Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

5. Tipos Abstractos de Datos I

Stacks, Queues y Sets

Repaso

Tipos algebraicos

- Los tipos algebraicos son tipos nuevos
  - Se definen a través de ***constructores***
    - Los constructores pueden llevar argumentos
    - Cada constructor expresa a un grupo de elementos
  - Se acceden mediante ***pattern matching***
    - Los *constructores* se usan para preguntar

Tipos algebraicos

- Los tipos algebraicos se clasifican en
  - enumerativos
    - varios constructores sin argumentos (e.g. **`Direccion`**)
  - registros (o productos)
    - un único constructor con varios argumentos (e.g. **`Persona`**)
  - sumas (o variantes)
    - varios constructores con argumentos (e.g. **`Helado`**)
  - recursivos
    - suma que usa el mismo tipo como argumento (e.g. **`Listas`**)

Recursión estructural

- Las definiciones recursivas estructurales
  - Tienen un caso por cada constructor
  - En el recursivo, usan *la misma función* en la parte recursiva
  - Solo hace falta pensar cómo agregar lo que falta
- La metodología propuesta requiere seguir ciertos pasos
  - Decidir si usar recursión y *sobre qué estructura*
  - Plantear el esquema
  - Resolver los casos recursivos
  - Completar con los casos base

Tipos abstractos de datos: motivación

Limitaciones de los tipos algebraicos

- ¿Qué limitaciones tienen los tipos algebraicos?
  - ¿Qué sucede si cambio la estructura del tipo algebraico?
    - Ej. `Helado`
  - ¿Cómo forzar ciertas formas de uso del tipo de datos?
    - Ej. `Persona`
  - ¿Cómo puedo ocultar información sobre el tipo?

Limitaciones de los tipos algebraicos

- Los tipos algebraicos no sirven para todo
  - El código es muy dependiente de la estructura
    - Y por ello, es complicado cambiar la estructura
    - Ej. `Helado`
  - No se puede restringir el uso de constructores
    - Se pueden construir cosas inválidas
    - Ej. `Persona`
- Precisamos alguna herramienta nueva

Tipos abstractos de datos

- Los tipos abstractos de datos (TADs) son una forma de definir tipos, con las siguientes características
  - Se definen mediante una ***interfaz*** de operaciones
    - nombre y tipo
    - comportamiento esperado

    } contrato
  - *No* se conoce *cómo* se implementan las operaciones
    - O sea, *oculta la representación interna* al usuario
- ¿Cómo definirlos en Haskell?

Tipos abstractos de datos en Haskell

Módulos en Haskell

- Los TADs en Haskell utilizan el mecanismo de *módulos*
- Un ***módulo*** es un grupo de definiciones que tiene
  - Nombre, y una lista de las operaciones que son visibles
  - Implementación de cada una de las operaciones y otras auxiliares necesarias
  - Opcionalmente, indica qué otros módulos utiliza
- Sintaxis

```
module <nombre> (<interfaz>) where
    <implementación>
```

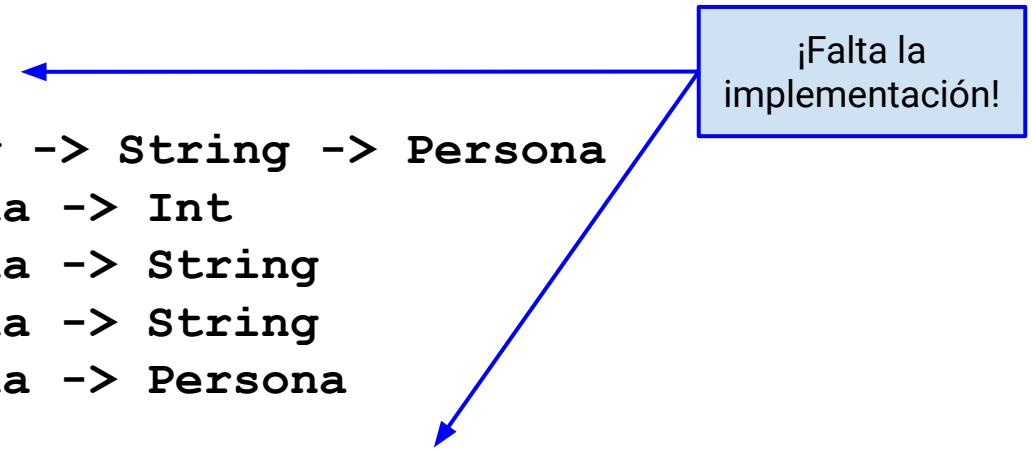
Módulos en Haskell

- Ejemplo: TAD Persona

```
module Persona
   (Persona, nacer, edad, nombre, apellido, crecer)
 where

data Persona = …

nacer    :: String -> String -> Persona
edad     :: Persona -> Int
nombre   :: Persona -> String
apellido :: Persona -> String
crecer   :: Persona -> Persona
…
```



