APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Por favor, reportar errores, omisiones y sugerencias a dfridlender@gmail.com

Estructuras de Datos

La elección de las **estructuras de datos** o **tipos de datos** son determinantes a la hora de diseñar un programa. Una elección correcta podrá dar lugar a programas elegantes, breves, eficientes, legibles, fáciles de mantener, reusables. Una elección equivocada o simplemente apurada puede tener como consecuencia programas sin ninguna de las virtudes mencionadas, incorrectos (que no resuelven el problema) y cuyas fallas, además, son difíciles de encontrar, no hablemos de reparar.

Un lenguaje de programación normalmente proporciona ciertos tipos básicos, e incluso maneras de definir ciertos tipos más complejos. Podemos llamar a éstos **tipos concretos**. Distintos lenguajes proporcionan distintos tipos concretos, es un concepto relativo al lenguaje de programación que se utilice.

Los **tipos abstractos** no están asociados al lenguaje de programación que uno utilice sino más bien al problema que uno intenta resolver. Los tipos abstractos surgen cuando uno analiza el problema, e identifica qué información debe manipular el programa, y cómo debe manipularla. Se denominan tipos abstractos para resaltar el hecho de que uno está pensando en las propiedades del tipo de dato, y no en su implementación. Lo que uno elabora al pensar en tipos abstractos es independiente de toda posible implementación del mismo.

Por ejemplo, si debemos escribir un programa que informe cuál es el camino más corto para trasladarse en auto de una ciudad a otra del país, surgirá naturalmente pensar que la información que dicho programa maneja es un grafo, donde los nodos son ciudades y cruces de rutas y las aristas son segmentos de rutas con sus longitudes. Grafo será entonces un tipo abstracto que se utilizará para resolver el problema, a pesar de que usualmente los lenguajes de programación no cuentan con un tipo concreto Grafo.

Finalmente, cuando el problema tenga que ser programado para su ejecución por una computadora, los tipos abstractos deberán ser implementados en el lenguaje de programación utilizando tipos concretos. Ésta será la **implementación** o **representación** del tipo abstracto. Normalmente, un mismo tipo abstracto puede ser implementado de numerosas maneras diferentes.

Sin embargo, identificar y especificar los tipos abstractos involucrados es muy importante porque proporciona un entendimiento cabal del problema que se intenta resolver (en vez de abocarse directamente a intentar resolver un problema no entendido completamente); porque permite escribir los algoritmos en un lenguaje comprensible por el ser humano; permite poner a prueba tempranamente si la solución que se está diseñando está bien encaminada; facilita el desarrollo del programa en equipo; y proporciona una

solución **independiente de la representación** del tipo abstracto. Esto último quiere decir que uno obtiene una solución que admite diferentes maneras de implementar el tipo abstracto; en particular, es provechoso para facilitar el reemplazo de una implementación del tipo abstracto por otra en caso de que uno lo considere luego conveniente.

En esta parte de la materia, desarrollaremos la técnica de la especificación de tipos abstractos de datos e ilustraremos sobre posibles implementaciones.

Primeramente haremos un breve repaso de los tipos concretos usualmente proporcionados por los lenguajes de programación imperativos. Además de los tipos básicos enteros, naturales, caracteres, booleanos, strings, flotantes, etc. los lenguajes imperativos normalmente proporcionan maneras de definir tipos compuestos. Describiremos arreglos, listas, tuplas y punteros.

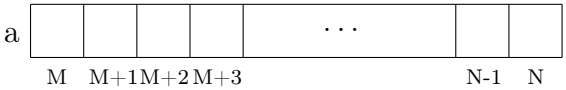
Luego introduciremos una notación para describir tipos abstractos de datos (TADs) y la aplicaremos en algunos ejemplos para ilustrar su utilización.

Tipos Concretos

Arreglos. Dado un tipo **T**, normalmente declaramos arreglos de la siguiente forma:

type tarray = **array**[M..N] **of T**
var a: tarray

El tipo tarray así definido corresponde al producto Cartesiano $\mathbf{T}^{\text{N-M+1}}$. Los arreglos se alojan normalmente en **espacios contiguos** de memoria. El arreglo a declarado recientemente suele representarse gráficamente de la siguiente manera:



En la representación gráfica se observa una celda para cada índice entre M y N, que al desplegarse en forma adyacente sugiere efectivamente que se alojan en espacios contiguos de memoria. El valor alojado en la celda i se obtiene evaluando la expresión a[i] y se modifica asignando a a[i] (ejemplo, a[i]:= e). Al estar alojado en espacio contiguo de memoria acceder o modificar cualquier celda lleva tiempo *constante*. Es decir, el tiempo de acceso al valor de una celda, o el tiempo de modificación de una celda no depende del número de celdas (pero sí puede depender del tipo **T** ya que lo que se está accediendo o modificando es un elemento de ese tipo).

