Algorítmos y Estructuras de Datos II
Trabajo Práctico 2 (Diseño)
Grupo 6

Bayardo, Julián
julian@bayardo.com.ar
850/13

Cuneo, Christian
chriscuneo93@gmail.com
755/13

Gambaccini, Ezequiel
ezequiel.gambaccini@gmail.com
715/13

Lebrero Rial, Ignacio Manuel
ignaciolebrero@gmail.com
751/13

28 de Octubre del 2014

1. Módulo Iterador de Vector a Puntero (α)

```
*******************************************************************************
************                          Interfaz                         ************
*******************************************************************************

VectorPointer(α) es Vector(puntero(α))

Este iterador deberia estar en Vector

itVectorPointer(α)
Se explica con iterador unidireccional

Se considera n igual a longitud(vec).

*******************************************************************************
************                        Operaciones                        ************
*******************************************************************************

CrearIt(in vec : VectorPointer(α)) → res : itVectorPointer(α)
Pre ≡ {true}
Post ≡ {res =obs CrearItUni(in)}
Complejidad: O(1)
Descripcion: Crea un iterador unidireccional del VectorPointer(α).

HayMas?(in it : itVectorPointer(α)) → res : bool
Pre ≡ {true}
Post ≡ {res =obs HayMas?(it)}
Complejidad: O(n)
Descripcion: Devuelve true si y solo si en el iterador quedan elementos para avanzar.

Actual(in it: itVectorPointer(α)) → res : puntero(α)
Pre ≡ {HayMas?(it)}
Post ≡ {alias(res =obs Actual(it))}
Complejidad: O(1)
Descripcion: Devuelve el elemento actual del iterador.
Aliasing: res y su contenido no son modificables.

Avanzar(in/out it : itVectorPointer(α))
Pre ≡ {it = it₀ ∧ HayMas?(it)}
Post ≡ {it =obs Avanzar(it₀)}
Complejidad: O(n)
Descripcion: Avanza a la posicion siguiente del iterador.

*******************************************************************************
************                      Representación                       ************
*******************************************************************************

itVectorPointer(α) se representa con iter(α),
    donde iter(α) es: tupla(
                        actual : nat,
                        len    : nat,
                        vec    : puntero(VectorPointer(α)))

Rep: ^(iter(α) → boolean
(∀e : ^(iter(α)) Rep(e) ≡ true ⟺
    (e.actual < e.len) ∧ (¬(e.vec =obs NULL) ∧ₗ e.len = Longitud(*e.vec))

Abs: ^(iter(α)) i → IteradorUnidireccional(α)                          {Rep(i)}
(∀e : ^(iter(α)) Abs(e) =obs m /
    Longitud(Siguientes(m)) =obs (e.len - e.actual) ∧
    (∀i : nat) (i < e.len ∧ₗ *e.vec[i] =obs Siguientes(m)[i])
```

```
*****************************************************************************
************                      Algoritmos                         ************
*****************************************************************************
iCrearIt(in input_vec : VectorPointer(α)) → res: itVectorPointer(α)
    res ← ⟨actual: 0, len: Longitud(input_vec), vec: &input_vec⟩
end function


iActual(in it: iter(α)) → res: puntero(α)
    res ← (it.vec)[iter.actual]
end function

iHayMas?(in it : iter(α)) → res : bool
    var next : nat ← it.actual

    while next < len ∧ (*it.vec)[next] == NULL do
        next ← next + 1
    end while

    res ← next != len
end function

iAvanzar(in/out it : iter(α))
    var next : nat ← it.actual + 1

    while (*it.vec)[next] == NULL do
        next ← next + 1
    end while

    it.actual ← next
end function
```

2. Módulo Conjunto Rápido de Strings

```
*******************************************************************************
************                        Interfaz                         ************
*******************************************************************************

Se explica con: conj(String)
Géneros: conjRapidoString

*******************************************************************************
************                      Operaciones                        ************
*******************************************************************************

Vacio() → res : conjRapidoString
Pre ≡ {true}
Post ≡ {res =obs ϕ}
Complejidad: O(1)
Descripcion: Crea un conjunto vacio.

Agregar(in/out c : conjRapidoString, in s : string)
Pre ≡ {true}
Post ≡ {c =obs Ag(s, c)}
Complejidad: O(long(s))
Descripcion: Agrega un string al conjunto.

Pertenece?(in c : conjRapidoString, in s : string) → res : bool
Pre ≡ {true}
Post ≡ {res =obs s ∈ c}
Complejidad: O(long(s))
Descripcion: Se fija si s pertenece al conjunto o no.

CrearIt(in c : conjRapidoString) → res : itConj(string)
Pre ≡ {true}
Post ≡ {alias(esPermutación?(SecuSuby(res), c)) ∧ vacia?(Anteriores(res))}
Complejidad: O(1)
Descripcion: Devuelve un iterador del conjunto.

*******************************************************************************
************                    Representación                       ************
*******************************************************************************

conjRapidoString se representa con conjRap, donde conjRap es DiccString(Bool)

Rep: ^(conjRap) → boolean
Rep(e) ≡ true ⟺ (∀s : string) Definido?(e, s) ⟹ι Obtener(e, s) =obs true

Abs: ^(conjRap) e → Conj(String)                                        {Rep(e)}
(∀e : ^(conjRap)) Abs(e) =obs c /
    (∀s : string) (Definido?(e, s) ∧ι Obtener(e, s)) =obs s ∈ c

*******************************************************************************
************                     Algoritmos                          ************
*******************************************************************************

iVacio() → res : conjRap
    res ← Crear()
end function

iAgregar(in/out c : conjRap, in s : string)
    if ¬Definido?(c, s) then
        Definir(c, s, true)
    end if
end function

iPertenece?(in c : conjRap, in s : string) → res : bool
    res ← Definido?(c, s)
end function

iCrearIt(in c : conjRap) → res : itConjString
    res ← Claves(c)
end function
```

