Diseño I: Elección de estructuras

Clase práctica

Algoritmos y Estructuras de Datos II

Diferencias entre especificación y diseño

- En la etapa de especificación:
  - Nos ocupamos del '¿Qué?'.
  - Lo explicamos usando TADs.
  - Ese '¿Qué?' se explica desde una perspectiva denotacional, usando lógica de primer orden y axiomas recursivos.

- En la etapa de diseño:
  - Nos ocupamos del '¿Cómo?'.
  - Lo explicamos con *módulos de abstracción*.
  - Ese '¿Cómo?' lo explicaremos desde una perspectiva operacional, usando un lenguaje de programación imperativo.
  - Tenemos un *contexto de uso* que nos fuerza a tomar decisiones respecto de la estructura.

Partes de un módulo

Un módulo de abstracción se divide en tres secciones:

1. Interfaz: sección accesible a los usuarios del módulo (ej. otros módulos) que detalla cada operación exportada:
   - Signatura: qué recibe y qué devuelve.
   - Precondición: condiciones requeridas sobre los datos de entrada.
   - Postcondición: condiciones que se prometen sobre la salida.
   - Descripción (¡importante!).
   - Complejidad.
   - Aspectos de *aliasing*.

2. Representación: sección no accesible a los usuarios del módulo que detalla la estructura de representación y los algoritmos justificando las complejidades deseadas. Define también *Rep* y *Abs*.

3. Servicios usados: detalla las suposiciones que se hacen sobre los servicios usados de otros módulos. Estos requisitos justifican las complejidades de la sección anterior (que a su vez justifican las de la interfaz).
   - Ejemplo: cola implementada sobre una lista enlazada.

Apuntes

- No se olviden de los apuntes:
  - Apunte de diseño.
  - Apunte de módulos básicos (módulos completos).
- Veamos un poco estos apuntes...
- En general no les vamos a pedir módulos completos.
- Sólo por hoy: diseñar un módulo (casi) completo.

Ejercicio: ¡Subite!

Sistema de tarjetas de transporte para el subte:

- En las boleterías, se venden tarjetas de 1, 2, 5, 10 y 30 viajes.
- En la entrada a los andenes, hay molinetes con lectores de tarjetas.
- Cuando el usuario pasa su tarjeta por el lector:
  - Si la tarjeta está agotada o es inválida, se informa de esto en el visor y no se abre el molinete.
  - Si hay viajes en la tarjeta.
    - Se abre el molinete.
    - Se muestra el saldo restante en el visor.
    - Se actualiza en el sistema el nuevo saldo para esa tarjeta.
    - Se registra en el sistema el día y la hora del acceso.

- Nos piden tiempo de acceso sublineal (en la cantidad total de tarjetas y en peor caso) para la operación de usar la tarjeta.

TAD

TAD SUBITE

observadores básicos

tarjetas : subite $\longrightarrow$ conj(tarjeta)

crédito : subite s $\times$ tarjeta t $\longrightarrow$ nat $\quad \{t \in tarjetas(s)\}$

viajes : subite s $\times$ tarjeta t $\longrightarrow$ secu(FechaHora) $\quad \{t \in tarjetas(s)\}$

generadores

Crear : $\longrightarrow$ subite

NuevaTarjeta : subite s $\times$ nat c $\longrightarrow$ subite $\quad \{c \in \{1,2,5,10,30\}\}$

UsarTarjeta : subite s $\times$ tarjeta t $\times$ fechaHora $\longrightarrow$ subite

$\{t \in tarjetas(s) \wedge_L crédito(s,t) > 0\}$

axiomas

...

Fin TAD

Diseñanado...

¿Qué hacemos ahora?

- ¿Elegir estructura o definir la interfaz?
  ↪ La interfaz será la misma sin importar la estructura, y además nos puede dar ideas para esta última...

- ¿Hay que pensar cuáles serán las operaciones provistas por la interfaz?
  ↪ ¡Sí! No siempre salen directamente del TAD...

- Definamos entonces las operaciones y para cada una definir:
  - **Descripción**
  - Precondición
  - Postcondición
  - Complejidad (!)
  - Aliasing (!)

- ¿Cómo elegimos entonces las operaciones?
  ↪ Depende de muchas cosas (principalmente el contexto de uso).

Diseñanado...

Antes que nada arrancamos con la cabecera:

- **Módulo**: Subite
- **Se explica con**: TAD SUBITE
- **Géneros**: subite
- **Operaciones**: ...

Subite: Creación

Crear : $\longrightarrow$ subite

- Signatura: $\textsc{Crear}() \rightarrow$ res: subite
- Pre: { true }
- Post: { res = Crear() } (¿sombrerito?)
- Descripción: Crea una nueva instancia
- Complejidad: La definiremos más adelante.
- Aliasing: (no aplica)

Subite: Nuevas tarjetas

NuevaTarjeta : subite s $\times$ nat c $\longrightarrow$ subite $\{c \in \{1,2,5,10,30\}\}$

Algunas preguntas antes de pensar en la signatura...

