# 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. 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 catedra 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** $x$ : `Nat`, **in** $y$ : `Nat`) $\rightarrow$ $res$ : `coor`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} crearCoor(x, y)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Crea una nueva coordenada

Latitud(**in** $c$: `coor`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} latitud(c)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la latitud de la coordenada pasada por parametro

Longitud(**in** $c$: `coor`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} longitud(c)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la longitud de la coordenada pasada por parametro

distEuclidea(**in** $c1$ : `coor`, **in** $c2$ : `coor`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} distEuclidea(c1, c2)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la distancia euclidea entre las dos coordenadas

coordenadaArriba(**in** $c$: `coor`) $\rightarrow$ $res$ : `coor`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} coordenadaArriba(c)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la coordenada de arriba

coordenadaAbajo(**in** $c$: `coor`) $\rightarrow$ $res$ : `coor`
**Pre** $\equiv$ $\{latitud(c) > 0\}$
**Post** $\equiv$ $\{res =_{\text{obs}} coordenadaAbajo(c)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la coordenada de abajo

coordenadaALaDerecha(**in** $c$: `coor`) $\rightarrow$ $res$ : `coor`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} coordenadaALaDerecha(c)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve la coordenada de la derecha

coordenadaALaIzquierda(**in** $c$: `coor`) $\rightarrow$ $res$ : `coor`
**Pre** $\equiv$ $\{longitud(c) > 0\}$
**Post** $\equiv$ $\{res =_{\text{obs}} coordenadaALaIzquierda(c)\}$
**Complejidad:** $O(1)$

**Descripción:** Devuelve la coordenada de la izquierda

# Representación

`Coordenada` **se representa con** `estr`

donde `estr` es `tupla`($x$: `Nat`, $y$: `Nat` )

1) True
Abs(e): estre - > Coor Rep(e)
c : Coor tq e.x = latitud(c) y e.y = longitud(c)

# 3. Módulo Mapa

## Interfaz

**usa**: NAT, BOOL, COORDENADA, CONJ($\alpha$).

**se explica con**: MAPA.

**géneros**: `map`.

CREARMAPA() $\rightarrow$ *res* : `Mapa`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} crearMapa(x, y)$}
**Complejidad:** $O(1)$
**Descripción:** Crea un nuevo mapa

AGREGARCOORDENADA(**in/out** $m$: `map`, **in** $c$: `coor`) $\rightarrow$ *res* : `itConj(coor)`
**Pre** $\equiv$ {$m =_{\text{obs}} m_0$}
**Post** $\equiv$ {$m =_{\text{obs}} agregarCoor(c, m_0)$}
**Complejidad:** $O(\left(\sum_{c' \in coordenadas(m)}^{\ell} equal(c, c')\right))$
**Descripción:** Agrega una coordenada al mapa y devuelve el iterador a la coordenada agregada

COORDENADASS(**in** $m$: `map`) $\rightarrow$ *res* : `itConj(coor)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} coordenadas(m)$}
**Complejidad:** $O(1)$
**Descripción:** Devuelve un iterador al conjunto de coordenadas del mapa

## Representación

`Mapa` **se representa con** `estr`

donde `estr` es `tupla`(*coordenadas*: `ConjLineal`, *ancho*: `Nat` )

1) El ancho es igual al maximo de las coordenadas X.
1) e.Ancho = Max(TT1(e.Coordenadas))
Abs(e): estre - > Mapa Rep(e)
m : Mapa tq e.coordenadas = coordenadas(m)

# 4. Módulo Juego

## Interfaz

**usa**: MAPA, COORDENADA.

**se explica con**: JUEGO.

**géneros**: `juego`.

CREARJUEGO(**in** $m$: `mapa`) $\rightarrow$ $res$ : `juego`
**Pre** $\equiv$ {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** $\equiv$ $\{j =_{\text{obs}} j_0 \wedge puedoAgregarPokemon(c, j_0)\}$
**Post** $\equiv$ $\{j =_{\text{obs}} agregarPokemon(p, c, j_0)\}$
**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** $\equiv$ $\{j =_{\text{obs}} agregarJugador(j_0) \wedge res = \#jugadores(j_0) + \#expulsados(j_0)\}$
**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** $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} \neg estaConectado(id, j_0) \wedge posExistente(c, mapa(j_0))\}$
**Post** $\equiv$ $\{j =_{\text{obs}} conectarse(id, c, j_0)\}$
**Complejidad:** $O(log(EC))$
**Descripción:** Conecta al jugador pasado por parametro en la coordenada indicada

