# Algoritmos y Estructuras de Datos II

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

**De los creadores de sacarCompu...**

## Trabajo práctico 2

Diseño - DCNet

**Grupo 11**



| Integrante | LU | Correo electrónico |
|---|---|---|
| Frizzo, Franco | 013/14 | francofrizzo@gmail.com |
| Martínez, Manuela | 160/14 | martinez.manuela.22@gmail.com |
| Rabinowicz, Lucía | 105/14 | lu.rabinowicz@gmail.com |
| Weber, Andrés | 923/13 | herr.andyweber@gmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# 1. Módulo Diccionario AVL

## Interfaz

**parámetros formales**

**géneros** $\kappa$, $\sigma$

**función** $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\to res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$
**Complejidad:** $\Theta(equal(k_1, k_2))$
**Descripción:** función de igualdad de $\kappa$

**función** $\bullet \leq \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\to res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} (k_1 \leq k_2)\}$
**Complejidad:** $\Theta(order(k_1, k_2))$
**Descripción:** función de comparación por orden total estricto de $\kappa$

**función** COPIAR(**in** $k : \kappa$) $\to res : \kappa$
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} k\}$
**Complejidad:** $\Theta(copy(k))$
**Descripción:** función de copia de $\kappa$

**función** COPIAR(**in** $s : \sigma$) $\to res : \sigma$
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} s\}$
**Complejidad:** $\Theta(copy(s))$
**Descripción:** función de copia de $\sigma$

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

**géneros**: `diccAVL(`$\kappa$`, `$\sigma$`)`

### Operaciones de diccionario

VACIO() $\to res$ : `diccAVL(`$\kappa$`, `$\sigma$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}}$ vacío$\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Crea y devuelve un diccionario AVL vacío.

DEFINIR(**in**/**out** $d$: `diccAVL(`$\kappa$`, `$\sigma$`)`, **in** $k : \kappa$, **in** $s : \sigma$)
**Pre** $\equiv \{d =_{\text{obs}} d_0\}$
**Post** $\equiv \{d =_{\text{obs}}$ definir($k$, $s$, $d_0$)$\}$
**Complejidad:** $\Theta(\log(n) \times order(k) + copy(k) + copy(s))$
**Descripción:** Define en el diccionario la clave pasada por parámetro con el significado pasado por parámetro. En caso de que la clave ya esté definida, sobreescribe su significado con el nuevo.

BORRAR(**in**/**out** $d$: `diccAVL(`$\kappa$`, `$\sigma$`)`, **in** $k : \kappa$)
**Pre** $\equiv \{d =_{\text{obs}} d_0 \wedge$ def?($k$, $d$)$\}$
**Post** $\equiv \{d =_{\text{obs}}$ borrar($k$, $d_0$)$\}$
**Complejidad:** $\Theta(\log(n) \times order(k))$
**Descripción:** Elimina del diccionario la clave pasada por parámetro.

#CLAVES(**in** $d$: `diccAVL(`$\kappa$`, `$\sigma$`)`) $\to res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} \#(\text{claves}(d))\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve la cantidad de claves del diccionario.

DEFINIDO?(**in** $d$: `diccAVL(`$\kappa$`, `$\sigma$`)`, **in** $k$: $\kappa$) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ def?($k$, $d$)}
**Complejidad:** $\Theta(\log(n) \times order(k))$
**Descripción:** Devuelve true si y solo si la clave pasada por parámetro está definida en el diccionario.

OBTENER(**in** $d$: `diccAVL(`$\kappa$`, `$\sigma$`)`, **in** $k$: $\kappa$) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {def?($k$, $d$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ obtener($k$, $d$)}
**Complejidad:** $\Theta(\log(n) \times order(k))$
**Descripción:** Devuelve el significado con el que la clave pasada por parámetro está definida en el diccionario.
**Aliasing:** El significado se pasa por referencia. Modificarlo implica cambiarlo en la estructura interna del diccionario.