- Una vez que creamos nuestras tarjetas, ¿cómo hace el usuario para pedir información sobre las tarjetas creadas?
- ¿Tendría sentido pensar en una operación TARJETAS?
- ¿Devolvería las mismas por referencia o por copia?
- ¿Qué problemas tienen ambas opciones?

Idea: Que NuevaTarjeta devuelva la tarjeta recién creada.

Subite: Nuevas tarjetas

NuevaTarjeta : subite s $\times$ nat c $\longrightarrow$ subite $\{c \in \{1,2,5,10,30\}\}$

- Signatura: NUEVATARJETA(**inout** s: subite, **in** c: nat) $\rightarrow$ res: tarjeta
- Pre: $\{\ s = s_0 \wedge c \in \{1, 2, 5, 10, 30\}\ \}$
- Post: $\{\ s = \mathit{NuevaTarjeta}(s_0, c) \wedge_{\mathrm{L}}$
  $\mathit{res} = \mathit{dameUno}(\mathit{tarjetas}(s) - \mathit{tarjetas}(s_0))\ \}$
- Descripción: Agrega una nueva tarjeta al sistema y la retorna.
- Complejidad: La definiremos más adelante.
- Aliasing: (E.g) Se devuelve una referencia no modificable a la nueva tarjeta.

Subite: Usar tarjeta

UsarTarjeta : subite s $\times$ tarjeta t $\times$ fechaHora $\longrightarrow$ subite

$\{\text{t} \in \text{tarjetas(s)} \wedge_{\text{L}} \text{crédito(s,t)} > 0\}$

Supongamos que el contexto de uso nos indica lo siguiente:

- Queremos que UsarTarjeta indique si la misma era válida y tenía crédito (en lugar de restringir su uso).
- Y además, de usarla, que nos devuelva el nuevo saldo de la tarjeta.

¿Podemos alterar la signatura para mejorar la usabilidad del módulo?

Subite: Usar tarjeta

UsarTarjeta : subite s $\times$ tarjeta t $\times$ fechaHora $\longrightarrow$ subite

$\{\text{t} \in \text{tarjetas(s)} \wedge_{\text{L}} \text{crédito(s,t)} > 0\}$

- Signatura:
  USARTARJETA (**inout** s: subite, **in** t: tarjeta,
  in h: fechaHora, **out** c: nat) $\rightarrow$ res: bool
- Pre: $\{ s = s_0 \}$
- Post: $\{ res = (t \in tarjetas(s_0) \wedge_{\text{L}} \text{crédito}(s_0, t) > 0) \wedge_{\text{L}}$
  $(res \Rightarrow_{\text{L}} s = UsarTarjeta(s_0, t, h) \wedge_{\text{L}} \text{c} = \text{crédito}(s, t)) \wedge (\neg res \Rightarrow s = s_0) \}$
- Descripción: Si la tarjeta pasada por parámetro está registrada y todavia tiene crédito, entonces registra el uso y retorna true
- Complejidad: tiene que ser $< O(n)$, pero lo precisamos luego.
- Aliasing: idem...

Subite: Ver viajes y crédito

$$\begin{array}{lll} \text{viajes} : \text{subite } s \times \text{tarjeta } t \longrightarrow \text{secu(fechaHora)} & & \{\text{t} \in \text{tarjetas(s)}\} \\ \text{crédito} : \text{subite } s \times \text{tarjeta } t \longrightarrow \text{nat} & & \{\text{t} \in \text{tarjetas(s)}\} \end{array}$$

Podemos hacer algo parecido para ver los viajes y el crédito....

- Ver viajes:
  - VIAJES(**in** s: subite, **in** t: tarjeta, **out** vs: secu(fechaHora)) $\rightarrow$ res: bool
  - Pre: { true }
  - Post: { $(res = t \in tarjetas(s)) \wedge_{\mathrm{L}} (\text{res} \Rightarrow_{\mathrm{L}} \text{vs} = \text{viajes}(s, t))$ }
  - Descripción: Si la tarjeta pasada por parámetro está registrada, retorna true y asigna en *viajes* los viajes realizados
  - Complejidad: La definimos más adelante
  - Aliasing: (E.g) Se devuelve una referencia no modificable a la lista de viajes realizados.

Subite: Ver viajes y crédito

viajes : subite s $\times$ tarjeta t $\longrightarrow$ secu(fechaHora) $\{t \in \text{tarjetas}(s)\}$
crédito : subite s $\times$ tarjeta t $\longrightarrow$ nat $\{t \in \text{tarjetas}(s)\}$

Podemos hacer algo parecido para ver los viajes y el crédito....

