

Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Practico 2

Especificacion

Grupo De TP Algo2

Integrante	LU	Correo electrónico
Fernando Castro	627/12	fernandoarielcastro92@gmail.com
Philip Garrett	318/14	garrett.phg@gmail.com
Gabriel Salvo	564/14	gabrielsalvo.cap@gmail.com
Bernardo Tuso	792/14	btuso.95@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulos	3
2. Módulo Coordenada	4
3. Módulo Mapa	6
4. Módulo Juego	7
5. Módulo Diccionario Acotado(<i>coordenada</i> , σ)	9
5.0.1. Especificación de las operaciones auxiliares utilizadas en la interfaz	10
6. Módulo $\tilde{DiccionarioString}(\alpha)$	11
6.1. Interfaz	11
6.1.1. Operaciones básicas de Diccionario String(α)	11
6.1.2. Operaciones Básicas Del Iterador	12
6.1.3. Representación de $\tilde{DiccionarioString}(\alpha)$	13
6.1.4. Invariante de Representación \tilde{n}	13
6.1.5. Función de Abstracción	14
6.2. Algoritmos	14

1. Modulos

Esta es un disenio(no tengo enie, paja) de la especificacion del Trabajo Practico 2 del 2^{do} cuatrimestre del 2016 presentada por la cathedra para la realizacion del Trabajo Practico 2. Ver enunciado:

<http://www.dc.uba.ar/materias/aed2/2016/2c/descargas/tps/tp2/view>

2. Módulo Coordenada

Interfaz

usa: NAT, BOOL.

se explica con: COORDENADA.

géneros: *coor*.

CREARCOORDENADA(**in** $x : \text{Nat}$, **in** $y : \text{Nat}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{crearCoor}(x, y)\}$

Complejidad: $\Theta(1)$

Descripción: Crea una nueva coordenada

LATITUD(**in** $c : \text{coor}$) $\rightarrow res : \text{Nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{latitud}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la latitud de la coordenada pasada por parametro

LONGITUD(**in** $c : \text{coor}$) $\rightarrow res : \text{Nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{longitud}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la longitud de la coordenada pasada por parametro

DISTEUCLIDEA(**in** $c1 : \text{coor}$, **in** $c2 : \text{coor}$) $\rightarrow res : \text{Nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c1, c2)\}$

Complejidad: $O(1)$

Descripción: Devuelve la distancia euclidea entre las dos coordenadas

COORDENADAARRIBA(**in** $c : \text{coor}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{coordenadaArriba}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de arriba

COORDENADAABAJO(**in** $c : \text{coor}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{latitud}(c) > 0\}$

Post $\equiv \{res =_{\text{obs}} \text{coordenadaAbajo}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de abajo

COORDENADAALADERECHA(**in** $c : \text{coor}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{coordenadaALaDerecha}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la derecha

COORDENADAALAIZQUIERDA(**in** $c : \text{coor}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{longitud}(c) > 0\}$

Post $\equiv \{res =_{\text{obs}} \text{coordenadaALaIzquierda}(c)\}$

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la izquierda

Representación

Coordenada se representa con **estr**

donde **estr** es $\text{tupla}(x: \text{Nat}, y: \text{Nat})$

1) True

Abs(e): $\text{estre} \rightarrow \text{Coor Rep}(e)$

$c : \text{Coor}$ tq $e.x = \text{latitud}(c)$ y $e.y = \text{longitud}(c)$

3. Módulo Mapa

Interfaz

usa: NAT, BOOL, COORDENADA, CONJ(α).

se explica con: MAPA.

géneros: map.

CREARMAPA() $\rightarrow res : \text{Mapa}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{crearMapa}(x, y)\}$

Complejidad: $O(1)$

Descripción: Crea un nuevo mapa

AGREGARCOORDENADA(in/out $m : \text{map}$, in $c : \text{coord}$) $\rightarrow res : \text{itConj}(\text{coord})$

Pre $\equiv \{m =_{\text{obs}} m_0\}$

Post $\equiv \{m =_{\text{obs}} \text{agregarCoord}(c, m_0)\}$

Complejidad: $O\left(\sum_{c' \in \text{coordenadas}(m)}^{\ell} \text{equal}(c, c')\right)$

Descripción: Agrega una coordenada al mapa y devuelve el iterador a la coordenada agregada

COORDENADASS(in $m : \text{map}$) $\rightarrow res : \text{itConj}(\text{coord})$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{coordenadas}(m)\}$

Complejidad: $O(1)$

Descripción: Devuelve un iterador al conjunto de coordenadas del mapa

Representación

Mapa se representa con **estr**

donde **estr** es $\text{tupla}(\text{coordenadas} : \text{ConjLineal}, \text{ancho} : \text{Nat})$

1) El ancho es igual al maximo de las coordenadas X.

