DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 2

Algoritmos y Estructuras de Datos II

Grupo18

Integrante	LU	Correo electrónico
Castro Russo, Matias Nahuel	203/19	castronahuel14@gmail.com
Torsello, Juan Manuel	248/19	juantorsello@gmail.com
Capelo, Gianluca	83/19	gianluca.capelo@gmail.com
Yazlle, Máximo	310/19	myazlle99@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Extension Lista Enlazada

Interfaz

CONCATENAR(**in/out** a : `Lista(`α`)`,**in/out** b : `Lista(`α`)`)
Pre $\equiv \{a = a_0 \wedge b = b_0\}$
Post $\equiv \{a = a_0 \& b_0 \wedge b = a_0 \& b_0\}$
Complejidad: $O(1)$
Descripción: concatena la lista a con la lista b, las enlaza
Aliasing: Se genera aliasing en la lista a y b, las dos ahora comparten las mismas posiciones de memoria.

PERTENECE(**in/out** l : `Lista(`α`)`,**in** $elem$: α) $\rightarrow$ res : `Bool`
Pre $\equiv \{l = l_0\}$
Post $\equiv \{res =_{\text{obs}} Pertenece?(l_0, elem)\}$
Complejidad: $O(n)$
Descripción: devuelve True si y solo si, elem pertnece a la lista
Aliasing: No genera Aliasing

Concatenar(in/out a : `Lista(`α`)`,**in/out** b : `Lista(`α`)`

```
1: it_a ← CrearItUlt(a)                                          ▷ O(1)
2: it_b ← CrearIt(b)                                             ▷ O(1)
3:
4: siguiente(anterior(it_a)) ← siguiente(it_b)                   ▷ O(1)
5: anterior(siguiente(it_b)) ← anterior(pit_a)                   ▷ O(1)
```

Complejidad: O(1)

Pertenece(in/out a : `Lista(`α`)`,**in/out** e : $\alpha \rightarrow res$: Bool

```
1: it ← CrearIt(a)                                               ▷ O(1)
2: res ← False                                                   ▷ O(1)
3: while HaySiguiente(siguiente(it)) & !res do                   ▷ O(n)
4:     if siguiente(it) = e then                                 ▷ O(1)
5:         res ← True                                            ▷ O(1)
6:     end if
7: Avanzar(it)                                                   ▷ O(1)
8: end while
       return res                                                ▷ O(1)
```

Complejidad: O(Longitud(a))

---

2. Módulo Mapa

Interfaz

se explica con: MAPA

géneros: `mapa`

Operaciones básicas de mapa

CREAR(**in** hs : `ListaEnlazada(Nat)`,**in** vs : `ListaEnlazada(Nat)`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} agregarRios(hs, vs)\}$
Complejidad: $O(copy(hs), copy(vs))$
Descripción: crea un mapa
Aliasing: No genera Aliasing

HayRio(**in/out** m : `Mapa`,**in** c : `Casilla`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{obs} hayRio?(m, c)\}$
Complejidad: $O(|horizontales| + |verticales|)$
Descripción: verifica si hay rio o no, en la Casilla c
Aliasing: No genera Aliasing

UnirMapa(**in/out** m : `Mapa`,**in/out** $m2$: `Mapa`)
Pre $\equiv$ $\{m = m_0 \wedge m2 = m2_0\}$
Post $\equiv$ $\{m =_{obs} unirMapa(m_0, m2_0) \wedge m2 =_{obs} unirMapa(m_0, m2_0)\}$
Complejidad: $O(1)$
Descripción: Une al mapa m con el mapa m2.
Aliasing: Los rios de los mapas comparten la misma memoria.

Representación

TP 2Representación de mapa
Un mapa contiene rios infinitos horizontales y verticales. Los rios se representan como conjuntos lineales de naturales que indican la posición en los ejes de los ríos.

