Elección de estructuras

Algoritmos y Estructuras de Datos 2

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

¿Qué es elegir estructuras de datos?

- Elegir una estructura de representación.
  (Incluyendo Rep y Abs).
- Describir los algoritmos.
- Justificar que cumple con los requerimientos.
  (Ej. complejidad temporal).
- Explicitar los servicios usados.
  Justificar cómo podrían cumplirse.

Ejercicio: Padrón

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 largo del nombre está acotado por 20 caracteres.
- 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.

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	$\longrightarrow$	fecha	
DNIs	:	padron	$\longrightarrow$	conj(DNI)	
personaPorDNI	:	DNI d $\times$ padron p	$\longrightarrow$	persona	$\{d \in \text{DNIs}(p)\}$

generadores

crear	:	fecha *hoy*	$\longrightarrow$	padron	
avanzarDia	:	padron p	$\longrightarrow$	padron	
agregar	:	persona t $\times$ padron p	$\longrightarrow$	padron	$\left\{\begin{array}{l}\text{dni(t)} \notin \text{DNIs}(p) \land \text{código(t)} \notin \text{códigos}(p) \land \\ \text{nacimiento}(t) \leq \text{fechaActual}(p)\end{array}\right\}$
borrar	:	DNI d $\times$ padron p	$\longrightarrow$	padron	$\{d \in \text{DNIs}(p)\}$

otras operaciones

edad	:	DNI d $\times$ padron p	$\longrightarrow$	nat	$\{d \in \text{DNIs}(p)\}$
códigos	:	padron	$\longrightarrow$	conj(código)	
personaPorCodigo	:	código c $\times$ padron p	$\longrightarrow$	persona	$\{c \in \text{códigos(p)}\}$
jubilados	:	padron	$\longrightarrow$	nat	

Fin TAD

Las operaciones que nos piden

Nos piden que nos concentremos principalmente en las siguientes:

1. Agregar una persona nueva.
2. Dado un código, borrar a la persona.
3. Dado un código, encontrar todos los datos de la persona.
4. Dado un DNI, encontrar todos los datos de la persona.
5. Decir cuántas personas están en edad jubilatoria (i.e., tienen 65 años o más).

A trabajar...

Va una pequeña ayudita:

`persona` es `tupla` $\langle$*dni*: `nat`,
código: `string`,
nombre: `string`,
díaNac: `nat`,
añoNac: `nat`$\rangle$

`padron` **se representa con** `estr`, donde

`estr` es `tupla` $\langle$...,
...$\rangle$

Los requerimientos de complejidad temporal

Nos piden que respetemos las siguientes complejidades:

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. Decir cuántas personas están en edad jubilatoria en $O(1)$.

donde:

- n es la cantidad de personas en el sistema.
- ℓ es la longitud del código recibido cómo parámetro.

Recomendaciones

Algunas recomendaciones:

- Tener bien claro para qué sirve cada parte de la estructura y convencerse de que funciona antes de pensar los detalles más finos (Rep, Abs, algoritmos, etc).
- Esbozar los algoritmos y ver que las cosas más o menos cierren. Si algo no cierra, arreglarlo.
- El diseño es un proceso iterativo y suele involucrar prueba y error. No desalentarse si las cosas no cierran de entrada.
- Tener muy en cuenta los invariantes
  - ... de nuestra estructura, para no olvidarnos de mantenerlos.
  - ... de estructuras conocidas, para poder aprovecharlas.

Más operaciones y requerimientos

Además de lo anterior, nos piden que dada una edad, se pueda saber cuántas personas tienen esa edad con una complejidad temporal de $O(1)$. Sabiendo que la edad de las personas nunca supera los 200 años.

Más operaciones y requerimientos

Además de lo anterior, nos piden que dada una edad, se pueda saber cuántas personas tienen esa edad con una complejidad temporal de $O(1)$. Sabiendo que la edad de las personas nunca supera los 200 años.

- ¿Cómo sería si, en cambio, cuando se crea un padrón recibe como parámetro la edad máxima que puede tener una persona?

Más operaciones y requerimientos

Además de lo anterior nos piden que avanzar el día actual lo hagamos en $O(m)$, donde m es la cantidad de personas que cumplen años en el día al que se llega luego de pasar.

- ¿Qué agregamos?
- ¿Qué hace falta para mantenerlo?

A seguir pensando...