1) $e.\text{Ancho} = \text{Max}(\text{TT1}(e.\text{Coordenadas}))$

$\text{Abs}(e)$: estre - \rightarrow Mapa Rep(e)

$m : \text{Mapa}$ tq $e.\text{coordenadas} = \text{coordenadas}(m)$

4. Módulo Juego

Interfaz

usa: MAPA, COORDENADA.

se explica con: JUEGO.

géneros: juego.

CREARJUEGO(**in** m : mapa) $\rightarrow res$: juego

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{crearJuego}(m_0) \wedge \text{mapa}(res) =_{\text{obs}} m_0\}$

Complejidad: $\Theta(MUCHO)$

Descripción: Crea el nuevo juego, revisar la complejidad

AGREGARPOKEMON(**in/out** j : juego, **in** c : coor, **in** p : pokemon) $\rightarrow res$: itPokemon

Pre $\equiv \{j =_{\text{obs}} j_0 \wedge \text{puedoAgregarPokemon}(c, j_0)\}$

Post $\equiv \{j =_{\text{obs}} \text{agregarPokemon}(p, c, j_0)\}$

Complejidad: $O(|P| + EC * \log(EC))$

Descripción: EC es la maxima cantidad de jugadores esperando para atrapar un pokemon. —P— es el nombre mas largo para un pokemon en el juego

AGREGARJUGADOR(**in/out** j : juego) $\rightarrow res$: Nat

Pre $\equiv \{j =_{\text{obs}} j_0\}$

Post $\equiv \{j =_{\text{obs}} \text{agregarJugador}(j_0) \wedge res = \#jugadores(j_0) + \#expulsados(j_0)\}$

Complejidad: $O(J)$

Descripción: Agrega el jugador en el conjLineal, el iterador que devuelve el agregar se guarda en un vector donde la posicion es el id del jugador que voy a devolver

CONECTARSE(**in/out** j : juego, **in** id : Nat, **in** c : coor)

Pre $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_L \neg \text{estaConectado}(id, j_0) \wedge \text{posExistente}(c, \text{mapa}(j_0))\}$

Post $\equiv \{j =_{\text{obs}} \text{conectarse}(id, c, j_0)\}$

Complejidad: $O(\log(EC))$

Descripción: Conecta al jugador pasado por parametro en la coordenada indicada

DESCONECTARSE(**in/out** j : juego, **in** id : Nat)

Pre $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_L \text{estaConectado}(id, j_0)\}$

Post $\equiv \{j =_{\text{obs}} \text{desconectarse}(id, j_0)\}$

Complejidad: $O(\log(EC))$

Descripción: Desconecta al jugador pasado por parametro

MOVEVERSE(**in/out** j : juego, **in** id : Nat, **in** c : coor)

Pre $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_L \text{estaConectado}(id, j_0) \wedge \text{posExistente}(c, \text{mapa}(j_0))\}$

Post $\equiv \{j =_{\text{obs}} \text{moverse}(c, id, j_0)\}$

Complejidad: $O((PS + PC) * |P| + \log(EC))$

Descripción: Mueve al jugador pasado por parametro a la coordenada indicada

Representación

Juego se representa con estr

donde **estr** es `tupla(pokemones: diccTrie, jugadores: conjLineal , jugadoresPorPosicion: conjHash , pokemonesPorPosicion: conjHash , mapa: Mapa , pT: Nat)`

Jugador se representa con jug

donde **jug** es `tupla(id: Nat, posicion: Coordenada , estaConectado: Bool , sanciones: Nat , pokeCapturados: ConjLineal)`

Pokemon se representa con poke

donde **poke** es `tupla(tipo: String, contador: Nat , jugadoresEnRango: diccHeap , salvaje: Bool)`