3. Módulo Diccionario de Strings (α)

```
*******************************************************************************
************                          Interfaz                         ************
*******************************************************************************

Parametros formales
    generos α
    funcion Copiar(in a: α) → res: α
        Pre ≡ {true}
        Post ≡ {res =obs a}
        Complejidad: O(copy(a))
        Descripcion: funcion de copia de α's.

    funcion • = • (in a1 : α, a2 : α) → res : bool
        Pre ≡ {true}
        Post ≡ {res =obs (a1 = a2)}
        Complejidad: O(equal(a1, a2))
        Descripcion: funcion de igualdad de αs

Se explica con: dicc(string, α)
Generos: DiccString(α)

*******************************************************************************
************                        Operaciones                        ************
*******************************************************************************

Crear() → res : DiccString(α)
Pre ≡ {true}
Post ≡ {res =obs vacio}
Complejidad: O(1)
Descripcion: Crea un diccionario vacio.

Definir(in/out d : DiccString(α), in k : string, in v : α)
Pre ≡ {d =obs d₀}
Post ≡ {d =obs definir(k, v, d₀)}
Complejidad: O(long(k))
Descripcion: Inserta una clave en el diccionario.

Definido?(in d : DiccString(α), in k : string) → res : bool
Pre ≡ {true}
Post ≡ {res =obs def?(k, d)}
Complejidad: O(long(k))
Descripcion: Dice si una clave esta definido.

Obtener(in d : DiccString(α), in k : string) → res : α
Pre ≡ {def?(k, d)}
Post ≡ {res =obs obtener(k, d)}
Complejidad: O(long(k))
Descripcion: Obtiene el significado de la clave en el diccionario

Claves(in d : DiccString(α)) → res : itConj(string)
Pre ≡ {true}
Post ≡ {alias(esPermutación?(SecuSuby(res), claves(d))) ∧ vacia?(Anteriores(res))}
Complejidad: O(1)
Descripcion: Devuelve un iterador a las claves del diccionario.

Copia(in d: DiccString(α)) → res : DiccString(α)
Pre ≡ {true}
Post ≡ {res =obs d}
Complejidad: O(#Claves(d) * long(e) * Copia(Obtener(d, s))), donde e es la clave mas larga
            y s pertenece a Claves(d)
Descripcion: Crea un diccionario por copia.
```

```
*****************************************************************************
***********                    Representación                      ***********
*****************************************************************************

DiccString(α) se representa con cabeza(α)
    donde cabeza(α) es tupla(
                        claves: conj(string),
                        significados: trie(α)
                        )

    donde trie(α) es tupla(
                        continuacion : array_estatico[256] de puntero(trie(α)),
                        significado  : puntero(α)
                        )

Rep: cabeza(α) t → bool
Rep(t) ≡ true ⟺
    ((∀s: string)(s ∈ t.claves ⟺ def?'(t.significados, s))) ∧
    No hay ciclos en significados ∧
    ((∀i : nat) (i < 256) ⟹ι t.continuacion[i] =obs NULL) ⟹ t.significado =obs NULL ∧ι
    (∀i : nat) (i < 256 ∧ι ¬(t.continuacion[i] =obs NULL)) ⟹ι Rep(*(t.continuacion[i]))

Abs: ^(cabeza(α)) t → dicc(string, α)                                   {Rep(t)}
(∀t : ^(cabeza(α))) Abs(t) =obs d /
    (∀c : string)
        def?(c, d) =obs def'?(c, t.significados) ∧
        def'?(c, t.significados) ⟹ι obt'(c, t.significados) =obs obtener(c, d)

def?': string × ^(trie(α)) → bool
def?'(s, t) ≡
    if vacia?(s) then ¬(t.significado =obs NULL)
    else
        if t.continuacion[ord(prim(s))] =obs NULL then false
        else def?(fin(s), t.continuacion[ord(prim(s))]) fi
    fi

obt': string s × ^(trie(α)) t → α                                   {def?'(s, t)}
obt'(s, t) ≡
    if vacia?(s) then t.significado
    else obt'(fin(s), t.continuacion[ord(prim(s))]) fi

*****************************************************************************
***********                     Algoritmos                         ***********
*****************************************************************************

iCrear() → res : cabeza(α)
    res ← ⟨claves: Vacio(),
         significados: ⟨continuacion: arreglo_estatico[256] de puntero(trie(α)),
             significado: NULL⟩
        ⟩
end function

iDefinir(in/out d : cabeza(α), in k : string, in v : α)
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    var nuevo: bool ← false
    while i < longitud(k) do
        if t→continuacion[ord(k[i])] == NULL then
            t→continuacion[ord(k[i])] ← &Crear()
            nuevo ← true
        end if
        t ← *t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    t→significado ← &v
    if nuevo then
        AgregarRapido(d.claves, k)
    fi
end function
```

Como se puede observar el bucle principal depende exlusivamente de la cantidad de caracteres del vector, por lo que el mejor caso es que el mismo tenga 0 caracteres en donde automaticamente saldra y definira la palabra vacia y su costo sera O(1), pero tanto el caso promedio como el peor sera en donde el largo del vector es mayor a 0 entonces dado que todas las operaciones dentro del bucle tienen costo O(1) tendra un costo equivalente al tamaño del vector (O(long(k))) ya que tendra que recorrerlo para definir letra por letra dentro de la estructura hasta llegar a su fin y definir el significado que tiene costo O(1). A esto se le suma la complejidad de AgregarRapido de la clave al conjunto de claves (si esta no estaba definida), al ser la clave un string esto cuesta el copiar del vector de char, que es la longitud del vector por el costo de copiar cada char que es 1, quedando entonces la longitud del string k.

```
iDefinido?(in d : cabeza(α), in k : string) → res : bool
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    bool listo ← False
    while i < longitud(k) ∧ t→continuacion[ord(k[i])] != NULL do
        t ← t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    res ← t→significado != NULL ∧ i == longitud(k)
end function
```

En el peor caso(si el string esta definido dentro de la estructura y el mismo tiene largo mayor a 0) dado que las operaciones dentro del bucle tienen costo O(1) y se repetiran long(k) veces, para ir recorriendo la estructura, este tendra un costo O(long(k)) pero en el caso de que no este definido el costo sera O(long(k) - n) con n cantidad de caracteres que no pertenecen al prefijo perteneciente a la estructura por lo que el peor caso sera O(long(k)).

```
iObtener(in d : cabeza(α), in k : string) → res : α
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    while i < longitud(k) do
        t ← t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    res ← *t→significado
end function
```

En el peor caso dado que las operaciones dentro del bucle tienen costo O(1) y se repetiran long(k) veces, para ir recorriendo la estructura, este tendra un costo O(long(k)).

```
iClaves(in d: cabeza(α)) → res : itConj(string)
    res ← CrearIt(d.claves)
end function
```

```
iCopia(in d: cabeza(α)) → res : cabeza(α)
    var i : nat ← 0
    var c : cabeza(α) ← Crear()
    var it : itConj(string) ← CrearIt(d.claves)
    while HaySiguiente(it)
        Definir(c, Siguiente(it), Copiar(Obtener(d, Siguiente(it))))
        Avanzar(it)
    end while
    res ← c
end function
```

Al copiar se itera por todas las claves y se definen los significados creando una copia de cada uno. Por lo tanto se multiplica el cardinal del conjunto de claves por el obtener de cada claves por el copiar de cada significado. Por lo tanto para el peor caso tomamos el peor obtener que seria el de la clave mas larga y la copia de todos los significados.

