

# Elección de estructuras

## Algoritmos y Estructuras de Datos 2

Departamento de Computación,  
Facultad de Ciencias Exactas y Naturales,  
Universidad de Buenos Aires

### Parte 1

#### Elección y justificación de estructuras

##### Enunciado

El enunciado se puede ver en las slides de la clase. Copiamos acá simplemente los requerimientos de complejidad para cada operación solicitada:

1. Agregar una persona nueva en  $O(\ell + \log n)$ .
2. Dado un código, borrar a la persona en  $O(\ell + \log n)$ .
3. Dado un código, encontrar los datos de la persona en  $O(\ell)$ .
4. Dado un DNI, encontrar los datos de la persona en  $O(\log n)$ .
5. Dada una edad, decir cuántos tienen esa edad en  $O(1)$ .
6. Decir cuántas personas estan en edad jubilatoria en  $O(1)$ .
7. Avanzar el día actual en  $O(m)$ .

Donde  $n$  es la cantidad de personas en el sistema,  $\ell$  es la longitud del código recibido como parámetro,  $m$  es la cantidad de personas que cumplen años el día al que se llega al avanzar el día y  $M$  es la edad máxima de una persona puede tener.

##### Esquema de solución

Lo aquí escrito **no** es una resolución completa y satisfactoria del ejercicio sino simplemente un esquema que permite llegar fácilmente a la misma (ver además los comentarios en la última hoja).

##### Estructura

- *porCódigo* Diccionario de código a **persona**, implementado sobre Trie.
- *porDNI* Diccionario de DNI en **persona**, implementado sobre árbol balanceado (AVL).
- *cantPorEdad* Arreglo de  $M$  posiciones que guarda en la posición  $i$  cuántas personas tienen  $i$  años actualmente.
- *cumplenEn* Arreglo de 365 posiciones, en cada posición tenemos el conjunto de personas que cumplen años ese día.
- *día* Entero con el día actual.
- *año* Entero con el año actual.
- *jubilados* Entero con la cantidad de jubilados.

padrón se representa con *estr*, donde

*estr* es tupla  $\langle$ *porCódigo*: diccTrie(string, persona)  
    *porDNI*: diccAVL(nat, persona)  
    *cantPorEdad*: arreglo\_dimensionable(nat)  
    *cumplenEn*: arreglo\_dimensionable(conjAVL(persona))  
    *día*: nat  
    *año*: nat  
    *jubilados*: nat $\rangle$

y *persona* es tupla  $\langle$ *nombre*: string, *código*: string, *dni*: nat, *día*: nat, *año*: nat $\rangle$

### Idea de los algoritmos

1. *p* es la persona a agregar  
    definir en *porCódigo* el código de la persona con *p* como definición  
    definir en *porDNI* el DNI de la persona con *p* como definición  
     $y \leftarrow$  edad actual de la persona  
     $\text{cantPorEdad}[y] \leftarrow \text{cantPorEdad}[y] + 1$   
    **if**  $y \geq 65$  **then**  
         $\text{jubilados} \leftarrow \text{jubilados} + 1$   
    **end if**  
     $d \leftarrow$  día de nacimiento de *p*  
    agregar *p* al conjunto *cumpleEn*[*d*]

Las actualizaciones de los diccionarios son  $O(\ell)$  y  $O(\log n)$  en ese orden y el agregado al conjunto en la última línea es  $O(\log n)$ . Todos los otros pasos son  $O(1)$ .

2. *p* es la persona a eliminar  
    borrar el código de *p* de *porCódigo*  
    borrar el dni de *p* de *porDNI*  
     $y \leftarrow$  edad actual de *p*  
     $\text{cantPorEdad}[y] \leftarrow \text{cantPorEdad}[y] - 1$   
    **if**  $y \geq 65$  **then**  
         $\text{jubilados} \leftarrow \text{jubilados} - 1$   
    **end if**  
     $d \leftarrow$  día de nacimiento de *p*  
    borrar *p* del conjunto *cumpleEn*[*d*]

Las actualizaciones de los diccionarios son  $O(\ell)$  y  $O(\log n)$  en ese orden y el borrado del conjunto en la última línea es  $O(\log n)$ . Todos los otros pasos son  $O(1)$ .