Rep: estr -> bool

- 1) La suma de toos los significados de pokemons es igual al PT
- 2) Todos las posiciones de jugPorPosicion esta contenida en el heap
- 3) Idem pokePorPosicion
- 4) Todo jugador que esta conectado y no expulsado, existe en jugPorPosicion
- 5) Para cada posicion hay un jugador en jugPorPosicion que pertenece a jugadores
- 6) Para cada pos en pokePorPosicion hay pokemon en pokemones
- 7) Para cada posicion en jugadoresEnRango, sus jugadores estan contenidos en jugadores
- 8) Para cada jugador en jugadores: si no esta expulsado, sus pokemones estan contenidos en pokemones del juego y no estan en pokemonesPorPosicion; y si esta conectado, su posicion pertenece al mapa del juego
- 9) Para cada pokemon en pokemones, si es salvaje: su contador es menor a 10, su posicion pertenece al mapa del juego y pertenece a pokemonEnPosicion

Abs(e): estre - > Jugo Rep(e) pGo: Juego tq e.mapa = mapa(pGo) y e.jugadores = jugadores(pGo) y luego

(Para todo j : jugador) j pertenece e.jugadores impluego

j.sanciones = sanciones(j, pGo) ((j pertenece expulsados(pGo) y j.sanciones != 10)

oluego (j.pokesCapturados = pokemones(j,pGo) y j.estaConectado = estaConectad(j,pGo)

y j.estaConectado impluego j.pos = posicion(j,pGo))) y

(Para todo p : pokemon) p pertenece c.pokemones impluego (Para todo j : Jugador)

j pertenece e.jugadores y luego p pertenece pokemones(j,pGo) o [(Para todo c : coord)

c pertenece e.mapa.coordenadas y luego p = pokemonEnPos(c,pGo) y cantMovParaCap(c,pGo)

p.contador]

5. Módulo Diccionario Acotado(*coordenada*, σ)

El módulo Diccionario Acotado provee un diccionario por posiciones en el que se puede definir, borrar, y testear si hay un valor en una posición en tiempo $O(1)$.

El principal costo de pago al crear la estructura, dado de cuesta tiempo lineal *ancho* por *largo*.

Interfaz

parámetros formales

géneros *coordenada*, σ

se explica con: `DICCACOTADO(Nat, σ)`,

géneros: `DiccAc(coordenada, σ)`.

Trabajo Practico 2 Operaciones básicas de tabla

VACÍO(**in** *Nat*: *a* *ncho*, **in** *Nat*: *l* *argo*) \rightarrow *res* : `DiccAc(coordenada, σ)`

Pre $\equiv \{\text{ancho} > 0 \wedge \text{largo} > 0\}$

Post $\equiv \{res =_{\text{obs}} \text{vacío}(\text{ancho} * \text{largo})\}$

Complejidad: $\Theta(\text{ancho} * \text{largo})$

Descripción: genera un diccionario vacía.

DEFINIR(**in/out** *t*: `DiccAc(coordenada, σ)`, **in** *c*: *coordenada*, **in** *s*: σ)

Pre $\equiv \{t =_{\text{obs}} t_0\}$

Post $\equiv \{t =_{\text{obs}} \text{definir}(t, c, s)\}$

Complejidad: $\Theta(1)$

Descripción: define el significado *s* en la tabla, en la posición representada por *c*.

Aliasing: Hay aliasing, pero no se como explicarlo TODO

DEFINIDO?(**in** *t*: `tabla(coordenada, σ)`, **in** *c*: *coordenada*) \rightarrow *res* : `bool`

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{def?}(t, c)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve `true` si y sólo *c* tiene un valor en la tabla.

SIGNIFICADO(**in** *t*: `tabla(coordenada, σ)`, **in** *c*: *coordenada*) \rightarrow *res* : σ

Pre $\equiv \{\text{def?}(t, c)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Significado}(t, c))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve el valor en la posición *c* de *t*.

BORRAR(**in/out** *t*: `tabla(coordenada, σ)`, **in** *c*: *coordenada*)

Pre $\equiv \{t = t_0 \wedge \text{def?}(t, c)\}$

Post $\equiv \{t =_{\text{obs}} \text{borrar}(t_0, c)\}$

Complejidad: $\Theta(1)$

Descripción: elimina el valor en la posición *c* en *t*.