DESCONECTARSE(**in/out** $j$: `juego`, **in** $id$: `Nat`)
**Pre** $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} estaConectado(id, j_0)\}$
**Post** $\equiv$ $\{j =_{\text{obs}} desconectarse(id, j_0)\}$
**Complejidad:** $O(log(EC))$
**Descripción:** Desconecta al jugador pasado por parametro

MOVERSE(**in/out** $j$: `juego`, **in** $id$: `Nat`, **in** $c$: `coor`)
**Pre** $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} estaConectado(id, j_0) \wedge posExistente(c, mapa(j_0))\}$
**Post** $\equiv$ $\{j =_{\text{obs}} moverse(c, id, j_0)\}$
**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) yluego
(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 yluego p pertenece pokemones(j,pGo) o [(Para todo c : coord)
c pertenece e.mapa.coordenadas yluego p = pokemonEnPos(c,pGo) y cantMovParaCap(c,pGo)
p.contador]

# 5. Módulo Tabla de Valores(*coordenada*, $\sigma$)

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

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

## Interfaz

**parámetros formales**
**géneros** $coordenada, \sigma$

**se explica con**: DICCIONARIO($\kappa, \sigma$),

**géneros**: `tabla`($coordenada, \sigma$).

Trabajo Practico 2Operaciones básicas de tabla

VACÍO(**in** $Nat$ : `a` ncho, **in** $Nat$ : `l` argo) $\rightarrow$ $res$ : `tabla`($coordenada, \sigma$)
**Pre** $\equiv$ {ancho $> 0 \wedge$ largo $> 0$}
**Post** $\equiv$ {$res =_{\text{obs}}$ vacio}
**Complejidad:** $\Theta$(*ancho*largo*)
**Descripción:** genera una tabla vacía.

DEFINIR(**in/out** $t$ : `tabla(`$coordenada, \sigma$`)`, **in** $c$ : $coordenada$, **in** $s$ : $\sigma$)
**Pre** $\equiv$ {$t =_{\text{obs}} t_0$}
**Post** $\equiv$ {$t =_{\text{obs}}$ definir($t, c, s$)}
**Complejidad:** $\Theta(1)$
**Descripción:** define el significado $s$ en la tabla, en la posicion representada por $c$.
**Aliasing:** Hay alising, pero no se como explicarlo TODO

DEFINIDO?(**in** $t$ : `tabla(`$coordenada, \sigma$`)`, **in** $c$ : $coordenada$) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ def?($t$, $c$)}
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve `true` si y sólo $c$ tiene un valor en la tabla.

SIGNIFICADO(**in** $t$ : `tabla(`$coordenada, \sigma$`)`, **in** $c$ : $coordenada$) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {def?($t$, $c$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Significado($t$, $c$))}
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve el valor en la posicion $c$ de $t$.

BORRAR(**in/out** $t$ : `tabla(`$coordenada, \sigma$`)`, **in** $c$ : $coordenada$)
**Pre** $\equiv$ {$t = t_0 \wedge$ def?($t$, $c$)}
**Post** $\equiv$ {$t =_{\text{obs}}$ borrar($t_0, c$)}
**Complejidad:** $\Theta(1)$
**Descripción:** elimina el valor en la posicion $c$ en $t$.

# 6. MÃ³duloDiccionarioString($\alpha$)

Se representa mediante un Ã¡rbol 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) mÃ¡s larga y definir un significado en el mismo tiempo mÃ¡s el tiempo de copy(s) ya que los significados se almacenan por copia.

## 6.1. Interfaz

**parametros formales**
**gÃ©neros**: $\alpha$.
**funcion**: COPIAR(**in** $s$: $\alpha$) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ s}
**Complejidad:** $O(copy(s))$
**Descripción:** funcion de copia de $\alpha$.
**se explica con**: DICCIONARIO(STRING,$\alpha$).
**gÃ©neros**: `diccString(`$\alpha$`)`, `itDiccString(`$\alpha$`)`.

