Tipos estructurados de datos homogéneos. Tablas

PID_00268297

Autores que han participado colectivamente en esta obra
Lluís Beltrà
Kenneth Capseta
Maria Jesús Marco Galindo
Antonio Ponce

Idea y dirección de la obra
Maria Jesús Marco Galindo

Diseño y edición gráfica
Asunción Muñoz

Material docente de la UOC

Universitat Oberta de Catalunya

Segunda edición: julio 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Contenidos

Tipos estructurados de datos homogéneos
Tablas

Los tipos de datos que hemos visto hasta ahora, a medida que trabajamos con problemas más complejos resultan insuficientes e ineficientes. Por este motivo utilizamos otro tipo de datos más estructurado que nos permite representar mejor algunos de estos problemas. Se trata de las **tablas**.

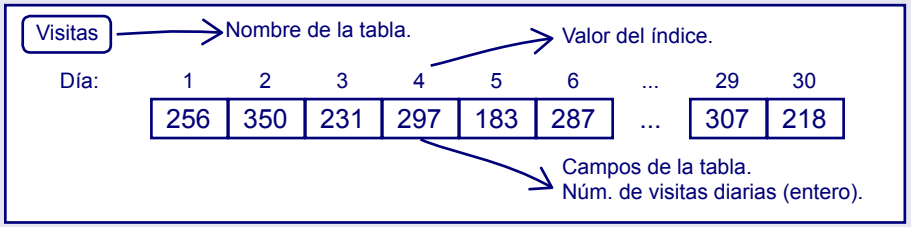
Las tablas sirven para agrupar elementos que son del mismo tipo de datos simple (**estructura homogénea**) con un orden específico y bajo un único nombre.

¿Qué es una TABLA?

Una TABLA es un tipo estructurado de datos que permite agrupar información del mismo tipo (homogénea).

Ejemplo

Imaginemos que somos los gestores de una importante plataforma web y necesitamos tener un control de las visitas que recibe la web mensualmente. Para almacenar eficientemente estos datos, seguramente los registraríamos de un modo similar a:



Observad que hemos creado una tabla con el nombre “visitas”, que tiene 30 campos, correspondientes a cada día del mes, y en cada uno de estos campos almacenará el número de visitas recibidas ese día quue es un valor entero.

Fijaos también que cada campo tiene asignado un índice, que es la posición de este campo dentro de la tabla.

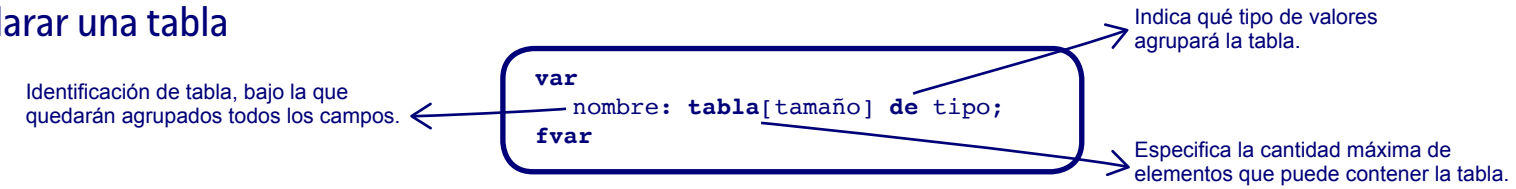
¿Cómo representamos este registro en notación algorítmica?

Podemos representar esta estructura con una tabla y los datos a almacenar (el número de visitas diarias) serán los campos de esta tabla:

```
var
    visitas: tabla[30] de entero;
fvar
```

Observad que con las tablas hemos sido capaces de almacenar 30 valores en una sola variable. Sin embargo, sin las tablas tendríamos que haber creado 30 variables distintas, una para cada valor. Usando la tabla hemos ganado eficiencia y claridad.

Sintaxis para declarar una tabla



Hay que recordar que una tabla solo puede almacenar datos que sean del mismo tipo. En este caso, hemos almacenado enteros pero podríamos haber almacenado cadenas, reales, etc. Por ejemplo, podemos usar una tabla para almacenar el nombre de todos los alumnos de un aula:

```
var
    alumnos: tabla[30] de cadena;
fvar
```

Acceso a los campos de una tabla

Para referirnos a cada elemento, simplemente tendremos que indicar el nombre de la tabla y la posición que el elemento ocupa dentro de ella con el índice.