- Ver crédito:
  - CRÉDITO(**in** s: subite, **in** t: tarjeta, **out** cred: nat) $\rightarrow$ res: bool
  - Pre: { true }
  - Post: { $(res = t \in tarjetas(s)) \wedge_{\text{L}}$ (res $\Rightarrow_{\text{L}}$ cred = crédito(s, t)) }
  - Descripción: Si la tarjeta pasada por parámetro está registrada, retorna true y asigna el crédito en *cred* si no retorna false
  - Complejidad: La definimos más adelante
  - Aliasing: -

Subite: Elección de estructuras

- Elijamos la estructura, sabiendo que la única restricción de complejidad que tenemos es en utilizar la tarjeta ($< O(n)$)
- Subite se podría representar con `estr`, donde:
  - `estr` es `dicc(tarjeta, datosTarjeta)`
  - `tarjeta` es `nat`
  - `datosTarjeta` es `tupla`$\langle$
    `crédito: nat,`
    `viajes: secuencia(fechaHora)`$\rangle$
- ¿Así cumple con las complejidades requeridas?
- Recordemos que utilizar la tarjeta implica:
  - **Buscar la tarjeta**
  - Descontarle el saldo
  - Agregarle un viaje
- Escribamos el algoritmo y veamos...

Subite: Algoritmo de UsarTarjeta

```
function iUsarTarjeta(inout e: estr, in t: tarjeta, in h: fechaHora, out crédito: nat)
    if ¬definido(e, t) then                                          ▷ O(definido)
        res ← false
    else
        datos ← obtener(e, t)                                        ▷ O(obtener)
        if datos.credito = 0 then
            res ← false
        else
            datos.credito ← datos.credito − 1
            agregar(datos.viajes, h)                                 ▷ O(agregar)
            credito ← datos.credito
            res ← true
        end if
    end if
    devolver res
end function
```

Ejercicio: Requisitos de complejidad

- Ok, entonces *definido*, *obtener* y *agregar* tienen que ser sublineales...
- Para el agregar, si implementamos la secuencia con una lista enlazada, el agregar al final cuesta O(1).
- ¿Y el diccionario?
  - Vectores de tuplas
  - Tabla de Hash (lo van a ver en la teórica y en el labo):
    Permite buscar «casi» en $O(1)$
  - Árbol ABB:
    Si agregan con una distribución uniforme permite buscar en $O(log\ n)$
  - Árbol AVL:
    Permite buscar en $O(log\ n)$
- Si elegimos implementar la secuencia con una lista enlazada y el diccionario con un árbol AVL, entonces... touché!

Subite: Algoritmo de UsarTarjeta (con compliejidades)

```
function iUsarTarjeta(inout e: estr, in t: tarjeta, in h: fechaHora, out crédito: nat)
    if ¬definido(e, t) then                                          ▷ O(log n)
        res ← false
    else
        datos ← obtener(e, t)                                        ▷ O(log n)
        if datos.credito = 0 then
            res ← false
        else
            datos.credito ← datos.credito − 1
            agregar(datos.viajes, h)                                 ▷ O(1)
            credito ← datos.credito
            res ← true
        end if
    end if
    devolver res
end function                                                         ▷ T(n) ∈ O(log n)
```

Subite: estructura elegida

Finalmente:

- `subite` se representa con `estr`, donde:
  - `estr` es `diccLog(tarjeta, datosTarjeta)`
  - `tarjeta` es `nat`
  - `datosTarjeta` es `tupla`⟨
    `crédito: nat,`
    `viajes: listaEnlazada(fechaHora)`⟩
- Y donde `diccLog(a,b)` implementa un diccionario que provee complejidades logarítmicas para búsqueda, inserción y borrado (por ejemplo, podría estar definido sobre un árbol binario de búsqueda balanceado con invariante de AVL).
- Deberíamos diseñar `diccLog`, pero eso lo damos por hecho... ;-)

- Restan completar el resto de los algoritmos...
- Resta escribir el invariante de representación y la función de abstracción (queda de tarea)

Subite: adicionales para discutir (1)

Supongamos que el TAD Subite especificaba también la siguiente operación:

tarjetaMasUsada : subite $\longrightarrow$ tarjeta $\{ \neg\emptyset(\text{tarjetas}(t)) \}$

Observación: si hay más de una posible respuesta, devuelve la que se usó por último.

- ¿Podemos responder esto con nuestra estructura?
- ¿Qué complejidad tendría esa operación?
- ¿Se puede hacer mejor? ¿Se podría hacer en $O(1)$? ¡Ojo! Sin empeorar las otras operaciones...

Subite: adicionales para discutir (1)

Supongamos que el TAD Subite especificacba también la siguiente operación:

tarjetaMasUsada : subite $\longrightarrow$ tarjeta $\{ \neg\emptyset(\text{tarjetas}(t)) \}$

Observación: si hay más de una posible respuesta, devuelve la que se usó por último.

- Una opción posible:
  - `estr es tupla⟨`
    `tarjetas: diccLog(tarjeta, datosTarjeta),`
    `masUsada: tarjeta ⟩`
  - `tarjeta es nat`
  - `datosTarjeta es tupla⟨`
    `crédito: nat,`
    `viajes: listaEnlazada(fechaHora)⟩`
- ¿Cómo serían los algoritmos?

Fin