5.0.1. Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD DICCAcOTADO(NAT,SIGNIFICADO)

géneros diccAc(Nat,Significado)

exporta diccAc(Nat,Significado), generadores, observadores, borrar, claves

usa NAT, BOOL, CONJ(NAT)

,

igualdad observacional

$$(\forall d, d' : \text{Dicc}(\text{Nat}, \sigma)) \left(d =_{\text{obs}} d' \iff \left((\forall c : \text{Nat}) (\text{enRango}(c,d) =_{\text{obs}} \text{enRango}(c,d') \wedge_L) \right) \right)$$

observadores básicos

enRango : Nat × diccAc(Nat × significado) → Bool

def? : Nat c × diccAc(Nat × significado) d → Bool {enRango(c,d)}

obtener : Nat c × diccAc(Nat × significado) d → significado {enRango(c,d) ∧_L def?(c,d)}

generadores

vacio : Nat r → diccAc(Nat,significado) {r > 0}

definir : Nat c × significado s × diccAc(Nat × significado) d → enRango(c,d)

otras operaciones

borrar : Nat c × diccAc(Nat × significado) d → diccAc(Nat,significado) {enRango(c,d) ∧_L def?(c,d)}

claves : diccAc(Nat × significado) → conj(Nat)

axiomas ∀c, k: Nat ∀d: diccAc(Nat,significado) ∀s: significado

enRango(c, vacio(r)) ≡ c < r

def?(c, vacio(r)) ≡ false

enRango(c, definir(k,s,d)) ≡ enRango(c,d)

def?(c, definir(k,s,d)) ≡ c = k ∨ def?(c,d)

obtener(c, definir(k,s,d)) ≡ **if** c = k **then** s **else** obtener(c,d) **fi**

borrar(c, definir(k,s,d)) ≡ **if** c = k **then**
 if def?(c,d) **then** borrar(c,d) **else** d **fi**
 else
 definir(k,s,borrar(c,d))
 fi

claves(vacio) ≡ ∅

claves(definir(c,s,d)) ≡ Ag(c,claves(d))

Fin TAD

6. \tilde{A}^3 duloDiccionarioString(α)

Se representa mediante un Árbol n-ario con invariante de trie. Las claves son strings y permite acceder a un significado en un tiempo en el peor caso igual a la longitud de la palabra (string) más larga y definir un significado en el mismo tiempo más el tiempo de copy(s) ya que los significados se almacenan por copia.

6.1. Interfaz

parametros formales

gñeros: α .

funcion: COPIAR(**in** $s : \alpha$) $\rightarrow res : \alpha$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} s\}$

Complejidad: $O(\text{copy}(s))$

Descripción: funcion de copia de α .

se explica con: DICCIONARIO(STRING, α).

gñeros: diccString(α), itDiccString(α).

6.1.1. Operaciones básicas de Diccionario String(α)

CREARDICCIONARIO()

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} \text{vacío}\}$

Complejidad: $O(1)$ Justificación: Se crea un arreglo de posiciones inicializadas con null y una lista vacía

Descripción: Crea un diccionario vacío.

DEFINIDO?(**in** $d : \text{diccString}(\alpha)$, **in** $c : \text{string}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} \text{def?}(d, c)\}$

Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave, recorriéndola por una parte de la clave (caracteres)

Descripción: Devuelve true si la clave está definida en el diccionario y false en caso contrario.

DEFINIR(**in/out** $d : \text{diccString}(\alpha)$, **in** $c : \text{string}$, **in** $s : \alpha$)

Pre $\equiv \{d =_{obs} d_0\}$

Post $\equiv \{d =_{obs} \text{definir}(c, s, d_0)\}$

Complejidad: $O(|n_m| + \text{copy}(s))$ Justificación: Debe definir la clave, recorriéndola por una parte de la clave y después

Descripción: Define la clave con el significado

Aliasing: Almacena una copia de s .

OBTENER(**in** $d : \text{diccString}(\alpha)$, **in** $c : \text{string}$) $\rightarrow res : \alpha$

Pre $\equiv \{\text{def?}(c, d)\}$

Post $\equiv \{\text{alias}(res =_{obs} \text{obtener}(c, d))\}$

Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave, recorriéndola por una parte de la clave (caracteres)

Descripción: Devuelve el significado correspondiente a la clave.