Hace referencia al nombre que le hemos puesto en la tabla.

```
nombreTabla[índice]
```

Entero positivo o expresión que da como resultado un número entero positivo. Debe estar dentro de los límites marcados por el tamaño máximo de la tabla.

Por ejemplo, siguiendo la declaración que hemos visto anteriormente donde almacenábamos las visitas diarias de la web:

Estas referencias **serían correctas**

```
visitas[2]
visitas[num]
visitas[30]
```

- `visitas[2]`: Accedemos al segundo campo de la tabla.
- `visitas[num]`: En caso de que *num* sea una variable de tipo entero comprendido entre 1 y 30, accederemos al campo de la tabla que tenga la posición indicada.
- `visitas[30]`: Accedemos al trigésimo campo de la tabla. En nuestro ejemplo, es donde se guarda el número de visitas del último día del mes.

En cambio, estas referencias **NO serán correctas**

```
visitas
visitas[32]
visitas["dos"]
```

- `visitas`: No podemos hacer referencia a la tabla sin indicar algún campo.
- `visitas[32]`: No podemos hacer referencia a un índice que sobrepasa los límites establecidos por el tamaño máximo de la tabla.
- `visitas["dos"]`: El índice debe ser de tipo entero.

Concepto clave

Las **tablas** nos permiten agrupar una serie de elementos todos ellos del mismo tipo. De este modo podemos realizar ciertos tratamientos de manera más eficiente. Las tablas abren una nueva posibilidad con el acceso directo a sus elementos.

¿En qué casos nos son útiles las tablas?

Para representar datos formados por una serie de elementos (por una secuencia) del mismo tipo. Este es el caso por ejemplo de vectores, matrices o frases.

¿Cómo podemos asignar y consultar los campos de una tabla?

Ejemplo

Podemos utilizar la asignación como hacíamos con otros tipos de datos para asignar valores a cada campo de la tabla por separado:

```
visitas[1] := 520;
```

Asignamos la primera posición de la tabla *visitas* al número 520.

Esto significa que el primer día del mes la web ha obtenido 520 visitas.

En cuanto a la consulta, si queremos utilizar el valor de un campo concreto, podemos acceder a la tabla de forma directa:

```
x := 2 * visitas[1];
```

Consultamos el número de visitas el primer día del mes y lo duplicamos, de forma que *x* valdrá 1040.

También podemos usar las operaciones de lectura y escritura:

```
escribirPantalla(visitas[1];
visitas[12] := leerTeclado();
```

- `escribirPantalla(visitas[1];`: Escribimos por pantalla el número de visitas del primer día del mes.
- `visitas[12] := leerTeclado();`: Leemos por teclado el número de visitas del día 12.º día del mes y lo guardamos en la posición correspondiente de la tabla.

¿Cómo podemos consultar la longitud de una tabla?

La longitud de una tabla ha quedado determinada en su declaración:

```
var
    alumnos: tabla[30] de cadena;
fvar
```

En este caso, la tabla puede contener hasta 30 elementos, que sería su longitud, su capacidad máxima.

Algunos lenguajes de programación permiten utilizar tablas sin indicar su longitud inicialmente. Son tablas que van creciendo a medida que se van añadiendo elementos. En este caso, disponen de funciones predefinidas para conocer la longitud de una tabla en un momento dado.

En notación algorítmica, por comodidad, disponemos también de la siguiente función predefinida que retorna la longitud de una tabla:

```
l := longitudTabla(t);
```

Retorna la longitud de la tabla *t*, es decir, la medida con la que la hemos definido inicialmente, y la guarda en la variable entera *l*."

Tablas de dimensiones diversas

Hasta ahora hemos trabajado con tablas de una sola dimensión, que hemos representado como un vector. Pues bien, en algunas situaciones nos convendrá trabajar con tablas de dos o más dimensiones, lo que podemos representar como **matrices**.

Ejemplo

Siguiendo con el ejemplo inicial, donde se almacenan las visitas diarias que recibe la página web en un mes, si queremos tener un control de las visitas recibidas durante todo el año, deberemos utilizar una matriz similar a:

Visitas → Nombre de la tabla.

Valor del índice horizontal.

