Implementaciones del TAD Persona

A continuación se muestran las diferentes versiones del Tipo Abstracto de Datos Persona vistas en la clase.

Versión 1 (PersonaV1)

Esta es la primera implementación, que guarda el nombre, apellido y edad en campos separados.

Haskell

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

-- Representación interna: data Persona = P Nombre Apellido Edad
data Persona = P String String Int

-- Crea una nueva persona con edad 0.
nacer :: String -> String -> Persona
nacer n a = P n a 0

-- Devuelve la edad de la persona.
edad :: Persona -> Int
edad (P _ _ e) = e

-- Devuelve el nombre de la persona.
nombre :: Persona -> String
nombre (P n _ _) = n

-- Devuelve el apellido de la persona.
apellido :: Persona -> String
apellido (P _ a _) = a

-- Incrementa la edad de la persona en 1.
crecer :: Persona -> Persona
crecer (P n a e) = P n a (e + 1)
```

Versión 2 (PersonaV2)

Esta versión optimiza el almacenamiento concatenando el nombre y el apellido en un solo String. Las funciones nombre y apellido deben procesar este string para obtener el dato correspondiente.

Haskell

```
module PersonaV2
 ( Persona,
   nacer,
   edad,
   nombre,
   apellido,
   crecer,
 )
```

```
where

-- Representación interna: data Persona = P NombreCompleto Edad
data Persona = P String Int

-- Funciones auxiliares no exportadas
obtenerHastaElEspacio :: String -> String
obtenerHastaElEspacio [] = error "No hay nombre"
obtenerHastaElEspacio (c : cs) =
 if c == ' '
  then ""
  else c : obtenerHastaElEspacio cs

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

-- Crea una nueva persona concatenando nombre y apellido.
nacer :: String -> String -> Persona
nacer n a = P (n ++ " " ++ a) 0

-- Devuelve la edad de la persona.
edad :: Persona -> Int
edad (P _ e) = e

-- Extrae el nombre del string concatenado.
nombre :: Persona -> String
nombre (P na _) = obtenerHastaElEspacio na

-- Extrae el apellido del string concatenado.
apellido :: Persona -> String
apellido (P na _) = obtenerDesdeElEspacio na

-- Incrementa la edad de la persona en 1.
crecer :: Persona -> Persona
crecer (P na e) = P na (e + 1)
```

Versión 1.1 (con Invariante de Representación)

Esta versión es una mejora de PersonaV1 que introduce validaciones para garantizar un **invariante de representación**: la edad siempre es >= 0 y los nombres/apellidos no pueden ser vacíos ni contener espacios.

Haskell

```
module PersonaV1_1
 ( Persona,
  nacer,
  edad,
  nombre,
  apellido,
```

```
  crecer,
 )
where

-- INV.REP.:
-- * el nombre y el apellido no son vacíos y no contienen espacios [cite: 201]
-- * la edad es >= 0 [cite: 202]
data Persona = P String String Int

-- Funciones auxiliares de validación
noVacioSinEspacios :: String -> Bool
noVacioSinEspacios s = s /= "" && not (elem ' ' s)

esNombreValido :: String -> Bool
esNombreValido n = noVacioSinEspacios n

esApellidoValido :: String -> Bool
esApellidoValido a = noVacioSinEspacios a

-- La función `nacer` ahora valida los datos antes de crear la persona.
nacer :: String -> String -> Persona
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

-- El resto de las funciones son iguales a PersonaV1
edad :: Persona -> Int
edad (P _ _ e) = e

nombre :: Persona -> String
nombre (P n _ _) = n

apellido :: Persona -> String
apellido (P _ a _) = a

crecer :: Persona -> Persona
crecer (P n a e) = P n a (e + 1)
```

Código de Usuario de TADs

Estos son ejemplos de cómo un usuario podría interactuar con los TADs Persona y Termometro usando únicamente su interfaz.

Usuario del TAD Persona

Haskell

```
import Persona -- (Puede ser PersonaV1, V2 o V1_1, el código no cambia)
```

```
-- Usa `crecer` n veces sobre una persona.
crecerVeces :: Int -> Persona -> Persona
crecerVeces 0 p = p
crecerVeces n p = crecer (crecerVeces (n - 1) p)