`mapa` se representa con `estr`

donde `estr` es `tupla`(*horizontales*: `ListaEnlazada(Nat)`,
verticales: `ListaEnlazada(Nat)`)

casilla = <Nat,Nat>

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

Abs : estr m $\longrightarrow$ mapa {Rep(m)}
Abs(m) $\equiv$ agregarRios(nuevoMapa, estr.verticales, estr.horizontales)

Algoritmos

---

crear(**in** hs : `ListaEnlazada(Nat)`, **in** vs : `ListaEnlazada(Nat)`) $\rightarrow$ res : mapa

1: $estr.horizontales \leftarrow hs$
2: $estr.verticales \leftarrow vs$
3: **return** estr

Complejidad: $O(copy(hs) + copy(vs))$

HayRio(**in** m : `mapa`, **in** c : `Casilla`) $\rightarrow$ res : Bool

1: res $\leftarrow$ (Pertenece(m.horizontales,$\pi_1(c)$) || (Pertenece(m.verticales,$\pi_2(c)$)) ▷ $O(|horizontales| + |verticales|)$
2: **return** res

Complejidad: $O(|horizontales| + |verticales|)$

---

unirMapa(**in/out** m_1 : `Mapa`, **in/out** m_2 : `Mapa`)

1: concatenar(m$_1$.$verticales, m_2.verticales$) ▷ $\mathcal{O}(1)$
2: concatenar(m$_1$.$horizontales, m_2.horizontales$) ▷ $\mathcal{O}(1)$

Complejidad: $O(1)$

---

3. PARTIDA

`partida` se representa con `estr`

donde `estr` es `tupla`(*mapa*: `Mapa` ,
construcciones: `ListaEnlazada(Construccion)` ,
popularidad: `Nat` ,
antiguedad: `Nat` ,
seConstruyo: `Bool`)

Construccion = <id : String, antiguedad : Nat, cas: Casilla>
Casilla = <Nat,Nat>

Donde el String se refiere al tipo de construccion ("Casa" o "Comercio"), el Nat se refiere a la antiguedad que tiene esa construccion(no su nivel) y la casilla a su posicion.

3.1. Descripción de la estructura

Nuestra estructura de Partida es una tupla que está compuesta por:

"mapa":

Es una tupla de dos listas enlazadas de naturales (que representan los ríos horizontales y verticales en el mismo). Esto nos permite concatenar en O(1) dos mapas ya que la función de concatenar, en la lista enlazada extendida, enlaza listas en O(1) (siendo la inserción en una lista O(1)).

En la lista puede haber repetidos cuando se unen dos partidas con ríos en la misma posicion pero no nos importa ya que tomamos como criterio que va a haber un río en un mapa cuando aparezca al menos una vez.

"construcciones":

Es una lista enlazada de "Construcción", donde Construcción es un renombre para una tupla que están compuestas por un String, un Nat y una Casilla. Donde el String se refiere al tipo de construccion ("Casa" o "Comercio"), el Nat se refiere a la antiguedad que tiene esa construccion(no su nivel) y la casilla, que es una tupla de dos Nat, a su posicion.

Al ser una lista enlazada nos permite agregar casas y comercios en O(1) que es lo que deseamos. A la vez de permitirnos unir la partida en O(1) ya que enlazarlas es O(1).

Al agregar construcciones, por ejemplo, en la unión pueden aparecen elementos que tengan la misma casilla que otros, pero no nos interesa. Al preguntar por una Casilla, por ejemplo el nivel, ya está contemplado cuál debe devolver.

"popularidad":

Es un Nat que nos devuelve la cantidad de partidas que fueron unidas.

"antigüedad":

Es un Nat que nos devuelve la cantidad de turnos de la partida.

"seConstruyó":

Es un Bool que vale true cuando se construyó en este turno y false en el caso contrario.

Aclaración : Cuando en complejidad de los algoritmos hablamos de n nos referimos al largo de la lista construcciones, la que puede tener elementos con misma posicion dado al algoritmo Unir.

Interfaz

se explica con: SimCity

generos: `SimCity`

Operaciones básicas de partida

EMPEZARPARTIDA(**in** m : `Mapa`) $\rightarrow$ res : `SimCity`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = empezarPartida(m)\}$
Complejidad: $O(Copy(Mapa))$
Descripción: empieza una partida
Aliasing: No genera Aliasing

AGREGARCASA(**in/out** s : `SimCity`,**in** c : `Casilla`)
Pre $\equiv$ $\{s = s_0 \wedge sePuedeConstruir(s_0, c)\}$
Post $\equiv$ $\{agregarCasa(s_0, c) = s\}$
Complejidad: $O(1)$
Descripción: Agrega una casa con posicion c a construcciones
Aliasing: agregamos una construccion con posicion c por copia