Repaso: iteradores

Si quiero recorrer todos los elementos contenidos en una estructura c voy a utilizar un iterador de la siguiente forma:

$it \leftarrow CrearIt(c)$
while $HaySiguiente(it)$ **do**
    $elem \leftarrow Siguiente(it)$
    $Avanzar(it)$
end while

Cada una de estas operaciones tienen un costo asociado y **no siempre es $\mathbf{O(1)}$**. La responsabilidad sobre el funcionamiento del iterador y sobre cómo se garantizan sus complejidades temporales es de ustedes a menos que sea un módulo de los *módulos básicos*.

Estructura

`padron` **se representa con** `estr`, donde

`estr` **es** `tupla` ⟨*porCódigo*: `diccTrie`(`string`, `persona`)
porDNI: `diccAVL`(`nat`, `persona`)
cantPorEdad: arreglo_dimensionable(`nat`)
cumplenEn: arreglo_dimensionable(`conjAVL`(`persona`))
día: nat
año: nat
jubilados: nat⟩

Algoritmos

function iBuscarPorDNI(in e: estr, in dni: nat) → res : Persona
    res ← *obtener*(*e.porDNI*, *dni*) ▷ $O(log\ n)$
end function

function iBuscarPorCódigo(in e: estr, in cod: string) → res : Persona
    res ← *obtener*(*e.porCodigo*, *cod*) ▷ $O(\ell)$
end function

- En ambos casos devuelvo a la persona por referencia no modificable.

Algoritmos

function iJubilados(in e: estr) → res : nat
    res ← *e.jubilados* ▷ $O(1)$
end function

function iTienenAnios(in e: estr, in edad: nat) → res : nat
    res ← *e.cantPorEdad*[*edad*] ▷ $O(1)$
end function

Algoritmos

function IAGREGAR(inout e: estr, in p: persona)
  definir(*e.porDNI*, *p.DNI*, *p*) ▷ $O(log\ n + copy(p))$
  definir(*e.porCodigo*, *p.Codigo*, *p*) ▷ $O(\ell + copy(p))$
  $y \leftarrow calcularEdad(p, e)$ ▷ $O(1)$
  if $y \geq 65$ **then** ▷ $O(1)$
    $e.jubilados \leftarrow e.jubilados + 1$
  end if
  $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] + 1$ ▷ $O(1)$
  Agregar(*e.CumplenEn*[*p.diaNac*], *p*) ▷ $O(log\ m + copy(p))$
end function

La complejidad del algoritmo es $O(log\ n + \ell + copy(p) + log\ m)$.
Notemos que:

- $m \leq n$ por lo cual $O(\log m) \leq O(\log n)$
- $O(copy(p)) = O(\ell)$

Por lo tanto la complejidad es $O(log\ n + \ell)$

Algoritmos

function iBorrar(inout e: estr, in cod: string)
  $p \leftarrow obtener(e.porCodigo, cod)$ ▷ $O(\ell)$. Devuelve por referencia
  $borrar(e.porDNI, p.DNI)$ ▷ $O(log\ n)$
  $y \leftarrow calcularEdad(p, e)$ ▷ $O(1)$
  if $y \geq 65$ **then** ▷ $O(1)$
    $e.jubilados \leftarrow e.jubilados - 1$
  end if
  $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] - 1$ ▷ $O(1)$
  $eliminar(e.CumplenEn[p.diaNac], p)$ ▷ $O(log\ m)$
  $borrar(e.porCodigo, cod)$ ▷ $O(\ell)$
end function

La complejidad del algoritmo es $O(log\ n + \ell + log\ m)$.
Notemos que:

- $m \leq n$ por lo cual $O(\log m) \leq O(\log n)$

Por lo tanto la complejidad es $O(log\ n + \ell)$

Algoritmos

```
function iAvanzarDia(inout e: estr)
    avanzarUnDia(e)                                              ▷ O(1)
    it ← CrearIt(e.cumpleEn[e.dia])                              ▷ O(1)
    while HaySiguiente(it) do                                    ▷ O(m)
        p ← Siguiente(it)
        y ← calcularEdad(p, e)                                   ▷ O(1)
        if y = 65 then                                           ▷ O(1)
            e.jubilados ← e.jubilados + 1
        end if
        e.cantPorEdad[y − 1] ← e.cantPorEdad[y − 1] − 1          ▷ O(1)
        e.cantPorEdad[y] ← e.cantPorEdad[y] + 1                  ▷ O(1)
        Avanzar(it)
    end while
end function
```

Las operaciones en azul son de un *Iterador* de *ConjAVL*. La interfaz de este iterador será similar a la interfaz del *ItConj* del módulo **Conjunto Lineal**. Luego, un conjunto implementado sobre AVL nos permite crear el iterador en $O(1)$ e iterar sobre todos sus elementos con costo total $O(cantidad\_de\_elementos)$. La complejidad del algoritmo es $O(m)$.