Los arreglos tienen longitud prefijada: N-M+1. Normalmente N > M, pero se puede admitir también N=M (longitud 1) e incluso N < M (longitud 0). El tamaño total del arreglo (espacio ocupado en memoria) es la longitud del mismo multiplicada por el tamaño de cada celda, que depende del tipo **T**. Es decir, el espacio que ocupa es *del orden de n* donde *n* es la longitud del arreglo (diremos que el espacio que ocupa es *lineal* en el número de celdas).

A veces conviene utilizar índices que no sean simplemente enteros. Por ejemplo:

type tindex = **array**['a'..'z'] **of nat**
var page: tindex

Dada una guía teléfonica, el arreglo page podría servir para informar en qué página de la misma aparecen listadas las personas cuyos nombres comienzan con cada letra.

Por ejemplo, page['g'] = 271 significaría que en la página 271 de la guía comienzan los nombres de personas cuya primera letra es la letra g.

Otras posibilidades son

type tweek = (sunday, monday, tuesday, wednesday, thursday, friday, saturday)
type tcalendar = **array** [monday..friday] **of T**
var cal: tcalendar

En este caso, la variable cal es un arreglo con cinco celdas, una para cada día hábil de la semana. Por ejemplo, con un **T** adecuado, cal podría almacenar las tareas a desarrollar cada uno de esos días.

Esto muestra que se puede utilizar una variedad de conjuntos como índices de arreglos. Lo importante es que haya una clara noción de "el siguiente de", cosa que ocurre con enteros (el siguiente de 4 es 5), caracteres (el siguiente de 'h' es 'i') y tipos enumerados como tweek (el siguiente de wednesday es thursday).

También es frecuente la utilización de arreglos multidimensionales, ejemplos:

type tarray1 = **array**[1..N,1..M] **of T**
type tarray2 = **array**[1..N,'a'..'z',sunday..saturday] **of T**
var b: tarray2

dando lugar a celdas que se acceden con notaciones tales como b[3,'k',tuesday]. Los arreglos bidimensionales se suelen denominar matrices y se grafican como tales. Para enfatizar el hecho de que un arreglo es unidimensional se lo suele denominar vector.

Para inicializar y modificar arreglos es muy común utilizar el comando **for**. Por ejemplo, el siguiente comando inicializa todas las celdas de a con el valor 0.

for i:= M **to** N **do** a[i]:= 0 **od**

Intuitivamente, el comando dice que para todo valor de i entre M y N, ambos inclusive, se asigna 0 a la celda a[i].

El comando **for** adquiere en general la forma

for i:= M **to** N **do** c **od**
for l:= 'a' **to** 'z' **do** c **od**
for d:= tuesday **to** friday **do** c **od**

donde en los tres ejemplos, c, llamado **el cuerpo** del **for**, es cualquier comando que **no modifica** el valor de la variable que se usa como índice[5] (i, l y d respectivamente en estos ejemplos).

El comando **for** se puede traducir directamente usando el comando **do**. El primer ejemplo se traduce a:

var i: **int**
i:= M
do i $\leq$ N $\rightarrow$ c
i:=i+1
od

[5]Lamentablemente, la mayoría de los lenguajes de programación permiten que dicha variable sea modificada en el cuerpo del **for**. Se considera una **muy mala práctica** de programación escribir un **for** en el que eso ocurre.

Puede observar que cuando M>N, el comando c no se ejecuta. Dado que el índice i **no es modificado** en el comando c, la terminación del **for** es clara (a menos que c contenga a su vez algún ciclo que pueda no terminar). El uso correcto del **for** en vez de su equivalente en términos del **do** permite convencerse fácilmente de la terminación.