# Representación

`diccAVL(`$\kappa$`, `$\sigma$`)` **se representa con** `estrAVL`

donde `estrAVL` es `tupla`(*raiz*: `puntero(nodo)`,
*cantNodos*: `nat`)

donde `nodo` es `tupla`(*clave*: $\kappa$,
*significado*: $\sigma$,
*padre*: `puntero(nodo)`,
*izq*: `puntero(nodo)`,
*der*: `puntero(nodo)`,
*altSubarbol*: `nat`)

Rep : estrAVL $\longrightarrow$ bool

Rep($e$) $\equiv$ true $\iff$ $e.cantNodos = \#(\text{nodos}(e)) \wedge$
$\#(\text{claves}(e)) = \#(\text{nodos}(e)) \wedge$
$(\forall\, n : \text{nodo})\ (n \in \text{nodos}(e) \Rightarrow_{\text{L}} ($
$n.altSubarbol = \text{altura}(\text{subárbol}(\&n)) \wedge$
$\text{máx}(\text{altura}(\text{subárbol}(n.izq)), \text{altura}(\text{subárbol}(n.der))) -$
$\text{mín}(\text{altura}(\text{subárbol}(n.izq)), \text{altura}(\text{subárbol}(n.der))) \leq 1 \wedge$
$(\forall\, n' : \text{nodo})\ ((n' \in \text{nodos}(\text{subárbol}(n))) \Rightarrow (n' \in \text{nodos}(\text{subárbol}(n.der)) \Leftrightarrow \neg(n'.clave \leq n.clave))) \wedge$
$*(n.izq) \neq *(n.der) \wedge (n.padre = \text{NULL} \Leftrightarrow \&n = e.raiz) \wedge_{\text{L}}$
$((\&n \neq e.raiz) \Rightarrow_{\text{L}} (\forall\, n' : \text{nodo})(n.padre = \&n' \Leftrightarrow n'.izq = \&n \vee n'.der = \&n))))$

Abs : estrAVL $e$ $\longrightarrow$ dicc($\kappa$, $\sigma$)) {Rep($e$)}

Abs($e$) $=_{\text{obs}}$ $d$: dicc($\kappa$, $\sigma$)) | $(\forall\, \kappa : \text{k})\ ((\text{def?}(k, d)) =_{\text{obs}} (k \in \text{claves}(e)) \wedge_{\text{L}} (\text{def?}(k, d) \Rightarrow_{\text{L}} \text{obtener}(k, d) =_{\text{obs}}$ significado($k$, $e$)))

hijos : nodo $\longrightarrow$ conj(nodo)

hijos($n$) $\equiv$ **if** $n.izq = \text{NULL}$ **then** $\emptyset$ **else** $\text{Ag}(*(n.izq), \text{hijos}(*(n.izq)))$ **fi**
$\cup$ **if** $n.der = \text{NULL}$ **then** $\emptyset$ **else** $\text{Ag}(*(n.der), \text{hijos}(*(n.der)))$ **fi**

nodos : estrAVL $\longrightarrow$ conj(nodo)

nodos($e$) $\equiv$ **if** $e.raiz = \text{NULL}$ **then** $\emptyset$ **else** $\text{Ag}(*(e.raiz), \text{hijos}(*(e.raiz)))$ **fi**

subárbol : puntero(nodo) $\longrightarrow$ estrAVL

subárbol($p$) $\equiv$ $\langle p, 1 + \#(\text{hijos}(*(p)))\rangle$

claves : estrAVL $\longrightarrow$ conj($\kappa$)

claves($e$) $\equiv$ **if** $e.raiz = \text{NULL}$ **then**
$\emptyset$
**else**
Ag($e.raiz \to clave$, claves(subárbol($e.raiz \to izq$)) $\cup$ claves(subárbol($e.raiz \to der$)))
**fi**