Algoritmos

```
function iAvanzarDia(inout e: estr)
    avanzarUnDia(e)                                              ▷ O(1)
    for p in e.cumpleEn[e.dia] do                                ▷ O(m)
        y ← calcularEdad(p, e)                                   ▷ O(1)
        if y = 65 then                                           ▷ O(1)
            e.jubilados ← e.jubilados + 1
        end if
        e.cantPorEdad[y − 1] ← e.cantPorEdad[y − 1] − 1          ▷ O(1)
        e.cantPorEdad[y] ← e.cantPorEdad[y] + 1                  ▷ O(1)
    end for
end function
```

El lenguaje de pseudocódigo nos permite ocultar las operaciones del iterador en un **for** . . . **in** . . . **do**. Debemos tener en cuenta que esto es sólo *azúcar sintáctico*, y que, a la hora de calcular la complejidad del algoritmo, debemos considerar las operaciones del iterador como en la diapositiva anterior.

Adicionales

¿Y si nos piden modificar la información de una persona utilizando como clave su DNI en O(log n) y utilizando como clave su código en $O(\ell)$?

Entonces:

- ¿Qué agregamos?
- ¿Qué hace falta para mantenerlo?

Estructura

`padron` **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(conjPersonas)
día: nat
año: nat
jubilados: nat⟩

donde conjPersonas es `conjAVL`(`itDiccTrie(string, persona)`)

donde itDiccTrie es un iterador que me permite acceder a la persona por referencia y modificar su información.

Adicionales

- ¿Y si nos piden que se pueda borrar una persona a partir de su DNI en $O(\log n)$?.
- ¿Y si nos piden que se pueda borrar una persona a partir de su código en $O(\ell)$?.

Extra I

- Iterar las *edades relevantes*, estas son únicamente edades de personas que están en el sistema. Esto significa que la interfaz del módulo debe dar operaciones que permitan recorrer todas las edades válidas.
- Ej.: Si A tiene 20 años y B tiene 21 años, 20 y 21 son edades relevantes, mientras que 22 no lo es.
- La complejidad de iterar todas las edades debe ser en $O(x)$, donde x es la cantidad de edades relevantes.

- 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 cumples válidos

`padron` **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(conjPersonas)
edadesRelevantes: lista(nat)
día: nat
año: nat
jubilados: nat⟩

donde conjPersonas es `conjAVL`(`itDiccTrie(string, persona)`)

Algoritmos

function IAGREGAR(inout e: estr, in p: persona) **(Versión anterior)**
    $ite \leftarrow definir(e.porCodigo, p.Codigo, p)$
    $definir(e.porDNI, p.DNI, ite)$
    $y \leftarrow calcularEdad(p, e)$
    if $y \geq 65$ **then**
        $e.jubilados \leftarrow e.jubilados + 1$
    end if
    $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] + 1$
    $Agregar(e.CumplenEn[p.diaNac], ite)$
end function

Algoritmos

function IAGREGAR(inout e: estr, in p: persona) **(Versión nueva)**
    $ite \leftarrow definir(e.porCodigo, p.Codigo, p)$
    $definir(e.porDNI, p.DNI, ite)$
    $y \leftarrow calcularEdad(p, e)$
    if $y \geq 65$ **then**
        $e.jubilados \leftarrow e.jubilados + 1$
    end if
    $Agregar(e.CumplenEn[p.diaNac], ite)$
    $t \leftarrow e.cantPorEdad[y]$ ▷ recibo la tupla por referencia.
    if $t.second = 0$ **then**
        $t.first \leftarrow e.edadesRelevantes.AgregarAdelante(y)$
    end if
    $t.second \leftarrow t.second + 1$
end function

Algoritmos

function IAVANZARDIA(inout e: estr) **(Versión anterior)**
    avanzarUnDia(*e*)
    for `p` **in** `e.cumpleEn[e.dia]` **do**
        $y \leftarrow$ *calcularEdad*(p, e)
        if $y = 65$ **then**
            $e.jubilados \leftarrow e.jubilados + 1$
        end if
        $e.cantPorEdad[y-1] \leftarrow e.cantPorEdad[y-1] - 1$
        $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] + 1$
    end for
end function