Módulos en Haskell

```
data Persona

nacer     :: String -> String -> Persona
edad      :: Persona -> Int
nombre    :: Persona -> String
apellido  :: Persona -> String
crecer    :: Persona -> Persona
```

- La interfaz es la única manera de usar un elemento
- Así, define TODAS las formas posibles de uso
- No hace falta conocer la implementación para usarlo

```
import Persona

ff :: Persona
ff = crecerVeces 53 (nacer "Fidel" "ML")

crecerVeces :: Int -> Persona -> Persona
crecerVeces 0 p = p
crecerVeces n p = crecer (crecerVeces (n-1) p)
```

Módulos en Haskell

```
data Persona

nacer    :: String -> String -> Persona
edad     :: Persona -> Int
nombre   :: Persona -> String
apellido :: Persona -> String
crecer   :: Persona -> Persona
```

- Observaciones
  - Cada módulo va en su propio archivo
  - El nombre del archivo es igual al del módulo
  - Para usar un módulo debe usarse **`import`**
  - Solamente pueden usarse los elementos de la interfaz
- ***Para conocer un TAD alcanza con conocer su interfaz***
  - Pueden definirse muchas operaciones con las operaciones de interfaz, SIN conocer la implementación

Módulos en Haskell

```
data Persona

nacer    :: String -> String -> Persona
edad     :: Persona -> Int
nombre   :: Persona -> String
apellido :: Persona -> String
crecer   :: Persona -> Persona
```

- Una vez más:
  - *La interfaz es la única manera de usar un elemento*

```
import Persona

nacerMuchas :: [(String,String)] -> [Persona]
nacerMuchas []           = []
nacerMuchas ((n,a):nas) = nacer n a : nacerMuchas nas

nombres :: [Persona] -> [String]
nombres []     = []
nombres (p:ps) = nombre p : nombres ps
```

Tipos abstractos de datos: implementación, parte 1

Implementaciones de TADs

- Pueden darse varias implementaciones del mismo TAD
  - Ejemplo: tipo abstracto **`Persona`**

```
module PersonaV1   -- Versión 1
  …
```

```
module PersonaV2   -- Versión 2
  …
```

Implementaciones de TADs

- Pueden darse varias implementaciones del mismo TAD
  - Ejemplo: tipo abstracto **`Persona`**

```
module PersonaV1   -- Versión 1
  …
```

```
module PersonaV2   -- Versión 2
  …
```

- Pueden impedirse ciertos comportamientos
  - Invariantes de representación garantizados

Implementaciones de TADs

- Pueden darse varias implementaciones del mismo TAD

```
module PersonaV1  -- Versión 1
  (Persona, nacer, edad, nombre, apellido, crecer)
 where

data Persona = P String String Edad

nacer    n a       = P n a 0
edad     (P _ _ e) = e
nombre   (P n _ _) = n
apellido (P _ a _) = a
crecer   (P n a e) = P n a (e+1)
```

Implementaciones de TADs

- Pueden darse varias implementaciones del mismo TAD

```
module PersonaV2  -- Versión 2
  (Persona, nacer, edad, nombre, apellido, crecer)
 where

data Persona = P String Edad

nacer    n a       = P (n++" "++a) 0
edad      (P _  e) = e
nombre    (P na _) = obtenerHastaElEspacio na
apellido  (P na _) = obtenerDesdeElEspacio na
crecer    (P na e) = P na (e+1)
…
```

Implementaciones de TADs

- Pueden darse varias implementaciones del mismo TAD

```
…                    -- Persona versión 2
obtenerHastaElEspacio []     = error "No hay nombre"
obtenerHastaElEspacio (c:cs) =
      if c==' ' then ""
                 else c : obtenerHastaElEspacio cs

obtenerDesdeElEspacio []     = error "No hay apellido"
obtenerDesdeElEspacio (c:cs) =
      if c==' ' then cs
                 else obtenerDesdeElEspacio cs
```