AGREGARCOMERCIO(**in/out** s : `SimCity`,**in** c : `Casilla`)
Pre $\equiv$ $\{s = s_0 \wedge sePuedeConstruir(s_0, c)\}$
Post $\equiv$ $\{agregarComercio(s_0, c) = s\}$
Complejidad: $O(1)$
Descripción: Agrega un comercio en la posición c a construcciones
Aliasing: agregamos una construccion con posicion c por copia

AVANZARTURNO(**in/out** s : `SimCity`)
Pre $\equiv$ $\{s = s_0 \wedge huboConstruccion(s)\}$
Post $\equiv$ $\{avanzarTurno(s_0) = s\}$
Complejidad: $O(n)$
Descripción: Aumenta en uno la antiguedad de la partida y cada antiguedad interna de todas las construcciones
Aliasing: No genera aliasing

UNIR(**in/out** s : `SimCity`,**in** $s2$: `SimCity`)
Pre $\equiv$ $\{s = s_1 \wedge AptoParaUnion(s_1, s_2)\}$
Post $\equiv$ $\{Unir(s_1, s_2) = s\}$
Complejidad: $O(1)$ sin contar la complejidad de copia del segundo simcity
Descripción: Une dos partidas de simCity
Aliasing: No genera aliasing

Aclaracion: AptoParaUnion es la misma precondicion que tiene el generador unir en el tad.

NIVEL(**in/out** s : `SimCity`,**in** c : `Casilla`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{HayConstruccion?(s, c)\}$
Post $\equiv$ $\{nivel(s, c) = res\}$
Complejidad: $O(n)$
Descripción: Devuelve el nivel de una construccion
Aliasing: No genera aliasing, devuelve el nivel por copia

POPULARIDAD(**in/out** s : `SimCity`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = popularidad(s)\}$
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de uniones
Aliasing: No genera aliasing, devuelve la popularidad por copia

ANTIGUEDAD(**in/out** s : `SimCity`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res=antiguedad(s)}
Complejidad: $O(1)$
Descripción: Devuelve cantidad de turnos que pasaron en la partida.
Aliasing: No genera aliasing, devuelve la antiguedad por copia

VERCASAS(**in/out** s : `SimCity`) $\rightarrow$ res : `conj(casillas)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = casas(s)\}$

Complejidad: $O(n^2)$
Descripción: Devuelve el conjunto de casas de una partida
Aliasing: No genera aliasing, devuelve el conjunto de casillas donde hay casas por copia

verComercios(**in/out** s : `SimCity`) $\rightarrow$ res : `conj(casillas)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = comercios(s)\}$
Complejidad: $O(n^2)$
Descripción: Devuelve el conjunto de comercios de una partida
Aliasing: No genera aliasing, devuelve el conjunto de casillas donde hay comercios por copia

verMapa(**in/out** s : `SimCity`) $\rightarrow$ res : `Mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = mapa(s)\}$
Complejidad: $O(|horizontales|^2 + |verticales|^2)$
Descripción: Devuelve el mapa sin repetidos
Aliasing: No genera aliasing, devuelve el Mapa por copia

huboConstrucción(**in/out** s : `SimCity`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = huboConstruccin(s)\}$
Complejidad: $O(1)$
Descripción: devuelve true si se construyó en el turno actual de la partida
Aliasing: No genera aliasing, se devuelve por copia.

Algoritmos

---

crearPartida(**in** m : `mapa`) $\rightarrow$ res : estr

1: estr.popularidad $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
2: estr.antiguedad $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
3: estr.seConstruyo $\leftarrow$ False ▷ $\mathcal{O}(1)$
4: estr.mapa $\leftarrow$ m ▷ $\mathcal{O}(|m|)$
5: estr.construcciones $\leftarrow$ Vacia() ▷ $\mathcal{O}(1)$
6: **return** estr ▷ $\mathcal{O}(|m|)$

Complejidad: O(|m|)

---

agregarCasa(**in/out** e : `estr`, **in** cas : `casilla`)

1: agregarAtras(e.construcciones, $<$ "$Casa$", $0, cas >$) ▷ $\mathcal{O}(1)$
2: e.seConstruyo $\leftarrow$ True

Complejidad: O(1)

