicarse el mismo criterio y continuar con el proceso hasta poder resolverse sin necesidad de descomposición.

Otra de las funciones del análisis es la de reducir el problema estudiado, o parte de él, a problemas ya conocidos y resueltos y de esa forma poder usar algunas de las soluciones que se conocen para dichos problemas.

La *abstracción* en el análisis de un problema consiste en individualizar los objetos de la realidad que interesan y el comportamiento de dichos objetos que se traducirán luego en las operaciones que permiten su manipulación.

Como ya sabemos un tipo de datos es un conjunto de valores, acompañado de un conjunto de operaciones que determinan el comportamiento de esos valores. Lo que se está haciendo conceptualmente es individualizar los tipos de datos presentes en el problema. En particular estos tipos de datos identificados se dicen *abstractos* porque no se está trabajando aún con una implementación en particular. Lo que hacemos en el análisis es establecer las características que tendrán estos tipos de datos, pero sin definir aún estructuras de datos para su implementación. Lo que interesa es definir qué operaciones interesa realizar sobre los elementos, pero sin definir aún cómo se implementarán dichas operaciones.

Recién a la hora de implementar la solución del problema es que se realiza una implementación de los tipos abstractos identificados en la realidad, para lo que se elegirán las estructuras de datos más adecuadas para representarlos y se definirán algoritmos lo más eficiente posibles para manipular dichas estructuras. Veremos incluso que para un mismo tipo abstracto de datos, pueden existir diferentes estructuras posibles para su implementación. Incluso sucede que el análisis puede realizarse en forma independiente de la elección del lenguaje de programación concreto a utilizar.

Los tipos abstractos de datos que pueden aparecer en las diferentes realidades no son muchos, y podemos agruparlos en familias de acuerdo al comportamiento.

Categorías de Tipos Abstractos de Datos

Podemos clasificar los Tipos Abstractos de Datos en las siguientes categorías:

- **Elementales**: incluye aquellos tipos abstractos de datos que son indivisibles, y en ella se encuentran los ya conocidos en anteriores cursos de Programación:
  - ***Enteros***
  - ***Reales***
  - ***Caracteres***
  - ***Booleanos***
  - ***Enumerados***
- **Intermedios:** incluye aquellos tipos abstractos de datos que representan objetos de la realidad del problema a resolver:
  - ***Producto Cartesiano***
  - ***Unión discriminada***
- **Colecciones:** incluye aquellos tipos abstractos de datos que representan agrupaciones de elementos (ya sean elementales o intermedios) dentro de la realidad del problema a resolver.

En este curso nos concentraremos particularmente en el análisis de los diferentes tipos de colecciones que podemos representar en una realidad. En general, todas las colecciones se construyen de la misma manera, en forma inductiva, originalmente es una colección vacía y luego le vamos agregando de a un nuevo elemento.

Lo que diferencia a los distintos tipos de colección es su comportamiento. Dependiendo de la forma como se agrupan los elementos de la colección y el tratamiento de los mismos se definen diferentes tipos de colecciones.

Al agrupar los elementos en colecciones surgen los siguientes criterios a tener en cuenta:

- repetición de elementos
- orden de los elementos
- claves que identifican a los elementos

Repetición de Elementos

Al agrupar elementos de un determinado tipo, puede importar la cantidad de veces que ocurre cada uno de ellos en la agrupación o simplemente saber si está en la agrupación sin importar cuántas veces.

Ejemplo:

- En una clase, se quiere calcular la edad promedio de los alumnos. Para resolver este problema, la entrada es una colección de números naturales entre los cuales importa la repetición de elementos, ya que si hay dos alumnos con la misma edad se deberá considerar a ambos.
- En una clase, se quiere saber de qué edades no hay alumnos. Para resolver este problema, la entrada es una colección de números naturales, entre los cuales no importa la repetición de elementos, ya que el resultado va a ser el mismo si hay un alumno de 25 años o hay 5 con dicha edad. Lo que importa es representar la pertenencia o ausencia de un elemento.

Orden de los Elementos

Hay otras situaciones en las cuales importa la forma en la cual los elementos se ordenan dentro de la colección. En el ejemplo del cálculo de la edad promedio de los alumnos de una clase, no importa el orden en que se procesen dichas edades, ya que el resultado será el mismo. Pero hay otras situaciones en la cuales el orden sí es relevante.

Ejemplo:

- Se quiere representar a las personas que esperan frente a una ventanilla de una caja en un banco. En este caso importa el orden en que estas personas fueron llegando a la cola. Es decir debe ser posible determinar cual es el primero, cuál es el segundo y así sucesivamente.

Claves de Identificación

En la mayoría de los casos interesa saber si los elementos de una agrupación se pueden identificar por una clave. Nótese que en este tipo de colección no se admitirán elementos repetidos dentro de la misma.