3. Busco el código de la persona en *porCódigo* ( $O(\ell)$ ).
4. Busco el dni de la persona en *porDNI* ( $O(\log n)$ ).
5. Accedo al arreglo *cantPorEdad* en  $O(1)$  y tengo exactamente el dato.
6. Tengo el dato *jubilados*.
7. actualizo la fecha actual con operaciones aritméticas  
    **for**  $p \in \text{porCumple}[x]$  **do**  
         $y \leftarrow$  edad actual de *p*  
         $\text{cantPorEdad}[y - 1] \leftarrow \text{cantPorEdad}[y - 1] - 1$   
         $\text{cantPorEdad}[y] \leftarrow \text{cantPorEdad}[y] + 1$   
        **if**  $y = 65$  **then**  
             $\text{jubilados} \leftarrow \text{jubilados} + 1$   
        **end if**  
    **end for**

El **for** itera por el conjunto de personas que cumplen el día nuevo (i.e., hace *m* iteraciones) y dentro del mismo todas las operaciones son  $O(1)$ .

## Parte 2

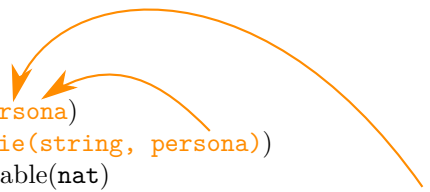
En la parte dos de esta clase extendemos el conjunto de restricciones, lo cual nos obliga a modificar la estructura y los algoritmos.

### Quitando la redundancia

En la estructura como está definida al principio de la solución, tenemos repetida la información de las personas en tres lugares distintos: *porCódigo*, *porDNI* y *cumplenEn*. Esto podría resolverse teniendo la información en un solo lugar y teniendo en el resto de los lugares iteradores. Por ejemplo:

patrón se representa con *estr*, donde

```
estr es tupla { porCódigo: diccTrie(string, persona)
               porDNI: diccAVL(nat, itDiccTrie(string, persona))
               cantPorEdad: arreglo_dimensionable(nat)
               cumplenEn: arreglo_dimensionable(diccAVL(nat, itDiccTrie(string, persona)))
               día: nat
               año: nat
               jubilados: nat }
```



En esta situación, buscar una persona por DNI implicaría hacer la búsqueda en *porDNI*, obtener un iterador a *porCódigo* y obtener el siguiente significado por referencia para poder devolverlo. Asimismo, al agregar una persona debo si o si primero agregarla en *porCódigo*, pedir que el *diccTrie* devuelva un iterador a la tupla insertada e insertar este iterador en *porDNI* y en *cumpleEn*.

Con esta implementación podría, por ejemplo, modificar la información de una persona utilizando como clave su dni en  $\mathcal{O}(\log n)$ ; ya que puedo buscar en *porDNI* en esa complejidad y obtener un iterador que me permite acceder a la persona por referencia y luego modificar su información. Como hay una sola copia de la persona, con esta actualización alcanza. En la versión anterior de la estructura, necesito actualizar cada copia, por lo que tengo que acceder a *porCódigo* y *porDNI*. La complejidad es entonces  $\mathcal{O}(\ell + \log n)$ .

**Para pensar:** ¿como modifico la estructura para poder modificar a una persona en  $\mathcal{O}(\ell)$  si accedo por código y  $\mathcal{O}(\log n)$  si accedo por dni?

### Borrado rápido

Considerando que los iteradores suelen permitir borrar el siguiente en  $\mathcal{O}(1)$  y que en esta estructura tengo iteradores del dni al trie, ¿puedo borrar una persona por DNI en  $\mathcal{O}(\log n)$ ?

En la versión anterior, borrar una persona implicaba buscarla en *porDNI* y *porCódigo*, dando una complejidad de  $\mathcal{O}(\ell + \log n)$ . Ahora tengo un iterador desde *porDNI* a *porCódigo*, por lo que una vez que lo encontré por dni en  $\mathcal{O}(\log n)$ , no necesito pagar el costo de buscar a la persona en *porCódigo*. ¿No debería poder borrar ese nodo del trie en  $\mathcal{O}(1)$ ?

La respuesta es no. Esto se debe a que además de borrar el significado del nodo del trie y potencialmente ese nodo (si es hoja), también hay que recorrer la rama hasta ese nodo de forma inversa para borrar cualquier prefijo que no tenga definición. Potencialmente esto toma  $\mathcal{O}(\ell)$ .

