

# Elección de estructuras

## Iteradores

### Algoritmos y Estructuras de Datos 2

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

# Menú del día

Repaso clase anterior

Nuevos desafíos

# ¿Se acuerdan?

Nos encargaron implementar un PADRÓN que mantiene una base de datos de personas, con DNI, nombre, fecha de nacimiento y un código de identificación alfanumérico.

- ▶ El DNI es un entero (y es único).
- ▶ El nombre es un string.
- ▶ El código de identificación es un string (y es único).
- ▶ La fecha de nacimiento es un día de 1 a 365 (sin bisiestos) y un año.
- ▶ Sabemos además la fecha actual y por lo tanto la edad de cada persona (la cual sabemos que nunca supera los 200 años).

Además de poder agregar y eliminar personas del PADRÓN se desea poder realizar otras consultas en forma eficiente.

# Especificación

## TAD PADRON

### observadores básicos

fechaActual	: padron	→ fecha	
DNI	: padron	→ conj(DNI)	
nombre	: DNI $d \times$ padron $p$	→ nombre	$\{d \in \text{DNIs}(p)\}$
edad	: DNI $d \times$ padron $p$	→ nat	$\{d \in \text{DNIs}(p)\}$
código	: DNI $d \times$ padron $p$	→ código	$\{d \in \text{DNIs}(p)\}$

### generadores

crear	: fecha <i>hoy</i>	→ padron	
avanzDia	: padron $p$	→ padron	$\{\text{sePuedeAvanzar}(p)\}$
agregar	: persona $t \times$ padron $p$	→ padron	$\left\{ \begin{array}{l} \text{dni}(t) \notin \text{DNIs}(p) \wedge \text{código}(t) \notin \text{códigos}(p) \wedge \\ \text{nacimiento}(t) \leq \text{fechaActual}(p) \end{array} \right\}$
borrar	: DNI $d \times$ padron $p$	→ padron	$\{d \in \text{DNIs}(p)\}$

### otras operaciones

códigos	: padron	→ conj(código)	
persona	: código $c \times$ padron $p$	→ persona	$\{c \in \text{códigos}(p)\}$
tienenAños	: nat $\times$ padron	→ nat	
jubilados	: padron	→ nat	

Fin TAD

# Los requerimientos de complejidad temporal

Nos piden que respetemos las siguientes complejidades:

1. Agregar una persona nueva en  $O(\ell + \log n)$
2. Borrar una persona a partir de su código 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 están 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

# Donde estábamos ...

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 $\rangle$

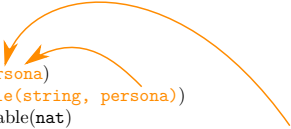
y *persona* **es** tupla  $\langle$ *nombre*: string, *código*: string,  
*dni*: nat, *día*: nat, *año*: nat  $\rangle$

# Sin repetidos

Con iteradores evitamos la repetición de datos y permitimos la modificación de una persona en  $O(\ell)$  si la búsqueda es por código o bien en  $O(\log n)$  si la búsqueda es por DNI.

padrón se representa con estr, donde

estr es tupla  $\langle$  *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  $\rangle$



## Extra I

- ▶ Iterar las edades importantes, estas son únicamente edades de personas en el sistema
  - ▶ Ej.: Si A tiene 20 años y B tiene 21 años, 20 y 21 son edades importantes, mientras que 22 no lo es
  - ▶ La complejidad de la operación debe ser en  $O(x)$ , donde  $x$  es la cantidad de años especiales
- 
- ▶ Al dar una solución con iteradores implica que la suma del costo de crear el iterador y avanzarlo todo lo necesario para llegar al final debe pertenecer a la clase de complejidad del requerimiento, es decir, en  $O(x)$
  - ▶ Tener en cuenta que no se exige que se iteren en orden

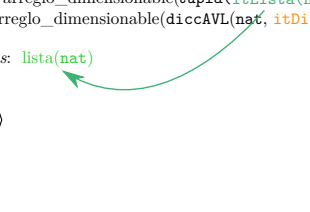


# Agregamos cumple válidos

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 }
```

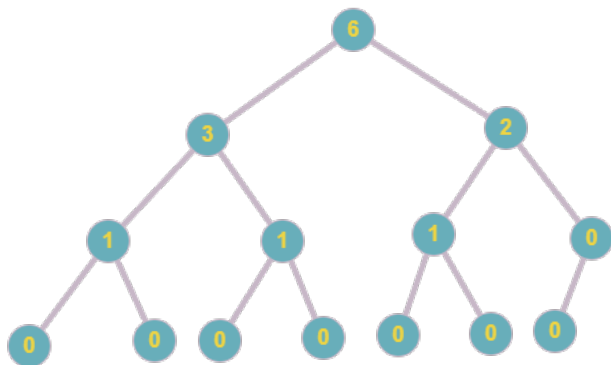


## Extra II

Queremos saber:

- ▶ Dado un DNI, cuantas personas hay con DNI mayor o igual  $O(\log n)$
- ▶ Dada una cadena  $\ell$ , cantidad de personas cuyo nombre empieza con ella  $O(|\ell|)$

Cada nodo tiene el tamaño de su subárbol derecho



## Extra III

Queremos saber:

- ▶ Dada una edad, saber qué DNIs tienen esa edad en  $O(1)$

Tener en cuenta las complejidades anteriores. En particular:

- ▶ Agregar una persona nueva en  $O(\ell + \log n)$ .
- ▶ Avanzar el día actual en  $O(m)$ .

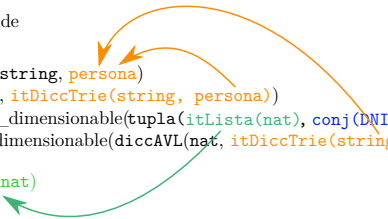
Siendo  $m$  es la cantidad de personas que cumplen años el día al que se llegan

# Reemplazamos el natural anterior por un conjunto

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)))

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

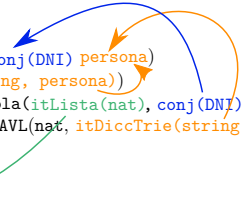


Acá tenemos la información almacenada, pero cómo pasamos de día?

# Reemplazamos el natural anterior por un conjunto

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



Cómo queda el algoritmo de pasar de día?

## Extra IV

- ▶ Relajamos el límite de edad de 200. Las edades no estarán más acotadas

## Extra IV

- ▶ Relajamos el límite de edad de 200. Las edades no estarán más acotadas
- ▶ El nuevo requerimiento de complejidad para agregar persona será  $O(\ell + \log n + x)$
- ▶ No se pide más conocer la cantidad de DNIs para una edad en particular, pero...



## Extra IV

- ▶ Relajamos el límite de edad de 200. Las edades no estarán más acotadas
- ▶ El nuevo requerimiento de complejidad para agregar persona será  $O(\ell + \log n + x)$
- ▶ No se pide más conocer la cantidad de DNIs para una edad en particular, pero...
- ▶ 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  $O(x)$

El desafío es avanzar el día actual en  $O(m)$

# Cant. por edad solo tendrá cumplidos válidos

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)⟩
```

# Resumen

- ▶ Aprovechamos al máximo cada estructura
- ▶ Ejercitamos el uso de iteradores en diversos contextos
- ▶ Con esto deberían poder terminar la práctica completa de diseño