4. Módulo Cola de Prioridad(α)

```
*****************************************************************************
************                        Interfaz                         ************
*****************************************************************************

Parametros Formales
    géneros α
    función • < • (in a1 : α, a2 : α) → res : bool
        Pre ≡ {true}
        Post ≡ {res =obs (a1 < a2)}
        Complejidad: O(lower(a1, a2))
        Descripción: funcion de comparación de αs

Se explica con: ColaDePrioridadExtendida(α), IteradorUnidireccional(α)
Generos: ColaPrioridad(α), itCola(α)

*****************************************************************************
************                      Operaciones                        ************
*****************************************************************************

Crear() → res : ColaPrioridad(α)
Pre ≡ {true}
Post ≡ {res =obs vacia}
Complejidad: O(1)
Descripción: Crea una cola vacia.

Encolar(in/out t : ColaPrioridad(α), in e : α) → res : itCola(α)
Pre ≡ {¬(e ∈ t) ∧ t₀ =obs t}
Post ≡ {t =obs encolar(e, t₀) ∧ alias(Actual(res) =obs e)}
Complejidad: O(log(Tamaño(t)))
Descripción: Inserta un elemento en la cola y devuelve un iterador posicionado en el elemento
agregado.
Aliasing: El iterador se invalida sii se borra el elemento utizando Desencolar o Borrar

Desencolar(in/out t : ColaPrioridad(α)) → res : α
Pre ≡ {¬vacia?(t) ∧ t =obs t₀}
Post ≡ {t =obs desencolar(t₀) ∧ res =obs proximo(t₀)}
Complejidad: O(log(Tamaño(t)))
Descripción: Desencola el elemento con mayor prioridad.

Tamaño(in t : ColaPrioridad(α)) → res : nat
Pre ≡ {true}
Post ≡ {res =obs #t}
Complejidad: O(1)
Descripción: Devuelve la cantidad de elementos en la cola.

Borrar(in/out t : ColaPrioridad(α), in i : itCola(α)) → res : α
Pre ≡ {t₀ =obs t}
Post ≡ {t =obs borrar(Actual(i), t₀)}
Complejidad: O(log(Tamaño(t)))
Descripción: Borra el elemento al que apunta el iterador.
Aliasing: Invalida el iterador.

TAD ColaDePrioridadExtendida(α)
    extiende colaPrior(α)
    otras operaciones (exportadas)
        #• : colaPrior(α) → nat
        • ∈ • : α × colaPrior(α) → bool
        borrar : α e × colaPrior(α) c → colaPrior(α) {e ∈ c}
    axiomas
        #c ≡ if vacia?(c) then 0 else 1 + #desencolar(c) fi
        x ∈ c ≡ ¬vacia?(c) ∧ₗ (x =obs proximo(c) ∨ₗ x ∈ desencolar(c))
        borrar(e, c) ≡
            if e =obs proximo(c) then
                if e ∈ desencolar(c) then borrar(e, desencolar(c))
                else desencolar(c) fi
            else encolar(proximo(c), borrar(e, desencolar(c))) fi
Fin TAD
```

```
*******************************************************************************
************                      Representación                     ************
*******************************************************************************

ColaPrioridad(α) se representa con estr(α)
    donde estr(α) es tupla(
                        cabeza : puntero(nodo(α)),
                        ultimo : puntero(nodo(α)),
                        tamaño : nat)
    donde nodo(α) es tupla(
                        arr    : puntero(nodo(α)),
                        izq    : puntero(nodo(α)),
                        der    : puntero(nodo(α)),
                        dato   : α)

Rep: ^(estr(α)) → boolean
Rep(e) ≡ true ⟺
    (e.cabeza =obs NULL ⟺ e.ultimo =obs NULL) ∧
    e.tamaño =obs Tamaño(e.cabeza) ∧
    InvPadres(e.cab) ∧
    No hay ciclos en el arbol ∧ι
    MaxHeap(e.cab) ∧
    Balanceado(e.cab) ∧
    Izquierdista(e.cab) ∧ι
    (¬(e.cabeza =obs NULL) ⟹ι
        e.cabeza→arr =obs NULL ∧
        e.ultimo ES EL ULTIMO AGREGADO)

InvPadres: puntero(nodo(α)) → bool
InvPadres(p) ≡
    if p =obs NULL then true
    else
        (¬(p→izq =obs NULL) ⟹ι p→izq→arr =obs p) ∧
        (¬(p→der =obs NULL) ⟹ι p→der→arr =obs p) ∧
        InvPadres(p→izq) ∧
        InvPadres(p→der)
    fi

MaxHeap: puntero(nodo(α)) → bool
MaxHeap(p) ≡
    if p =obs NULL then true
    else
        (¬(p→izq =obs NULL) ⟹ι p→izq→dato < p→dato) ∧
        (¬(p→der =obs NULL) ⟹ι p→der→dato < p→dato) ∧