Y si tuviera los iteradores al revés (del Trie al AVL), ¿no puedo borrar una persona por código en  $\mathcal{O}(\ell)$ ?

Nuevamente, a pesar de que no tengo que pagar el costo de buscar a la persona en el AVL, si tengo que hacer las rotaciones del árbol para mantenerlo balanceado. Esto puede tomar a lo sumo  $\mathcal{O}(\log n)$  operaciones.

## Iteraciones salteadas

Queremos ahora iterar las edades importantes en  $\mathcal{O}(x)$ , siendo  $x$  la cantidad de años en los que cumple al menos una persona.

Cuando decimos *iterar* nos referimos a que la interfaz del módulo debe dar operaciones que permitan recorrer ese conjunto en la complejidad pedida. Si damos una solución con iteradores (que es lo esperado), la suma del costo de crear el iterador y avanzar el iterador todo lo necesario para llegar al final debe pertenecer a la clase de complejidad del requerimiento. Tener en cuenta que no se exige que se iteren en orden.

Con la estructura que tenemos hasta el momento, la única forma en que podemos recorrer el conjunto de edades con personas que tienen esa edad es recorriendo todo *cantPorEdad* y filtrando aquellos lugares donde el valor es 0. Esto es  $\mathcal{O}(M)$ . Tener en cuenta que si bien  $M$  está fijo, no es una constante, ya que puede cambiar de un padrón a otro. Para poder no recorrer todas las edades, deberíamos tener una estructura que solo tenga las válidas.

Esto podría ser una lista. Ahora, esta lista es necesario actualizarla cada vez que avanza un día, ya que puede haber edades nuevas o puede que haya que eliminar alguna edad. No podemos pagar el costo de buscar el nodo correspondiente en la lista, por lo que va a ser necesario tener punteros a los nodos. Proponemos la siguiente estructura:

padrón se representa con *estr*, donde

```
estr es tupla { porCódigo: diccTrie(string, persona)
               porDNI: diccAVL(nat, itDiccTrie(string, persona))
               cantPorEdad: arreglo_dimensionable(tupla(itLista(nat), nat))
               cumplenEn: arreglo_dimensionable(diccAVL(nat, itDiccTrie(string, persona)))

               cumpleValidos: lista(nat)
               día: nat
               año: nat
               jubilados: nat }
```

Para que esto funcione es necesario mantener actualizada la lista *cumpleValidos* de forma que cumpla el siguiente invariante: todo nat en la lista es una posición en *cumplenEn* y el diccionario en esa posición no es vacío y para toda posición en *cumpleEn* que no tiene un diccionario vacío, la posición está en *cumpleValidos*. Esta situación solamente cambia en dos situaciones: cuando avanza el día y cuando se agrega una persona.

Veamos el caso cuando avanza el día:

---

### Algorithm 1 avanzarDia

---

```
actualizar fecha
for all persona p que cumple años en la fecha actualizada buscado en cumplenEn do                                 $\triangleright \mathcal{O}(m)$ 
    it, cant = cantPorEdad[p.edad]                                 $\triangleright$  cant me lo pasan por referencia
    cant = cant - 1
    if cant == 0 then it.borrarSiguiente()
    end if
    it, cant = cantPorEdad[p.edad + 1]
    if cant == 0 then
        cantPorEdad[p.edad + 1].first = cumpleValidos.agregarAdelante(1)
        cant = 1
    else
        cant = cant + 1
    end if
    actualizar edad en persona
end for
```

---

Pensar el algoritmo para agregar una nueva persona.

## DNI's mayores o iguales

Queremos saber cuantos DNI's hay mayores o iguales a un DNI válido en  $\mathcal{O}(\log n)$ . Dada la inmediatez requerida, es necesario tener pre-calculado este valor. También es necesario poder actualizarlo rápido. Lo que se propone hacer acá es modificar el diccAVL usado en *porDNI*. Aprovechando que la estructura mantiene los DNI's ordenados en su estructura de árbol, en cada nodo podemos agregar un **nat** que nos diga cuantos DNI's hay mayores al DNI guardado ahí. O sea, el tamaño del sub-árbol derecho. De esta manera, poder responder la pregunta cuesta solo el tiempo de encontrar el nodo, que en un árbol AVL es  $\mathcal{O}(\log n)$ .