Aliasing: Devuelve el significado almacenado en el diccionario, por lo que se es modificable si y sólo si lo es.

ELIMINAR(**in/out** $d : \text{diccString}(\alpha)$, **in** $c : \text{string}$)

Pre $\equiv \{d =_{obs} d_0 \wedge \text{def?}(d, c)\}$

Post $\equiv \{d =_{obs} \text{borrar}(d_0, c)\}$

Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave, recorriéndola por una parte de la clave (caracteres) y eliminarla

Descripción: Borra la clave del diccionario si está definido.

CREARITCLAVES(**in** $d : \text{diccString}(\alpha)$) $\rightarrow res : \text{itConj}(\text{String})$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c)) \wedge \text{vacía?}(\text{Anteriores}(res))\}$

Complejidad: $O(1)$

Descripción: Crea un Iterador de Conjunto en base a la interfaz del iterador de Conjunto Lineal

6.1.2. Operaciones Básicas Del Iterador

Este iterador permite recorrer el trie sobre el que está implementado el diccionario para obtener de cada clave los significados. Las claves de los elementos iterados no pueden modificarse nunca por cuestiones de implementación. *El iterador se invalida si se eliminan los elementos siguientes del iterador sin utilizar la función EliminarSiguiente o si se modifican los elementos anteriores del iterador sin utilizar la función Anterior.*

CREARIT(in $d: \text{DiccString}(\alpha) \rightarrow res: \text{itDiccString}(\alpha)$)

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(\text{esPermutación}(\text{SecuSuby}(res), d)) \wedge \text{vacía}(\text{Anteriores}(res))\}$

Complejidad: $O(1)$

Descripción: crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).

Aliasing: El iterador se invalida si y sólo si se eliminan los elementos siguientes del iterador sin utilizar la función EliminarSiguiente o si se modifican los elementos anteriores del iterador sin utilizar las funciones del iterador. @

HAYSIGUIENTE(in $it: \text{itDiccString}(\alpha) \rightarrow res: \text{bool}$)

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} \text{haySiguiente?}(it)\}$

Complejidad: $O(1)$

Descripción: devuelve true si y sólo si hay elementos para avanzar. @

HAYANTERIOR(in $it: \text{itDiccString}(\alpha) \rightarrow res: \text{bool}$)

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} \text{hayAnterior?}(it)\}$

Complejidad: $O(1)$

Descripción: devuelve true si y sólo si hay elementos para retroceder. @

SIGUIENTESIGNIFICADO(in $it: \text{itDiccString}(\alpha) \rightarrow res: \alpha$)

Pre $\equiv \{\text{haySiguiente?}(it)\}$

Post $\equiv \{\text{alias}(res =_{obs} \text{haySiguiente?}(it).\text{significado})\}$

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento siguiente del iterador

Aliasing: res es modificable si y sólo si se modifican los elementos siguientes. @

ANTERIORESIGNIFICADO(in $it: \text{itDiccString}(\alpha) \rightarrow res: \alpha$)

Pre $\equiv \{\text{hayAnterior?}(it)\}$

Post $\equiv \{\text{alias}(res =_{obs} \text{hayAnterior?}(it).\text{significado})\}$

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento anterior del iterador

Aliasing: res es modificable si y sólo si se modifican los elementos anteriores. @

AVANZAR(in/out $it: \text{itDiccString}(\alpha)$)

Pre $\equiv \{it = it_0 \wedge \text{haySiguiente?}(it)\}$

Post $\equiv \{it =_{obs} \text{avanzar}(it_0)\}$

Complejidad: $O(1)$

Descripción: avanza a la posición siguiente del iterador.

RETROCEDER(in/out $it: \text{itDiccString}(\alpha)$)

Pre $\equiv \{it = it_0 \wedge \text{hayAnterior?}(it)\}$

Post $\equiv \{it =_{obs} \text{retroceder}(it_0)\}$

Complejidad: $O(1)$

Descripción: retrocede a la posición anterior del iterador.

