1. Módulo Diccionario Lineal(κ, σ)

El módulo Diccionario Lineal provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definición se puede hacer en tiempo $O(1)$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer y eliminar los elementos de d como si fuera una secuencia de pares κ, σ.

Para describir la complejidad de las operaciones, vamos a llamar $copy(k)$ al costo de copiar el elemento $k \in \kappa \cup \sigma$ y $equal(k_1, k_2)$ al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales (i.e., *copy* y *equal* son funciones de $\kappa \cup \sigma$ y $\kappa \times \kappa$ en $\mathbb{N}$, respectivamente).[1]

Interfaz

parámetros formales

géneros κ, σ

función $\bullet = \bullet(\textbf{in } k_1 : \kappa, \textbf{in } k_2 : \kappa) \to res$: `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) $\to res$: κ
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} k\}$
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función COPIAR(**in** $s : \sigma$) $\to res$: σ
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} s\}$
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

se explica con: DICCIONARIO(κ, σ)

géneros: `dicc`(κ, σ).

Operaciones básicas de diccionario

VACÍO() $\to res$: `dicc`(κ, σ)
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{vacio}\}$
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$, **in** $s : \sigma$) $\to res$: `itDicc`(κ, σ)
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s) \wedge \text{haySiguiente}(res) \wedge_L \text{Siguiente}(res) = \langle k, s \rangle \wedge \text{alias}(\text{esPermutación}(\text{SecuSuby}(res), d))\}$
Complejidad: $\Theta\left(\sum_{k' \in K} equal(k, k') + copy(k) + copy(s)\right)$, donde $K = \text{claves}(d)$
Descripción: define la clave k con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

DEFINIRRAPIDO(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$, **in** $s : \sigma$) $\to res$: `itDicc`(κ, σ)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \neg\text{definido?}(d, k)\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s) \wedge \text{haySiguiente}(res) \wedge_L \text{Siguiente}(res) = \langle k, s \rangle \wedge \text{esPermutación}(\text{SecuSuby}(res), d)\}$
Complejidad: $\Theta(copy(k) + copy(s))$
Descripción: define la clave $k \notin \text{claves}(d)$ con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.

[1] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* y *equal* en función del tamaño de k. A la hora de usarlo, habrá que realizar la traducción.

Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

Definido?(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde K = claves(d)
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

Significado(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Significado(d, k))}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde K = claves(d)
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

Borrar(**in/out** d: `dicc(`κ, σ`)`, **in** k: κ)
Pre $\equiv$ {$d = d_0 \wedge$ def?(d, k)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(d_0, k)}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde K = claves(d)
Descripción: elimina la clave k y su significado de d.

#Claves(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ #claves(d)}
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de claves del diccionario.

Copiar(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `dicc(`κ, σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ d}
Complejidad: $\Theta\left(\sum_{k \in K} (copy(k) + copy(\text{significado}(k, d)))\right)$, donde K = claves(d)
Descripción: genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** d_1 : `dicc(`κ, σ`)`, **in** d_2 : `dicc(`κ, σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $c_1 = c_2$}
Complejidad: $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} equal(\langle k_1, s_1 \rangle, \langle k_2, s_2 \rangle)\right)$, donde K_i = claves(d_i) y s_i = significado(d_i, k_i), $i \in \{1, 2\}$.
Descripción: compara d_1 y d_2 por igualdad, cuando σ posee operación de igualdad.
Requiere: $\bullet = \bullet$(**in** $s_1 : \sigma$, **in** $s_2 : \sigma$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ ($s_1 = s_2$)}
Complejidad: $\Theta(equal(s_1, s_2))$
Descripción: función de igualdad de σ's

Especificación de las operaciones auxiliares utilizadas en la interfaz

esPermutacion? : secu(tupla(κ, σ)) $\times$ dicc(κ, σ) $\longrightarrow$ bool

esPermutacion?(s, d) $\equiv$ d = secuADicc(s) $\wedge$ #claves(d) = long(s)

secuADicc : secu(tupla(κ, σ)) $\longrightarrow$ dicc(κ, σ)

secuADicc(s) $\equiv$ **if** vacia?(s) **then** vacio **else** definir(Π_1(prim(s)), Π_2(prim(s)), secuADict(fin(s))) **fi**

Representación

Representación del diccionario

El diccionario se representa con dos listas enlazadas, una conteniendo las claves y otras los significados. Ambas listas deben tener igual largo ya que las claves y los significados se aparean por posición.