Implementaciones de TADs

- Pueden impedirse ciertos comportamientos

```
module PersonaV1  -- Versión 1.1
  (Persona, nacer, edad, nombre, apellido, crecer)
where

data Persona = P String String Int
  {- INV.REP.:
      * el nombre y el apellido no son vacíos y no
         contienen espacios
      * la edad es >= 0
  -}
…
```

Implementaciones de TADs

- Pueden impedirse ciertos comportamientos

```
module Persona  -- Versión 1.1
   (Persona, nacer, edad, nombre, apellido, crecer)
…

nacer    n a        =
     if not (esNombreValido n)
      then error "El nombre no es adecuado"
      else if not (esApellidoValido a)
            then error "El apellido no es adecuado"
            else P n a 0
…
```

Implementaciones de TADs

- Pueden impedirse ciertos comportamientos

```
…                -- Persona Versión 1.1
esNombreValido      n = noVacioSinEspacios n
esApellidoValido    a = noVacioSinEspacios a
noVacioSinEspacios s = s/="" && not (elem ' ' s)
…
```

Tipos abstractos de datos: propiedades

Propiedades de TADs

- Implementaciones del mismo TAD son *intercambiables*

```
import PersonaV1
--import PersonaV2

yo = crecerVeces 53 (nacer "Fidel" "ML")

crecerVeces 0 p = p
crecerVeces n p = crecer (crecerVeces (n-1) p)
```

  - Si se cambia por el otro **`import`**, no cambia NADA

Propiedades de TADs

- No pueden usarse operaciones que NO son de la interfaz
  - E.g.: el constructor, funciones auxiliares

```
import PersonaV1
--import PersonaV2

yo = P "Fidel" "ML" 53                          -- ERROR

validarPersona p = esNombreValido (nombre p) -- ERROR
```

  - De esta forma, puede saberse que se cumple el invariante
    - ¿Se pueden tener personas con edad negativa?
      ¿Por qué no?

Diseño de TADs

- Diseñar un TAD es determinar cuál es su interfaz
  - No es tarea sencilla
  - Requiere pensar en todos los usos que se le quiere dar
  - Una interfaz mal diseñada puede impedir o dificultar ciertas tareas
  - No es tema de esta materia
    - Aunque quizás diseñen algún TAD, no se evalúa

Ejemplo de TAD

- Ejemplo: TAD Termómetro
  - Interfaz

```
data Termometro

nuevoT          :: Termometro
ingresarT       :: Int -> Termometro -> Termometro
sinTempsT       :: Termometro -> Bool
ultimaT         :: Termometro -> Int          -- PARCIAL
quitarUltimaT   :: Termometro -> Termometro   -- PARCIAL
maxT            :: Termometro -> Int          -- PARCIAL
```

  - ¿Qué puede hacerse con esta interfaz?

Ejemplo de TAD

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Usuario

```
import Termometro

ingresarTemps        :: [Int] -> Termometro -> Termometro
ultimasTempsDeTodos  :: [Termometro] -> [Int]
todasLasTemps        :: Termometro -> [Int]
cuantasNegativas     :: Termometro -> Int
dameNRecientes       :: Int -> Termometro -> [Int]
```

  - ¿Cómo puedo implementarlas?

Ejemplo de TAD

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Usuario

```
todasLasTemps          :: Termometro -> [Int]
todasLasTemps term =
  if sinTempsT term
   then []
   else ultimaT term : todasLasTemps (quitarUltimaT term)
                     -- todasLasTemps term  -- ¡¡ERROR!!
```

  - ¡Se usa una forma de recursión!
    - **Atención** a no generar repeticiones infinitas

Ejemplo de TAD

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Usuario

```
todasLasTempsMAL          :: Termometro -> [Int]
todasLasTempsMAL nuevoT = []    -- ERROR
todasLasTempsMAL t       =
        ultimaT t : todasLasTempsMAL (quitarUltimaT t)
```

  - ¿Qué tiene mal?