altura : estrAVL $\longrightarrow$ nat

altura($e$) $\equiv$ **if** $e.raiz = \text{NULL}$ **then**
0
**else**
1 + máx(altura(subárbol($e.raiz \to izq$)), altura(subárbol($e.raiz \to der$)))
**fi**

significado : estrAVL $e \times \kappa\ k \longrightarrow \sigma$ $\{k \in \text{claves}(e)\}$

significado($e, k$) $\equiv$ **if** $e.raiz \to clave = k$ **then**
$e.raiz \to significado$
**else**
**if** $k \in$ claves(subárbol($e.raiz \to izq$)) **then**
significado($k$, subárbol($e.raiz \to izq$))
**else**
significado($k$, subárbol($e.raiz \to der$))
**fi**
**fi**

# Algoritmos

iVacio() $\to res$ : `estrAVL`

1 $res \leftarrow \langle \text{NULL}, 0 \rangle$ ▷ $\Theta(1)$

**Complejidad:** $\Theta(1)$

iDefinir(**in/out** $e$ : `estrAVL`, **in** $k : \kappa$, **in** $s : \sigma$)

1 `puntero(nodo)` $padre \leftarrow$ NULL ▷ $\Theta(1)$
2 `puntero(nodo)` $lugar \leftarrow$ Buscar($e$, $k$, $padre$) ▷ $\Theta(\log(n) \times order(k))$
3 **if** $lugar \neq$ NULL **then** ▷ $\Theta(1)$
4 | ($lugar \to significado$) $\leftarrow$ Copiar($s$) ▷ $\Theta(copy(s))$
5 **else**
6 | `puntero(nodo)` $nuevo \leftarrow$ &$\langle$Copiar($k$), Copiar($s$), NULL, NULL, NULL, 1$\rangle$ // Reservamos memoria para el nuevo nodo
▷ $\Theta(copy(k) + copy(s))$
7 | **if** $k \leq (padre \to clave)$ **then** ▷ $\Theta(order(k))$
8 | | ($padre \to izq$) $\leftarrow nuevo$ ▷ $\Theta(1)$
9 | **else**
10 | | ($padre \to der$) $\leftarrow nuevo$ ▷ $\Theta(1)$
11 | **end**
12 | ($nuevo \to padre$) $\leftarrow padre$ ▷ $\Theta(1)$
13 | RebalancearArbol ($padre$) ▷ $\Theta(\log(n))$
14 | $e.cantNodos++$ ▷ $\Theta(1)$
15 **end**

**Complejidad:** $\Theta(\log(n) \times order(k) + copy(k) + copy(s))$

**Justificación:** La función tiene llamadas a funciones con complejidad $\Theta(\log(n) \times order(k)$ y $\Theta(copy(k) + copy(s))$.

| iObtener(**in** $e$ : `estrAVL`, **in** $k$ : $\kappa$) $\rightarrow$ $res$ : $\sigma$ | |
|---|---|
| 1 `puntero(nodo)` $padre \leftarrow$ NULL | ▷ $\Theta(1)$ |
| 2 `puntero(nodo)` $lugar \leftarrow$ Buscar($e$, $k$, $padre$) | ▷ $\Theta(\log(n) \times order(k))$ |
| 3 $res \leftarrow (lugar \rightarrow significado)$ | ▷ $\Theta(1)$ |

**Complejidad:** $\Theta(\log(n) \times order(k))$

**Justificación:** La función tiene llamadas a funciones con complejidad $\Theta(\log(n) \times order(k)$ y $\Theta(copy(k) + copy(s))$.

| #Claves(**in** $e$ : `estrAVL`) $\rightarrow$ $res$ : `nat` | |
|---|---|
| 1 $res \leftarrow e.cantNodos$ | ▷ $\Theta(1)$ |