`dicc`(κ, σ) **se representa con** `dic`

donde `dic` es `tupla`(*claves*: `lista`(κ), *significados*: `lista`(σ))

Rep : dic $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$
sin claves repetidas: #claves(secuADicc(d.claves)) = long(d.claves) $\wedge$
long(d.claves) = long(d.significados)

Abs : dicc d $\longrightarrow$ dicc(κ, σ) {Rep(d)}
Abs(d) $\equiv$ **if** vacía?(d.claves) **then**
vacío
else
definir(prim(d).claves, prim(d).significado, Abs((fin(d.claves), fin(d.significados)))
fi

Algoritmos

---

iVacío() $\rightarrow$ res : dic

1: $res \leftarrow \langle$ secu::Vacia(), secu::Vacia() $\rangle$

Complejidad: $\Theta(1)$

---

iDefinir(**in/out** d: `dic`, **in** k: κ, **in** s: σ) $\rightarrow$ res : $itDicc$

if la clave k está en d.claves **then**
itC, itS $\leftarrow$ buscar(d, k)
*itS $\leftarrow$ s
$res \leftarrow \langle$ itC, itS $\rangle$
else$res \leftarrow$DefinirRapido(d, k, s)
end if
Complejidad: $\Theta\left(\sum_{k' \in K} equal(k, k') + copy(k) + copy(s)\right)$
Justificación: Se debe revisar si la clave está. Esto toma $\emptyset(n)$. Si la clave está definida, se recorren las listas hasta encontrar la clave ($\mathcal{O}(n)$). Luego se reemplaza el significado anterior por el nuevo ($\mathcal{O}(copy(\sigma))$). Si la clave no está, se deben copiar clave y significado al final ($\mathcal{O}(copy(\kappa) + copy(\sigma))$).

---

iDefinirRapido(**in/out** d: `dic`, **in** k: κ, **in** s: σ) $\rightarrow$ res : $itDicc$

itC $\leftarrow$ d.claves.AgregarAdelante(k)
itS $\leftarrow$ d.significados.AgregarAdelante(s)
res $\leftarrow \langle$ itC, its $\rangle$
Complejidad: $\Theta\left(copy(k) + copy(s)\right)$
Justificación: La clave no está, solo se deben copiar clave y significado al final de las listas internas ($\mathcal{O}(copy(\kappa) + copy(\sigma))$).

---

iDefinido?(in/out d: `dic`, **in** k: κ) $\rightarrow$ res : $bool$

itC, itS $\leftarrow$ buscar(d, k)
$res \leftarrow$ (itC == d.claves.end())
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$
Justificación: Se recorre la lista de claves comparando con la clave parámetro.

---

iSignificado(in/out d: `dic`, **in** k: κ) $\rightarrow$ res : σ

itC, itS $\leftarrow$ buscar(d, k)
$res \leftarrow$ *itS
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$
Justificación: Se recorre la lista de claves comparando con la clave parámetro. El significado se devuelve por referencia.

---

iBorrar(in/out d: `dic`, **in** k: κ) $\rightarrow$ res : σ

itC, itS $\leftarrow$ buscar(d, k)
d.claves.borrar(itC)
d.significado.borrar(itS)
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$
Justificación: Se recorre la lista de claves comparando con la clave parámetro. El significado se devuelve por referencia.

---

iClaves(in/out d: `dic`) $\rightarrow$ res : nat

$res \leftarrow$ d.claves.longitud()
Complejidad: Asume que la lista permite responder cantidad de claves en $\mathcal{O}(1)$

---

iBuscar(in/out d: `dic`, **in** k: κ) $\rightarrow$ res : tupla($itLista(\kappa), itLista(\sigma)$)

itC $\leftarrow$ d.claves.begin()
itS $\leftarrow$ d.significado.begin()
while *itC != k **do**
    ++itC
    ++itS
end while
$res \leftarrow$ $\langle$ itC, itS $\rangle$
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$
Justificación: Se recorre la lista de claves comparando con la clave parámetro. Se devuelven dos iteradores por copia. Se asume que su copia es $\mathcal{O}(1)$.