        MaxHeap(p→izq) ∧
        MaxHeap(p→der)
    fi

Balanceado: puntero(nodo(α)) → bool
Balanceado(p) ≡
    (p =obs NULL) vι (|Altura(p→izq) - Altura(p→der)| ≤ 1) ∧
    Balanceado(p→izq) ∧ Balanceado(p→der)

Altura: puntero(nodo(α)) → nat
Altura(p) ≡ if p =obs NULL then 0 else 1 + max(Altura(p→izq), Altura(p→der)) fi

Izquierdista: puntero(nodo(α)) → bool
Izquierdista(p) ≡
    (p =obs NULL) vι (¬(p→der =obs NULL) ⟹ι ¬(p→izq =obs NULL)) ∧
    Izquierdista(p→izq) ∧ Izquierdista(p→der)

Tamaño: puntero(nodo(α)) → nat
Tamaño(p) ≡ if p =obs NULL then 0 else 1 + Tamaño(p→izq) + Tamaño(p→der) fi

Abs: ^(estr(α)) e → ColaDePrioridadExtendida(α)                          {Rep(e)}
(∀e : ^(estr(α))) Abs(e) =obs t /
    vacia?(t) =obs (e.tam =obs 0) ∧
    ¬vacia?(t) ⟹ι
        proximo(t) =obs e.cab→dato ∧
        (∀e : α) (((e ∈ t) ∧ ¬(e =obs proximo(t))) ⟺ (e ∈ desencolar(t)))
```

```
*******************************************************************************
************                      Algoritmos                       ************
*******************************************************************************

iCrear() → res : estr(α)
    res ← ⟨cabeza: NULL, ultimo: NULL, tamaño: 0⟩
end function

iTamaño(in t : estr(α)) → res : nat
    res ← t.tamaño
end function

iEncolar(in/out t : estr(α), in e : α) → res : itCola(α)
    var tmp : puntero(nodo(α)) ← &⟨arr: NULL, izq: NULL, der: NULL, dato: e⟩

    if Tamaño(t) == 0 then
        tmp→arr ← NULL
        t.cabeza ← tmp
    else if Tamaño(t) == 1 then
        tmp→arr ← t.cabeza
        t.cabeza→izq ← tmp
    else
        if t.ultimo→arr→izq == t.ultimo then
            tmp→arr ← t.ultimo→arr
            t.ultimo→arr→der ← tmp
        else
            var cur : puntero(nodo(α)) ← t.ultimo

            while cur→arr != NULL ∧ cur→arr→izq != cur do
                cur ← cur→arr
            end while

            if cur→arr != NULL then
                cur ← cur→arr→der
            fi

            while cur→izq != NULL do
                cur ← cur→izq
            end while

            tmp→arr ← cur
            cur→izq ← tmp
        end if
    end if

    t.ultimo ← tmp
    t.tamaño++

    Subir(t.ultimo)
    res ← crearIter(t, p)
end function

En el peor caso, tenemos un árbol con más de 1 elemento, en el que el último nodo
agregado está a la derecha de su padre, fundamentalmente el peor es cuando el
árbol es completo.
En este caso, observemos que lo que sucederá es que subiremos hasta la raíz del
árbol, y luego se bajaremos hacia el último nodo a la izquierda del árbol. Es
decir, recorreremos dos veces la altura del árbol (2*log(#t)). Luego, en
absolutamente todos los casos haremos a lo sumo log(#t) pasos para restablecer
el invariante utilizando Subir. Es decir, haremos 3*log(#t), por lo que en el
peor de los casos es O(log(#t)).

iDesencolar(in/out t : estr(α)) → res : α
    res ← Eliminar(t, t.cabeza)
end function
```

```
iBorrar(in/out t : estr(α), in i : itCola(α)) → res : α
    res ← Eliminar(t, i)
end function
```

```
Pre ≡ {¬(p =obs NULL) ∧ₗ p es un puntero de la estructura de datos ∧ t₀ =obs t}
Post ≡ {t =obs borrar(p→dato, t₀) ∧ res =obs p→dato}
Descripcion: Elimina el dato al que apunta el puntero.
Complejidad: O(log(Tamaño(t)))
iEliminar(in/out t : estr(α), in p : puntero(nodo(α))) → res : α
    res ← p→dato

    if Tamaño(t) == 1 then
        t.ultimo ← NULL
        t.cabeza ← NULL
    else
        p→dato ← t.ultimo→dato

        if t.ultimo→arr→izq == t.ultimo then
            var cur : puntero(nodo(α)) = t.ultimo

            while cur→arr != NULL ∧ cur→arr→der != cur do
                cur ← cur→arr
            end while

            if cur→arr != NULL then
                cur ← cur→arr→izq
            fi

            while cur→der != NULL do
                cur ← cur→der
            end while

            t.ultimo→arr→izq ← NULL
        else
            t.ultimo ← t.ultimo→arr→izq
            t.ultimo→arr→der ← NULL
        end if

        Bajar(p)
    end if

    t.tamaño--
end function
```

