Elección de estructuras

Algoritmos y Estructuras de Datos 2

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

Repaso

- En el etapa de diseño:
  - Nos ocupamos del **¿cómo?**
  - Lo plasmamos en *módulos de abstracción*
  - Ese **¿cómo?** se implementa usando el paradigma imperativo.

- Los módulos de abstracción tienen
  - Servicios exportados e interfaz (público)
  - Estructura de representación, Rep, Abs, algoritmos (privado)
  - Justif. de complejidades (esto también es privado) (¿por qué?)
  - Servicios usados (esto también es privado) (¿por qué?)

¿Qué es elegir estructuras de datos?

- Estructura de representación y algoritmos
  - Elegir un buen combo para cumplir los requerimientos (e.g., complejidad temporal)
- Justificación de complejidades
  - Podemos basarnos en estructuras conocidas (e.g., AVL, Trie, etc.)
- Servicios usados
  - Exigir requisitos cumplibles (e.g., justificar la cumplibilidad pedida)

¿Qué insumos tenemos?

- Apunte de diseño (para saber qué y cómo escribir)
- Estructuras vistas en la teórica (módulos incompletos)
- Apunte de módulos básicos (módulos completos)
- Nuestra experiencia en programación (hoy esperamos agregar algunos trucos a nuestra carpeta de elección de estructuras).
- Próximamente tendremos también algoritmos de ordenamiento y otras técnicas algorítmicas (i.e., “dividir y conquistar”).

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)	
nombre	:	DNI d $\times$ padron p	$\longrightarrow$	nombre	$\{d \in \text{DNIs}(p)\}$
edad	:	DNI d $\times$ padron p	$\longrightarrow$	nat	$\{d \in \text{DNIs}(p)\}$
código	:	DNI d $\times$ padron p	$\longrightarrow$	código	$\{d \in \text{DNIs}(p)\}$
díaCumple	:	DNI d $\times$ padron p	$\longrightarrow$	nat	$\{d \in \text{DNIs}(p)\}$

generadores

crear	:	fecha *hoy*	$\longrightarrow$	padron	
avanzDia	:	padron p	$\longrightarrow$	padron	$\{\text{sePuedeAvanzar}(p)\}$
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

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

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 en recontra-pseudo-código 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.

A trabajar...

Va una pequeña ayudita:

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

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

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

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 $O(1)$

Pensemos que estructura elegir si:

1. Sabemos que la edad de las personas nunca supera los 200 años.
2. Cuando se crea un padrón además de las fecha, recibe como parámetro la edad máxima que puede tener una persona. (Abría que modificar el TAD también).

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...

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) $\rightarrow$ res : nat
    $res \leftarrow e.jubilados$ ▷ $O(1)$
end function

function iTienenAnios(in e: estr, in edad: nat) $\rightarrow$ res : nat
    $res \leftarrow e.cantPorEdad[edad]$ ▷ $O(1)$
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
  $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] + 1$ ▷ $O(1)$
  $Agregar(e.CumplenEn[p.dia], p)$ ▷ $O(log\ m + copy(p))$
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
  $borrar(e.porDNI, p.DNI)$ ▷ $O(log\ n + borrar(p))$
  $y \leftarrow calcularEdad(p, e)$ ▷ $O(1)$
  if $y \geq 65$ **then** ▷ $O(1)$
    $e.jubilados \leftarrow e.jubilados - 1$
  end if
  $e.cantPorEdad[y] \leftarrow e.cantPorEdad[y] - 1$ ▷ $O(1)$
  $eliminar(e.CumplenEn[p.dia], p)$ ▷ $O(log\ m + borrar(p))$
  $borrar(e.porCodigo, cod)$ ▷ $O(\ell + borrar(p))$
end function

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

- $m \leq n$ por lo cual $O(\log m) \leq O(\log n)$
- $O(\text{borrar(p)}) = O(\ell)$

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

Algoritmos

```
function iAvanzarDia(inout e: estr)
    avanzarUnDia(e)                                              ▷ O(1)
    for p in e.cumpleEn[e.dia] do                                ▷ O(log 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
```

Entonces la complejidad del algoritmo es $O(\log m)$

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?

Quitando redundancia - Iteradores

- Tenemos repetida la información de las personas en tres lugares distintos: porCódigo, porDNI y cumplenEn.
- Podríamos quitar la redundacia teniendo la información en un solo lugar y teniendo en el resto de los lugares iteradores.
- Esto nos permite mejorar la complejidad de modificar una persona.

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 tuvieramos los iteradores al revés (del trie al AVL), ¿no puedo borrar una persona a partir de su código en $O(\ell)$?.