DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 2

Algoritmos y Estructuras de Datos II

Grupo: 21

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Walter, Nicolás	272/14	nicowalter25@gmail.com
Sticco, Patricio Bernardo	337/14	pbsticco@hotmail.com
Len, Julián	467/14	julianlen@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Diseño del Tipo DiccionarioString(σ)

1.1. Especificación

Se usa el Tad Diccionario(κ, σ) especificado en el apunte de Tads básicos.

1.2. Aspectos de la interfaz

1.2.1. Interfaz

parámetros formales

género κ, σ

función $\bullet = \bullet$(**in** a_1: κ, **in** a_2: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} (a_1 = a_2)$ }
Complejidad: $\Theta(equals(a_1, a_2))$
Descripción: función de igualdad de κ's

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

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

Se explica con especificación de Diccionario(κ, σ), Iterador Bidireccional(Tupla(κ, σ))

Género `diccString(`κ`,`σ`)`

Operaciones básicas de diccionario

Definido?(**in** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} def?(d, k)$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve true si y sólo si k está definido en el diccionario.

Obtener(**in** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $def?(d, k)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} obtener(d, k))$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res no es modificable.

Vacio() $\longrightarrow res : diccString(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacio()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Genera un diccionario vacío.

Definir(**in/out** d: $diccString(\kappa, \sigma)$, **in** k: κ, **in** s: σ)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} definir(k, s, d_0)$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*

Descripción: Define la clave k con el significado s en el diccionario.

BORRAR(**in/out** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $d =_{\text{d}_0} \wedge def?(k, d)$ }
Post $\equiv$ { $d =_{\text{obs}} borrar(k,\text{d}_0)$}
Complejidad: $\mathcal{O}(|k|)$ *$|k|$ es la longitud de la clave.*
Descripción: Elimina la clave k del diccionario.

Operaciones básicas del iterador

CREARIT(**in** d: $diccString(\kappa, \sigma)$) $\longrightarrow res : itdiccString(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(esPermutacion(SecuSuby(res), d)) \wedge vacia?(Anteriores(res))$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Crea un iterador del diccionario de forma tal que se puedan recorrer sus elementos aplicando iterativamente SIGUIENTE(no ponemos la operacion SIGUIENTE en la interfaz pues no la usamos).

HAYSIGUIENTE(**in** it: $itdiccString(\kappa, \sigma)$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} HaySiguiente?(it)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si y solo si en el iterador quedan elementos para avanzar.

SIGUIENTESIGNIFICADO(**in** it: $itdiccString(\kappa, \sigma)$) $\longrightarrow res : \sigma$
Pre $\equiv$ { $HaySiguiente?(it)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} Siguiente(it).significado)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el significado del elemento siguiente del iterador.
Aliasing: res no es modificable.

AVANZAR(**in/out** it: $itdiccString(\kappa, \sigma)$)
Pre $\equiv$ { $it =_{\text{obs}} it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ { $it =_{\text{obs}} Avanzar(it_0)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza a la posicion siguiente del iterador.

1.3. Pautas de implementación

1.3.1. Estructura de representación

$dicc\_trie(\kappa, \sigma)$ **se representa con** $puntero(nodo)$
donde $nodo$ **es**
tupla(
significado: ***Puntero(*σ*)*** $\times$
caracteres: ***arreglo[256]*** *de* $puntero(nodo)$ $\times$
padre: ***Puntero(nodo)***
)

1.3.2. Justificación

1.3.3. Invariante de Representación

Informal

- Todas las posiciones del arreglo de caracteres están definidas.
- No hay claves de 0 caracteres. El significado de la raíz es NULL.
- No hay ciclos en la estructura. Es decir, existe una cota superior sobre la cantidad de niveles posibles del árbol.

- Dado un nodo cualquiera del trie, existe un único camino desde la raíz hasta el nodo.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1)($\forall i : nat$)($i < 256 \Rightarrow$ definido?(e $\rightarrow$ caracteres,i)) $\wedge_{\text{L}}$
(2)(e $\rightarrow$ $significado = NULL$) $\wedge_{\text{L}}$
(2)($\exists$ n:nat)(finaliza(e,n)) $\wedge_{\text{L}}$
(3)($\forall$ p,q: puntero(nodo))(p $\in$ punteros(e) $\wedge q \in (punteros(e) - \{p\}) \Rightarrow$ p$\neq$ q) $\wedge_{\text{L}}$
)