Ejemplo de TAD

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Usuario

```
todasLasTempsMAL          :: Termometro -> [Int]
todasLasTempsMAL nuevoT = []    -- ERROR
todasLasTempsMAL t        =
        ultimaT t : todasLasTempsMAL (quitarRecienteT t)
```

  - ¡NO SE PUEDE hacer pattern matching sobre TADs!
    - La función **`nuevoT`** NO es un constructor

Tipos abstractos de datos: implementación, parte 2

Roles para entender un TAD

- Un TAD puede ser visto desde 3 puntos de vista (roles)
  - Diseñador
    - Decide las operaciones de la interfaz
  - Usuario
    - Usa las operaciones de la interfaz
  - Implementador
    - Decide la representación interna
    - Fija el invariante de representación
    - Provee la implementación de las operaciones

Roles para entender un TAD

- Un TAD puede ser visto desde 3 puntos de vista (roles)
  - Ya vimos los roles de diseñador y de usuario
    - No haremos diseño, pero sí extenso uso de TADs
  - El rol de implementador es el más delicado
    - ¿Qué debe tener en cuenta el implementador?
    - ¿Cómo decidir la representación y la implementación?
    - Debemos profundizar en las nociones de
      - Eficiencia
      - El rol de los invariantes de representación

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador

…

```
data Termometro = T [Int]

nuevoT                 = T []
ingresarT   t (T ts)   = T (t:ts)
sinTempsT     (T ts)   = null ts
ultimaT       (T ts)   = head ts
quitarUltimaT (T ts)   = T (tail ts)
maxT          (T ts)   = maximum ts
```

  - ¿Se puede mejorar? ¿Cómo saber qué mejorar?

Roles para entender

```
data Termometro

nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Rol de implementador: Termometro
  - ¿Se puede mejorar? ¿Cómo saber qué mejorar?
  - La operación para calcular el máximo puede llevar mucho
    - Precisamos alguna forma de saber medir la ***eficiencia***
    - Precisamos usar la representación para mejorarla
    - Precisamos alguna forma de garantizar que la información de la representación cumple ciertas propiedades
      - Invariantes de representación
      - ¿Cómo usarlos al implementar operaciones?

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de mejora
    - Calcular el máximo es caro (después veremos cómo saberlo)
    - ¿Y si guardamos el máximo?

```
…
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts)
  -}
…
```

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

…

```
nuevoT                  = T []          Nothing
ingresarT   t (T ts m)  = T (t:ts)      (maxAlIngresar t m)
sinTempsT     (T ts _)  = null ts
ultimaT       (T ts _)  = head ts
quitarUltimaT (T ts m)  = T (tail ts)   (maxAlQuitar m ts)
maxT          (T _  m)  = fromJust m
```

…

¿Y por qué esto es correcto?
¡El Invariante de Representación!

¿Cómo determinar qué hacer en estos casos?
¡El Invariante de Representación!

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int         -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro  -- PARCIAL
maxT           :: Termometro -> Int         -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

```
...
maxAlIngresar :: Int -> Maybe Int -> Maybe Int
maxAlIngresar t Nothing   = Just t
maxAlIngresar t (Just t') = Just (max t t')
...
```

El nuevo máximo al ingresar tiene que elegir entre el máximo anterior y el valor nuevo

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int         -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro  -- PARCIAL
maxT           :: Termometro -> Int         -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

…

```
maxAlQuitar :: Maybe Int -> [Int] -> Maybe Int
 -- PRECOND: la lista no es vacía y el maybe no es Nothing
maxAlQuitar (Just t') (t:ts) =
  if null ts then Nothing
             else if t==t' then Just (maximum ts)
                           else Just t'
```

El nuevo máximo al quitar tiene que considerar si sacar o no el que estaba

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

…

```
maxAlQuitar :: Maybe Int -> [Int ] -> Maybe Int
  -- PRECOND: la lista no es vacía
maxAlQuitar (Just t') (t:ts) =
  if null ts then Nothing
             else if t==t' then Just (maximum ts)
                           else Just t'
```

`t` es el elemento que se va a quitar…
(¡Observar el invariante!)

El nuevo máximo al quitar tiene que considerar si sacar o no el que estaba

Roles para entender

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int         -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro  -- PARCIAL
maxT           :: Termometro -> Int         -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador: intento de

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

…