**Complejidad:** $\Theta(1)$

| Definido?(**in** $e$ : `estrAVL`, **in** $k$ : $\kappa$) $\rightarrow$ $res$ : `bool` | |
|---|---|
| 1 `puntero(nodo)` $padre \leftarrow$ NULL | ▷ $\Theta(1)$ |
| 2 `puntero(nodo)` $lugar \leftarrow$ Buscar($e$, $k$, $padre$) | ▷ $\Theta(\log(n) \times order(k))$ |
| 3 $res \leftarrow (lugar \neq$ NULL) | ▷ $\Theta(1)$ |

**Complejidad:** $\Theta(\log(n) \times order(k))$

iBorrar(**in/out** $e$ : `estrAVL`, **in** $k : \kappa$)

```
1  puntero(nodo) padre ← NULL                                              ▷ Θ(1)
2  puntero(nodo) lugar ← Buscar(e, k, padre)                               ▷ Θ(log(n) × order(k))
3  if lugar → izq = NULL ∧ lugar → der = NULL then                         ▷ Θ(1)
4      if padre ≠ NULL then                                                ▷ Θ(1)
5          if padre → izq = lugar then                                     ▷ Θ(1)
6              (padre → izq) ← NULL                                        ▷ Θ(1)
7          else
8              (padre → der) ← NULL                                        ▷ Θ(1)
9          end
10         RebalancearArbol(padre)                                         ▷ Θ(log(n))
11     else
12         e.raiz = NULL                                                   ▷ Θ(1)
13     end
14 else if lugar → der = NULL then                                         ▷ Θ(1)
15     (lugar → izq → padre) ← padre                                       ▷ Θ(1)
16     if padre ≠ NULL then                                                ▷ Θ(1)
17         if padre → izq = lugar then                                     ▷ Θ(1)
18             (padre → izq) ← (lugar → izq)                               ▷ Θ(1)
19         else
20             (padre → der) ← (lugar → izq)                               ▷ Θ(1)
21         end
22         RebalancearArbol(padre)                                         ▷ Θ(log(n))
23     else
24         e.raiz ← lugar → izq                                            ▷ Θ(1)
25     end
26 else if lugar → izq = NULL then                                         ▷ Θ(1)
27     (lugar → der → padre) ← padre                                       ▷ Θ(1)
28     if padre ≠ NULL then                                                ▷ Θ(1)
29         if padre → izq = lugar then                                     ▷ Θ(1)
30             (padre → izq) ← (lugar → der)                               ▷ Θ(1)
31         else
32             (padre → der) ← (lugar → der)                               ▷ Θ(1)
33         end
34         RebalancearArbol(padre)                                         ▷ Θ(log(n))
35     else
36         e.raiz ← lugar → izq                                            ▷ Θ(1)
37     end
```