---

agregarComercio(**in/out** e : `estr`, **in** cas : `casilla`)

1: agregarAtras(e.construcciones, $<$ "$Comercio$", $0, cas >$) ▷ $\mathcal{O}(1)$
2: e.seConstruyo $\leftarrow$ True

Complejidad: O(1)

---

avanzarTurno(**in**/**out** e : `estr`)

```
1: it ← CrearIt(e.construcciones)                          ▷ O(1)
2: while HaySiguiente(it) do                               ▷ O(n)
3:     π2(siguiente(it)) ← π2(siguiente(it)) + 1           ▷ O(1)
4:     avanzar(it)                                         ▷ O(1)
5: end while
6: antiguedad ← antiguedad+1                               ▷ O(1)
7: e.seConstruyo ← False
```

Complejidad: O(n)

Unir(**in**/**out** e_0 : `estr`, **in**/**out** e_1 : `estr`)

```
1: concatenar(e0.construcciones, e1.construcciones)        ▷ O(1)
2: concatenar(e0.mapa, e1.mapa)                            ▷ O(1)
3:
4: popularidad(e0) ← popularidad(e0)+popularidad(e1) +1    ▷ O(1)
5: antiguedad(e0) ← max(antiguedad(e0), antiguedad(e1))    ▷ O(1)
6:
7: if e0.seConstruyo || e1.seConstruyo then                ▷ O(1)
8:     e.seConstruyo ← True                                ▷ O(1)
9: end if
```

Complejidad: O(1)

---

antiguedad(**in**/**out** e : `estr`) → res : nat

```
1: return e.antiguedad                                     ▷ O(1)
```

Complejidad: O(1)

nivel(**in**/**out** e : `estr`, **in** c : `Casilla`) → res : nat

```
1: hayComercio ← false                                                          ▷ O(1)
2: hayCasa ← false                                                              ▷ O(1)
3: resCasa ← 0                                                                  ▷ O(1)
4: resComercio ← 0                                                              ▷ O(1)
5:  punt ← CrearIt(e.construcciones)                                            ▷ O(1)
6:
7: while HaySiguiente(siguiente(punt)) do                                       ▷ O(n)
8:     if π3(siguiente(punt))=c then                                            ▷ O(1)
9:         if π1(siguiente(punt))="Casa" then                                   ▷ O(1)
10:            resCasa ← max(π2(siguiente(punt)), resCasa)                      ▷ O(1)
11:            hayCasa ← true                                                   ▷ O(1)
12:        else
13:            if π1(siguiente(punt))="Comercio" & !hayCasa then                ▷ O(1)
14:                resCom ← max(π2(siguiente(punt)), resCom)                    ▷ O(1)
15:                hayComercio ← true                                           ▷ O(1)
16:            end if
17:        end if
18:    end if
19:    avanzar(punt)                                                            ▷ O(1)
20: end while
21:
22: if !hayCasa then                                                            ▷ O(1)
23:    hayQueVerManhattan ← true                                                ▷ O(1)
24:    maxMan ← 0                                                               ▷ O(1)
25:    puntMan ← CrearIt(e.construcciones)                                      ▷ O(1)
26: end if
27:
28: while hayQueVerManhattan & HaySiguiente(siguiente(puntMan)) do              ▷ O(n)
29:    if π1(siguiente(puntMan))="Casa" & distanciaManhattan(c, π3(siguiente(puntMan)) ≤3 then  ▷ O(1)
30:        maxMan ← max(maxMan,π2(siguiente(puntMan)))                          ▷ O(1)
31:    end if
32:    avanzar(puntMan)                                                         ▷ O(1)
33: end while
34:
35: if HayQueVerManhattan then                                                  ▷ O(1)
36:    return max(resCom,maxMan)                                                ▷ O(1)
37: else
38:    return resCasa                                                           ▷ O(1)
39: end if
```

Complejidad: O(n)

---

distanciaManhattan(**in** c : `Casilla` **in** $cMan$: `Casilla` → res : nat

```
1: res ← |π1(c) − π1(cMan)| + |π2(c) − π2(cMan)|                                ▷ O(1)
2: return res                                                                   ▷ O(1)
```