6.1.3. Representaci3ndeDiccionarioString(α)

Diccionario String(α) se representa con estr

donde estr es tupla(raiz: arreglo(puntero(Nodo)), listaIterable: lista(puntero(Nodo)))

donde Nodo es tupla(arbolTrie: arreglo(puntero(Nodo)),
info: α ,
infoValida: bool,
infoEnLista: iterador(listaIterable))

6.1.4. Invariante de Representaci3n

- (I) Raiz es la raiz del arbol con invariante de trie y es un arreglo de 27 posiciones.
- (II) Cada uno de los elementos de la lista tiene que ser un puntero a un Nodo del trie.
- (III) Nodo es una tupla que contiene un arreglo de 27 posiciones con un puntero a otro Nodo en cada posicion ,un elemento info que es el alfa que contiene esa clave del arbol, un elemento infoValida y un elemento iterador que es un puntero a un nodo de la lista enlazada.
- (IV) El iterador a la lista enlazada de cada nodo tiene que apuntar al elemento de la lista que apunta al mismo Nodo.
- (V) Cada uno de los nodos de la lista apunta a un nodo del arbol cuyo infoEnLista apunta al mismo nodo de la lista.

($\forall c$: diccString((α)))()
Rep : estr \rightarrow bool
Rep(#2) \equiv true \iff
longitud(e.raiz)==27 \wedge_L
($\forall i \in [0..longitud(e.raiz))$)
(((\neg e.raiz[i] == NULL) \Rightarrow_L nodoValido(raiz[i])) \wedge (*e.raiz[i].infoValida == true \Rightarrow_L
iteradorValido(raiz[i])) \wedge
listaValida(e.listaIterable)

nodoValido : puntero(Nodo) *nodo* \rightarrow bool
iteradorValido : puntero(Nodo) *nodo* \rightarrow bool

nodoValido(*nodo*) \equiv longitud(*nodo.arbolTrie) == 27 \wedge_L
($\forall i \in [0..longitud(*nodo.arbolTrie))$)
((\neg *nodo.arbolTrie[i] == NULL) \Rightarrow_L nodoValido(*nodo.arbolTrie[i]))

iteradorValido(*nodo*) \equiv PunteroValido(*nodo*) \wedge_L
($\forall i \in [0..longitud(*nodo.arbolTrie))$)
((*nodo.arbolTrie[i].infoValida == true) \Rightarrow_L iteradorValido(*nodo.arbolTrie[i]))

PunteroValido(*nodo*) \equiv El iterador perteneciente al nodo (infoEnLista) apunta a un nodo de listaIterable (lista(puntero(Nodo))) cuyo puntero apunta al mismo nodo pasado por par3metro. Es decir se trata de una referencia circular.

listaValida(lista) \equiv Cada nodo de la lista tiene un puntero a un nodo de la estructura cuyo infoEnLista (iterador) apunta al mismo nodo. Es decir se trata de una referencia circular.

6.1.5. Función de Abstracción

$Abs : \text{estr } e \rightarrow \text{diccString}(\alpha)$ {Rep(#3)}
 $Abs(\#3) =_{\text{obs}} d : \text{diccString}(\alpha) \mid (\forall s : \text{string})(\text{def?}(d, s) =_{\text{obs}}$
 $\quad \text{Definido?}(d, s) \wedge$
 $\quad \text{def?}(d, s) \Rightarrow_L \text{obtener}(s, d) =_{\text{obs}}$
 $\quad \text{Obtener}(d, s)$
 $\quad)$

6.2. Algoritmos

style=alg

iCrearDiccionario $(\rightarrow res : \text{estr})$

Pre $\equiv \text{true}$

$\text{arreglo}(\text{puntero}(\text{Nodo})) : res.raiz \leftarrow \text{CrearArreglo}(27)$ $\triangleright O(1)$

$nat : i \leftarrow 0$ $\triangleright O(1)$

while $i < \text{long}(res.raiz)$ **do** $\triangleright O(1)$

$res.raiz[i] \leftarrow \text{NULL}$ $\triangleright O(1)$

end while

$res.listaIterable \leftarrow \text{Vacía}()$ $\triangleright O(1)$

Complejidad: $O(1)$

Justificación: $\tilde{O}(1)$: Crea un arreglo de 27 posiciones y lo recorre inicializándolo en NULL. Luego crea una lista vacía.