1.3.4. Función de Abstracción

Abs : roseTree(estrDato) r $\longrightarrow$ dicc_trie(σ) {Rep(r)}
($\forall$ r:roseTree(estrDato)) Abs(r) $=_{\text{obs}}$ d : dicc_trie(σ) /
($\forall$ k : secu($letra$))(def?(k, d) $=_{\text{obs}}$ esta?(k, r)) $\wedge$ (def?(c, d) $\Rightarrow$ (obtener(k, d) $=_{\text{obs}}$ buscar(k, r)))

Funciones Auxiliares

1.3.5. Algoritmos

```
1: function iVacio( )⟶ res : estr                                   ▷ O(1)
2:     var arreglo(puntero(nodo)) letras ← crearArreglo[256]
3:     for i ← 0 to 255 do                                          ▷ O(1)
4:     letras[i] ← NULL                                             ▷ O(1)
5:     end for
6:     var nodo nuevo ← <NULL,letras,NULL>                          ▷ O(1)
7:     res ← &nuevo                                                 ▷ O(1)
8: end function
```

```
1: function iDefinir(in/out d: estr, in k: string, in s: σ)
2:     nat i ← 0                                                    ▷ O(1)
3:     puntero(nodo) actual ← d                                     ▷ O(1)
4:     while (i < |k|) do                                           ▷ O(|k|)
5:         if actual ⟶ caracteres[ord(k[i])] = NULL then            ▷ O(1)
6:             puntero(nodo) anterior ← actual
7:             actual ⟶ caracteres[ord(k[i])] ← iVacio()            ▷ O(1)
8:             actual ⟶ padre ← anterior
9:         else
10:            actual ← (actual ⟶ caracteres[ord(k[i])])            ▷ O(1)
11:        end if
12:        i ← i + 1                                                ▷ O(1)
13:    end while
14:    actual ⟶ significado ← &copiar(s)                            ▷ O(1)
15: end function
```

```
1: function iObtener(in d: estr, in k: string)⟶ res : σ
2:     nat i ← 0                                                    ▷ O(1)
3:     puntero actual ← d                                           ▷ O(1)
4:     while i < |k| do                                             ▷ O(|k|)
5:         actual ← (actual ⟶ caracteres[ord(k[i])])                ▷ O(1)
6:         i ← i + 1
7:     end while
8:     res ← *(actual ⟶ significado)
9: end function
```

```
1: function iBorrar(in/out d: estr, in k: string)
2:     puntero(nodo) actual ← d
3:     for i ← 0 to |k|                                                    ▷ O(1)
4:     actual ← (actual⟶caracteres[ord(k[i])])                             ▷ O(1)
5:     end for
6:     (actual⟶significado) ← NULL var puntero(nodo) camino ← NULL
7:     while (actual⟶significado = NULL) or todosNULL(actual⟶caracteres) do  ▷ O(|k|)
8:         camino ← actual                                                 ▷ O(1)
9:         actual ← (actual ⟶padre)
10:        delete camino
11:    end while
```

```
1: function iDefinido?(in d: estr, in k: string)⟶ res : bool
2:     nat i ← 0                                                           ▷ O(1)
3:     puntero actual ← d                                                  ▷ O(1)
4:     bool def ← true                                                     ▷ O(1)
5:     while (i < |k| and def) do                                          ▷ O(|k|)
6:         if actual ⟶ caracteres[ord(k[i])] = NULL then                   ▷ O(1)
7:             def ← false                                                 ▷ O(1)
8:         else
9:             actual ← actual ⟶ caracteres[ord(k[i])]                     ▷ O(1)
10:            i ← i + 1                                                   ▷ O(1)
11:        end if
12:    end while
13:    res ← def ∧ ¬(actual ⟶ significado(NULL))                           ▷ O(1)
14: end function=0
```

1.4. Servicios Usados

Requerimientos sobre el Tipo

- La función |**x**| debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- La función |**x**| debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- Las operaciones deben realizarse por referencia.
- Debe proveer una operación **Copia** que devuelve una nueva instancia de la secuencia pero que es independiente de la actual, con complejidad $\mathcal{O}(n)$ en el caso peor.
- Debe proveer un **iterador** para avanzar que comienza en el primero elemento de la secuencia.
- Debe proveer un **iterador** para retroceder que comienza en el último elemento de la secuencia.
- Las operaciones **CrearIt, Siguiente, Anterior, TieneSiguiente, TieneAnterior** deben tener complejidad $\mathcal{O}(1)$ en el caso peor.

Donde n es la longitud de la palabra.