```
maxAlQuitar :: Maybe Int -> [Int ] -> Maybe Int
  -- PRECOND: la lista no es vacía
maxAlQuitar (Just t) (t':ts) =
  if null ts then Nothing
             else if t==t' then Just (maximum ts)
                           else Just t
```

  - ¿Se puede hacer mejor?

Eficiencia

Eficiencia

- Medir la eficiencia es necesario para saber cómo mejorar
  - ¿Pero qué medimos para entender la eficiencia?
  - Miraremos cómo se comporta la función a medir
    - En el peor caso posible
    - En función de la cantidad de datos de la estructura
  - Usaremos una clasificación para esto
    - En base al comportamiento general en peor caso
    - Indicador **grueso** de eficiencia

Eficiencia

- Clasificación para medir eficiencia
  - Constante
  - Lineal
  - Cuadrática
  - Cúbica
  - …
  - Exponencial
  - Hyperexponencial
  - …

Eficiencia

- Clasificación: ***costo constante***
  - La función tarda siempre lo mismo
  - No depende de la cantidad de elementos de la estructura
  - Ejemplos:
    - `head`
    - `tail`
    - `null`

Eficiencia

- Clasificación: ***costo lineal***
  - Por cada elemento de la estructura, solamente se hacen operaciones de costo constante
  - Ejemplos:
    - `length`
    - `sum`
    - `elem`

Eficiencia

- Clasificación: ***costo lineal***
  - Por cada elemento de la estructura, se hacen operaciones de costo constante
  - Otro ejemplo

```
unoConTodos :: a -> [a] -> [(a,a)]  -- lineal
unoConTodos x []     = []
unoConTodos x (y:ys) = (x,y) : unoConTodos x ys
                     -- constante
```

Eficiencia

```
unoConTodos :: a -> [a] -> [(a,a)]  -- lineal
unoConTodos x []     = []
unoConTodos x (y:ys) = (x,y) : unoConTodos x ys xys
                     -- constante
```

- Clasificación: ***costo cuadrático***
  - Por cada elemento de la estructura, se hacen operaciones de costo lineal (en peor caso)
  - Ejemplo:

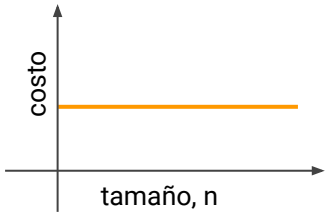
```
todosConTodos :: [a] -> [(a,a)]      -- cuadrática
todosConTodos xs = cadaConTodos xs xs
  where cadaConTodos []     _      = []
        cadaConTodos (x:xs) todos =
            unoConTodos x todos ++ cadaConTodos xs todos
            -- lineal
```

Eficiencia

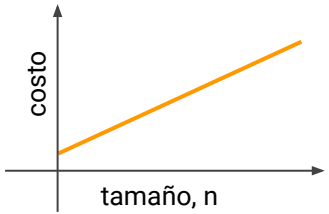
- Clasificación: ***costo cúbico***
  - Por cada elemento de la estructura, se hacen operaciones de costo cuadrático
  - También existen operaciones que elevan a la cuarta, a la quinta, etc. (usando una operación del costo previo por cada elemento)
  - Por ahora nos manejaremos solamente con las 3 primeras: constante, lineal, cuadrática

Eficiencia

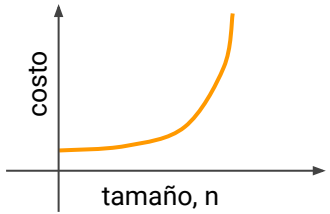
- Clasificación para medir eficiencia
  - Notación matemática: big O (O grande)
    - Constante, O(1)
    - Lineal, O(n)
    - Cuadrática, O(n^2)



Constante, O(1)



Lineal, O(n)



Cuadrática, O(n^2)

Tipos abstractos de datos: ejemplos

Eficiencia

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int         -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro  -- PARCIAL
maxT           :: Termometro -> Int         -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador, con eficiencia

```
...
data Termometro = T [Int]

nuevoT                = T []          -- constante
ingresarT   t (T ts)  = T (t:ts)      -- constante
sinTempsT     (T ts)  = null ts       -- constante
ultimaT       (T ts)  = head ts       -- constante
quitarUltimaT (T ts)  = T (tail ts)   -- constante
maxT          (T ts)  = maximum ts    -- lineal
```

Eficiencia

```
data Termometro
nuevoT        :: Termometro
ingresarT     :: Int -> Termometro -> Termometro
sinTempsT     :: Termometro -> Bool
ultimaT       :: Termometro -> Int          -- PARCIAL
quitarUltimaT :: Termometro -> Termometro -- PARCIAL
maxT          :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador, con eficien

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

```
...
nuevoT                  = T []        Nothing                -- constante
ingresarT   t (T ts m)  = T (t:ts)    (maxAlIngresar t m)    -- constante
sinTempsT     (T ts _)  = null ts                          -- constante
ultimaT       (T ts _)  = head ts                          -- constante
quitarUltimaT (T ts m)  = T (tail ts) (maxAlQuitar m ts)   -- lineal
maxT          (T _  m)  = fromJust m                       -- constante
...
```

  - ¡Cambiamos una lineal por otra!