Algoritmos

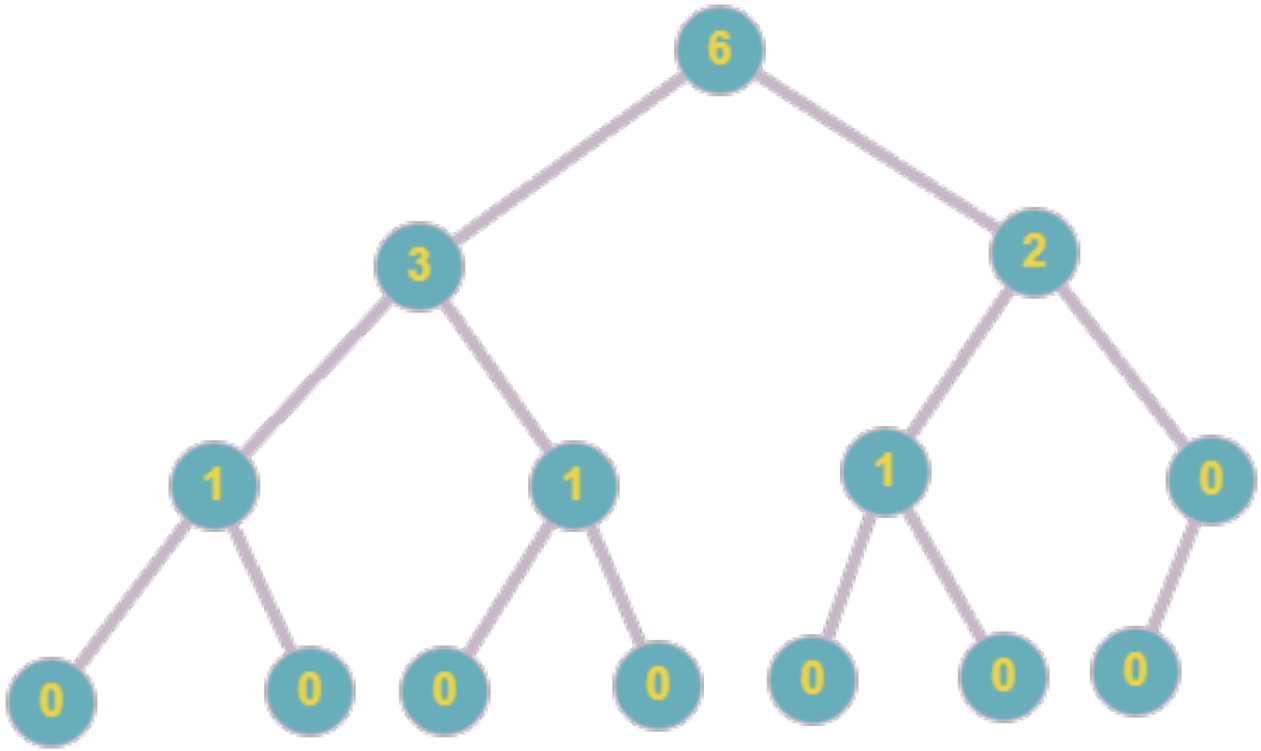
```
function iAvanzarDia(inout e: estr) (Versión nueva)
    avanzarUnDia(e)
    for p in e.cumpleEn[e.dia] do
        y ← calcularEdad(p, e)
        if y = 65 then
            e.jubilados ← e.jubilados + 1
        end if
        t ← e.cantPorEdad[y]
        t.second ← t.second − 1
        if t.second = 0 then
            t.second.EliminarSiguiente()
        end if
        t ← e.cantPorEdad[y + 1]
        if t.second = 0 then
            t.first ← e.edadesRelevantes.AgregarAdelante(y + 1)
        end if
        t.second ← t.second + 1
    end for
end function
```

Extra II

Queremos saber:

- Dado un DNI, cuantas personas hay con DNI mayor o igual $O(\log n)$

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 $O(\ell + \log n)$.
- Borrar una persona 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

`padron` **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(nat)))
cumplenEn: arreglo_dimensionable(conjPersona)
edadesRelevantes: lista(nat)
día: nat
año: nat
jubilados: nat⟩

donde conjPersona es `conjAVL`(`itDiccTrie(string,persona)`)

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

Reemplazamos el natural anterior por un conjunto

`padron` **se representa con** `estr`, donde

`estr` **es** `tupla` ⟨*porCódigo*: `diccTrie`(`string`, infoDniPersona)
porDNI: `diccAVL`(`nat`, itDiccTrie(string, infoDniPersona)
cantPorEdad: arreglo_dimensionable(tupla(itLista(nat), conj(nat)))
cumplenEn: arreglo_dimensionable(conjPersona)
edadesRelevantes: lista(nat)
día: nat
año: nat
jubilados: nat⟩

donde
conjPersona es `conjAVL`(itDiccTrie(string,infoDniPersona))
infoDniPersona es tupla(itConj(nat), persona)

Cómo queda el algoritmo de pasar de día?