### 6.1.1. Operaciones bÃ¡sicas de Diccionario String($\alpha$)

CREARDICCIONARIO()
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ vacÃo()}
**Complejidad:** $O(1)$ JustificaciÃ³n : $SÃ³locreaunarreglode27posicionesinicializadasconnullyunalistavacÃa$
**Descripción:**$Creaundiccionariovacío.$@

DEFINIDO?(**in** $d$: `diccString`($\alpha$), **in** $c$: `string`)) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ def?($d$, $c$)}
**Complejidad:** $O(|n_m|)$ JustificaciÃ³n : $Debeaccederalaclavec, recorriendounaporunalaspartesdelaclave(caracteres)$
**Descripción:**$Devuelvetruesilaclaveestá"definidaeneldiccionarioyfalseencasocontrario.$@

DEFINIR(**in/out** $d$: `diccString`($\alpha$), **in** $c$: `string`, **in** $s$: $\alpha$)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$}
**Post** $\equiv$ {$d =_{\text{obs}}$ definir($c$, $s$, $d_0$)}
**Complejidad:** $O(|n_m|+copy(s))$ JustificaciÃ³n : $Debedefinirlaclavec, recorriendounaporunalaspartesdelaclaveydespuÃ©$
**Descripción:**$Definelaclavecconelsignificados$
**Aliasing:**$Almacenaunacopiade$s.@

OBTENER(**in** $d$: `diccString`($\alpha$), **in** $c$: `string`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {def?($c$, $d$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ obtener($c$, $d$))}
**Complejidad:** $O(|n_m|)$ JustificaciÃ³n : $Debeaccederalaclavec, recorriendounaporunalaspartesdelaclave(caracteres)$
**Descripción:**$Devuelveelsignificadocorrespondientealaclavec.$@
**Aliasing:**$Devuelveelsignificadoalmacenadoeneldiccionario, porlocualesesmodificableysólosidloes.$@

ELIMINAR(**in/out** $d$: `diccString(`$\alpha$`)`, **in** $c$: `string`)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?($d$, $c$)}
**Post** $\equiv$ {$d =_{\text{obs}}$ borrar($d_0$, $c$)}
**Complejidad:** $O(|n_m|)$ JustificaciÃ³n : $Debeaccederalaclavec, recorriendounaporunalaspartesdelaclave(caracteres)einval$
**Descripción:**$Borralaclavecdeldiccionarioysusignificado.$@

CREARITCLAVES(**in** $d$: `diccString(`$\alpha$`)`) $\rightarrow$ $res$ : `itConj(String)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(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 BÃ¡sicas 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 implementaciÃ³n.*Eliteradoresuniteradordelista*, *querecorrelistaIterableporloquesusoperacionessonidenticasaella*.

CrearIt(**in** $d$ : `diccString(`$\alpha$`)`) $\rightarrow$ $res$ : `itDiccString(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias(esPermutaciÃ³n($SecuSuby$(res),d))$\wedge$ vacia?(Anteriores($res$))}
**Complejidad:** $O(1)$
**Descripción:** crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalÃºe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
**Aliasing:** El iterador se invalida si y sÃ³*losiseeliminaelelementosiguientedeliteradorsinutilizarlafunciÃ³nEliminarSigui* *ancambiarcompletamenteantecualquieroperaciÃ³nquemodifiquedsinutilizarlasfuncionesdeliterador*.@

HaySiguiente(**in** $it$ : `itDiccString(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ haySiguiente?($it$)}
**Complejidad:** $O(1)$
**Descripción:** devuelve true si y sÃ³*losieneliteradortodavÃaquedanelementosparaavanzar*.@

HayAnterior(**in** $it$ : `itDiccString(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ hayAnterior?($it$)}
**Complejidad:** $O(1)$
**Descripción:** devuelve true si y sÃ³*losieneliteradortodavÃaquedanelementospararetroceder*.@

SiguienteSignificado(**in** $it$ : `itDiccString(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {haySiguiente?(it)}
**Post** $\equiv$ {alias(res $=_{obs}$ haySiguiente?($it$).significado)}
**Complejidad:** $O(1)$
**Descripción:** devuelve el significado del elemento siguiente del iterador
**Aliasing:** res es modificable si y sÃ³*losiitesmodficable*.@

AnteriorSignificado(**in** $it$ : `itDiccString(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {hayAnterior?(it)}
**Post** $\equiv$ {alias(res $=_{obs}$ haySiguiente?($it$).significado)}
**Complejidad:** $O(1)$
**Descripción:** devuelve el significado del elemento anterior del iterador
**Aliasing:** res es modificable si y sÃ³*losiitesmodficable*.@

Avanzar(**in/out** $it$ : `itDiccString(`$\alpha$`)`)
**Pre** $\equiv$ {it = $\text{it}_0 \wedge$ haySiguiente?(it)}
**Post** $\equiv$ {it $=_{obs}$ avanzar($\text{it}_0$)}
**Complejidad:** $O(1)$
**Descripción:** avanza a la posicion siguiente del iterador.

