

Elección de estructuras

Algoritmos y Estructuras de Datos 2

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

Menú del día

- 1 Repaso de módulos básicos
- 2 Algunas estructuras básicas que necesitamos
- 3 Ejercitación general

¿Qué es elegir estructuras de datos?

- En el etapa de diseño: ¿**cómo** implementamos las estructuras que definimos en los TADs?
- Vamos a diseñar **módulos**
- Parte pública: lo que el usuario del módulo puede ver. Lo más importante es la **interfaz**, es decir, las operaciones que dispone.
Por cada operación tenemos:
 - Signatura
 - Pre/post condición
 - Complejidad
 - Descripción
 - Aspectos de aliasing

¿Qué es elegir estructuras de datos?

- Parte privada: lo que no nos interesa que vea el usuario
 - Estructura de representación
 - Invariante de representación / función de abstracción
 - Justificación de complejidades
 - Algoritmos (podemos basarnos en las estructuras elegidas)
 - Servicios usados (esto también es privado, ¿por qué?)

Fuentes de consulta

- 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

¿Qué estructuras vimos en la materia?

Complejidades de algunas operaciones (peores casos)

Pensar contextos de uso y memoria empleada

	Indexar	Buscar	Insertar	Eliminar
Array	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Lista enlazada	-	$O(n)$	$O(1)$	$O(1)$
ABB	-	$O(n)$	$O(n)$	$O(n)$
AVL	-	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tries (*)	-	$O(n *m)$	$O(n *m)$	$O(n *m)$

(*) La complejidad corresponde a la versión naif de la estructura, siendo m la cantidad de elementos por nivel

Unificando

- ¿Cómo relacionamos nuestros TADs con las estructuras anteriores?
- Nuestro módulo implementará un TAD y en su estructura de representación utilizaremos una o varias estructuras de las mencionadas que explicarán las complejidades de la interfaz
- Siempre debemos mantener el pre y la post condición

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 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 hoy	→ 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

Requerimientos

Nos piden que nos concentremos en las siguientes operaciones con sus respectivas 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 Dada una edad, decir cuántos tienen esa edad en $O(1)$
- 6 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 como 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 cierran. 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...

persona es tupla $\langle \text{dni: nat},$
 código: string,
 nombre: string,
 día: nat,
 año: nat \rangle

padron **se representa con** estr, donde

estr es tupla $\langle \dots,$
 $\dots \rangle$

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...

Hasta ahora...

- *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

Bien escrito quedaría:

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


Evitemos información repetida

- ¿Qué sucede cuando actualizamos los datos de una persona?
- ¿Cómo podemos evitar repetir información?

Agregamos iteradores

padrón 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}
```



Adicionales

¿Y si nos piden mejorar alguna de las siguientes complejidades?

- Poder borrar una persona a partir de su código en $O(\ell)$
- Poder borrar una persona a partir de su DNI en $O(\log n)$

Entonces:

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

Resumen

Deberían poder contestar las siguientes preguntas:

- ¿qué es un módulo? ¿qué partes lo componen?
- ¿por qué hay partes privadas y partes públicas?
- ¿qué estructuras vimos en la materia?
¿en qué contexto las utilizarían?
- ¿pudieron entender la estructura del problema planteado?
- ¿en qué nos puede ayudar el uso de iteradores?