Día: / Mes:	1	2	3	4	5	6	...	29	30
Enero	256	350	231	297	183	287	...	307	218
Febrero	315	198	197	216	187	214	...	297	246
Marzo	248	378	245	310	232	265	...	365	202
Abril	218	316	273	263	169	224	...	386	265
...									
Noviembre	269	305	243	219	194	237	...	360	246
Diciembre	276	363	211	248	146	230	...	309	286

Valor del índice vertical.

Campos de la tabla.
Núm. de visitas diarias (entero).

Como vemos, estamos trabajando con una matriz de dos dimensiones: en un eje tendremos los días que tiene un mes y en el otro eje representaremos los meses del año.

Así pues, tendremos dos índices distintos, que representarán, cada uno, uno de los ejes.

¿Cómo representamos la matriz bidimensional en algorítmica?

Con la notación algorítmica, la podemos representar como una tabla bidimensional de este modo:

```
var
    visitas: tabla[12,30] de entero;
fvar
```

Nombre de la tabla.

Tipo de datos que almacena la tabla.

Índice que representa los meses.

Índice que representa los días.

Sintaxis para declarar una tabla multidimensional

Hemos visto un ejemplo con una tabla de dos dimensiones, pero podríamos trabajar con tablas de tres, cuatro o más dimensiones.

La declaración de una tabla **n-dimensional** se realiza con la sintaxis siguiente:

```
var
   nombre: tabla[tamaño1, tamaño2, ..., tamañoN] de tipo;
fvar
```

Las operaciones de asignación y consulta se realizan de modo parecido a como lo hacíamos con las tablas de una sola dimensión.

Ejemplo

```
visitas[2,3] := 230;
```

→ Asignamos un total de 230 visitas al día 3 de febrero.

¿Cómo representamos las tablas en JavaScript?

En lenguaje JavaScript (JS) las tablas se representan con el tipo **Array**. Un **Array** es un conjunto ordenado de valores al que se accede mediante un índice (que indica la posición dentro del **Array** y que se identifica por un nombre).

Sintaxis para declarar una tabla en JS

```
var nombreTabla=Array(tamañoTabla);
```

A diferencia de la definición de tabla que hemos visto en notación algorítmica, un **Array** en JavaScript puede contener elementos de diferentes tipos, por eso no se indica el tipo de los campos en la declaración.

De todas formas, aunque JS lo permita, no es una buena práctica definir un **Array** con valores de diferentes tipos. Es mejor evitarlo.

Acceso a los campos de una tabla JS

Accederemos a los valores a través del índice, teniendo en cuenta que el índice del primer elemento de un **Array** siempre es cero (0), es decir, las posiciones de un **Array** de longitud n son 0, 1, 2, 3,…n-1. Es muy habitual en los lenguajes de programación que el índice empiece a contar a partir de 0.

Asignaremos valores al **Array** accediendo de la misma manera:

```
nombreTabla[índice]=valorCampo;
```

Recordad que el lenguaje JavaScript tiene muchas más opciones que la notación algorítmica. Aquí adoptaremos la más sencilla.

Encontraréis otras opciones para declarar e inicializar un **Array** en la guía de JavaScript de Mozilla.

Ejemplo

```
1  //Esta definición devolverá una tabla vacía de longitud 30
2  var visitas = Array(30);
3  console.log(visitas.length);
4  //Rellenamos la tabla con los datos de los siete primeros días
5  //Atenció ! El primer elemento siempre es 0
6  visitas[0] = 256;
7  visitas[1] = 350;
8  visitas[2] = 231;
9  visitas[3] = 297;
10 visitas[4] = 183;
11 visitas[5] = 267;
12 visitas[6] = 321;
13 //Muestra todos los campos de la tabla
14 console.log(visitas);
15 //Muestra las visitas del día 4
16 console.log(visitas[3]);
17 //Muestra las visitas del último día de la semana
18 console.log(visitas[6]);
19 //Mostra el primer campo de la tabla
20 console.log(visitas[0]);
21 //Mostrem el último campo de la tabla que no hemos llenado aún
22 console.log(visitas[29]);
```

El acceso a cada uno de los elementos de un **Array** lo hacemos indicando su posición a través del índice.

¿Qué pasaría si...

...intentamos añadir un valor a la posición siguiente a la que hemos establecido como longitud máxima del **Array** en su definición? En este caso, el **Array** se amplía automáticamente y alarga su longitud en una posición más.

Ejemplo

En nuestro caso, ya que lo hemos definido con longitud 30, esta sería la 31.ª posición, lo que es lo mismo que decir **visitas[30]** (recordad que empezamos numerando por 0 y por eso la última posición tiene el índice 30).