En el peor caso, tenemos un árbol con más de 1 elemento, en el que el último nodo agregado está a la izquierda de su padre, fundamentalmente el peor es cuando el árbol tiene un nodo de más para ser completo.
En este caso, observemos que lo que sucederá es que subiremos hasta la raíz del árbol, y luego se bajaremos hacia el último nodo a la derecha del árbol. Es decir, recorreremos dos veces la altura del árbol (2*log(#t)). Luego, en absolutamente todos los casos haremos a lo sumo log(#t) pasos para restablecer el invariante utilizando Bajar. Es decir, haremos 3*log(#t), por lo que en el peor de los casos es O(log(#t)).

```
Pre ≡ {¬(p =obs NULL)}
Post ≡ {Se restableció el rep de la estructura de datos}
Complejidad: O(log(#t))
Descripción: Restablece el invariante si el nodo al que apunta p está fuera del
lugar que le corresponde.
iSubir(in/out p : puntero(nodo(α)))
    while p→arr != NULL ∧ p→arr→dato < p→dato do
        var tmp : α ← p→arr→dato
        p→arr→dato ← p→dato
        p→dato ← tmp
        p ← p→arr
    end while
end function
```

Observemos que en el peor caso, se va a iterar hasta que p→arr sea NULL. Pero el invariante de representación asegura que el único caso en que esto sucede es que

el nodo sea la raiz. Es decir, desde una posición arbitraria la máxima cantidad de pasos es la altura del árbol, por lo que en el peor caso, la complejidad de iSubir es log(#t).

```
Pre ≡ {¬(p =obs NULL)}
Post ≡ {Se restableció el rep de la estructura de datos}
Complejidad: O(log(#t))
Descripción: Restablece el invariante si el nodo al que apunta p está fuera del
lugar que le corresponde.
iBajar(in/out p : puntero(nodo(α)))
    while    (p→izq != NULL ∧ p→dato < p→izq→dato) ∨
             (p→der != NULL ∧ p→der→dato < p→dato) do
        if p→izq != NULL then
            var tmp : α ← p→izq→dato
            p→izq→dato ← p→dato
            p→dato ← tmp
        else
            if p→der != NULL then
                var tmp : α ← p→der→dato
                p→der→dato ← p→dato
                p→dato ← p→der→dato
            end if
        end if
    end while
end function
```

Observemos que en el peor caso, se va a iterar hasta que tanto p→izq como p→der. sean NULL. Pero el invariante de representación asegura que el único caso en que esto sucede es que el nodo al que se llega sea una hoja. Es decir, desde una posición arbitraria la máxima cantidad de pasos es la altura del árbol, por lo que en el peor caso, la complejidad de iBajar es log(#t).

```
*****************************************************************************
**********                    Operaciones del iterador            ***********
*****************************************************************************
```

Ninguna.

```
*****************************************************************************
**********                   Representación del iterador          ***********
*****************************************************************************
```

itCola(α) se representa con it(α)
    donde it(α) es puntero(nodo(α))

```
Rep: ^(it(α)) → boolean
Rep(e) ≡ true ⟺ ¬(e =obs NULL)

Abs: ^(it(α)) → IteradorUnidireccional(α)
(∀e : ^(it(α))) Abs(e) =obs i / Siguientes(i) =obs *e • <>
```

```
*****************************************************************************
**********                 Algorítmos del Iterador                ***********
*****************************************************************************
```

```
Pre ≡ {p es un puntero en la estructura de t}
Post ≡ {itCola es un iterador posicionado en p}
Complejidad: O(1)
Descripción: Crea un puntero disfrazado de iterador.
crearIter(in t : colaPrioridad(α), p : puntero(nodo(α))) → res : itCola(α)
    res ← p
end function
```

5. Módulo Restricción

```
*****************************************************************************
************                      Interfaz                       ************
*****************************************************************************

Se explica con: Restricción
Géneros: restricción

*****************************************************************************
************                    Operaciones                      ************
*****************************************************************************

Var(in s : string) → res : restricción
Pre ≡ {true}
Post ≡ {res =obs ⟨s⟩}
Complejidad: O(1)
Descripcion: Crea una nueva restricción

And(in r1 : restricción, in r2 : restricción) → res : restricción
Pre ≡ {true}
Post ≡ {res =obs r1 AND r2}
Complejidad: O(1)
Descripcion: Crea una nueva restricción que tiene que cumplir con r1 y r2

Or(in r1 : restricción, in r2 : restricción) → res : restricción
Pre ≡ {true}
Post ≡ {res =obs r1 OR r2}
Complejidad: O(1)
Descripcion: Crea una nueva restricción que tiene que cumplir con r1 o r2

Not(in r : restricción) → res : restricción
Pre ≡ {true}
Post ≡ {res =obs NOT r}
Complejidad: O(1)
Descripcion: Crea una nueva restricción que no tiene que cumplir con r

Verifica?(in tags : conjRapidoString, in r : restricción) → res : bool
Pre ≡ {true}
Post ≡ {res =obs verifica?(tags, r)}
Complejidad: O(#r)
Descripcion: Evalua si la restricción r evalua a verdadero asumiendo que las
variables que figuran en tags son verdaderas.


Copiar(in r: restricción) → res : restricción
Pre ≡ {true}
Post ≡ {res =obs r}
Complejidad: O(R)
Descripcion: Devuelve una copia de la restricción.

*****************************************************************************
************                    Representación                   ************
*****************************************************************************

restricción se representa con estr
    donde estr es tupla(
                    tipo  : Enumerado(VAR, AND, OR, NOT),
                    op1   : puntero(estr),
                    op2   : puntero(estr),
                    valor : string)

Rep: ^(estr) → boolean
Rep(e) ≡ true ⟺
    No hay ciclos en el arbol ∧ι
    Ningún nodo de más abajo en el arbol tiene punteros a e ∧ι
    (e.tipo =obs VAR ⟺ (e.op1 =obs NULL ∧ e.op2 =obs NULL)) ∧
    (e.tipo =obs NOT ⟺ (¬(e.op1 =obs NULL) ∧ e.op2 =obs NULL)) ∧
    ((e.tipo =obs AND ∨ e.tipo =obs OR) ⟺ (¬(e.op1 =obs NULL) ∧ ¬(e.op2 =obs NULL))) ∧
    (¬(e.op1 =obs NULL) ⟹ι Rep(*e.op1)) ∧
    (¬(e.op2 =obs NULL) ⟹ι Rep(*e.op2)) ∧ (Long(e.valor) <= 64)
```

```
Abs: ^(estr) → Restricción                                                    {Rep(e)}
Abs(e) ≡
    if e.tipo =obs VAR then
        ⟨e.valor⟩
    else if e.tipo =obs AND then
        Abs(*e.op1) AND Abs(*e.op2)
    else if e.tipo =obs OR then
        Abs(*e.op1) OR Abs(*e.op2)
    else
        NOT Abs(*e.op1)
    fi fi fi

#•: ^(estr) → nat
#(e) ≡
    if e.tipo =obs VAR then
        1
    else if e.tipo =obs AND then
        1 + #(*e.op1) + #(*e.op2)
    else if e.tipo =obs OR then
        1 + #(*e.op1) + #(*e.op2)
    else
        1 + #(*e.op1)
    fi fi fi

*****************************************************************************
************                      Algoritmos                       ************
*****************************************************************************

iVar(in s : string) → res : estr
    res ← ⟨tipo: VAR, op1: NULL, op2: NULL, valor: s⟩
end function

iAnd(in r1 : estr, in r2 : estr) → res : estr
    res ← ⟨tipo: AND, op1: &r1, op2: &r2, valor: ""⟩
end function

iOr(in r1 : estr, in r2 : estr) → res : estr
    res ← ⟨tipo: OR, op1: &r1, op2: &r2, valor: ""⟩
end function

iNot(in r : estr) → res : estr
    res ← ⟨tipo: NOT, op1: &r, op2: NULL, valor: ""⟩
end function

iVerifica?(in tags : conj(string), in r : estr) → res : bool
    case
        [] tipo == AND
            res ← Verifica?(tags, *r.op1) ∧ Verifica?(tags, *r.op2)
        [] tipo == OR
            res ← Verifica?(tags, *r.op1) ∨ Verifica?(tags, *r.op2)
        [] tipo == NOT
            res ← ¬Verifica?(tags, *r.op1)
        [] tipo == VAR
            res ← Pertenece?(tags, r.valor)
    end case
end function
```