Retroceder(**in/out** $it$ : `itDiccString(`$\alpha$`)`)
**Pre** $\equiv$ {it = $\text{it}_0 \wedge$ hayAnterior?(it)}
**Post** $\equiv$ {it $=_{obs}$ hayAnterior?($\text{it}_0$)}
**Complejidad:** $O(1)$
**Descripción:** retrocede a la posicion anterior del iterador.

### 6.1.3. RepresentaciÃ³n de *DiccionarioString*($\alpha$)

`Diccionario String(`$\alpha$`)` **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*: $\alpha$,
*infoValida*: `bool`,
*infoEnLista*: `iterador(listaIterable)` )

### 6.1.4. Invariante de RepresentaciÃ³n

(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(($\alpha$)))()
Rep : estr $\longrightarrow$ bool
Rep(#2) $\equiv$ true $\Longleftrightarrow$
longitud(e.raiz)==27 $\wedge_{\text{L}}$
($\forall i \in$ [0..longitud(e.raiz)))
(((¬ e.raiz[i] == NULL) $\Rightarrow_{\text{L}}$ nodoValido(raiz[i])) $\wedge$ (∗e.raiz[i].infoValida == true $\Rightarrow_{\text{L}}$
iteradorValido(raiz[i]))) $\wedge$
listaValida(e.listaIterable)

nodoValido : puntero(Nodo) *nodo* $\longrightarrow$ bool
iteradorValido : puntero(Nodo) *nodo* $\longrightarrow$ bool

nodoValido(*nodo*) $\equiv$ longitud(*nodo.arbolTrie) == 27 $\wedge_{\text{L}}$
($\forall i \in$ [0..longitud(*nodo.arbolTrie)))
((¬ *nodo.arbolTrie[i] == NULL) $\Rightarrow_{\text{L}}$ nodoValido(*nodo.arbolTrie[i]))
iteradorValido(*nodo*) $\equiv$ PunteroValido(nodo) $\wedge_{\text{L}}$
($\forall i \in$ [0..longitud(*nodo.arbolTrie)))
((*nodo.arbolTrie[i].infoValida == true) $\Rightarrow_{\text{L}}$ 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 parÃ¡metro. 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 : estr $e$ $\longrightarrow$ diccString($\alpha$) {Rep(#3)}
Abs(#3) $=_{\text{obs}}$ d: diccString($\alpha$) | ($\forall s$: string)(def?($d$, $s$) $=_{\text{obs}}$
Definido?(d,s) $\wedge$
def?($d$, $s$) $\Rightarrow_{\text{L}}$ obtener($s$,$d$) $=_{\text{obs}}$
Obtener($d$,$s$)
)

## 6.2. Algoritmos

style=alg

---

**iCrearDiccionario**() $\rightarrow$ $res$ : $estr$
**Pre** $\equiv$ true

| | |
|---|---|
| $arreglo(puntero(Nodo))$ : $res.raiz \leftarrow CrearArreglo(27)$ | $\triangleright$ $O(1)$ |
| $nat$ : $i \leftarrow 0$ | $\triangleright$ $O(1)$ |
| **while** $i < long(res.raiz)$ **do** | $\triangleright$ $O(1)$ |
| $res.raiz[i] \leftarrow NULL$ | $\triangleright$ $O(1)$ |
| **end while** | |
| $res.listaIterable \leftarrow Vacia()$ | $\triangleright$ $O(1)$ |

Complejidad: $O(1)$
JustificaciÃ³n : Crea un arreglo de 27 posiciones y lo recorre inicializÃ¡ndolo en NULL. Luego crea una lista vacÃa.
**Post** $\equiv$ res $=_{obs}$ vacio()

---

**iDefinido?**(**in** $d$: `estr`), **in** $c$: `string`) $\rightarrow$ $res$ : $bool$
**Pre** $\equiv$ true

| | |
|---|---|
| $nat$ : $i \leftarrow 0$ | $\triangleright$ $O(1)$ |
| $nat$ : $letra \leftarrow ord(c[0])$ | $\triangleright$ $O(1)$ |
| $puntero(Nodo)$ : $arr \leftarrow d.raiz[letra]$ | $\triangleright$ $O(1)$ |
| **while** $i < longitud(c) \wedge \neg arr = NULL$ **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** | |
| **if** $i = longitud(c)$ **then** | $\triangleright$ $O(1)$ |
| $res \leftarrow (*arr).infoValida$ | $\triangleright$ $O(1)$ |
| **else** | |
| $res \leftarrow false$ | $\triangleright$ $O(1)$ |
| **end if** | |