Ejemplo:

- Se quiere representar a las personas que asistieron a un sorteo. Cada una de ellas tiene un número para el sorteo, el cual identifica a la persona, ya que no habrá dos participantes con el mismo número.

Familias de Tipos Abstractos de Datos

De acuerdo a los criterios anteriores, vamos a definir cuatro familias de tipos abstractos de datos para representar colecciones de elementos. Ellas son las siguientes:

- ***Secuencia***
- ***Bolsa***
- ***Set***
- ***Diccionario***

Vamos a describir el comportamiento de cada una de estas familias. Hablaremos de colecciones de elementos de un tipo genérico T, ya que para la definición de las familias no es relevante el tipo concreto de los elementos a almacenar.

También vamos a definir las principales operaciones que manipulan los elementos de las colecciones pertenecientes a dichas familias. Habitualmente se las llama operaciones básicas o primitivas de la colección. Estas no necesariamente son las únicas operaciones que tendrá la colección, aquí solamente presentamos las primitivas.

Las vamos a definir como funciones desde un punto de vista matemático, pero a la hora de implementarlas en un lenguaje de programación se deberá determinar en cada caso si es más conveniente realizar función o procedimiento. Generalmente sólo aquellas operaciones que modifiquen la colección en algún sentido se implementarán como procedimientos.

Secuencia

Una secuencia es una colección ordenada de elementos, en el sentido de que los elementos se almacenan linealmente (como en una fila). Habrá un primer elemento, seguido de un segundo elemento, un tercer elemento, etc. y cada uno de ellos ocupará una posición definida dentro de la secuencia.

Dentro de esta familia vamos a definir cuatro tipos abstractos de datos (secuencia, stack, queue, y deque). El nombre del primero de ellos coincide con el nombre de la familia por tratarse del tipo más simple de secuencia. No obstante, cada uno de los cuatro TAD representa a una secuencia específica sobre la cual interesa definir un comportamiento diferente en cada caso.

Secuencia

Se trata de una secuencia en la cual no interesa definir ningún comportamiento especial de sus elementos. Simplemente interesa tenerlos almacenados en forma lineal. Sus operaciones primitivas son las siguientes:

Crear: $\varnothing \rightarrow$ *Secuencia*
Crea una secuencia vacía.

InsFront: Secuencia x T$\rightarrow$ *Secuencia*
Agrega un elemento de tipo T a la secuencia.

EsVacía : Secuencia $\rightarrow$ *Boolean*
Determina si la secuencia está vacía o no.

Primero : Secuencia $\rightarrow$ *T*
Devuelve el primer elemento de la secuencia.
Precondición: la secuencia no es vacía.

Resto : Secuencia $\rightarrow$ *Secuencia*
Devuelve la secuencia sin su primer elemento.
Precondición: la secuencia no es vacía

Largo: Secuencia $\rightarrow$ *N*
Devuelve la cantidad de elementos de la secuencia.

K-ésimo: Secuencia x N $\rightarrow$ *T*
Devuelve el elemento que ocupa la posición K de la secuencia.
Precondición: El largo de la secuencia es mayor o igual a K.

Stack

Se trata de una secuencia cuyo comportamiento es como el de una pila de platos, sólo se puede agregar un elemento encima de toda la pila y solo se puede quitar el elemento que está encima de la pila. En ningún momento se puede acceder a elementos que están por debajo del tope de la pila. Sus operaciones primitivas son las siguientes:

Make: $\varnothing \rightarrow$ *Stack*
Crea un stack vacío.

Push: Stack x T $\rightarrow$ *Stack*
Agrega un elemento en el tope del stack

Empty: Stack → Boolean
Determina si el stack está vacío o no.

Top: Stack → T
Devuelve el elemento que se encuentra en el tope del stack.
Precondición: el stack no está vacío.

Pop: Stack → Stack
Devuelve el stack sin el elemento que está en el tope.
Precondición: el stack no está vacío.

Queue

Se trata de una secuencia cuyo comportamiento es como el de una cola de espera, sólo se puede agregar un elemento al final de la cola y solo se puede quitar el elemento que está al principio de la cola. En ningún momento se puede acceder a elementos que están entre medio de ellos en la cola. Sus operaciones primitivas son las siguientes:

Make : ∅ → Queue
Crea un queue vacío.

InsBack : Queue x T → Queue
Agrega un elemento al final del queue.

Empty : Queue → Bolean
Determina si el queue está vacío o no.

Front : Queue → T
Devuelve el primer elemento del queue.
Precondición: el queue no está vacío.

RemFront : Queue → Queue
Devuelve el queue sin su primer elemento.
Precondición: el queue no está vacío.