Complejidad: O(1)

verCasas(in/out e : `estr`**)** $\rightarrow$ res : conjuntoLineal(Casilla)

```
1: res ← Vacio()                                                    ▷ O(1)
2: it ← CrearIt(e.construcciones)                                   ▷ O(1)
3:
4: while HaySiguiente(Siguiente(it)) do                             ▷ O(n)
5:     if π1(Siguiente(it) = "Casa" then                            ▷ O(1)
6:         Agregar(res,π3(Siguiente(it)))                           ▷ O(n)
7:     end if
8:     Avanzar(it)                                                  ▷ O(1)
9: end while
       return res
```

Complejidad: O(n^2)

verComercio(in/out e : `estr`**)** $\rightarrow$ res : conjuntoLineal(Casilla)

```
1: res ← Vacio()                                                    ▷ O(1)
2: it ← CrearIt(e.construcciones)                                   ▷ O(1)
3: Casas ← verCasas(e)                                              ▷ O(n^2 + n) ∈ O(n^2)
4:
5: while HaySiguiente(Siguiente(it)) do                             ▷ O(n)
6:     if π1(Siguiente(it)) = "Comercio" & !(Pertenece(Casas,π1(Siguiente(it))) then   ▷ O(n)
7:         Agregar(res,π3(Siguiente(it)))                           ▷ O(n)
8:     end if
9:     Avanzar(it)                                                  ▷ O(1)
10: end while
       return res
```

Complejidad: O(n^2)

verMapa(in/out e : `estr`**)** $\rightarrow$ res : Mapa

```
1: resHs ← Vacio()                                                  ▷ O(1)
2: resVs ← Vacio()                                                  ▷ O(1)
3: it_0 ← CrearIt(e.mapa.horizontales)                              ▷ O(1)
4: it_1 ← CrearIt(e.mapa.verticales)                                ▷ O(1)
5:
6: while HaySiguiente(Siguiente(it_0)) do                           ▷ O(|hs|)
7:     if !pertenece(resHs, Siguiente(it_0)) then                   ▷ O(|hs|)
8:         AgregarAtras(resHs,(Siguiente(it_0)))                    ▷ O(1)
9:     end if
10:    Avanzar(it_0)                                                ▷ O(1)
11: end while
12:
13: while HaySiguiente(Siguiente(it_1)) do                          ▷ O(|vs|)
14:    if !pertenece(resVs, Siguiente(it_1)) then                   ▷ O(|vs|)
15:        AgregarAtras(resVs,(Siguiente(it_1)))                    ▷ O(1)
16:    end if
17:    Avanzar(it_1)                                                ▷ O(1)
18: end while
       return Crear(resHs,resVs)
```

Complejidad: O($|hs|^2 + |vs|^2$)

huboConstrucción(**in/out** e : `estr`) $\rightarrow$ res : bool

1: **return** e.seConstruyo ▷ $\mathcal{O}(1)$

Complejidad: O(1)

Representación

Invariante de Representacion

1. No deben coincidir rios y construcciones.

2. El string de las construcciones sea "*casa*" o "*comercio*" .

3. *popularidad* sea mayor o igual a la mayor cantidad de veces que se repite una posicion en *construcciones*. Esto debe porque solo agregan casas o comercios cuando no se hayan agregado antes, pero en el momento de unir partidas no pasa lo mismo se sobrecarga la estructura dando posibilidad a que aparezcan repetidos .

4. *SeConstruyo* sea verdadero si y solo si, hay alguna construccion con antiguedad cero.

5. Verificar que no haya ninguna construccion con antiguedad mayor a la *antiguedad*

6. Verificar que todas las antiguedades pertenecientes a *construcciones* sean validas, ya que una de las restricciones del simcity, es que en todo turno tiene que haber por lo menos una nueva construccion

Utilizaremos el tad secuencia para explicar la lista doblemente enlazada construcciones.

string ES secu(char)
Casilla ES tupla(nat,nat)
construccion ES tupla(tipo:string,antiguedadConstruccion:nat,posicion:Casilla)

1. ($\forall$*construc*: construccion)(esta?(construc,e.construcciones)$\rightarrow$
$\neg(\pi_1(\pi_3(construc)) \in mapa.horizonateles) \wedge \neg(\pi_2(\pi_3(construc)) \in mapa.verticales)$