La complejidad de iVerifica? es O(#r), siendo #r la cantidad de restricciones anidadas. Esto se debe a que para verificar una restriccion vale para un conj(string), si esta no es un Var, debe verificar que valga para sus restricciones anidadas en op1 y/o op2 (depende si la restriccion es de tipo NOT o no), recursivamente, hasta llegar a un Var. Luego, cada restriccion devuelve el resultado anidado, por lo que se pasaria por cada nodo 2 veces, ergo, O(2R), y por algebra de ordenes, esto es equivalente a O(R)

```
iCopiar(in r: estr) → res: estr
    case
        [] tipo == AND
            res ← ⟨tipo: AND, op1: &Copiar(*r.op1), op2: &Copiar(*r.op2), valor: ""⟩
        [] tipo == OR
            res ← ⟨tipo: OR, op1: &Copiar(*r.op1), op2: &Copiar(*r.op2), valor: ""⟩
        [] tipo == NOT
            res ← ⟨tipo: Not, op1: &Copiar(*r.op1), op2: NULL, valor: ""⟩
        [] tipo == VAR
            res ← ⟨tipo: VAR, op1: NULL, op2: NULL, valor: Copiar(s)⟩
    end case
end function
```

La complejidad de este algoritmo es O(R) porque se tienen que copiar todos los nodos.
Como el valor esta acotado a 64 caracteres por enunciado, O(Copiar(s)) ≡ O(1)

6. Módulo Mapa

****
********** Interfaz **********
****

Se explica con: Mapa
Géneros: Mapa

****
********** Operaciones **********
****

Crear() → res: Mapa
Pre ≡ {true}
Post ≡ {res =obs vacío}
Complejidad: O(1)
Descripcion: Crea un mapa vacío

Agregar(in/out m : Mapa, in e : estacion)
Pre ≡ {e ∉ estaciones(m) ∧ m_0 =obs m}
Post ≡ {m =obs agregar(e, m_0)}
Complejidad: O(#estaciones(m) + long(e))
Descripcion: Agrega una estación al mapa

Conectar(in/out m : Mapa, in e1 : estacion, in e2 : estacion, in r : restriccion)
Pre ≡ {e1, e2 ∈ estaciones(m) ∧ₗ ¬conectadas?(m, e1, e2) ∧ m_0 =obs m}
Post ≡ {m =obs conectar(e1, e2, r, m)}
Complejidad: O(long(e1) + long(e2) + S)
Descripcion: Conecta 2 estaciones del mapa entre con una restriccion para ese vinculo.

Estaciones(in m : Mapa) → res : itConj(string)
Pre ≡ {true}
Post ≡ {alias(esPermutación?(SecuSuby(res), estaciones(m))) ∧ vacia?(Anteriores(res))}
Complejidad: O(1)
Descripcion: Devuelve un iterador al conjunto de las estaciones del mapa.
Aliasing: No se pueden modificar los contenidos del iterador.

Conectadas?(in m : Mapa, in e1 : estacion, in e2 : estacion) → res : bool
Pre ≡ {e1, e2 ∈ estaciones(m)}
Post ≡ {res =obs conectadas?(m, e1, e2)}
Complejidad: O(long(e1) + long(e2))
Descripcion: Indica si 2 estaciones estan conectadas.

idSenda(in m : Mapa, in e1 : estacion, in e2 : estacion) → res : nat
Pre ≡ {e1, e2 ∈ estaciones(m) ∧ₗ conectadas?(m, e1, e2)}
Post ≡ {res es el id de la senda en nuestra representación}
Complejidad: O(long(e1) + long(e2))
Descripcion: Devuelve el id de la senda entre e1 y e2.

Sendas(in m : Mapa) → res : itVectorPointer(restriccion)
Pre ≡ {true}
Post ≡ {res itera sobre las restricciones de todas las sendas del mapa}
Complejidad: O(1)
Descripcion: Devuelve las restricciones de las sendas del mapa