Eficiencia

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador, con eficien

```
data Termometro = T [Int] (Maybe Int)
  {- INV.REP.: en T ts m,
       * si ts es vacío, m es Nothing
       * si no, m es Just (maximum ts) -}
```

…

```
maxAlQuitar :: Maybe Int -> [Int] -> Maybe Int
  -- PRECOND: la lista no es vacía  -- lineal
maxAlQuitar (Just t') (t:ts) =
  if null ts then Nothing
             else if t==t' then Just (maximum ts)
                           else Just t'
```

¡Solamente importa el peor caso!

Eficiencia

```
data Termometro

nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int          -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro   -- PARCIAL
maxT           :: Termometro -> Int          -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador, con eficiencia
  - Observación
    - Precisamos un máximo relativo a cada elemento…

```
…
data Termometro = T [Int] [Int]
  {- INV.REP.: en T ts ms
       * ts y ms tienen la misma longitud
       * cada m de ms es el máximo de la sublista de ts desde esa pos.
  -}
…
```

Eficiencia

```
data Termometro
nuevoT         :: Termometro
ingresarT      :: Int -> Termometro -> Termometro
sinTempsT      :: Termometro -> Bool
ultimaT        :: Termometro -> Int         -- PARCIAL
quitarUltimaT  :: Termometro -> Termometro  -- PARCIAL
maxT           :: Termometro -> Int         -- PARCIAL
```

- Ejemplo: TAD Termómetro
  - Implementador, con eficien

```
data Termometro = T [Int] [Int]
  {- INV.REP.: en T ts ms
       * ts y ms tienen la misma longitud
       * cada m de ms es el máximo de la
         sublista de ts desde esa pos. -}
```

```
...
nuevoT                  = ...
ingresarT   t (T ts ms) = ...
sinTempsT     (T ts _ ) = ...
ultimaT       (T ts _ ) = ...
quitarUltimaT (T ts ms) = ...
maxT          (T _  ms) = ...
...
```

  - Hacer las operaciones y calcular su costo

TADs clásicos

- Existen muchas estructuras de datos clásicas
- Entre las más simples, veremos 3
  - Stacks (pilas)
  - Queues (colas)
  - Sets (conjuntos)
- Los detalles los verán en la práctica

TADs clásicos: Stacks

- Stack (pila)
  - Se pueden ingresar, consultar y eliminar elementos
  - La propiedad que cumplen es
    - “el último que entra es el primero que sale”
      (en inglés, *Last In - First Out* o LIFO)
  - Tienen muchas aplicaciones en programación
    - ejecución de lenguajes,
    - problemas de balanceo,
    - etc.

TADs clásicos: Queues

- Queue (cola) (se pronuncia *kiú*)
  - Se pueden ingresar, consultar y eliminar elementos
  - La propiedad que cumplen es
    - "el primero que entra es el primero que sale"
      (en inglés, *First In - First Out* o FIFO)
  - Suelen tener aplicaciones comerciales o en sistemas
    - sistemas operativos,
    - atención de cajas,
    - etc.

TADs clásicos: Sets

- Set (conjunto)
  - Se pueden ingresar elementos y consultar existencia
  - La propiedad que cumplen es
    - “no hay elementos repetidos en un conjunto”
  - Se usan en aplicaciones donde se deben mantener elementos sin repetición

Criterios de elección para implementaciones

- ¿Cómo elegir una implementación específica?
  - Sabemos que en alguna implementación una operación es costosa, y en otra, es otra operación la costosa
  - En general, hay un balance
    - alguna operación es más cara y se elige cuál
    - o todas son baratas, pero se usa más memoria
- Depende de qué operaciones sean de uso más frecuente
  - ¡Debe tenerse en cuenta el contexto de uso!

Resumen

Resumen

- Tipos abstractos de datos
  - Interfaz + implementaciones
  - **Roles:** diseñador, usuario, implementador
  - TADs clásicos: Stacks, Queues, Sets
- Eficiencia
  - Modelo de peor caso
  - Clasificación: constante, lineal, cuadrática