Complejidad: $O(|n_m|)$
JustificaciÃ³n : Toma el primer caracter y encuentra su posiciÃ³n en el arreglo raÃz. Luego itera sobre los caracteres restantes has
**Post** $\equiv$ res $=_{obs}$ def?(d,c)

---

**iDefinir**(**in/out** $d$: `estr`, **in** $c$: `string`, **in** $s$: $\alpha$)
**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** | $\triangleright\ O(1)$ |
| $Nodo : nuevo$ | $\triangleright\ O(1)$ |
| $arreglo(puntero(Nodo)) : nuevo.arbolTrie \leftarrow CrearArrelgo(27)$ | $\triangleright\ O(1)$ |
| $nuevo.infoValida \leftarrow false$ | $\triangleright\ O(1)$ |
| $d.raiz[letra] \leftarrow puntero(nuevo)$ | $\triangleright\ O(1)$ |
| **end if** | |
| $puntero(Nodo) : arr \leftarrow d.raiz[letra]$ | $\triangleright\ O(1)$ |
| **while** $i < longitud(c)$ **do** | $\triangleright\ O(\lvert n_m \rvert)$ |
| $i \leftarrow i + 1$ | $\triangleright\ O(1)$ |
| $letra \leftarrow ord(c[i])$ | $\triangleright\ O(1)$ |
| **if** $arr.arbolTrie[letra] = NULL$ **then** | $\triangleright\ O(1)$ |
| $Nodo : nuevoHijo$ | $\triangleright\ O(1)$ |
| $arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie \leftarrow CrearArrelgo(27)$ | $\triangleright\ O(1)$ |
| $nuevoHijo.infoValida \leftarrow false$ | $\triangleright\ O(1)$ |
| $arr.arbolTrie[letra] \leftarrow puntero(nuevoHijo)$ | $\triangleright\ O(1)$ |
| **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: $O(|n_m| + copy(s))$
JustificaciÃ³n : Itera sobre la cantidad de caracteres del String c y en caso de que algÃºn caracter no estÃ© 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** | $\triangleright\ O(\lvert n_m \rvert)$ |
| $i \leftarrow i + 1$ | $\triangleright\ O(1)$ |
| $letra \leftarrow ord(c[i])$ | $\triangleright\ O(1)$ |
| $arr \leftarrow (*arr).arbolTrie[letra]$ | $\triangleright\ O(1)$ |
| **end while** | |
| $res \leftarrow (*arr).info$ | $\triangleright\ O(1)$ |

Complejidad: $O(|n_m|)$
JustificaciÃ³n$nenelarregloraÃz.Luegoiterasobreloscaracteresrestanteshas$
**Post** $\equiv$ alias(res $=_{obs}$ obtener(c,d)

**iEliminar**(**in/out** $d$: `estr`, **in** $c$: `string`)
**Pre** $\equiv$ d $=_{obs}$ $d_0$ $\wedge$ def?(d,c)

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

Complejidad: $O(|n_m|)$
JustificaciÃ³n : Toma el primer caracter y encuentra su posiciÃ³n en el arreglo raÃz. Luego crea una pila en $O(1)$. Recorre el resto de
**Post** $\equiv$ d $=_{obs}$ borrar($d_0$,c)

---

**tieneHermanos**(**in** $nodo$: `puntero(Nodo)`) $\rightarrow$ $res$ : $bool$
**Pre** $\equiv$ nodo!=NULL

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

Complejidad: $O(1)$
JustificaciÃ³n : 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..longitud(*nodo.arbolTrie)) (*nodo.arbolTrie[i] != NULL))

---

**tieneHermanosEInfo**(**in** $nodo$: `puntero(Nodo)`) $\rightarrow$ $res$ : $bool$
**Pre** $\equiv$ nodo!=NULL

| | |
|---|---|
| $res \leftarrow tieneHermanos(nodo) \wedge (*nodo).infoValida = true$ | $\triangleright\ O(1)$ |

Complejidad: $O(1)$
JustificaciÃ³n : Llama a la funciÃ³n $tieneHermanos$ que es $O(1)$ y verifica ademÃ!‘s que el nodo contenga información válida.
**Post** $\equiv$ res $=_{obs}$ ($\exists i \in$ [0..longitud(*nodo.arbolTrie)) (*nodo.arbolTrie[i] != NULL)) $\wedge$ (*nodo).infoValida = true