-- Ejemplo de uso:
fidel :: Persona
fidel = crecerVeces 53 (nacer "Fidel" "ML")

-- Crea una lista de personas a partir de una lista de tuplas (nombre, apellido).
nacerMuchas :: [(String, String)] -> [Persona]
nacerMuchas [] = []
nacerMuchas ((n, a) : nas) = nacer n a : nacerMuchas nas

-- Obtiene los nombres de una lista de personas.
nombres :: [Persona] -> [String]
nombres [] = []
nombres (p : ps) = nombre p : nombres ps
```

Usuario del TAD Termometro

Haskell

```
import Termometro

-- Devuelve todas las temperaturas de un termómetro.
-- Se basa en la recursión sobre el TAD.
todasLasTemps :: Termometro -> [Int]
todasLasTemps term =
  if sinTempsT term
    then []
    else ultimaT term : todasLasTemps (quitarUltimaT term)

-- ¡¡IMPLEMENTACIÓN INCORRECTA!!
-- No se puede hacer pattern matching (e.g., sobre `nuevoT`) con un TAD.
-- La función `nuevoT` NO es un constructor. [cite: 326]
todasLasTempsMAL :: Termometro -> [Int]
todasLasTempsMAL nuevoT = [] -- ESTO ES UN ERROR DE CONCEPTO [cite: 326]
todasLasTempsMAL t = ultimaT t : todasLasTempsMAL (quitarUltimaT t)
```

Implementaciones del TAD Termometro

A continuación, las implementaciones del TAD Termometro con sus análisis de eficiencia.

Versión 1 (Lista simple)

Representación simple usando una lista de enteros. La operación maxT es **lineal** (O(n)) mientras que el resto son **constantes** (O(1)).

Haskell

```
module TermometroV1
  ( Termometro,
    nuevoT,
    ingresarT,
    sinTempsT,
    ultimaT,
    quitarUltimaT,
```

```
  maxT,
 )
where

-- Representación interna: T [Temperaturas]
data Termometro = T [Int]

nuevoT :: Termometro
nuevoT = T []

ingresarT :: Int -> Termometro -> Termometro
ingresarT t (T ts) = T (t : ts)

sinTempsT :: Termometro -> Bool
sinTempsT (T ts) = null ts

ultimaT :: Termometro -> Int
ultimaT (T ts) = head ts

quitarUltimaT :: Termometro -> Termometro
quitarUltimaT (T ts) = T (tail ts)

maxT :: Termometro -> Int
maxT (T ts) = maximum ts
```

Versión 2 (Lista y máximo cacheado)

Se guarda el máximo actual para que la operación maxT sea **constante** (O(1)). Sin embargo, el costo se traslada a quitarUltimaT, que ahora pasa a ser **lineal** (O(n)) en el peor caso.

Haskell

```
module TermometroV2
 ( Termometro,
   nuevoT,
   ingresarT,
   sinTempsT,
   ultimaT,
   quitarUltimaT,
   maxT,
 )
where

import Data.Maybe (fromJust)

-- INV.REP.: en T ts m,
-- * si ts es vacío, m es Nothing [cite: 423]
-- * si no, m es Just (maximum ts) [cite: 423]
data Termometro = T [Int] (Maybe Int)

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

```
-- PRECOND: la lista no es vacía y el maybe no es Nothing. [cite: 507]
maxAlQuitar :: Maybe Int -> [Int] -> Maybe Int
maxAlQuitar (Just t') (t : ts) =
 if null ts
  then Nothing
  else
   if t == t'
    then Just (maximum ts) -- Peor caso, costo lineal [cite: 744, 747]
    else Just t'

-- Implementación de la interfaz
nuevoT :: Termometro
nuevoT = T [] Nothing

ingresarT :: Int -> Termometro -> Termometro
ingresarT t (T ts m) = T (t : ts) (maxAlIngresar t m)

sinTempsT :: Termometro -> Bool
sinTempsT (T ts _) = null ts

ultimaT :: Termometro -> Int
ultimaT (T ts _) = head ts

quitarUltimaT :: Termometro -> Termometro
quitarUltimaT (T ts m) = T (tail ts) (maxAlQuitar m ts)

maxT :: Termometro -> Int
maxT (T _ m) = fromJust m
```