Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2014

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

Trabajo Práctico 2

Diseño

Grupo 17

Integrante	LU	Correo electrónico
Alejandro Candioti	784/13	amcandio@gmail.com
Guido Tamborindeguy	584/13	guido@tamborindeguy.com.ar
Martin Jedwabny	885/13	martinj@live.com.ar
Lucas Puterman	830/13	lucasputerman@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Mapa

1.1. Interfaz

se explica con: MAPA.
géneros: `map`.

1.1.1. Operaciones básicas de mapa

VACIO() $\rightarrow$ *res* : `map`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio()}
Complejidad: $O(1)$
Descripción: crea un mapa nuevo

AGREGAR(**in** *e* : `string`, **in/out** *m* : `map`)
Pre $\equiv$ {$e \notin$ estaciones(m) $\wedge$ $|e| > 0$ $\wedge$ $m =_{\text{obs}} m_0$}
Post $\equiv$ {$m =_{\text{obs}}$ agregar(e, m_0)}
Complejidad: $O(|e|)$
Descripción: agrega una estacion al mapa

CONECTAR(**in** *e*1 : `string`, **in** *e*2 : `string`, **in** *r* : `restriccion`, **in/out** *m* : `map`)
Pre $\equiv$ {$e1 \neq e2 \wedge$ ($e1 \in$ estaciones(m) $\wedge$ $e2 \in$ estaciones(m)) $\wedge_{\text{L}}$ ($\neg$ conectadas?($e1$, $e2$)) $\wedge$ $m =_{\text{obs}} m_0$}
Post $\equiv$ {$m =_{\text{obs}}$ conectar($e1$, $e2$, r, m_0)}
Complejidad: $O(|e1| + |e2|)$
Descripción: conecta dos estaciones previamente agregadas con su respectiva restriccion para la senda que forman

ESTA?(**in** *e* : `string`, **in** *m* : `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ $e \in$ estaciones(m)}
Complejidad: $O(|e|)$
Descripción: verifica si la estacion fue agregada a la ciudad

CONECTADAS?(**in** *e*1 : `string`, **in** *e*2 : `string`, **in** *m* : `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {($e1 \in$ estaciones(m) $\wedge$ $e2 \in$ estaciones(m)) $\wedge$ $m =_{\text{obs}} m_0$}
Post $\equiv$ {*res* $=_{\text{obs}}$ conectadas?($e1$, $e2$, m)}
Complejidad: $O(|e1| + |e2|)$
Descripción: Se fija si las dos estaciones estan conectadas segun el mapa

RESTRICCION(**in** *e*1 : `string`, **in** *e*2 : `string`, **in** *m* : `map`) $\rightarrow$ *res* : `restriccion`
Pre $\equiv$ {($e1 \in$ estaciones(m) $\wedge$ $e2 \in$ estaciones(m)) $\wedge_{\text{L}}$ (conectadas?($e1$, $e2$))}
Post $\equiv$ {*res* $=_{\text{obs}}$ restriccion($e1$, $e2$, m)}
Complejidad: $O(|e1| + |e2|)$
Descripción: devuelve la restriccion correspondiente a la senda que conecta las dos estaciones en el mapa

ESTACIONES(**in** *m* : `map`) $\rightarrow$ *res* : `itLista(string)`
Pre $\equiv$ {true}
Post $\equiv$ {SecuSuby(res) $=_{\text{obs}}$ estaciones(m)}
Complejidad: $O(1)$
Descripción: devuelve un iterador para las estaciones del mapa

SENDAS(**in** *m* : `map`) $\rightarrow$ *res* : `itLista(tupla(string, string))`
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion?(res, sendas(m))}
Complejidad: $O(1)$
Descripción: devuelve un iterador para los pares de estaciones que forman una senda

1.1.2. Operaciones Auxiliares del TAD

sendas : mapa m $\longrightarrow$ lista(tupla(string, string))
sendas(m) $\equiv$ compararTodos(estaciones(m), estaciones(m))

compararTodos : lista(estacion) $l1$ $\times$ lista(estacion) $l2$ $\times$ mapa m $\longrightarrow$ lista(tupla(string, string))
{estan?(l1, estaciones(m)) $\wedge$ estan?(l2, estaciones(m))}
compararTodos($l1, l2, m$) $\equiv$ **if** vacia?($l1$) **then**
<>
else
compararUno(prim($l1$), $l2$) & compararTodos(fin($l1$, $l2$))
fi
compararUno : estacion e $\times$ lista(estacion) l $\times$ mapa m $\longrightarrow$ lista(tupla(string, string))
{esta?(e, estaciones(m)) $\wedge$ estan?(l, estaciones(m))}
compararUno(e, l, m) $\equiv$ **if** vacia?(l) **then**
<>
else
if $e1$ >prim(l) $\wedge$ conectadas?($e1, prim$(l), m) **then**
<$e1, prim$(l)> $\bullet$ compararUno(e, fin(l))
else
compararUno(e, fin(l))
fi
fi
estan? : lista(α) $l1$ $\times$ lista(α) $l2$ $\longrightarrow$ bool
estan?($l1, l2$) $\equiv$ vacia?($l2$) $\vee_{\text{L}}$ ($\neg$vacia?($l1$) $\wedge_{\text{L}}$ (esta?(prim($l1$), $l2$) $\wedge$ estan?(fin($l1$), sacar(prim($l1$), $l2$))))

sacar : α e $\times$ lista(α) l $\longrightarrow$ bool
sacar(e, l) $\equiv$ **if** vacia?(l) **then** <> **else** **if** prim($l = e$) **then** sacar(e, fin(l)) **else** $e \bullet$ sacar(e, fin(l)) **fi** **fi**
Obs: α debe ser comparable con la funcion =.

esPermutacion? : lista(α) $l1$ $\times$ lista(α) $l2$ $\longrightarrow$ bool
esPermutacion?($l1, l2$) $\equiv$ estan?($l1, l2$) $\wedge$ estan?($l2, l1$)

1.2. Representacion

1.2.1. Representación de mapa

`map` se representa con `estr`

donde `estr` es `tupla`(*estaciones*: `lista(string)`,
sendas: `lista(tupla(e1: string, e2: string))`,
restricciones: $\texttt{dicc}_T$`(`$\texttt{dicc}_T$`(restriccion))`)

1.2.2. Invariante de Representación

(I) Las estaciones son las mismas que las claves de las sendas

(II) No esta definida una clave del diccionario dentro de sus diccionarios hijos

(III) Los diccionarios hijos de una clave estan definidos en el diccionario original

(IV) Las claves de los hijos diccionarios de un item del diccionario tienen menor orden lexicografico que el padre

(V) Las combinaciones definidas en las restricciones son las mismas que la lista de sendas

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ ($\forall a$: string)($a \in e$.estaciones <=> def?(a, e.restricciones)) $\wedge$
($\forall c$, s: string)(def?(c, e.restricciones) $\Rightarrow_{\text{L}}$
($\neg$def?(c, obtener(c, e.restricciones)) $\wedge$
(def?(s, obtener(c, e.restricciones)) $\Rightarrow_{\text{L}}$ (def?(s, e) $\wedge$ $s < c$)))) $\wedge$
($\forall c$, s: string)(def?(c, e.restricciones) $\Rightarrow_{\text{L}}$
(def?(s, obtener(c, e.restricciones)) <=><c, s>$\in$ e.sendas))

1.2.3. Función de Abstracción

$$\text{Abs} : \text{estr } e \longrightarrow \text{mapa} \qquad \{\text{Rep}(e)\}$$

$$\begin{aligned}\text{Abs}(e) =_{\text{obs}} \text{m: mapa} \mid\ & \text{m.estaciones} =_{\text{obs}} \text{e.estaciones} \wedge \\ & (\forall c, s: \text{string})((c \in \text{estaciones}(e) \wedge s \in \text{estaciones}(e) \wedge c < s) \Rightarrow \\ & ((\text{def?}(s, \text{obtener}(c, e)) =_{\text{obs}} \text{conectadas?}(c, s, m)) \wedge_{\text{L}} \\ & (\text{def?}(s, \text{obtener}(c, e)) \Rightarrow_{\text{L}} (\text{obtener}(s, \text{obtener}(c, e)) =_{\text{obs}} \text{restriccion}(c, s, m)))))\end{aligned}$$