Post $\equiv res =_{\text{obs}} \text{vacío}()$

iDefinido? $(\text{in } d : \text{estr}, \text{in } c : \text{string}) \rightarrow res : \text{bool}$

Pre $\equiv \text{true}$

$nat : i \leftarrow 0$ $\triangleright O(1)$

$nat : letra \leftarrow \text{ord}(c[0])$ $\triangleright O(1)$

$\text{puntero}(\text{Nodo}) : arr \leftarrow d.raiz[letra]$ $\triangleright O(1)$

while $i < \text{longitud}(c) \wedge \neg arr = \text{NULL}$ **do** $\triangleright O(|n_m|)$

$i \leftarrow i + 1$ $\triangleright O(1)$

$letra \leftarrow \text{ord}(c[i])$ $\triangleright O(1)$

$arr \leftarrow (*arr).arbolTrie[letra]$ $\triangleright O(1)$

end while

if $i = \text{longitud}(c)$ **then** $\triangleright O(1)$

$res \leftarrow (*arr).infoValida$ $\triangleright O(1)$

else

$res \leftarrow \text{false}$ $\triangleright O(1)$

end if

Complejidad: $O(|n_m|)$

Justificación: $\tilde{O}(1)$: Toma el primer caracter y encuentra su posición en el arreglo. Luego itera sobre los caracteres restantes hasta

Post $\equiv res =_{\text{obs}} \text{def?}(d, c)$

iDefinir(in/out d : estr, in c : string, in s : α)

Pre $\equiv d =_{obs} d_0$

$nat : i \leftarrow 0$	$\triangleright O(1)$
$nat : letra \leftarrow ord(c[0])$	$\triangleright O(1)$
if $d.raiz[letra] = NULL$ then	$\triangleright O(1)$
$Nodo : nuevo$	$\triangleright O(1)$
$arreglo(puntero(Nodo)) : nuevo.arbolTrie \leftarrow CrearArreglo(27)$	$\triangleright O(1)$
$nuevo.infoValida \leftarrow false$	$\triangleright O(1)$
$d.raiz[letra] \leftarrow puntero(nuevo)$	$\triangleright O(1)$
end if	
$puntero(Nodo) : arr \leftarrow d.raiz[letra]$	$\triangleright O(1)$
while $i < longitud(c)$ do	$\triangleright O(n_m)$
$i \leftarrow i + 1$	$\triangleright O(1)$
$letra \leftarrow ord(c[i])$	$\triangleright O(1)$
if $arr.arbolTrie[letra] = NULL$ then	$\triangleright O(1)$
$Nodo : nuevoHijo$	$\triangleright O(1)$
$arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie \leftarrow CrearArreglo(27)$	$\triangleright O(1)$
$nuevoHijo.infoValida \leftarrow false$	$\triangleright O(1)$
$arr.arbolTrie[letra] \leftarrow puntero(nuevoHijo)$	$\triangleright O(1)$
end if	
$arr \leftarrow (*arr).arbolTrie[letra]$	$\triangleright O(1)$
end while	
$(*arr).info \leftarrow s$	$\triangleright O(copy(s))$
if $\neg(*arr).infoValida$ then	$\triangleright O(1)$
$itLista(puntero(Nodo))it \leftarrow AgregarAdelante(d.listaIterable, NULL)$	$\triangleright O(1)$
$(*arr).infoValida \leftarrow true$	$\triangleright O(1)$
$(*arr).infoEnLista \leftarrow it$	$\triangleright O(1)$
$siguiente(it) \leftarrow puntero(*arr)$	$\triangleright O(1)$
end if	

Complejidad: $O(|n_m| + copy(s))$

Justificaci3n : Itera sobre la cantidad de caracteres del String c y en caso de que alg3n caracter no est3 definido crea un

Post $\equiv d =_{obs} definir(c,s,d_0)$

iObtener(in d : estr, in c : string) $\rightarrow res : \alpha$

Pre $\equiv def?(c,d)$

$nat : i \leftarrow 0$	$\triangleright O(1)$
$nat : letra \leftarrow ord(c[0])$	$\triangleright O(1)$
$puntero(Nodo) : arr \leftarrow d.raiz[letra]$	$\triangleright O(1)$
while $i < longitud(c)$ do	$\triangleright O(n_m)$
$i \leftarrow i + 1$	$\triangleright O(1)$
$letra \leftarrow ord(c[i])$	$\triangleright O(1)$
$arr \leftarrow (*arr).arbolTrie[letra]$	$\triangleright O(1)$
end while	
$res \leftarrow (*arr).info$	$\triangleright O(1)$