Se asumirá que la variable que se utiliza como índice es local al **for**. Dicha variable es declarada por el propio **for** y deja de existir al finalizar la ejecución del mismo.[6]

Si bien es muy útil para recorrer las celdas de un arreglo, el comando **for** puede también utilizarse en otros contextos: cada vez que es necesario repetir un comando c para diferentes valores de i. Por ejemplo, si e y e' son, por ejemplo, expresiones enteras, el comando **for** i:= e **to** e' **do** c **od**, ejecuta repetidamente c para distintos valores de i: desde el valor de e hasta el de e'. Además de ser c un comando que no modique i, se requiere que el valor de e' no dependa de i. Estos requerimientos permiten garantizar la terminación del ciclo (siempre que c termine) e incluso predecir el número exacto de veces que se ejecutará.

Para traducir en términos del **do** los otros ejemplos mencionados en la página anterior es necesario una noción de "el siguiente de" que a su vez da lugar a una noción de "menor o igual que".

Una variante del **for** es usando **downto** en vez de **to**. Por ejemplo, si c es un comando que no modifica el valor de i,

for i:= M **downto** N **do** c **od**

se traduce a

```
var i: int
i:= M
do i ≥ N → c
            i:=i-1
od
```

donde se observa que el comando c no se ejecuta cuando M<N.

En el ejemplo del arreglo tridimensional b declarado más arriba, si se quiere inicializar todo el arreglo (asumiendo que **T** es, por ejemplo, **int**), se lo puede hacer a través de 3 ciclos **for** anidados:

```
for i:= M to N do
    for l:= 'a' to 'z' do
        for d:= tuesday to friday do
            b[i,k,d]:= 0
        od
    od
od
```

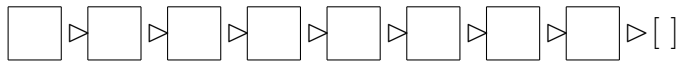
Por último, decimos que un invariante de **for** i:= M **to** N **do** c **od** es un predicado $\mathcal{I}(i)$, tal que $\mathcal{I}$(M) vale antes de la ejecución del **for** y la validez de $\mathcal{I}(i) \wedge M \leq i \leq N$ antes de cada ejecución de c garantiza la validez $\mathcal{I}(i+1)$ después de dicha ejecución de c. Entonces, si N ≥ M-1, al finalizar el **for** se cumple $\mathcal{I}$(N+1).

[6]Esta propiedad no la cumple su traducción en términos del **do**.

Listas. Algunos lenguajes de programación permiten declarar listas:

type tlist = **list of T**
var l: tlist

El tipo tlist así definido corresponde a la unión de los productos Cartesianos $\mathbf{T}^i$, es decir, corresponde a $\bigcup_{i=0}^{\infty} \mathbf{T}^i$. Así, a diferencia del arreglo cuya longitud está predeterminada, el número de elementos de una lista no lo está. A priori, puede contener una cantidad arbitraria de elementos de **T** (en la práctica evidentemente existirán limitaciones de espacio). Otra diferencia importante con el arreglo es que la lista **no** necesariamente se aloja en **espacios contiguos** de memoria. La lista l declarada recientemente puede representarse gráficamente de la siguiente manera:



En la representación gráfica se ve una celda para cada elemento de la lista, que al desplegarse con el símbolo ▷ sugiere la existencia de una flecha desde una celda a la siguiente. Una lista puede modificarse agregando (por ejemplo, l:= e ▷ l) o quitando un elemento (por ejemplo, l:= tail(l)).

Son justamente estas operaciones las que dificultan alojar una lista en espacios contiguos de memoria. Al agregarse un elemento a una lista, no hay ninguna garantía de que haya espacio libre justo en la posición de memoria adyacente a donde se encuentra el primer elemento de la lista. Si se quisiera alojar la nueva lista en espacios contiguos habría que copiar la lista entera en una parte de la memoria donde haya suficiente espacio libre para toda la lista. En principio, esto es posible (aunque presenta numerosos inconvenientes), pero no es lo que normalmente se hace ya que significa que las modificaciones requerirían justamente copiar toda la lista y por lo tanto serían *lineales* en el número de celdas (sería *del orden de n* donde n es la longitud de la lista). En lugar de esto, se aloja el elemento que se quiere agregar en una nueva posición de memoria (cualquiera que esté libre) y se mantiene la información que dice en qué posición de la memoria se encuentran los siguientes elementos de la lista. Así, estas modificaciones resultan *constantes* en lugar de *lineales*, o sea, no dependen del número de celdas de la lista.

Luego de una secuencia de modificaciones, los elementos de una lista pueden quedar desperdigados en la memoria. Siempre se puede recorrer la lista ya que se cuenta con la información necesaria, como sugiere la representación gráfica, para ir de cada elemento de la lista al siguiente.