1.3. Algoritmos

iVacio () → res: estr

```
res.estaciones ← Vacia()                                   O(1)
res.sendas ← CrearDicc()                                   O(1)
```

Complejidad : $O(1)$

iAgregar (**in** e: `string`, **in/out** m: `estr`)

```
Agregar(e, m.estaciones)                                   O(1)
Definir(e, m.sendas)                                       O(|e|)
```

Complejidad : $O(|e|)$

iConectar (**in** $e1$: `string`, **in** $e2$: `string`, **in** r: `restriccion`, **in/out** m: `estr`)

```
if e1 < e2 then                                                          O(1)
  Conectar(e2, e1, r, m)                                                 O(|e1| + |e2|)
else
  if ¬Definido?(e1, m.restricciones) then                                O(|e1|)
    Definir(e1, CrearDicc(), m.restricciones)                            O(|e1|)
  end if
  Definir(e2, r, Obtener(e1, m.restricciones))                           O(|e1| + |e2|)
  AgregarAdelante(m.sendas, <e1, e2>)
end if
```

Complejidad : $O(|e1| + |e2|)$

iEsta? (**in** e: `string`, **in** m: `map`) → res: bool

```
res← Definido?(e, m.restricciones)                                       O(|e|)
```

Complejidad : $O(|e|)$

iConectadas? (**in** $e1$: `string`, **in** $e2$: `string`, **in** m: `map`) → res: bool

```
res← Definido?(e1, m.restricciones) ∧L
     Definido?(e2, Obtener(e1, m.restricciones))                         O(|e1| + |e2|)
```

Complejidad : $O(|e1| + |e2|)$

iRestriccion (**in** $e1$: `string`, **in** $e2$: `string`, **in** m : `map`) → res: restriccion

```
if e1 < e2 then                                                    O(1)
  Restriccion(e2, e1, m)                                           O(|e1| + |e2|)
else
  res← Obtener(e2, Obtener(e1, m.restricciones))                   O(|e1| + |e2|)
end if
```

Complejidad : $O(|e1| + |e2|)$

iEstaciones (**in** m : `map`) → res: itLista(string)

```
res← CrearIt(e.estaciones)                                         O(1)
```

Complejidad : $O(1)$

iSendas (**in** m : `map`) → res: itLista(tupla(string, string))

```
res← CrearIt(e.sendas)                                             O(1)
```

Complejidad : $O(1)$

2. Módulo DiccionarioString(α)

2.1. Interfaz

parámetros formales
géneros α
se explica con: DICCIONARIO(STRING, α).
géneros: $\texttt{dicc}_T(\alpha)$.

2.1.1. Operaciones básicas

CREARDICC() $\rightarrow$ *res* : $\texttt{dicc}_T(\alpha)$
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio}
Complejidad: $O(1)$
Descripción: crea un diccionario vacio

DEFINIDO?(**in** *c*: `string`, **in** *d*: $\texttt{dicc}_T(\alpha)$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(*c*, *d*)}
Complejidad: $O(|c|)$
Descripción: devuelve si la clave fue previamente definida en el diccionario

DEFINIR(**in** *c*: `string` , **in** *s*: α, **in/out** *d*: $\texttt{dicc}_T(\alpha)$)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: $O(|c|+$ copy(s))
Descripción: define la clave c con el significado s en d

OBTENER(**in** *c*: `string`, **in** *d*: $\texttt{dicc}_T(\alpha)$) $\rightarrow$ *res* : α
Pre $\equiv$ {def?(*c*, *d*)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(*c*, *d*))}
Complejidad: $O(|c|)$
Descripción: devuelve el significado correspondiente a la clave en el diccinario
Aliasing: *res* es modificable si y solo si *d* es modificable.

Obs: copy es una función de α en que devuelve el costo de copiar un elemento del género α.

2.2. Representación

2.2.1. Representación de DiccionarioString(α)

$\texttt{dicc}_T(\alpha)$ **se representa con** `estr donde estr es puntero(nodo)`

donde `nodo` es `tupla`(*significado*: `puntero`(α) ,
caracteres: `arreglo[256] de puntero(nodo)`)

2.2.2. Invariante de Representación

(I) Todas las posiciones del arreglo de caracteres están definidas.

(II) No hay claves de 0 caracteres. Esto es, el nodo raiz tiene el campo significado `NULL`.

(III) No hay ciclos en el trie. Esto es, existe un número natural n tal que la cantidad de niveles del árbol está acotada por n.

(IV) Dado un nodo cualquiera del trie, existe un único camino desde la raíz hasta dicho nodo.

Rep : estr $\longrightarrow$ bool

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

$\text{finaliza} : \text{estr } e \times \text{nat} \longrightarrow \text{bool}$ $\{(\forall i\text{: nat})\ (i < 256 \Rightarrow \text{definido?}(e \to \mathit{caracteres}, i))\}$
$\text{finaliza}(e, n) \equiv n > 0 \wedge_{\text{L}} (e = \text{NULL} \vee_{\text{L}} \text{finalizaAux}(e \to \mathit{caracteres}, n-1, 0))$

$\text{finalizaAux} : \text{ad(puntero(nodo))}\ a \times \text{nat} \times \text{nat}\ k \longrightarrow \text{bool}$ $\{k \leq \text{tam}(a)\}$
$\text{finalizaAux}(a, n, k) \equiv \textbf{if } k = \text{tam}(a) \textbf{ then } \text{true } \textbf{else } \text{finaliza}(e \to \mathit{caracteres}[k], n) \wedge \text{finalizaAux}(a, n, k+1) \textbf{ fi}$

$\text{punteros} : \text{estr } e \longrightarrow \text{multiconj(puntero(nodo))}$
$\text{punteros}(e) \equiv \textbf{if } e = \text{NULL } \textbf{then } \emptyset \textbf{ else } \text{punterosAux}(e \to \mathit{caracteres}, 0) \textbf{ fi}$

$\text{punterosAux} : \text{ad(puntero(nodo))}\ a \times \text{nat}\ k \longrightarrow \text{multiconj(puntero(nodo))}$ $\{k \leq \text{tam}(a)\}$
$\text{punterosAux}(a, k) \equiv \textbf{if } k = \text{tam}(a) \textbf{ then}$
$\emptyset$
else
(**if** $a[k] = \text{NULL}$ **then** $\emptyset$ **else** $\text{Ag}(a[k], \text{punteros}(a[k]))$ **fi**) $\cup$ $\text{punterosAux}(a, k+1)$
fi

2.2.3. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{dicc}_T(\alpha)$ $\{\text{Rep}(e)\}$
$\text{Abs}(e) =_{\text{obs}} \text{d: dicc}_T(\alpha) \mid (\forall c\text{: string})(\text{def?}(c, d) = \text{esClave?}(c, e) \wedge_{\text{L}}$
$(\text{def?}(c, d) \Rightarrow_{\text{L}} \text{obtener}(c, d) = \text{significado}(c, e)))$

$\text{esClave?} : \text{string } c \times \text{estr } e \longrightarrow \text{bool}$ $\{\text{Rep}(e)\}$
$\text{esClave?}(c, e) \equiv \textbf{if } \text{vacía?}(c) \textbf{ then}$
$e \to \mathit{significado} \neq \text{NULL}$
else
$e \to \mathit{caracteres}[\text{ord}(\text{prim}(c))] \neq \text{NULL} \wedge_{\text{L}} \text{esClave?}(\text{fin}(c), e \to \mathit{caracteres}[\text{ord}(\text{prim}(c))])$
fi
$\text{significado} : \text{string } c \times \text{estr } e \longrightarrow \alpha$ $\{\text{Rep}(e) \wedge \text{esClave?}(c, e)\}$
$\text{significado}(c, e) \equiv \textbf{if } \text{vacía?}(c) \textbf{ then } *(e \to \mathit{significado}) \textbf{ else } \text{significado}(\text{fin}(c), e \to \mathit{caracteres}[\text{ord}(\text{prim}(c))]) \textbf{ fi}$