¿Que pasa con las actualizaciones? Este valor se modifica cada vez que se registra una nueva persona o cada vez que se la elimina. Cada vez que sucede esto se agrega o elimina un nodo del árbol. Una vez que se hacen alguna de esas operaciones, se puede recorrer desde ese nodo hacia arriba restando o sumando 1 a cada nat en los nodos camino a la raíz hasta el primer hijo izq.

*El caso de cantidad de nombres que tienen otro nombre como sufijos es análogo.*

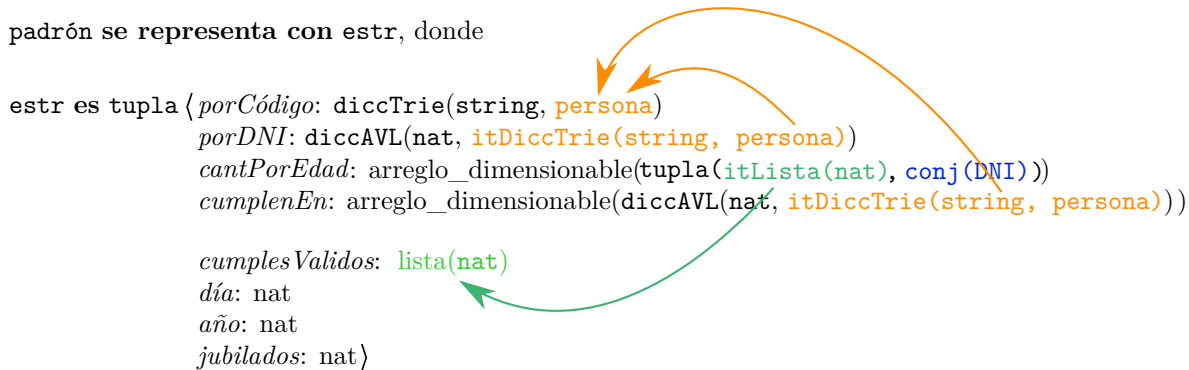
## DNI's por edad

Ahora también nos piden saber qué DNI's hay con una edad en particular en  $\mathcal{O}(1)$ .

El primer paso para esto es almacenar los DNI's por edad. Proponemos la siguiente estructura:

padrón se representa con **estr**, donde

```
estr es tupla {  
  porCódigo: diccTrie(string, persona)  
  porDNI: diccAVL(nat, itDiccTrie(string, persona))  
  cantPorEdad: arreglo_dimensionable(tupla(itLista(nat), conj(DNI)))  
  cumplenEn: arreglo_dimensionable(diccAVL(nat, itDiccTrie(string, persona)))  
  
  cumpleValidos: lista(nat)  
  día: nat  
  año: nat  
  jubilados: nat  
}
```



El conjunto agregado cumple con el invariante de contener todos los DNI's de personas que tienen esa edad. Como puedo llegar a él en  $\mathcal{O}(1)$  por la edad buscada y puedo devolverlo por referencia, esto resuelve el problema. El asunto ahora es mantener actualizado el conjunto.

El conjunto sufre modificaciones en dos situaciones, cuando se agrega o elimina una persona en el sistema y cuando una persona cumple años (al avanzar el día). El caso más complicado es el de avanzar un día. En ese caso hay que eliminar a la persona del conjunto donde estaba antes y agregarla al nuevo. Si consideramos que el conjunto en *cantPorEdad* es un conjunto sobre lista, podemos agregar en  $\mathcal{O}(1)$ . La dificultad está en eliminar. ¿Como resolvemos esto? Sí, ¡iteradores!. En este caso, como el conjunto se implementa sobre lista, sí podemos asumir que el iterador puede eliminar el siguiente en  $\mathcal{O}(1)$ .

Nos queda la siguiente estructura:

padrón se representa con *estr*, donde

```
estr es tupla { porCódigo: diccTrie(string, tupla(itConj(DNI) persona)
               porDNI: diccAVL(nat, itDiccTrie(string, persona))
               cantPorEdad: arreglo_dimensionable(tupla(itLista(nat), conj(DNI)))
               cumplenEn: arreglo_dimensionable(diccAVL(nat, itDiccTrie(string, persona)))

               cumpleValidos: lista(nat)
               día: nat
               año: nat
               jubilados: nat }
```

De esta forma, al actualizar la edad de una persona, tengo un iterador al conjunto que lo contenía por tener la edad anterior y puedo borrarlo en  $\mathcal{O}(1)$ .