En este caso, el **Array** se amplía automáticamente y tendrá una longitud de 31:

```
1  //Inicialmente hemos definido una tabla de 30 elementos
2  var visitas = Array (30);
3  console.log(visitas.length);
4  //Nos damos cuenta que el mes tiene 31 días y queremos añadir que
5  //día 31 ha tenido 400 visitas a la página web
6  visitas[30] = 400;
7  //Comprueba  que se ha añadido automáticamente una posición
8  console.log(visitas.length);
9  console.log(visitas[30]);
10
```

¿Y si hacemos lo mismo en la posición 33 (visitas[32])? Pues ahora también se amplía su longitud hasta 33, aunque visitas[31] la dejará vacía:

```
10
11 //Y, si llenamos por error la posición 33?
12 visitas[32] = 325;
13 console.log(visitas[32]);
14 //Comprobamos que la longitud se ha ampliado a 33
15 console.log(visitas.length);
16 //pero la posición 31 quedará undefined
17 console.log(visitas[31]);
18
```

Para inicializar la tabla de visitas mensuales a la página web a cero podemos utilizar la estructura iterativa:

```
1  var visitas = Array(30);
2  var i;
3  //Iteramos para recorrer todas las posiciones del Array
4  //e inicializamos los campos
5  for (i = 0; i <30; i++){ //Recordad: la primera posición es 0
6    visitas[i] = 0;
7    console.log(visitas[i]);
8  }
9
```

También podemos utilizar la propiedad predefinida del tipo **Array**, ***length***. Una propiedad es un atributo que se aplica a un tipo de datos y, por tanto, se puede consultar. Este es un concepto relacionado con el paradigma de la orientación a objetos, que se introduce en etapas posteriores del aprendizaje de la programación.

```
9
10 //Iteramos para recorrer todas las posiciones del Array
11 //e inicializamos los campos utilizando la propiedad
12 //que nos indica su longitud
13 for (i = 0; i <visitas.length; i++){ //Recordad: la primera posición es 0
14   visitas[i] = 0;
15   console.log(visitas[i]);
16 }
17
```

Esta propiedad predefinida aplicada a un **Array** devuelve la longitud del **Array**.

Tablas de varias dimensiones

La correspondencia de las **tablas de tablas** de la notación algorítmica **es el tipo Array multidimensionales.**

Ejemplo

Hemos visto que en **JavaScript** un **Array** puede contener cualquier objeto; sí, incluso otro **Array**.

Para almacenar las visitas recibidas a una web, durante un año, en una matriz, pues, podemos hacer:

```
1  //Definimos la matriz anual con 12 tablas de meses de 30 días
2  var visitasAnuales = Array(12);
3  var i;
4  for (i = 0; i <12; i++){
5    visitasAnuales[i] = Array(30); //Cada elemento del array de 12 meses es
6                                   //un array de 30 días
7  }
8
9  //Si queremos acceder a las visitas obtenidas el día 3 del mes 4
10 //mes abril está en la posición 3 i día 3 está en la posición 2
11 console.log(visiteaAnuales[3][4]); //retornará undefined porque aún
12                                    // no le hemos asignado ningún valor
13 //también podemos hacer aquí asignaciones de valores directamente:
14 //asignamos al día 3 de abril un total de 360 visitas
15  visitasAnuales[3][2] = 360;
16  console.log(visitasAnuales[3][2]); //ahora retornará 360
17
```

Podemos inicializar los elementos de la tabla a 0 utilizando la estructura iterativa.

```
//Definimos la matriz anual con 12 tablas de meses de 30 días
var visitasAnuales = Array(12);
var i, j;
for (i = 0; i <12; i++){
  visitasAnuales[i] = Array(30); //Cada elemento del array de 12 meses es
                                 //un array de 30 días
}

//Podemos inicializar todos los campos de la matriz a 0
for(i = 0; i<12; i++){
  for (j = 0; j<30; j++){
    visitasAnuales[i][j]=0;
    console.log(visitasAnuales[i][j]);
  }
}
```

Fijaos que el comportamiento del tipo **Array** en JS permite representar las tablas tal como las hemos definido algorítmicamente, pero también otras posibilidades que de momento no utilizaremos.

Podéis probar estos códigos en la web de PythonTutor.