Copiar(in m : Mapa) → res : Mapa
Pre ≡ {true}
Post ≡ {res =obs m}
Complejidad: O(Copiar(m.sendas) + Copiar(m.estaciones) + Copiar(m.conexiones))
Descripcion: Devuelve una copia del mapa.
Aliasing: El nuevo mapa comparte las restricciones ya agregadas al mapa original.

```
*****************************************************************************
***********                    Representación                     ***********
*****************************************************************************

Mapa se representa con estr,
    donde estr es: tupla(
                        sendas     : VectorPointer(Restriccion),
                        conexiones : DiccString(DiccString(Int)),
                        estaciones : conj(string)
                        )

Rep: ^(estr) → boolean
Rep(e) ≡ true ⟺
    claves(e.conexiones) ⊂ e.estaciones ∧ι
    ((∀e1 : string)
        e1 ∈ claves(e.conexiones) ∧ι
        claves(obtener(e1, e.conexiones)) ⊂ e.estaciones ∧ι
        ((∀e2 : string)
            (e1 ∈ claves(e.conexiones) ∧ι e2 ∈ claves(obtener(e1, e.conexiones)) ⟺
                e2 ∈ claves(e.conexiones) ∧ι e1 ∈ claves(obtener(e1, e.conexiones))) ∧ι
            (e1 ∈ claves(e.conexiones) ∧ι e2 ∈ claves(obtener(e1, e.conexiones)) ⟹ι
                obtener(e2, obtener(e1, e.conexiones)) < longitud(e.sendas))))

Abs: ^(estr) → mapa                                                {Rep(e)}
(∀e : ^(estr)) Abs(e) =obs m /
    (e.estaciones =obs estaciones(m)) ∧
    ((∀ e1, e2: string)
        definido?(e.conexiones, e1) ∧ι
        (definido?(obtener(e.conexiones, e1), e2) =obs conectadas?(e1, e2, m)) ∧
        (conectadas?(e1, e2, m) ⟹ι *(e.sendas)[i] =obs restriccion(e1, e2, m)))

*****************************************************************************
***********                     Algoritmos                        ***********
*****************************************************************************

iCrear() → res : estr
    res ← ⟨sendas: Vacia(), conexiones: Vacio(), estaciones: Vacio()⟩
end function

iAgregar(in/out m : estr, in e : estacion)
    Agregar(m.estaciones, e)
end function

iConectar(in/out m : estr, in e1 : estacion, in e2 : estacion, in r : restriccion)
    var i : nat ← Longitud(m.sendas)
    AgregarAtras(m.sendas, &r)

    if ¬Definido?(m.conexiones, e1) then
        Definir(m.conexiones, e1, Vacio())
    end if

    if ¬Definido?(m.conexiones, e2) then
        Definir(m.conexiones, e2, Vacio())
    end if

    Definir(Obtener(m.conexiones, e1), e2, i)
    Definir(Obtener(m.conexiones, e2), e1, i)
end function

iEstaciones(in m : estr) → res : itConj(string)
    res ← CrearIt(m.estaciones)
end function

iConectadas?(in m : estr, in e1 : estacion, in e2 : estacion) → res : bool
    res ← Definido?(m.conexiones, e1) ∧ι Definido?(obtener(m.conexiones, e1), e2)
end function

iidSenda(in m : estr, in e1 : estacion, in e2 : estacion) → res : nat
    res ← Obtener(Obtener(m.conexiones, e1), e2)
end function
```

```
iSendas(in m : estr) → res : itVectorPointer(Restriccion)
    res ← CrearIt(m.sendas)
end function

iCopiar(in m : estr) → res : estr
    res ← ⟨    sendas     : Copiar(m.sendas),
            conexiones : Copiar(m.conexiones),
            estaciones : Copiar(m.estaciones)⟩
end function
```

7. Módulo Ciudad

****
********** Interfaz **********
****

Se explica con: Ciudad
Géneros: Ciudad

****
********** Operaciones **********
****

Crear(in m : Mapa) → res : Ciudad
Pre ≡ {true}
Post ≡ {res =obs Crear(m)} actualizar con la copia de mapa.
Complejidad: O(#estaciones(m) * $|e_m|$)
Descripcion: Crea una nueva ciudad a partir de un mapa.
Aliasing: Se realiza una copia del mapa.

Entrar(in ts : conjRapidoString, in e : estacion, in/out c : Ciudad)
Pre ≡ {e ∈ estaciones(mapa(c)) ∧ c =obs c_0}
Post ≡ {c =obs entrar(ts, e, c_0)}
Complejidad: O(long(e) + S*R + N_{total})
Descripcion: Agrega un conjunto de caracteristicas nuevo a la ciudad, creando un nuevo robot.

Mover(in rur : nat, in e : estacion, in/out c : Ciudad)
Pre ≡ {u ∈ robots(c) ∧ e ∈ estaciones(c) ∧$_l$
conectadas?(estacion(u, c), e, mapa(c)) ∧ c_0 =obs c}
Post ≡ {c =obs mover(rur, e, c_0)}
Complejidad: O(long(e) + long(estacion(u, c)) + log(#robotsEn(estacion(u,c), c)) +
log(#robotsEn(e, c)))
Descripcion: Mueve a un robot de una estacion a otra.

Inspeccion(in e : estacion, in/out c : Ciudad)
Pre ≡ {e ∈ estaciones(c)}
Post ≡ {c =obs inspección(e, c)}
Complejidad: O(long(e) + log(#robotsEn(e, c)))
Descripcion: Remueve al robot mas infractor en la estacion e de la ciudad.

ProximoRUR(in c : ciudad) → res : nat
Pre ≡ {true}
Post ≡ {res =obs ProximoRUR(c)}
Complejidad: O(1)
Descripcion: Devuelve el proximo rur disponible.

Mapa(in c : ciudad) → res : Mapa
Pre ≡ {true}
Post ≡ {res =obs Mapa(c)}
Complejidad: O(Copiar(c.mapa))
Descripcion: Devuelve el mapa de la ciudad.
Aliasing: res es una copia de mapa

Robots(in c : ciudad) → res : ItVectorPointer(robot)
Pre ≡ {true}
Post ≡ {alias(res =obs Iterador Unidireccional(Puntero(robot)))}
Complejidad: O(1)
Descripcion: Devuelve un iterador de los robots que hay en la ciudad.
Aliasing: res no es modificable.

Estacion(in c : ciudad, in u : nat) → res : estacion
Pre ≡ {u ∈ robots(c)}
Post ≡ {res =obs estacion(u, c)}
Complejidad: O(1)
Descripcion: Devuelve la estacion donde esta el robot con rur u.

Tags(in c : ciudad, in u : nat) → res : itConj(string)
Pre ≡ {u ∈ robots(c)}
Post ≡ {alias(esPermutacion?(SecuSuby(res), tags(u, c)) ∧ vacia?(Anteriores(res)))}
Complejidad: O(1)
Descripcion: Devuelve un iterador a los tags del robot u.

Aliasing: res no es modificable.

```
#Infracciones(in c : ciudad, in u : nat) → res : nat
Pre ≡ {u ∈ robots(c)}
Post ≡ {res =obs #infracciones(u, c)}
Complejidad: O(1)
Descripcion: Devuelve la cantidad de infracciones del robot con rur u.
```

****
********** Representación **********
****

Ciudad se representa con city,

```
    donde city es: tupla(
                        robots           : VectorPointer(robot),
                        mapa             : Mapa,
                        robotsEnEstacion : DiccString(colaPrioridad(robot)))
    donde robot es: tupla(
                        rur                 : nat,
                        infracciones        : nat,
                        tags                : Puntero(conjRapidoString),
                        estacion            : string,
                        infringe_restriccion : Vector(Bool),
                        mi_estacion         : itCola(robot)
    )

Pre ≡ {true}
Post ≡ {res =obs (r1 < r2)}
Complejidad: O(1)
Descripción: funcion de comparación de robots
• < • (in r1 : robot, r2 : robot) → res : bool
    if r1.infracciones < r2.infracciones then
        res ← true
    else
        if r1.infracciones == r2.infracciones then
            res ← r1.rur < r2.rur
        else
            res ← false
        end if
    end if
end function
```