Esto significa que para acceder al i-ésimo elemento de una lista es necesario recorrerla secuencialmente hasta encontrarlo, operación que resulta *del orden de i*. También significa que además de las celdas debe reservarse lugar para indicar las posiciones de memoria en que se encuentran las celdas. A pesar de ello, el espacio que ocupa una lista en memoria es *del orden de n* donde n es el número de celdas de la lista.

No siempre los lenguajes de programación tienen el tipo concreto lista. Los más importantes ofrecen, sin embargo, alguna forma de implementarlo. Veremos luego que se pueden implementar listas usando los tipos concretos tupla y puntero.

Por último, puede convenir a veces extender la notación del **for**. Por ejemplo, si se quiere ejecutar c una vez para cada elemento e de la lista l (del primero al último) se escribe:

for e ∈ l **do** c **od**

La misma notación puede utilizarse también al recorrer arreglos si el cuerpo del **for** no necesita referirse a las posiciones. Por ejemplo,

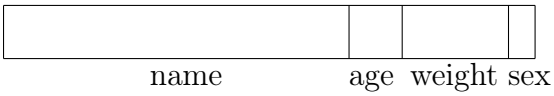
for e ∈ a **do** c **od**

ejecuta el cuerpo c para todos los elementos alojados en celdas de a, independientemente de la dimensión del arreglo a.

Tuplas. También llamados registros o estructuras, se utilizan para representar productos Cartesianos, pero ahora cuando los conjuntos entre los que se hace el producto son distintos, es decir, de la forma $\mathbf{T}_1 \times \mathbf{T}_2 \times \mathbf{T}_3$ donde los $\mathbf{T}_i$ pueden ser tipos distintos. Se declaran de la siguiente forma

```
type tperson = tuple
                 name: string
                 age: nat
                 weight: real
                 sex: (male, female)
               end
var p: tperson
```

El tipo tperson así definido corresponde a **string** × **nat** × **real** × {male, female}, y name, age, weight y sex se llaman **campos**. Las tuplas se alojan normalmente en **espacios contiguos** de memoria. Se puede representar gráficamente de la siguiente manera:



En la misma se ven los campos de distinto tamaño porque cada uno de ellos puede ocupar un espacio diferente. Lo alojado en cada campo se obtiene evaluando las expresiones t.name, t.age, t.weight y t.sex, y se modifica utilizando la misma notación a la izquierda de la asigación (por ejemplo, t.name:= "John"). Al estar alojado en espacio contiguo de memoria acceder o modificar cualquier campo lleva tiempo *constante*. Es decir, el tiempo de acceso al valor de un campo, o el tiempo de modificación de un campo no depende del número de campos (pero sí puede depender del tipo $\mathbf{T}_i$ del campo en cuestión, ya que lo que se está accediendo o modificando es un elemento de ese tipo).

Claramente el espacio de memoria que ocupa una tupla es la suma de los espacios que ocupan sus campos.

Punteros. Dado un tipo T, se puede declarar el tipo "puntero a T". Por ejemplo, si tperson es el tipo definido más arriba

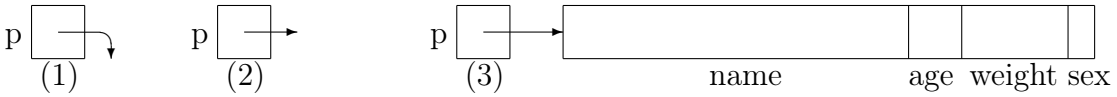
```
type tp_person = pointer to tperson
var p: tp_person
```

La variable p así declarada es un puntero a tperson. Esto significa que p puede almacenar una dirección de memoria donde se aloja una tperson. Las operaciones con punteros son las siguientes:

p:= e
alloc(p)
free(p)

El primer comando es una asignación: e es una expresión cuyo valor es la dirección de memoria de una tperson, la asignación tiene por efecto que dicha dirección sea alojada ahora en p. El segundo comando reserva un nuevo espacio de memoria capaz de almacenar una tperson, y la dirección de ese nuevo espacio de memoria se aloja en p. El tercer comando libera el espacio de memoria señalado por p, es decir, cuya dirección se encuentra alojada en p. Puede darse que p tenga como valor una dirección de memoria que no está actualmente reservada (por ejemplo, inmediatamente después de haber ejecutado free(p)). Para evitar permanecer en este estado, existe un valor especial que puede adoptar un puntero, llamado **null**. Cuando el valor de p es **null**, p no señala ninguna posición de memoria.