2. ($\forall$*construc*: construccion)(esta?(construc,e.construcciones)$\rightarrow$
$(\pi_1(construc) = "Casa") \vee (\pi_1(construc) = "Comercio"))$

3. ($\forall$*construc*: construccion)(esta?(construc,e.construcciones)$\rightarrow$
Repeticiones($e.construcciones$,π_3(construc)) $\leq e.popularidad$)

4. $e.SeConstruyo \Leftrightarrow$ ($\exists$*construc*: Construccion)(esta?(construc,e.construcciones) $\wedge(\pi_2(construcc) = 0))$

5. ($\forall$*construc*: construccion)(esta?(construc,e.construcciones)$\rightarrow$ ($\pi_2(construc) \leq e.antiguedad$)

6. ($\forall i$: N)(0 <i $\leq e.antiguedad \rightarrow_L$
(($\exists$*construc*: Construccion)(esta?(construc,e.construcciones) $\wedge_L \pi_2(construc) = i)$)

Rep(e) $\equiv$ true $\Longleftrightarrow$ (1)$\wedge$(2)$\wedge$(3)$\wedge$(4)$\wedge$(5)$\wedge$(6)

Auxiliares:

Repeticiones$(s, casilla) = \sum_{i=0}^{long(s)-1}$ **if** $\pi_3(iesimo(s)) = casilla)$ **then** 1 **else** 0 **fi**

iesiemo : secu(α) $s \times$ nat $i \longrightarrow \alpha$ $\{0 \leq i < long(s)\}$

iesimo(s,indice) $\equiv$ **if** indice $= 0$ **then** prim(s) **else** iesimo(fin(s),indice-1) **fi**

Función de abstracción

1. $e.mapa =_{obs} mapa(s)$

2. ($\forall construc$: construccion)(esta?(construc,e.construcciones) $\rightarrow \pi_3(construc) \in casas(s) \vee \pi_3(construc) \in comercios(s)$)

3. ($\forall cas$: Casilla)(cas $\in casas(s) \rightarrow$ ($\exists nivel$: Nat)(esta?($\langle$"$Casa$", $nivel, cas\rangle, e.construcciones$))

4. ($\forall cas$: Casilla)(cas $\in comercios(s) \rightarrow$ ($\exists n$: Nat)(esta?($\langle$"$Comercio$", $n, cas\rangle, e.construcciones$) $\wedge$ ($\neg(\exists n$: Nat)(esta?($\langle$"$Casa$", $n, cas\rangle, e.construcciones$)))

5. ($\forall cas$: casilla)(cas $\in casas(s) \rightarrow EsNivelMaximoDeCasa?(e.construcciones, Nivel(s, cas), cas)$)

$\wedge$

($\forall cas$: Casilla)(cas $\in comercios(s) \rightarrow Nivel(s, cas) = NivelComercioEstructura(e.construcciones, cas, 0$

6. $HuboConstruccion(s) = e.SeConstruyo$

7. $popularidad(s) = e.popularidad$

($\forall \widehat{estr}$: e)**Abs(e)**$=_{\text{obs}}$ $s : SimCity|$ (1)$\wedge$(2)$\wedge$(3)$\wedge$(4)$\wedge$(5)$\wedge$(6)$\wedge$(7)

Auxiliares :

EsNivelMaximoDeCasa?(c:secu<construccion>,n: nat,cas: Casilla) =
$esta?(< "Casa", n, cas >, c) \wedge (\forall nivel$: nat)(esta?(<"Casa",nivel,cas>,c) $\rightarrow nivel \leq n$)

NivelComercioEstructura : secu(construccion) $\times$ casilla $\times$ nat $\longrightarrow$ nat

NivelComercioEstructura(s,cas,n) $\equiv$ **if** vacia?(s) **then** n **else**
if $\pi_1(prim(s)) =$ "$Comercio$" $\wedge \pi_3(prim(s)) = cas$
then
NivelComercioEstructura(fin(s),cas,max{ $\pi_2(prim(s))$,n })
else
if $\pi_1(prim(s)) =$ "$Casa$" $\wedge \neg(\pi_3(prim(s)) = cas) \wedge DistanciaManhattan?(cas, \pi_3(prim(s))$
then NivelComercioEstructura(fin(s),cas,max{ $\pi_2(prim(s))$,n })
else NivelCasaEstructura(fin(s),cas,n) **fi**
fi