2.3. Algoritmos

iCrearDicc () → res: estr

```
(res→ significado) ← NULL                                   O(1)
(res→ caracteres) ← CrearArreglo(256)                       O(1)
for i ← 0 to 255 do                                         O(1)
    (res→ caracteres[i]) ← NULL                             O(1)
end for
```

Complejidad : $O(1)$

iDefinido? (**in** c: `string`, **in** d: `estr`) → res: bool

```
nat i ← 0                                                   O(1)
bool esta ← true                                            O(1)
puntero(nodo) actual ← d                                    O(1)
while i < Longitud(c) ∧_L esta do                           O(|c|)
    if actual→ caracteres[ord(c[i])] = NULL then            O(1)
```

```
            esta ← false                                                    O(1)
        end if
        actual ← (actual → caracteres[ord(c[i])])                           O(1)
        i ← i + 1                                                           O(1)
    end while
    res ← (esta ∧_L ¬(actual→ significado = NULL))                          O(1)
```

Complejidad : $O(|c|)$

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

```
    nat i ← 0                                                               O(1)
    puntero(nodo) actual ← d                                                O(1)
    while i < Longitud(c) do                                                O(|c|)
        if actual→ caracteres[ord(c[i])] = NULL then                        O(1)
            (actual→ caracteres[ord(c[i])]) ← CrearDicc()                   O(1)
        end if
        actual ← (actual→ caracteres[ord(c[i])])                            O(1)
        i ← i + 1                                                           O(1)
    end while
    (actual→ significado) ← &Copiar(s)                                      O(copy(s))
```

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

iObtener (**in** c: `string`, **in** d: `estr`) → res: α

```
    nat i ← 0                                                               O(1)
    puntero(nodo) actual ← d                                                O(1)
    while i < Longitud(c) do                                                O(|c|)
        actual ← (actual→ caracteres[ord(c[i])])                            O(1)
        i ← i + 1                                                           O(1)
    end while
    res ← *(actual→ significado)                                            O(1)
```

Complejidad : $O(|c|)$

2.4. Servicios Usados

α debe proveer la operación:
COPIAR(**in** s: α) → res : α
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} s$}
Complejidad: O(copy(s))
Donde se copia s, de modo que no haya aliasing entre s y res

3. Módulo Conjunto de String

3.1. Interfaz

se explica con: CONJ(STRING).
géneros: $\texttt{conj}_T$.

3.1.1. Operaciones básicas de conjunto de string

VACIO() $\rightarrow$ *res* : $\texttt{conj}_T$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset$}
Complejidad: $O(1)$
Descripción: crea un conjunto

AGREGAR(**in** s: `string`, **in/out** c: $\texttt{conj}_T$)
Pre $\equiv$ {true}
Post $\equiv$ {$c =_{\text{obs}} \text{Ag}(s, c)$}
Complejidad: $O(|s|)$
Descripción: agrega un string al conjunto

BORRAR(**in** s: `string`, **in/out** c: $\texttt{conj}_T$)
Pre $\equiv$ {$s \in c$}
Post $\equiv$ {$\neg s \in c$}
Complejidad: $O(|s|)$
Descripción: borra un string del conjunto

PERTENECE?(**in** s: `string`, **in** c: $\texttt{conj}_T$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s \in c$}
Complejidad: $O(|s|)$
Descripción: verifica si el string pertenece al conjunto

3.2. Representacion

3.2.1. Representación de conjunto de string

`map` **se representa con** `estr`

donde `estr` es $\texttt{dicc}_T$`(bool)`

3.2.2. Invariante de Representación

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ true

3.2.3. Función de Abstracción

Abs : estr e $\longrightarrow$ conj(string) {Rep(e)}
Abs(e) $=_{\text{obs}}$ c: conj(string) | ($\forall s$: string)($s \in c \Longleftrightarrow$ (def?(s, e) $\wedge_{\text{L}}$ obtener(s, e) = true))

3.3. Algoritmos

```
iVacio () → res: estr

    res ← CrearDicc()                                          O(1)
```

Complejidad : $O(1)$

iAgregar (**in** s: `string`, **in/out** e: `estr`)

```
Definir(s, true, e)                                    O(|s|)
```

Complejidad : $O(|s|)$

iBorrar (**in** s: `string`, **in/out** e: `estr`)

```
Definir(s, false, e)                                   O(|s|)
```

Complejidad : $O(|s|)$

iPertenece? (**in** s: `string`, **in/out** e: `estr`) → res: bool

```
res ← Definido?(s, e) ∧_L Obtener(s, e) = true         O(|s|)
```

Complejidad : $O(|s|)$

4. Módulo Cola de Prioridad(α)

4.1. Interfaz

se explica con: Cola de Prioridad(α), Iterador Unidireccional Modificable(α)
usa: Nat, bool
genero: colaPrior(α), itColaPrior(α)

4.1.1. Operaciones de Cola de Prioridad

VACIA() $\rightarrow$ *res* : `colaPrior(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacia}
Complejidad: O(1)
Descripción: Crea una cola de prioridad

VACIA?(**in** *c*: `colaPrior(`α`)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacia?(c)}
Complejidad: O(1)
Descripción: Dice si la cola no tiene ningun elemento

DESENCOLAR(**in/out** *c*: `colaPrior(`α`)`) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$vacia?(c) $\wedge$ $c =_{\text{obs}} c_0$}
Post $\equiv$ {*res* $=_{\text{obs}}$ proximo(c_0) $\wedge$ $c =_{\text{obs}}$ desencolar(c_0)}
Complejidad: O(log(tamano(c)))
Descripción: Quita el elemento mas prioritario

ENCOLAR(**in/out** *c*: `colaPrior(`α`)`, **in** a: α) $\rightarrow$ *res* : `itColaPrior(`α`)`
Pre $\equiv$ {$c =_{\text{obs}} c_0 \wedge \neg$esta($a$, c)}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(a,c_0) $\wedge$ Actual(*res*) $=_{\text{obs}}$ a}
Complejidad: O(log(tamano(c)))
Descripción: Agrega al elemento a a la cola de prioridad
Aliasing: El iterador se invalida si, y solo si se elimina el elemento siguiente del iterador sin llamar a la funcion Eliminar del mismo

4.1.2. Operaciones auxiliares del TAD

tamano : colaPrior(α) $\longrightarrow$ nat
tamano(c) $\equiv$ **if** vacia?(c) **then** 0 **else** 1 + tamano(desencolar(c)) **fi**

4.1.3. Operaciones del iterador

CREARIT(**in** *c*: `colaPrior(`α`)`) $\rightarrow$ *res* : `itColaPrior(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {Siguientes(*res*) $=_{\text{obs}} <> \wedge$ Anteriores(*res*) $=_{\text{obs}} <>$}
Complejidad: O(1)
Descripción: Crea un iterador
Aliasing: El iterador se invalida si, y solo si se elimina el elemento siguiente del iterador sin llamar a la funcion Eliminar del mismo

ELIMINAR(**in/out** *it*: `itColaPrior(`α`)`)
Pre $\equiv$ {$it =_{\text{obs}} it_0 \wedge$ HayMas?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Eliminar(it_0)}
Complejidad: O(1)
Descripción: elimina el elemento siguiente del iterador

HayMas?(**in** it : `itColaPrior(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res =$_{\text{obs}}$ HayMas?(it)}
Complejidad: O(1)
Descripción: Dice si hay siguiente

Actual(**in** it : `itColaPrior(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {res =$_{\text{obs}}$ Actual(it)}
Complejidad: O(1)
Descripción: Devuelve el elemento actual

4.2. Representación

`colaPrior(`α`)` **se representa con** `estr`

donde `estr` es `tupla(`*tam*: `nat`,
cabeza: `puntero(nodo)`)

donde `nodo` es `tupla(`*padre*: `puntero(nodo)`,
izq: `puntero(nodo)`,
der: `puntero(nodo)`,
dato: `puntero(`α`)`)

4.2.1. Invariante de representación

(I) Arbol Binario perfectamente balanceado

(II) El elemento con mayor prioridad se encuentra en el primer elemento

(III) Los hijos de un nodo son menores a su padre

(IV) Es izquierdista, o sea, el último nivel está lleno desde la izquiera

(V) Sea n un nodo, solo su padre tiene a n como hijo y además, su padre no puede ser a la vez parte de su descendencia (no hay ciclos)

(VI) El tam es igual a la cantidad de nodos de la cola, definido por la operacion tamano.