Escribimos el rep de ciudad informalmente:

• Las claves del diccionario e.robotsEnEstacion todas pertenecen a las estaciones definidas en e.mapa.
• Todos los robots pertenecientes a las colas de los significados de e.robotsEnEstacion tienen la clausula estacíon puesta al sigificado de la cola a la que pertenecen.
• Los elementos de e.robots que no sean NULL cumplen que
    • El rur es igual a su indice en e.robots.
    • El robot pertenece a la cola de prioridad en la entrada estacion de e.robotsEnEstacion (obviamente, la entrada debe estar también definida).
    • mi_estacion se corresponde con la posición del robot en la cola de prioridad asociada a la estación en la que se encuentra.
    • infringe_restriccion se corresponde con los caminos que hay en el mapa, y aparte se corresponde con si el robot verifica o no la restricción entre las sendas.
    • Los elementos en tags tienen una longitud menor o igual a 64.

Abs: ^(city) c → Ciudad {Rep(c)}

```
(∀e : ^(city)) Abs(e) =obs c /
    convertir(e.robots) =obs robots(c) ∧
    long(e.robots) =obs ProximoRUR(c) ∧
    e.mapa =obs mapa(c) ∧
    (∀u : rur)
        (u < Longitud(e.robots) ∧ι e.robots[u] != NULL) ⟹ι
            ((e.robots[u].estacion =obs estacion(u, c)) ∧
            (e.robots[u].tags =obs tags(u, c)) ∧
            (e.robots[u].infracciones =obs #infracciones(u, c)))
```

```
convertir: secu(puntero(α)) → conj(α)
convertir(xs) ≡
    if vacia?(xs) then ϕ
    else
        if prim(xs) =obs NULL then convertir(fin(xs))
        else Ag(prim(xs), convertir(fin(xs))) fi
    fi

*****************************************************************************
***********                       Algoritmos                        ***********
*****************************************************************************

iCrear(in m : Mapa) → res : city
    var robots_en_estacion: DiccString(ColaPrioridad(robot)) ← Crear()
    var it: itConj(string) ← Estaciones(m)

    while HaySiguiente(it) do
        Definir(robots_en_estacion, Siguiente(it), Crear())
        Avanzar(it)
    end while

    Definir(robots_en_estacion, *it, Crear())

    res ← ⟨robots: Vacia(), mapa: Copiar(m), robotsEnEstacion: robots_en_estacion⟩
end function

Recorrer todas las estaciones en el mapa es O(E). Definir cada entrada en
robots_en_estacion es O(|e_m|), siendo e_m el nombre de estacion mas largo, y hay
que hacerlo E veces. Por esto, la complejidad es O(|e_m| * E).

iEntrar(in ts : conjRapidoString, in e : estacion, in/out c : city)
    var rob : robot ← ⟨
        tags: &ts,
        infracciones: 0,
        rur: ProximoRUR(city),
        infringe_restriccion: Vacia(),
        estacion: e,
        mi_estacion: Encolar(obtener(robotsEnEstacion, e), rob)⟩

    var it : ItVectorPointer(Restriccion) ← Sendas(c.mapa)

    while HayMas?(it) do
        AgregarAtras(rob.infringe_restriccion, ¬Verifica?(ts, *Actual(it)))
        Avanzar(it)
    end while

    AgregarAtras(c.robots, &rob)
end function

La operacion obtener(robotsEnEstacion, e) tiene complejidad O(|e|). La operacion
Encolar de cola de prioridad es O(Log de la cantidad de elementos de la cola).
Como se evaluan todas las sendas para saber si un robot infringe o no, el recorrido
lineal del iterador es O(S), siendo S la cantidad de sendas que hay en el mapa.
Evaluar si un robot infringe o no cada senda (Verifica?) es O(R). Finalmente,
agregar un robot a robots de la ciudad es O(N) en el peor caso, siendo N la cantidad
de robots en la ciudad, como esta explicado en agregar atras de vector. Este O(N)
por algebra de ordenes acota superiormente al costo logaritmico de encolar.
Luego, la complejidad queda en |e| + S*R + N_total.

iMover(in rur : nat, in e : estacion, in/out c : city)
    var rob : puntero(robot) ← c.robots[rur]

    Borrar(Obtener(c.robotsEnEstacion, rob→estacion), rob→mi_estacion)

    var infringe : nat ← rob→infracciones
    var id_senda : nat ← idSenda(c.mapa, rob→estacion, e)

    if rob→infringe_restriccion[id_senda] then
        infringe++
```

```
    end if

    rob→infracciones ← infringe
    rob→estacion ← e
    rob→mi_estacion ← Encolar(obtener(c.robotsEnEstacion,  e), *rob)
end function
```

La complejidad de Obtener es O(|e1|), siendo e1 rob→estacion.
Borrar de la cola de prioridad teniendo el iterador al elemento es O(Log(#cola)), siendo #cola la cantidad de elementos de la cola. Todo el calculo de si infringe o no es O(1) porque ya esta precalculado. Luego, insertarlo en otra cola es O(|e|) para encontrarla en el diccionario y O(Log(#cola)) para insertarlo. Esto da como resultado O(|e1| + |e| + Log(#cola1) + Log(#cola2)).

```
iInspeccion(in e : estacion, in/out c : city)
    var cola : colaPrioridad(robot) ← Obtener(c.robotsEnEstacion, e)

    if tamaño(cola) > 0 then
        var rob : robot ← Desencolar(cola)
        c.robots[rob.rur] ← NULL
    end if
end function

iProximoRUR(in c : city) → res : nat
    res ← Longitud(c.robots)
end function

iMapa(in c : city) → res : Mapa
    res ← Copiar(c.mapa)
end function

iRobots(in c : city) → res : ItVectorPointer(robot)
    res ← CrearIt(c.robots)
end function

iEstacion(in c : city, in u : nat) → res : estacion
    res ← (c.robots[u])→estacion
end function

iTags(in c : city, in u : nat) → res : itConj(string)
    res ← CrearIt((c.robots[u])→tags)
end function

i#Infracciones(in c : city, in u : nat) → res : nat
    res ← (c.robots[u])→infracciones
end function
```