Hay distintas representaciones gráficas, una para cada una de las posibles situaciones:



En la situación (1), el valor de p es **null**, p no señala ninguna posición de memoria. En la situación (2) la posición de memoria señalada por p no está reservada, por ejemplo inmediatamente después de free(p). En la situación (3) el valor de p es la dirección de memoria donde se aloja la tperson representada gráficamente al final de la flecha, por ejemplo, inmediatamente después de alloc(p).

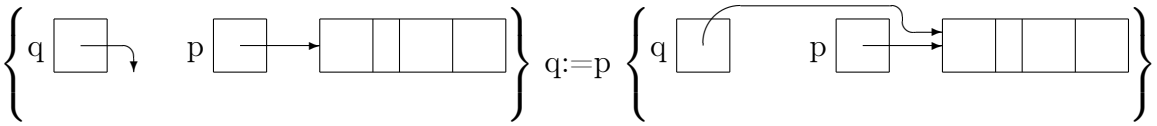
En la situación (3), ⋆p denota la tperson que se encuentra señalada por p, y por lo tanto, ⋆p.name, ⋆p.age, ⋆p.weight y ⋆p.sex, sus campos. Esta notación permite acceder a la información alojada en la tperson y modificarla mediante asignaciones a sus campos (por ejemplo, ⋆p.name:= “John”). Siempre en la situación (3), si r es una tperson, también se puede hacer ⋆p:= r alojando r en la posición señalada por p (pero perdiendo lo que había anteriormente en esa posición de memoria). Esta última asignación no es válida en algunos lenguajes (por ejemplo, no es válida en C). Una notación conveniente para acceder a los campos de una tupla señalada por un puntero es la flecha “→”. Así, en vez de escribir ⋆p.name, podemos escribir p→name tanto para leer ese campo como para modificarlo. Esta notación reemplaza el uso de dos operadores (“⋆” y “.”) por uno visualmente más apropiado (por ejemplo, p→name:= “John”).

La notación ⋆p y sus derivadas ⋆p.name, p→name, etc. sólo pueden utilizarse en la situación (3).

En la situación (2) el valor de p es inconsistente, no debe utilizarse ni accederse una dirección de memoria no reservada ya que no se sabe, a priori, qué hay en ella (en particular puede haber sido reservada para otro uso y al modificarlo se estaría corrompiendo información importante para tal uso). Los punteros que se encuentran en la situación (2) se llaman comúnmente referencias o punteros colgantes (dangling pointers).

En la situación (1) el valor de p es **null**, es decir que p no señala ninguna posición de memoria. Por ello, no tiene sentido intentar acceder a ella.

Como vemos, los punteros permiten manejar explícitamente direcciones de memoria. Esto no es sencillo, aparecen situaciones que con los tipos de datos usuales no se daban. Por ejemplo:



Como se ve, después de la asignación q y p señalan a la misma tupla, por lo que cualquier modificación en campos de $\star$q también modifican los de $\star$p (claro, ya que son los mismos) y viceversa. Estamos en presencia de lo que se llama **aliasing**, es decir, hay 2 nombres distintos ($\star$p y $\star$q) para el mismo objeto y al modificar uno se modifica el otro. Programar correctamente en presencia de aliasing es muy delicado y requiere gran atención.

Acceder o modificar lo señalado por un puntero es claramente *constante*, ya que el puntero contiene la dirección exacta donde se encuentra en la memoria. El *orden* de las operaciones alloc y free, en cambio, depende del compilador del lenguaje. Existen diferentes maneras -no triviales- de implementarlas.

Siempre hemos asumido que no es necesario ocuparse de reservar y liberar espacios de memoria para las variables. Los punteros como p y q son variables, así que **tampoco es necesario reservar y liberar espacio para ellos**. Pero las operaciones alloc y free son las responsables de reservar y liberar explícitamente espacio **para los objetos que p y q señalan**.

Esta posibilidad significa ciertas libertades: el programador puede decidir exactamente cuándo reservar espacio para una tupla. Por otro lado, significa también más responsabilidad: el programador es el que debe encargarse de liberar el espacio cuando deje de ser necesario.

Pero el verdadero beneficio de los punteros radica en que permiten una gran flexibilidad para representar estructuras complejas, y por lo tanto, para implementar diferentes tipos abstractos de datos.