1.1. Iterador Diccionario Lineal

Interfaz

parámetros formales

géneros κ, σ

función $\bullet = \bullet(\textbf{in } k_1 : \kappa, \textbf{in } k_2 : \kappa) \rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) $\rightarrow res$: κ
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} k\}$
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función COPIAR(**in** $s : \sigma$) $\rightarrow res$: σ
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} s\}$
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

se explica con: SECU(TUPLA(κ, σ))

géneros: `itDicc(`κ, σ`)`.

El iterador que presentamos permite recorrer el diccionario o hacer referencia a una tupla clave-significado dentro del diccionario.

CREARIT(**in** itC : `itLista(`κ`)`, **in** itS : `itLista(`σ`)`) $\rightarrow res$: `itDicc(`κ, σ`)`
Pre $\equiv$ {itC e itS refieren al mismo diccionario y están alineados en el mismo}
Post $\equiv$ {El iterador resultado genera la secuencia clave-significado de los elementos restantes por ver.}
Complejidad: $\Theta(1)$
Descripción: Crea un iterador del diccionario. Este método solo puede ser utilizado por el diccionario para generar un nuevo iterador.

DESREFERENCIAR(*)(**in** it : `itDicc(`κ, σ`)`) $\rightarrow res$: `tupla(`κ, σ`)`
Pre $\equiv$ {El iterador no llegó al final}
Post $\equiv$ {Devuelve la clave valor del diccionario por referencia.}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento al que apunta el iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

AVANZAR(++)(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {El iterador no llegó al final.}
Post $\equiv$ {El iterador apunta al siguiente elemento en el recorrido}
Complejidad: $\Theta(1)$
Descripción: avanza a la posición siguiente del iterador.

RETROCEDER(−)(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {El iterador no está al principio}
Post $\equiv$ {El ierador apunta a la posición anterior en el recorrido}
Complejidad: $\Theta(1)$
Descripción: retrocede a la posición anterior del iterador.

Representación

Representación del iterador

El iterador del diccionario es un par de iteradores a las listas correspondientes.

`itDicc(`κ`, `σ`)` **se representa con** `itDic`

donde `itDic` es `tupla(`*clave*: `itLista(`κ`)`, *significado*: `itLista(`σ`))`

Rep : itDic $\longrightarrow$ bool

Rep(it) $\equiv$ true $\iff$ **los iteradores apuntan al mismo diccionario** $\wedge$
obtener *it.clave en el diccionario al que apuntan los iteradores es igual a *it.significado

Abs : itDic s $\longrightarrow$ secu(tupla(κ, σ)) {Rep(s)}

Abs(s) $\equiv$ s está vacía ssi itDic.clave llegó al final del recorrido $\wedge$
prim(s) = $\langle$ *itDic.clave, *itDic.significado $\rangle$ $\wedge$
fin(s) = Abs($\langle$ ++itDic.clave, ++itDic.significado $\rangle$)

Algoritmos

---

iCrearIt(**in** itC : `itLista(`κ`)`, **in** itS : `itLista(`σ`)`) $\rightarrow$ res : itDic

1: $res \leftarrow \langle$ itC, itS $\rangle$

Complejidad: $\Theta(1)$
Justificacion: Se asume que copiar los iteradores de lista es $\mathcal{O}(1)$

---

iDesreferenciar(**in** it : `itDic(`κ`, `σ`)`) $\rightarrow$ res : tupla(κ, σ)

1: $res \leftarrow \langle$ *it.itC, *it.itS $\rangle$

Complejidad: $\Theta(1)$
Justificacion: Se asume que desreferenciar los iteradores de lista es $\mathcal{O}(1)$. Los elementos se devuelven por referencia.

---

iAvanzar(**in** it : `itDic(`κ`, `σ`)`)

1: ++it.itC
2: ++it.itS

Complejidad: $\Theta(1)$

---

iRetroceder(**in** it : `itDic(`κ`, `σ`)`)

1: –it.itC
2: –it.itS

Complejidad: $\Theta(1)$