Tener en cuenta que cuando se modifica la estructura cambiando un valor por una tupla, todos los accesos que antes usaban el valor directamente ahora también tienen que sacar el elemento de la tupla.

### Ya no hay más límite $M$

Se nos pide relajar la restricción de que las personas cumplen a lo sumo  $M$  años. A cambio, se modifican los siguientes requerimientos:

Estos cambios de restricciones implican un cambio respecto a lo presentado en clase. Este cambio responde a errores de enunciado.

- Agregar una persona debe tener costo  $\mathcal{O}(\ell + \log n + x)$
- En lugar de obtener los DNI para una edad en particular, se pide tener un iterador de tuplas (edad, conj(DNI)) que tenga como costo para iterar todas las edades con algún dni en  $\mathcal{O}(x)$ .
- No se pide más conocer la cantidad de dnis para una edad en particular ya que se obtiene del iterador anterior.

Este cambio de dominio nos dificulta la tarea ya que no podemos, al crear el padrón, crear un arreglo de tamaño  $M$  para tener acceso en  $\mathcal{O}(1)$  a información dada una edad. Podríamos crearlo de un tamaño arbitrario, pero habría que actualizarlo cada vez que una persona cumple más años que el máximo. En ese caso se nos hace imposible respetar la complejidad de *avanzarDia*. Lo mismo sucede en *agregarPersona*, ya que puede tener cualquier edad. Como consecuencia es que se relaja la restricción de obtener los dni por edad en  $\mathcal{O}(1)$ . No obstante sí debemos poder recorrer todas las edades con al menos un dni en  $\mathcal{O}(x)$ . Esto ya está resuelto en la última estructura. El problema que surge ahora es como encontrar la lista en la que debo agregar una persona cuando cumple años.

Proponemos la siguiente estructura:

padrón se representa con *estr*, donde

```
estr es tupla { porCódigo: diccTrie(string, tupla(itLista(conj(DNI)), itConj(DNI), persona))
               porDNI: diccAVL(nat, itDiccString(string, ...))
               cantPorEdad: lista(tupla(edad, conj(DNI)))
               cumplenEn: arreglo_dimensionable(diccAVL(nat, itDiccTrie(string, ...))
               día: nat
               año: nat
               jubilados: nat }
```

En esta estructura, *cantPorEdad* cumple con ser una lista donde los primeros elementos son las edades donde al menos una persona tiene esa edad y el conjunto que lo acompaña es el conjunto de DNIs de personas que tienen esa edad.

Además, la lista está ordenada por edad. Por otra parte, el primer elemento de los significados de *porCódigo* apunta a un nodo de *cantPorEdad* y el segundo elemento apunta a un nodo en el conjunto lineal dentro de *cantPorEdad* que acompaña a la edad de la persona.

Estos dos iteradores en los dos niveles de la estructura están para resolver el problema de ubicar a una persona en el conjunto correspondiente cuando cumple años. Dado que el invariante se cumple, para una persona, el primer *itLista* apunta al nodo de la edad que tiene. Cuando cumple años pasa a tener esa edad + 1. Esto significa que tiene que ir al nodo inmediatamente después (si es el de la edad + 1) o hay que crear un nodo nuevo (si es que no hay nadie con la edad + 1). En el primer caso, avanzo el iterador y agrego la persona en el conjunto del siguiente nodo. En el segundo, caso, dado que los iteradores en listas pueden permitir agregar un elemento adelante en  $\mathcal{O}(1)$ , puedo crear un nuevo nodo con la tupla  $\langle \text{edad} + 1, \{\text{dni}\} \rangle$ .

Finalmente, para mantener el invariante, debo eliminar el dni de la persona que cumplió años del conjunto donde estaba antes. Para eso uso el segundo iterador de la tupla, que permite eliminar en  $\mathcal{O}(1)$ . Luego debo actualizar ese iterador con el obtenido al agregar el dni al conjunto nuevo.