ColaALista : c:puntero(nodo) $\longrightarrow$ secu(puntero(nodo))
ColaALista(c) $\equiv$ (**if** c.izq == NULL **then** <> **else** ColaALista(c.izq) **fi** & **if** c.der == NULL **then** <> **else** ColaALista(c.der) **fi**) $\circ$ c

4.2.2. Función de abstracción

Abs : estr e $\rightarrow$ colaPrior(α) {Rep(e)}
($\forall$ e:estr) abs(e) $\equiv$ colaPrior | vacia?(colaPrior) $\equiv$ vacia?(e) $\wedge$ proximo(colaPrior) $\equiv$ desencolar(e) $\wedge$ encolar(colaPrior) $\equiv$ encolar(e)
Abs : estr e $\longrightarrow$ colaPrior(α) {Rep(e)}
Abs(e) =$_{\text{obs}}$ c: colaPrior(α) | vacia?(c) =$_{\text{obs}}$ e.cabeza = NULL $\wedge_{\text{L}}$
$\neg$vacia?(c) $\Rightarrow_{\text{L}}$ (
proximo(c) =$_{\text{obs}}$ *((*e.cabeza).dato) $\wedge$
desencolar(c) =$_{\text{obs}}$ unir(Abs(<Tamano(*((*e.cabeza).izq)),*((*e.cabeza).izq)>),
Abs(<Tamano(*((*e.cabeza).der)),*((*e.cabeza).der)>)))
unir : colaPrior(α) $c1$ $\times$ colaPrior(α) $c2$ $\longrightarrow$ colaPrior(α)
unir($c1, c2$) $\equiv$ **if** vacia?($c1$) **then** vacia **else** unir(encolar(proximo($c2$), $c1$), desencolar($c2$)) **fi**

Tamano : c:puntero(nodo) $\longrightarrow$ Nat)
Tamano(c) $\equiv$ 1 + **if** c.izq == NULL **then** 0 **else** Tamano(c.izq) **fi** + **if** c.der == NULL **then** 0 **else** Tamano(c.der) **fi**

4.3. Representación del iterador

`itColaPrior(`α`)` se representa con `iter`

donde `iter` es `tupla`(*siguiente*: `puntero(nodo)`, *cola*: `puntero(estr)`)

4.3.1. Invariante de Representación del iterador

Rep: iter → bool
Rep(it) ≡ true $\iff$ Rep(*(it.cola)) $\wedge_L$ (it.siguiente = NULL $\vee_L$ esta?(it.siguiente,ColaAListaa(it.cola.cabeza))

4.3.2. Función de Abstracción del iterador

Abs : iter it → itMod(α)
Abs(it) $=_{obs}$ m : itMod(α) | siguientes(m) ≡ Abs(<Tamano(*(it.siguiente)),*(it.siguiente)>) • <>∧
anteriores(m) ≡ <>

4.4. Algoritmos

4.4.1. Algoritmos de Cola de Prioridad

iVacia (**in** n: `nat`) → res: colaPrior

```
    res.tam ← 0                                                    O(1)
    res.cabeza ← NULL                                              O(1)
```

Complejidad : $O(1)$

iVacia? (**in** c: `estr`) → res: bool

```
    if c.cabeza = NULL:                                            O(1)
      res ← true                                                   O(1)
    else:
      res ← false                                                  O(1)
    endif
```

Complejidad : $O(1)$

iDesencolar (**in/out** c: : estr)) → res: α

```
    puntero(nodo) padreUltimo, ultimo, primerIzquierda, primerDerecha, izquierda, derecha \\
α dato , maximo \\
res ← c.cabeza.dato \\
if c.tamano > 1 :                                                  O(1)
    padreUltimo ← damePadre(c.tam −1)                              O(log(c.tam -1))
    if padreUltimo.der = NULL :                                    O(1)
        ultimo ← padreUltimo.izq                                   O(1)
        padreUltimo.izq ← NULL                                     O(1)
    else: \\
        ultimo ← padreUltimo.der                                   O(1)
        padreUltimo.der ← NULL                                     O(1)
    endif
```

```
    ultimo.padre ← NULL                                                    O(1)
    primerIzquierda  ← c.cabeza.izq                                        O(1)
    ultimo.izq ← primerIzquierda                                           O(1)
    if primerIzquierda != NULL :                                           O(1)
        primerIzquierda.padre = ultimo                                     O(1)
    endif
    primerDerecha  ← c.cabeza.der                                          O(1)
    ultimo.der  ← primerDerecha                                            O(1)
    if primerDerecha != NULL :                                             O(1)
        primerDerecha.padre  ← ultimo                                      O(1)
    endif
    c.cabeza  ← ultimo                                                     O(1)
    bool inPlace  ← false                                                  O(1)
    dato  ← ultimo.dato                                                    O(1)
    while ¬inPlace :                                                       O(log(c.tam))
        izquierda  ← ultimo.izq                                            O(1)
        derecha  ← ultimo.der                                              O(1)
        if derecha != NULL :                                               O(1)
            maximo  ← max(izquierda.v, derecha.v)                          O(1)
            if maximo < dato :                                             O(1)
                inPlace  ← true                                            O(1)
            else:
                if maximo = izquierda.dato :                               O(1)
                    swapConPadre(izquierda)                                O(1)
                else:
                    swapConPadre(derecha)                                  O(1)
                endif
            endif
        else:
            if izquierda = NULL :                                          O(1)
                inPlace  ← true                                            O(1)
            else:
                if izquierda.dato > dato :                                 O(1)
                    swapConPadre(izquierda)                                O(1)
                else:
                    inPlace  ← true                                        O(1)
                endif
            endif
        endif
else:\\
    c.cabeza  ← NULL                                                       O(1)
c.tam  ← c.tam −1                                                          O(1)
```

Complejidad : $O(log(c.tam-1)+17*O(1)+O(log(c.tam)*14)+2*O(1))=O(log(c.tam))$

iEncolar (**in/out** c: : estr, **in** d: a to: α) → res: itColaPrior(α)

```
    puntero(nodo) nodo = iAgregarAtras(c, dato)                            O(1)
    iLevantar(c, nodo, false)                                              O(log(c.tam))
    iter it  ← iCrearIt(c)                                                 O(1)
    it.siguiente ← nodo                                                    O(1)
    res ← it                                                               O(1)
```

Complejidad : $O(log(c.tam))$

```
pre ≡ {c.tam > 1 && 0 >= posicion < tam}
```

```
post ≡ {me devuelve el padre de la posicion que le pase}
iDamePadre (in/out c: : estr, in p: o sicion: Nat)) → res: puntero(nodo)

  Nat arraySize← 0                                                    O(1)
  Nat tam← posicion                                                   O(1)
  while tam > 0:                                                      O(log(posicion))
      arraySize← arraySize + 1                                        O(1)
      tam← (tam − 1)/2                                                O(1)
  Array(arraySize) array                                              O(1)

  //arraySize queda de tamaño log(posicion)
  //y posicion es en el peor de los casos c.tam

  Nat cant  ← 0                                                       O(1)
  while cant < arraySize:                                             O(log(posicion))
      array[cant]  ← −1                                               O(1)
      cant ← cant + 1                                                 O(1)

  Nat index← arraySize − 1                                            O(1)
  tam← posicion                                                       O(1)

  while tam > 0:                                                      O(log(posicion))
      array[index]← tam mod 2                                         O(1)
      index← index − 1                                                O(1)
      tam = (tam − 1)/2                                               O(1)

  index← 0                                                            O(1)
  puntero(nodo) actual← c.cabeza                                      O(1)

  while index < arraySize − 1:                                        O(log(posicion))
      if array[index] = 0:                                            O(1)
          actual← actual.der                                          O(1)
      else:
          actual← actual.izq                                          O(1)
      endif
      index← index + 1                                                O(1)

  return actual                                                       O(1)
```