Deque

Se trata de una secuencia en la cual los elementos se pueden agregar o sacar por cualquiera de los dos extremos. En ningún momento se puede acceder a elementos que están entre medio de ellos en la cola. Sus operaciones primitivas son las siguientes:

Make : ∅ → Deque
Crea un deque vacío.

Insfront : Deque x T → Deque
Agrega un elemento al principio del deque.

Insback : Deque x T → Deque
Agrega un elemento al final del deque.

Empty : Deque → Bolean
Determina si el deque está vacío o no.

Front : Deque → T
Devuelve el primer elemento del deque
Precondición: el deque no está vacío

Back : Deque → T
Devuelve el último elemento del deque
Precondición: el deque no está vacío

Remfront : Deque → Deque
Devuelve el deque sin el primer elemento
Precondición: el deque no está vacío

Remback : Deque → Deque
Devuelve el deque sin el último elemento
Precondición: el deque no está vacío

Bolsa

Esta familia define colecciones que representan agrupaciones de elementos en las cuales puede haber elementos repetidos y no importa el orden de los elementos.

Dentro de esta familia vamos a definir un único tipo abstracto de datos, cuyo nombre coincide con el de la familia.

Bolsa

Las operaciones primitivas de este TAD son las siguientes:

Crear : ∅ → Bolsa
Crea una bolsa vacía.

Insertar : Bolsa x T → Bolsa
Inserta un elemento de tipo T a la bolsa.

EsVacía: Bolsa → Boolean
Determina si la bolsa está vacía o no.

Pertenece : Bolsa x T → Boolean
Determina si el elemento de tipo T forma parte de la bolsa o no.

Borrar : Bolsa x T → Bolsa.
Devuelve la bolsa sin el elemento de tipo T especificado. En caso de estar repetido, elimina solamente una ocurrencia del elemento.
Precondición: el elemento especificado pertenece a la bolsa.

Set

Esta familia define colecciones que representan la noción matemática de conjunto. En un conjunto no hay elementos repetidos y no importa el orden de los elementos. Es importante notar que en teoría de conjuntos es posible definir conjuntos de elementos de diferente tipo. En este curso nos restringiremos solamente a definir conjuntos de elementos de un mismo tipo.

Dentro de esta familia vamos a definir un único tipo abstracto de datos, cuyo nombre coincide con el de la familia.

Set

Las operaciones primitivas de este TAD son las siguientes:

Crear : ∅ → Set
Crea un set vacío.

Insertar : Set x T → Set
Inserta un elemento de tipo T al set. Si el elemento ya pertenecía al set, esta operación no tiene efecto.

EsVacío: Set → Boolean
Determina si el set está vacío o no.

Pertenece : Set x T → Boolean
Determina si el elemento de tipo T forma parte del set o no.

Borrar : Set x T → Set.
Devuelve el Set sin el elemento de tipo T especificado.
Precondición: el elemento especificado pertenece al set.

Unión : Set x Set → Set
Dados dos sets, devuelve un set que contiene los elementos de cualquiera de ellos.

Intersección : Set x Set → Set
Dados dos sets, devuelve uno nuevo que contiene los elementos que están en ambos en forma simultánea.

Diferencia : Set x Set → Set
Dados dos sets, devuelve uno nuevo que contiene los elementos que están en el primero y no están en el segundo.

Diccionario

Esta familia define colecciones cuyos elementos poseen una clave que los identifica. Los elementos de este tipo de colección son parejas de la forma ***<clave, información>*** donde la clave es una función, generalmente una función selectora del tipo T, que identifica a los elementos de la colección y la información corresponde a los datos relevantes del objeto en la realidad del problema.

Dentro de esta familia vamos a definir un único tipo abstracto de datos, cuyo nombre coincide con el de la familia.

Diccionario

Llamaremos T al tipo de los elementos a almacenar en la colección y K al tipo de las claves. Las operaciones primitivas de este TAD son las siguientes:

Make: ∅ → Diccionario
Crea un diccionario vacío.

Member : Diccionario x K → Boolean
Determina si en el diccionario existe un elemento con la clave especificada.

Insert : Diccionario x T → Diccionario
Inserta un elemento de tipo T en el diccionario.
Precondición: el elemento a insertar no es miembro del diccionario.

Find : Diccionario x K → T
Dada la clave de un elemento devuelve el elemento con dicha clave
Precondición: el elemento es miembro del diccionario.

Modify : Diccionario x T → Diccionario
Sustituye el viejo elemento de tipo T en el diccionario por el nuevo elemento.
Precondición: el elemento a sustituir es miembro del diccionario.

Delete: Diccionario x K → Diccionario
Dada la clave de un elemento lo borra del diccionario
Precondición: el elemento es miembro del diccionario.