Complejidad: $O(|n_m|)$

Justificaci3n : Toma el primer caracter y encuentra su posici3nenelarreglora3z.Luegoiterasobreloscaracteresrestanteshas

Post $\equiv alias(res =_{obs} obtener(c,d))$

iEliminar(in/out d : estr, in c : string)

Pre $\equiv d =_{obs} d_0 \wedge \text{def?}(d, c)$

```

  nat : i  $\leftarrow$  0  $\triangleright O(1)$ 
  nat : letra  $\leftarrow \text{ord}(c[0])$   $\triangleright O(1)$ 
  puntero(Nodo) : arr  $\leftarrow d.\text{raiz}[letra]$   $\triangleright O(1)$ 
  pila(puntero(Nodo)) : pil  $\leftarrow \text{Vacia}()$   $\triangleright O(1)$ 
  while i < longitud(c) do  $\triangleright O(|n_m|)$ 
    i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
    letra  $\leftarrow \text{ord}(c[i])$   $\triangleright O(1)$ 
    arr  $\leftarrow (*arr).\text{arbolTrie}[letra]$   $\triangleright O(1)$ 
    Apilar(pil, arr)  $\triangleright O(1)$ 
  end while
  if tieneHermanos(arr) then  $\triangleright O(1)$ 
    (*arr).infoValida  $\leftarrow$  false  $\triangleright O(1)$ 
  else
    i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
    puntero(Nodo) : del  $\leftarrow \text{tope}(pil)$   $\triangleright O(1)$ 
    del  $\leftarrow \text{NULL}$   $\triangleright O(1)$ 
    Desapilar(pil)  $\triangleright O(1)$ 
    while i < longitud(c)  $\wedge$   $\neg$ tieneHermanosEInfo(*tope(pil)) do  $\triangleright O(|n_m|)$ 
      del  $\leftarrow \text{tope}(pil)$   $\triangleright O(1)$ 
      del  $\leftarrow \text{NULL}$   $\triangleright O(1)$ 
      Desapilar(pil)  $\triangleright O(1)$ 
      i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
    end while
    if i = longitud(c) then  $\triangleright O(1)$ 
      d.raiz[ord(c[0])]  $\leftarrow \text{NULL}$   $\triangleright O(1)$ 
    end if
  end if
end if

```

Complejidad: $O(|n_m|)$

Justificaci3n : Toma el primer caracter y encuentra su posici3nenelarreglora3z.LuegocreaunapilaenO(1).Recorreelrestode

Post $\equiv d =_{obs}$ borrar(d_0, c)

tieneHermanos(in nodo: puntero(Nodo)) $\rightarrow res$: bool

Pre $\equiv \text{nodo} \neq \text{NULL}$

```

  nat : i  $\leftarrow$  0  $\triangleright O(1)$ 
  nat : l  $\leftarrow \text{longitud}((*nodo).\text{arbolTrie})$   $\triangleright O(1)$ 
  while i < l  $\wedge$   $\neg((*nodo).\text{arbolTrie}[i] = \text{NULL})$  do  $\triangleright O(1)$ 
    i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
  end while
  res  $\leftarrow$  i < l  $\triangleright O(1)$ 

```

Complejidad: $O(1)$

Justificaci3n : Recorre el arreglo de 27 posiciones en caso de que todas las posiciones del mismo tengan NULL. Como es una

Post $\equiv res =_{obs} (\exists i \in [0..\text{longitud}(*nodo.\text{arbolTrie})] (*nodo.\text{arbolTrie}[i] \neq \text{NULL}))$

tieneHermanosEInfo(in nodo: puntero(Nodo)) $\rightarrow res$: bool

Pre $\equiv \text{nodo} \neq \text{NULL}$

res $\leftarrow \text{tieneHermanos}(nodo) \wedge (*nodo).\text{infoValida} = \text{true}$ $\triangleright O(1)$

Complejidad: $O(1)$

Justificaci3n : Llama a la funci3ntieneHermanosqueesO(1)yverificaadem3'squeelnodocontengainformaci3nv3'lida.

Post $\equiv res =_{obs} (\exists i \in [0..\text{longitud}(*nodo.\text{arbolTrie})] (*nodo.\text{arbolTrie}[i] \neq \text{NULL})) \wedge (*nodo).\text{infoValida} = \text{true}$