Complejidad : $2 * O(1) + O(log(posicion)) * 2 * O(1) + O(log(posicion)) * O(1) + 2 * O(1) + O(log(posicion)) * 3 * O(1) + 2 * O(1) + O(log(posicion)) * 4 * O(1) + O(1) = O(log(posicion) = O(log(c.tam))$

```
pre ≡ {el nodo esta dentro del arbol}
post ≡ {me devuelve el arbol con el dato del nodo que le pase en la posicion
 de su padre y el dato del padre en la posicion del nodo original}
iSwapConPadre (in/out c: : estr, in n: o do: puntero(nodo))

  puntero(nodo) padreObj← nodo.padre                                  O(1)
  puntero(nodo) abuelo← padreObj.padre                                O(1)
  Bool hermanoADer← padreObj.izq.dato = nodo.dato                     O(1)
  puntero(nodo) hermano
  if hermanoADer:
      hermano← padreObj.der                                           O(1)
  else:
      hermano← padreObj.izq                                           O(1)
  endif
```

```
puntero(nodo) hijoIzq← nodo.izq                                    O(1)
puntero(nodo) hijoDer← nodo.der                                    O(1)

if abuelo != NULL
    if(abuelo.izq.dato = padreObj.dato):
        abuelo.izq ← nodo                                          O(1)
    else:
        abuelo.der ← nodo                                          O(1)
endif

if hijoIzq != NULL
    hijoIzq.padre← padreObj                                        O(1)
endif

if hijoDer != NULL)
    hijoDer.padre ← padreObj                                       O(1)
endif

padreObj.izq← hijoIzq                                              O(1)
padreObj.der← hijoDer                                              O(1)
padreObj.padre← nodo                                               O(1)

if hermano != NULL
    hermano.padre← nodo                                            O(1)
endif

nodo.padre ← abuelo                                                O(1)

if (hermanoADer)
    nodo.der← hermano                                              O(1)
    nodo.izq← padreObj                                             O(1)
else:
    nodo.izq← hermano                                              O(1)
    nodo.der← padreObj                                             O(1)
endif

if (nodo.padre = NULL)
    c.cabeza ← nodo                                                O(1)
endif
```

Complejidad : $21 * O(1) = O(1)$

```
pre ≡ {el nodo a levantar se encuentra en el arbol}
post ≡ {si forceLevantar es true devuelve el arbol con el dato del
 nodo en la primer posicion del arbol y si no balancea el arbol}
```
iLevantar (**in/out** c: : estr, **in** n: o do: puntero(nodo), **in** f: o rceLevantar: bool)

```
    while nodo.padre != NULL ∧ (nodo.dato > nodo.padre ∨ forceLevantar):    O(log(c.tam))
        iSwapConPadre(c,nodo)                                                  O(1)
```

Complejidad : $O(log(c.tam))$

iAgregarAtras (**in/out** c: : estr, **in** d: : α) → res: puntero(nodo)

```
puntero(nodo) actual                                               O(1)
```

```
puntero(nodo) nuevo                                          O(1)
if c.tam = 0:                                                O(1)
    actual ← NULL                                            O(1)
else:
    actual ← iDamePadre(c.tam)                               O(1)
    nuevo.dato ← d                                           O(1)
    nuevo.padre  ← actual                                    O(1)
if c.tam = 0:                                                O(1)
    c.cabeza← nuevo                                          O(1)
else:
    if c.tam %2 == 0:                                        O(1)
        actual.der = nuevo                                   O(1)
    else:
        actual.izq = nuevo                                   O(1)
c.tam = c.tam + 1                                            O(1)
ret  ← nuevo                                                 O(1)
```

Complejidad : $O(11 * O(1)) = O(1)$

4.4.2. Algoritmos del iterador

iCrearIt (c : estr) → res: iter

```
iter iterador                                                O(1)
iterador.cola ← c                                            O(1)
iterador.siguiente ← NULL                                    O(1)
res ← iterador                                               O(1)
```

Complejidad : $O(1)$

iHayMas? (it : iter) → res: bool

```
if it.siguiente != NULL                                      O(1)
   res ← true                                                O(1)
else
   res ← false                                               O(1)
endif
```

Complejidad : $O(1)$

iEliminar (it : iter)

```
iLevantar(it.cola, it.siguiente, true)                       O(log(c.tam))
iDesencolar(it.cola)                                         O(log(c.tam))
it.siguiente ← NULL                                          O(1)
```

Complejidad : $O(2 * log(c.tam))$

iActual (it : iter) → res: α

```
res ← *(it.siguiente)
```

Complejidad : $O(1)$

5. Módulo Restriccion

5.1. Interfaz

se explica con: Restriccion
usa: tags, $dicc_t$(tags,bool), bool
genero: genero

5.1.1. Operaciones de Restriccion

CrearRestriccion(**in** t: `string`) → res : `restr`
Pre ≡ {t != AND ∧ t != OR ∧ t != NOT ∧ 0 <|t| <65}
Post ≡ {$res =_{obs} < t >$}
Complejidad: O(|r|)
Descripción: Crea una nueva restriccion en base a un tag unico

Negar(**in** r: `restr`) → res : `restr`
Pre ≡ {r =$_{obs}$ r_0}
Post ≡ {r =$_{obs}$ NOT r_0}
Complejidad: O(|r|)
Descripción: Niega la restriccion dada, generando otra

And(**in** r: `restr`, **in** $rOtra$: `restr`) → res : `restr`
Pre ≡ {r =$_{obs}$ r_0}
Post ≡ {r =$_{obs}$ r_0 AND rOtra}
Complejidad: O(|r|)
Descripción: Hace and de las dos restricciones, generando otra.

Or(**in** r: `restr`, **in** $rOtra$: `restr`) → res : `restr`
Pre ≡ {r =$_{obs}$ r_0}
Post ≡ {r =$_{obs}$ r_0 Or rOtra}
Complejidad: O(|r|)
Descripción: Hace or de las dos restricciones, generando otra.

Copiar(**in** r: `restr`) → res : `restr`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ r}
Complejidad: O(|r|)
Descripción: Crea una copia de la restricción

Verifica(**in** r: `restr`,**in** ts: $dicc_t$(`tags,bool`)) → res : `bool`
Pre ≡ {True}
Post ≡ {res == verifica?(ts,r)}
Complejidad: O(|r|)

5.2. Representación

5.2.1. Representación de Restricción

Representamos cada nodo del arbol con una clave y dos punteros a nodos hijos. Si la clave esta entre los strings (AND, OR, NOT), se considera al nodo como un nodo de operacion, y por lo tanto, si la clave es AND u OR sus dos punteros no pueden ser NULL, y si la clave es NOT, su puntero derecho debe ser NULL y su puntero izquiero no. Los nodos cuya clave no este entre las palabras (AND, OR y NOT) deberan tener como NULL a sus dos punteros.

`restr` **se representa con** `restr_tup`

donde `restr_tup` es `tupla`(*nombre*: `string` , *izq*: `puntero(restr)` , *der*: `puntero(restr)` , *tam*: `nat`)

5.2.2. Invariante de Representación

(I) El tamaño de un nodo se define como la suma del tamaño de sus hijos + 1.