Representación de Tipos Abstractos de Datos

Una vez realizado el análisis de TAD para una realidad, debemos elegir qué estructuras de datos usaremos para representar dichos TAD e implementar sus operaciones en un lenguaje de programación.

Generalmente, los TAD elementales o intermedios suelen representarse mediante los tipos de datos básicos o estructurados del lenguaje de programación a utilizar. El mayor problema consiste en la elección de estructuras de datos adecuadas para representar colecciones. En principio, cualquier estructura de datos es potencialmente adecuada para representar cualquier colección. Sin embargo, al hacer la elección debemos tener en cuenta las características más relevantes de esa colección y las operaciones más frecuentes que se realizan sobre ella.

Una vez determinado lo anterior, iremos descartando estructuras de datos de acuerdo al orden de los algoritmos implementados con ellas, y nos quedaremos con aquellas que permitan escribir algoritmos de menor orden. Los factores que influyen en la elección de una estructura para la colección, en orden de prioridad, son:

- la cardinalidad.
- el orden que tienen los algoritmos a implementar en dicha estructura.
- la frecuencia de uso de esas operaciones.

Cardinalidad

No es posible implementar una colección no acotada mediante una estructura acotada, ya que por más grande que sea la cota puede suceder que dicha estructura se llene y no sea posible agregar nuevos elementos a la colección. En este caso se descartan todas aquellas estructuras que son acotadas como posibles candidatas para la representación del TAD. Contrariamente, el hecho de que la colección sea acotada no descarta ninguna estructura de datos.

Orden de los Algoritmos

El siguiente factor a tener en cuenta es el orden de los algoritmos; es deseable que todas las operaciones del TAD identificado, sean implementadas lo más eficientemente posible. Por lo tanto la estructura de datos elegida debe ser aquella que implementa las operaciones del TAD de la forma más eficiente en comparación con cualquier otra estructura.

Frecuencia de las Operaciones

Con frecuencia se da el hecho de que para una determinada estructura de datos, la implementación de algunas operaciones se logra realizar en forma muy eficiente, en cambio la implementación del resto de las operaciones se realiza en forma poco eficiente.

Un criterio posible es querer realizar en forma eficiente la mayor cantidad de operaciones posible. Otro criterio posible sería preferir una estructura en la que las operaciones que se usan con mayor frecuencia se realicen en forma más eficiente. Los dos criterios mencionados no necesariamente deben ser excluyentes, pero en caso de serlo se suele priorizar el segundo.

Elección de Estructuras

Como se dijo anteriormente, en principio cualquier estructura de datos es apropiada para representar cualquier TAD colección. No obstante, de acuerdo a los tres factores anteriores, algunas estructuras suelen resultar más idóneas que otras para implementar las operaciones de los TAD colecciones. Presentamos aquí un breve resumen de las estructuras de datos que, en la mayoría de los casos, suelen resultar más apropiadas para cada familia de TAD colección estudiada. De cualquier manera es importante recordar que, dependiendo de las operaciones concretas de cada problema y de su frecuencia de uso, pueden surgir otras alternativas además de las contempladas aquí.

Secuencia

Cualquier estructura de datos lineal suele ser apropiada para implementar una secuencia. Si se asume una secuencia acotada, cualquier variante de arreglo puede resultar apropiada (arreglo simple, arreglo con tope, arreglo dinámico) dependiendo de las características concretas de la secuencia, aunque esto no descarta el uso de estructuras dinámicas. En el caso de una secuencia no acotada, necesariamente debemos usar estructuras dinámicas. En principio, cualquier variante de lista suele ser lo más adecuado. Para un Stack, puede bastar con una lista simple, debido a que la inserción y el borrado de elementos siempre se realiza por un extremo. Para un Queue o un Deque, puede resultar mejor una LPPF o una LDEPPF debido a que la inserción y el borrado se realizan siempre sobre los extremos.

Bolsa

Dado que en una bolsa puede haber elementos repetidos, descartamos las estructuras de Mapeo, Hash y ABB, dado que las mismas asumen que no hay repetición de elementos. Cualquier otra estructura es, en principio, apropiada para representar una bolsa.

Set

Dado que en un set no puede haber elementos repetidos, en principio cualquier estructura resulta apropiada para representarlo.

Diccionario

Las estructuras de Hash y ABB suelen ser las más apropiadas para representar un diccionario, dado que son idóneas para elementos que poseen una clave que los identifica. La estructura de Mapeo también puede ser usada si se asume un diccionario acotado. La estructura de Hash suele ser preferible dado que el orden de los algoritmos de búsqueda es menor que en un ABB. Sin embargo, la estructura de ABB es más eficiente cuando interesa recorrer los elementos del diccionario ordenados por clave de menor a mayor.