(II) El arbol no debe tener ciclos, o sea, existe n en Nat tal que n >al tamaño de cada uno de los nodos del árbol

(III) Los nodos hoja tienen clave disinta de (AND, OR, NOT)

(IV) Los nodos NO hoja tienen su clave entre los valores (AND, OR, NOT)

(V) Si el nodo tiene clave entre los valores (AND, OR) sus dos punteros son distintos de NULL

(VI) Si el nodo tiene como clave NOT, su puntero derecho debera ser NULL y su puntero izquierdo debera ser distinto de NULL

(VII) Cada nodo es un arbol en si mismo

5.2.3. Función de Abstracción

Abs : restr_tup → Restriccion

Abs(e) $=_{obs}$ r: restr_tup | ∀ cj : conj(tag)
Verifica?(r, cj) ⇔ Verifica?(e, cj)

5.3. Algoritmos

iCrearRestriccion (**in** t : `string`) → res: restr

```
restr restriccion
restriccion.nombre ← t                                  O(1)
restriccion.izq ← NULL                                  O(1)
restriccion.der ← NULL                                  O(1)
restriccion.tam ← 1                                     O(1)
res ← *restriccion                                      O(1)
```

Complejidad : $O(1)$

iNegar (**in/out** r : `restr`) → res: restr

```
restr not → <"NOT",NULL,NULL,NULL>                      O(1)
not→izq ← iCopiar(r)                                   O(r.tam)
not.tam ← r.tam + 1                                     O(1)
ret ← not                                               O(1)
```

Complejidad : $O(r.tam)$

iOr (**in** r : `restr`, **in** $rOtra$: `restr`) → res: restr

```
restr or → <"OR",NULL,NULL,NULL>                        O(1)
or→izq ← iCopiar(r)                                    O(r.tam)
or→ der ← iCopiar(rOtra)                               O(r.tam)
or.tam ← r.tam + rOtra.tam + 1                          O(1)
ret ← or                                                O(1)
```

Complejidad : $O(r.tam)$

iAnd (**in/out** r : `restr`, **in** $rOtra$: `restr`) → res: restr

```
estr and → <"AND",NULL,NULL,NULL>                                   O(1)
and→izq ← iCopiar(r)                                               O(r.tam)
and→ der ← iCopiar(rOtra)                                          O(r.tam)
and.tam  ← r.tam + rOtra.tam + 1                                   O(1)
ret ← and                                                          O(1)
```

Complejidad : $O(r.tam)$

iVerifica (**in** r : `restr`,**in** ts : $dicc_t$(`tags,bool`)) → res: bool

```
if r.nombre = "NOT" :                                              O(1)
  res ← (¬ iVerifica(r.izq,ts))                                    O(r.tam)
else if r.nombre  = "AND":                                         O(1)
  res ← (iVerifica(r.izq,ts) && iVerifica(r.der,ts))               O(r.tam)
else if r.nombre = "OR":                                           O(1)
  res ← (iVerifica(r.izq,ts) || iVerifica(r.der,ts))               O(r.tam)
else:
 res ← definido?(r.nombre, ts)
endif
```

Complejidad : $O(r.tam)$

5.3.1. Complejidad de iVerifica

Calcularemos la complejidad en base a m, la cantidad de nodos del árbol. En el peor caso de iVerifica (AND o OR) nos quedamos con dos subproblemas que tienen la mitad de tamaño que el original, así que a = 2 y c = 2. Más allá de esto, se hace una cantidad constante de comparaciones y asignaciones, por lo que podemos decir que f(n) ∈ O(1). Usamos el primer caso del Teorema Maestro, porque $n^{log_2(2)-\epsilon} = n^{1-\epsilon}$. Entonces, si elegimos, por ejemplo $\epsilon = 1$, nos queda $O(n^0) = O(1)$ y sabemos que f(n) tiene esa complejidad. Por lo tanto, utilizando el primer caso tenemos que $T(n) \in O(m^{log_2(2)}) = O(m)$.

iCopiar (**in** r : `restr`) → res: restr

```
restr copia
copia.nombre  ← r.nombre                                           O(64)
copia.tam  ← r.tam                                                 O(1)
copia.izq  ← NULL                                                  O(1)
copia.der  ← NULL                                                  O(1)
if r.izq != NULL
 copia.izq  ← iCopiar(&(r.izq))                                    O(r.izq→tam)
end if
if r.der != NULL
 copia.der  ← iCopiar(&(r.der))                                    O(copy(r.der→tam))
end if
```

Complejidad : $O(r.tam)$

5.3.2. Complejidad de iCopiar

La demostración de la complejidad es análoga a la de iVerifica ya que es la misma recursión y el caso base es O(1).

6. Modulo Ciudad

6.1. Interfaz

se explica con: CIUDAD, ITERADOR UNIDIRECCIONAL(NAT).
generos: `ciudad itVectorNat`.

6.1.1. Operaciones de Ciudad

CREAR(**in** m : `mapa`) $\rightarrow$ res : `ciudad`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crear(m)}
Complejidad: $O(1)$
Descripción: crea una ciudad robotica con un mapa preestablecido
Aliasing: el mapa se agrega a la ciudad por referencia y no se puede modificar

ENTRAR(**in** cs : `conj`$_T$`(string)`, **in** e : `string`, **in/out** c : `ciudad`)
Pre $\equiv$ {$e \in$ estaciones(mapa(c)) $\wedge$ $c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ entrar(cs, e, c_0)}
Complejidad: $O(|e_m|$ * E + ($|e_m|$ + R) * S + $N_{total})$
Descripción: agrega un robot a la ciudad, su rur es la cantidad de robots ya agregados, sus tags y estacion son los parametros pasados
Aliasing: se pasa todo por referencia

MOVER(**in** u : `nat`, **in** e : `string`, **in/out** c : `ciudad`)
Pre $\equiv$ {$e \in$ estaciones(mapa(c)) $\wedge$ $u \in$ robots(c) $\wedge$ conectadas?(e, estacion(u), c) $\wedge$ $c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ mover(u, e, c_0)}
Complejidad: $O(|e1| + |e2| + logN_{e1} + logN_{e2})$
Descripción: mueve un robot de una estacion a otra

INSPECCION(**in** e : `string`, **in/out** c : `ciudad`)
Pre $\equiv$ {$e \in$ estaciones(mapa(c)) $\wedge$ $c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ inspeccion(e, c_0)}
Complejidad: $O(|e| + logN_e)$
Descripción: realiza la inspeccion, eliminando si es necesario el robot mas infractor en la estacion

ROBOTS(**in** c : `ciudad`) $\rightarrow$ res : `itVectorNat`
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion?(Siguientes(res), conjALista(robots(c)))}
Complejidad: $O(1)$
Descripción: dada una ciudad, me da un iterador de sus rurs

ESTACIONES(**in** c : `ciudad`) $\rightarrow$ res : `itLista(string)`
Pre $\equiv$ {true}
Post $\equiv$ {SecuSuby(res) $=_{\text{obs}}$ estaciones(mapa(c))}
Complejidad: $O(1)$
Descripción: Dada una ciudad, me da un iterador de sus rurs

INFRACCIONES(**in** n : `nat`, **in** c : `ciudad`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$n \in$ robots(c)}
Post $\equiv$ {$res =_{\text{obs}}$ #infracciones(n, c)}
Complejidad: $O(1)$
Descripción: Dado un robot, me da sus infracciones

ESTACION(**in** n : `nat`, **in** c : `ciudad`) $\rightarrow$ res : `string`
Pre $\equiv$ {$n \in$ robots(c)}
Post $\equiv$ {$res =_{\text{obs}}$ estacion(n, c)}
Complejidad: $O(1)$

Descripción: Dado un robot, me da su estacion actual

TAGS(**in** n: `nat`, **in** c: `ciudad`) $\rightarrow$ res : $conj_T$
Pre $\equiv$ $\{n \in \text{robots}(c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{tags}(n, c)\}$
Complejidad: $O(1)$
Descripción: Dado un robot, me da sus caracteristicas

6.1.2. Operaciones de itVectorNat

CREARIT(**in** c: `Ciudad`) $\rightarrow$ res : `itVectorNat`
Pre $\equiv$ {true}
Post $\equiv$ {Siguientes(res) $=_{\text{obs}}$ $indices(vivos(c))$}
Complejidad: $O(1)$
Descripción: Crea y devuelve un iterador de los robots.

HAYMAS?(**in** it: `itVectorNat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{HayMas?}(it)\}$
Complejidad: $O(Longitud(Siguientes(it))$
Descripción: Informa si hay más elementos por iterar.

PRÓXIMO(**in/out** it: `itVectorNat`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{\text{HayMas?(it)} \land it =_{\text{obs}} it_0\}$
Post $\equiv$ $\{it =_{\text{obs}} \text{Avanzar}(it_0) \land res =_{\text{obs}} \text{Actual}(it_0)\}$
Complejidad: $O(1)$
Descripción: Avanza el iterador y devuelve el actual.

6.2. Representación

`ciudad` **se representa con** `estr`

donde `estr` es `tupla`(*#robots*: `nat`,
#infracciones: `vector(nat)`,
estacion: `vector(string)`,
tags: `vector(conj`$_T$`)`,
vivos: `vector(bool)`,
sendasPermitidas: `vector(dicc`$_T$`(dicc`$_T$`(bool)))`,
robotsEnEstacion: `dicc`$_T$`(colaPrior(tupla(nat, nat)))`,
sacarRobotDeEstacion: `vector(itColaPrior)`,
mapa: `map`)

6.2.1. Invariante de Representación

(I) Las longitudes de los vectores coinciden con la cantidad de robots

(II) Todas las estaciones estan en las estaciones del mapa de la ciudad

(III) Las estaciones padres en las sendas tienen orden lexicograficamente mayor a sus estaciones conectadas

(IV) Para cada robot vivo, el siguiente de su iterador en sacarRobotDeEstacion es una tupla cuyo segundo valor es el rur

(V) Para cada robot vivo, su estacion en la posicion correspondiente del vector esta dentro del mapa

(VI) Para cada robot vivo, sus sendasPermitidas corresponden a combinaciones de estaciones que estan conectadas segun el mapa y cuya restriccion es verificada por los tags del robot

(VII) Los indices que son true de los vivos, son los mismos que combinar todos los rurs de robotsEnEstacion

(VIII) Para cada robot de robotsEnEstacion, su estacion es la que indica el vector de estaciones y sus infracciones las que indica el vector de infracciones

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$
(long(e.#infracciones) = e.#robots $\wedge$ long(e.estacion) = e.#robots $\wedge$
long(e.tags) = e.#robots $\wedge$ long(e.vivos) = e.#robots $\wedge$
long(e.sendasPermitidas) = e.#robots $\wedge$ long(e.sacarRobotDeEstacion) = e.#robots)
$\wedge_{\text{L}}$

($\forall s$: string)((def?(s, e.sendasPermitidas) $\vee$
def?(s, e.robotsEnEstacion))
$\Rightarrow$ esta(s, estaciones(e.mapa)))
$\wedge_{\text{L}}$

($\forall s, t$: string)((def?(t, e.sendasPermitidas) $\wedge_{\text{L}}$ def?(s, obtener(t, e.sendasPermitidas)))
$\Rightarrow$ (esta(s, estaciones(e.mapa)) $\wedge$ s <t))
$\wedge_{\text{L}}$

($\forall i$: nat)((i <e.#robots $\wedge_{\text{L}}$ e.vivos[i] = true) $\Rightarrow_{\text{L}}$
((HaySiguiente?(e.sacarRobotDeEstacion[i]) $\wedge_{\text{L}}$ π_2(Siguiente(e.sacarRobotDeEstacion[i])) = i) $\wedge$
esta(e.estacion[i], estaciones(e.mapa)) $\wedge$
(($\forall s, t$: string)(
(def?(s, e.tags) $\Rightarrow$ 0 <long(s) <65) $\wedge$
((def?(s, e.sendasPermitidas[i]) $\wedge_{\text{L}}$ def?(t, obtener(s, e.sendasPermitidas[i]))) <=>
(conectadas?(s, t, e.mapa) $\wedge_{\text{L}}$
verifica?(e.tags[i], restriccion(s, t, e.mapa))))))))
$\wedge$

esPermutacion?(indices(e.vivos), combinar(e.robotsEnEstacion))
$\wedge$

($\forall s$: string)((def?(s, e.robotsEnEstacion) $\Rightarrow_{\text{L}}$
(($\forall t$: tupla(string, string))
((esta?(t, inorder(obtener(s, e.robotsEnEstacion)))) $\Rightarrow$
($\pi_1(t)$ <e.#robots $\wedge_{\text{L}}$
(e.estacion[$\pi_1(t)$] = s $\wedge$
e.#infracciones[$\pi_1(t)$] = $\pi_2(t)$)))))

combinar : dicc(string $\times$ colaPrior(tupla(nat $\times$ nat))) d $\longrightarrow$ lista(nat)
combinar(d) $\equiv$ auxCombinar(d, d.claves)

auxCombinar : dicc(string $\times$ colaPrior(tupla(nat $\times$ nat))) d $\times$ lista(string) l $\longrightarrow$ lista(nat) $\quad$ {$l \subseteq d$.claves}
auxCombinar(d, l) $\equiv$ **if** vacia?(l) **then**
<>
else
filtrar(colaALista(obtener(prim(l), d))) & auxCombinar(d, fin(l))
fi
filtrar : lista(tupla(nat $\times$ nat)) l $\longrightarrow$ lista(nat)
filtrar(l) $\equiv$ **if** vacia?(l) **then** <> **else** π_2(prim(l)) $\bullet$ filtrar(fin(l)) **fi**

indices : lista(bool) l $\longrightarrow$ lista(nat)
indices(l) $\equiv$ indicesDesde(l, 0)

indicesDesde : lista(bool) l $\times$ nat n $\longrightarrow$ lista(nat)

indicesDesde(l, n) $\equiv$ **if** vacia?(l) **then**
<>
else
if prim(l) = true **then** n •indicesDesde(l, n+1) **else** indicesDesde(l, n+1) **fi**
fi
esPermutacion? : lista(α) $l1$ × lista(α) $l2$ $\longrightarrow$ bool
esPermutacion?($l1, l2$) $\equiv$ estan?($l1, l2$) $\wedge$ estan?($l2, l1$)

estan? : lista(α) $l1$ × lista(α) $l2$ $\longrightarrow$ bool
estan?($l1, l2$) $\equiv$ vacia?($l2$) $\vee_{\text{L}}$ ($\neg$vacia?($l1$) $\wedge_{\text{L}}$ (esta?(prim($l1$), $l2$) $\wedge$ estan?(fin($l1$), sacar(prim($l1$), $l2$))))

sacar : α e × lista(α) l $\longrightarrow$ bool
sacar(e, l) $\equiv$ **if** vacia?(l) **then** <> **else** **if** prim($l = e$) **then** fin(l) **else** e • sacar(e, fin(l)) **fi** **fi**
Obs: α debe ser comparable con la funcion =.
conjALista : conj(α) c $\longrightarrow$ lista(α)
conjALista?(c) $\equiv$ **if** $\emptyset$?(c) **then** <> **else** dameUno(c) • conjALista(sinUno(c)) **fi**

6.2.2. Función de Abstracción

Abs : estr e $\longrightarrow$ ciudad {Rep(e)}
Abs(e) =$_{\text{obs}}$ c: ciudad | proximoRUR(c) =$_{\text{obs}}$ e.#robots $\wedge$
mapa(c) =$_{\text{obs}}$ e.mapa $\wedge$
robots(c) =$_{\text{obs}}$ listaAConj(indices(e.vivos)) $\wedge_{\text{L}}$
($\forall i$: nat)(($i \in$ robots(c)) $\Rightarrow_{\text{L}}$
(estacion(i, c) =$_{\text{obs}}$ e.estacion[i] $\wedge$
tags(i, c) =$_{\text{obs}}$ e.tags[i] $\wedge$
#infracciones(i, c) =$_{\text{obs}}$ e.#infracciones[i]))

Obs: indices fue previamente definido para el Rep.

listaAConj : lista(α) l $\longrightarrow$ conj(α)
listaAConj(l) $\equiv$ **if** vacia?(l) **then** $\emptyset$ **else** Ag(prim(l), listaAConj(fin(l))) **fi**

6.2.3. Representación de itVectorNat

`itVectorNat` **se representa con** `itVect`

donde `itVect` es `tupla`(*elementos*: `vector(bool)` , *actual*: `nat`)

6.2.4. Invariante de Representación de itVectorNat

Rep : itVectorNat $\longrightarrow$ bool
Rep(it) $\equiv$ true $\Longleftrightarrow$ true

6.2.5. Funcion abstracción de itVectorNat

Abs : itVectorNat it $\longrightarrow$ itUni(nat) {Rep(it)}
Abs(it) =$_{\text{obs}}$ u: itUni(nat) | Siguientes(u) = Abs(indicesDesdeIndice(it.actual, it.elementos)

indicesDesdeIndice : nat n × lista(bool) l $\longrightarrow$ conj(α)
indicesDesdeIndice(n, l) $\equiv$ **if** vacia?(l) **then** <> **else** n • indicesDesdeIndice(n+1, fin(l)) **fi**

6.3. Algoritmos

6.3.1. Algoritmos de Ciudad

```
iCrear (in m: map) → res: estr

  res.#robots ← 0                                                    O(1)
  res.#infracciones ← Vacia()                                        O(1)
  res.estacion ← Vacia()                                             O(1)
  res.tags ← Vacia()                                                 O(1)
  res.vivos ← Vacia()                                                O(1)
  res.sacarRobotDeEstacion ← Vacia()                                 O(1)
  res.map ← m                                                        O(1)
  res.sendasPermitidas ←                                             O(1)
  itLista it ← estaciones(m)                                         O(1)
  res.robotsEnEstacion ← crearDicc()                                 O(1)
  while haySiguientes?(it) do                                        O(1)
*#(estaciones(m))
    Definir(Siguiente(it), Vacia(), res.robotsEnEstacion)            O(|siguiente(it)|)
  endwhile
```

Complejidad : $O(\#(estaciones(m)) * E_m) donde E_m es el tamano de la estacion mas grande$

```
iEntrar (in ts: dictt(string, bool), in e: string, in/out c: ciudad)
  nat rurActual ← c.#robots                                                          O(1)
  c.#robots ← c.#robots + 1                                                          O(1)
  Agregar(0, c.#infracciones)                                                        O(|c.#infracciones|)
  Agregar(e, c.estacion)                                                             O(|c.estaciones|)
  Agregar(ts, c.tags)                                                                O(|c.tags|)
  Agregar(true, c.vivos)                                                             O(|c.vivos|)
  dicc_T estacionesPermitidas ← nuevoDict()                                          O(1)
  itLista(tupla(string, string)) iteradorSendas ← itSendas(c.mapa)                   O(1)
  while hayProximo(iteradorSendas):                                                  O(S)
    string estacion1 ← primero(siguiente(iteradorSendas))                            O(1)
    string estacion2 ← segundo(siguiente(iteradorSendas))                            O(1)
    if ¬ Definido?(estacionesPermitidas, estacion1):                                 O(|estacion1|)
      Definir(estacion1, nuevoDict(), estacionesPermitidas)                          O(|estacion1|)
    endif
    restr restriccion ← Restriccion(estacion1, estacion2, c.mapa)                    O(|estacion1|+|estacion2|)
    bool result ← Verifica?(restriccion, ts)                                         O(|restriccion|)
    Definir(estacion2, result, Obtener(estacionesPermitidas, estacion1))             O(|estacion1|+|estacion
    Avanzar(iteradorSendas)                                                          O(1)
  Agregar(c.estacionesPermitidas, estacionesPermitidas)                              O(|c.estacionesPermitidas|)
  colaPrior heapEstacion ← Obtener(e, c.robotsEnEstacion)                            O(|e|)
  itColaPrior ← Agregar((rurActual, 0), heapEstacion)                                O(log(|heapEstacion|))
  Agregar(c.sacarRobotDeEstacion, iterador)                                          O(|c.sacarRobotDeEstacion|)
```

Complejidad : $O(N_{total} + S * (|e_m| + |R|) + |e|)$

Complejidad de iEntrar: Por el invRep de heap sabemos que |c.'#'infracciones| = |c.estaciones| = |c.tags| = |c.vivos| = |c.estacionesPermitidas| = |c.sacarRobotDeEstacion|= N_{total} Por lo tanto, llamar a Agregar para cada una de esas listas, agrega un N_{total} a la complejidad final. Sabemos que itSendas devuelve un iterador de tupla(string, string), donde cada tupla representa cada pareja de estaciones que esta conectada, por lo tanto, el while tiene S ciclos. Sabemos que cada ciclo tiene complejidad O(|estacion1| + |estacion2| + |restriccion|). Entonces, si R es la restriccion de mayor tamaï¿½o en todo el mapa, y e_m es la estacion de mayor longitud en el mapa, podemos decir que la complejidad de la parte del while, le agrega un O(S * ($|e_m|$ + |R|)) a la complejidad total. La parte de objetener el heap dada la estacion, le agrega un total de |e| a la complejidad total. Tambien, insertar en la heap el nuevo elemento, toma

O(log(N_e)), donde N_e es la cantidad de robots en el heap de la estacion e. Como N_e <N_{total}, podemos decir que le agrega un O(log(N_e)) a la complejidad total. Resumiendo, la complejidad final es: O(N_{total} + S * ($|e_m|$ + |R|) + |e| + log(N_{total})) = O(N_{total} + S * ($|e_m|$ + |R|) + |e|)

iMover (**in** u: `nat`, **in** e: `string`, **in/out** c: `ciudad`)

```
string estacionAnterior = c.estacion[u]                                   O(1)
string min                                                                O(1)
string max                                                                O(1)
if estacionAnterior < e                                                   O(|estacionAnterior| + |e|)
  min = estacionAnterior                                                  O(1)
  max = e                                                                 O(1)
else
  min = e                                                                 O(1)
  max = estacionAnterior                                                  O(1)
eliminar(c.sacarRobotDeEstacion[u])                                       O(log(N_e))
bool result = Obtener(min, Obtener(max, c.estacionesPermitidas[u]))       O(|estacionAnterior| + |e|)
if not result                                                             O(1)
  c.#infracciones[u] = c.#infracciones + 1                                O(1)
colaPrior heapEstacion ← Obtener(e, c.robotsEnEstacion)                   O(|e|)
itColaPrior ← Agregar((rurActual, c.#infracciones[u]), heapEstacion)      O(log(|heapEstacion|))
c.sacarRobotDeEstacion[u] = iterador)                                     O(1)
c.estacion[u] = e                                                         O(1)
```

Complejidad : $O(log(N_e) + log(N_{estacionAnterior}) + |e| + |estacionAnterior|)$

iInspeccion (**in** e: `string`, **in/out** c: `ciudad`)

```
colaPrior heapEstacion = Obtener(e, c.robotsEnEstacion)                   O(|e|)
nat rur = Desencolar(heapEstacion)                                        O(log(N_e))
vivos[rur] = false                                                        O(1)
```

Complejidad : $O(log(N_e) + |e|)$

iRobots (**in** c: `estr`) → res: itVectorNat

```
res ← CrearIt(c.vivos)                                                    O(1)
```

Complejidad : $O(1)$

iEstaciones (**in** c: `estr`) → res: itVector(string)

```
res ← Estaciones(c.mapa)                                                  O(1)
```

Complejidad : $O(1)$

iInfracciones (**in** n: `nat`, **in** c: `estr`) → res: nat

```
res ← c.#infracciones[n]                                                  O(1)
```

Complejidad : $O(1)$

iEstacion (**in** n: `nat`, **in** c: `estr`) → res: string

```
res ← c.estacion[n]                                                    O(1)
```

Complejidad : $O(1)$

iTags (**in** n: `nat`, **in** c: `estr`) → res: $conj_T()$

```
res ← c.tags[n]                                                        O(1)
```

Complejidad : $O(1)$

6.3.2. Algoritmos de itVectorNat

iCrearIt (**in** v: `vector(bool)`) → res: itVectorNat

```
res.elementos ← v                                                      O(1)
res.actual ← 0                                                         O(1)
```

Complejidad : $O(1)$

iHayMás? (**in** it: `itVectorNat`) → res: bool

```
res ← false                                                            O(1)
nat i = it.actual                                                      O(1)
while (i < longitud(it.elementos) ∧ ¬ res)                             O(longitud(it.elementos))
    if it.elementos[i] = true then                                     O(1)
        res = true                                                     O(1)
    endif
    i++                                                                O(1)
```

Complejidad : $O(longitud(it.elementos))$

iProximo (**in/out** it: `itVectorNat`) → res: nat

```
bool encontreProximo                                                   O(1)
nat i = it.actual                                                      O(1)
while (i < longitud(it.elementos) ∧ ¬ encontreProximo)                 O(longitud(it.elementos))
    if it.elementos[i] = true then                                     O(1)
        it.actual = i                                                  O(1)
        res ← i
    endif
    i++                                                                O(1)
```

Complejidad : $O(longitud(it.elementos))$