iBorrar *(cont.)*

```
38 else
39 |  puntero(nodo) reemplazo ← (lugar → der)                                  ▷ Θ(1)
40 |  if (reemplazo → izq = NULL) then                                         ▷ Θ(1)
41 |  |  if padre ≠ NULL then                                                  ▷ Θ(1)
42 |  |  |  if (padre → izq) = lugar then                                     ▷ Θ(1)
43 |  |  |  |  (padre → izq) ← reemplazo                                      ▷ Θ(1)
44 |  |  |  else
45 |  |  |  |  (padre → der) ← reemplazo                                      ▷ Θ(1)
46 |  |  |  end
47 |  |  else
48 |  |  |  e.raiz ← reemplazo                                               ▷ Θ(1)
49 |  |  end
50 |  |  (reemplazo → padre) ← padre                                          ▷ Θ(1)
51 |  |  (reemplazo → izq) ← lugar → izq                                      ▷ Θ(1)
52 |  |  (lugar → izq → padre) ← reemplazo                                    ▷ Θ(1)
53 |  |  RebalancearArbol(reemplazo)                                          ▷ Θ(log(n))
54 |  else
55 |  |  while (reemplazo → izq) ≠ NULL do                                    ▷ Θ(log(n)) iteraciones
56 |  |  |  reemplazo ← (reemplazo → izq)                                     ▷ Θ(1)
57 |  |  end
58 |  |  puntero(nodo) padreReemplazo ← (reemplazo → padre)                   ▷ Θ(1)
59 |  |  if padre ≠ NULL then                                                 ▷ Θ(1)
60 |  |  |  if padre → izq = lugar then                                      ▷ Θ(1)
61 |  |  |  |  (padre → izq) ← reemplazo                                      ▷ Θ(1)
62 |  |  |  else
63 |  |  |  |  (padre → der) ← reemplazo                                      ▷ Θ(1)
64 |  |  |  end
65 |  |  else
66 |  |  |  e.raiz ← reemplazo                                               ▷ Θ(1)
67 |  |  end
68 |  |  (reemplazo → padre) ← padre                                          ▷ Θ(1)
69 |  |  (reemplazo → izq) ← lugar → izq                                      ▷ Θ(1)
70 |  |  (lugar → izq → padre) ← reemplazo                                    ▷ Θ(1)
71 |  |  (padreReemplazo → izq) ← (reemplazo → der)                           ▷ Θ(1)
72 |  |  if (reemplazo → der) ≠ NULL then                                     ▷ Θ(1)
73 |  |  |  (reemplazo → der → padre) ← padreReemplazo                        ▷ Θ(1)
74 |  |  end
75 |  |  (reemplazo → der) ← (lugar → der)                                    ▷ Θ(1)
76 |  |  (lugar → der → padre) ← reemplazo                                    ▷ Θ(1)
77 |  |  RebalancearArbol(reemplazo)                                          ▷ Θ(log(n))
78 |  end
79 end
80 delete(lugar)                                                              ▷ Θ(1)
```

**Complejidad:** $\Theta(\log(n) \times order(k))$

iBuscar(**in** $e$ : `estrAVL`, **in** $k : \kappa$, **out** $padre$ : `puntero(nodo)`) $\rightarrow$ $res$ : `puntero(nodo)`

```
1  padre ← NULL                                                          ▷ Θ(1)
2  actual ← e.raiz                                                       ▷ Θ(1)
3  while actual ≠ NULL ∧_L (actual → clave ≠ k) do                       ▷ Θ(log(n)) iteraciones
4  |  padre ← actual                                                     ▷ Θ(1)
5  |  if k ≤ (padre → clave) then                                        ▷ Θ(order(k))
6  |  |  actual ← (actual → izq)                                        ▷ Θ(1)
7  |  else
8  |  |  actual ← (actual → der)                                        ▷ Θ(1)
9  |  end
10 end
11 res ← actual                                                          ▷ Θ(1)
```

**Complejidad:** $\Theta(\log(n) \times order(k))$

iRecalcularAltura(**in** $n$ : `puntero(nodo)`)

```
1  if n → izq ≠ NULL ∧ n → der ≠ NULL then                               ▷ Θ(1)
2  |  (n → altSubarbol) ← 1 + max(n → izq → altSubarbol, n → der → altSubarbol)   ▷ Θ(1)
3  else if n → izq ≠ NULL then                                           ▷ Θ(1)
4  |  (n → altSubarbol) ← 1 + (n → izq → altSubarbol)                    ▷ Θ(1)
5  else if n → der ≠ NULL then                                           ▷ Θ(1)
6  |  (n → altSubarbol) ← 1 + (n → der → altSubarbol)                    ▷ Θ(1)
7  else
8  |  (n → altSubarbol) ← 1                                              ▷ Θ(1)
9  end
```

**Complejidad:** $\Theta(1)$

iFBD(**in** $n$ : `puntero(nodo)`) $\rightarrow$ $res$ : `int`

```
1  int altIzq ← n → izq = NULL ? 0 : n → izq → altSubarbol               ▷ Θ(1)
2  int altDer ← n → der = NULL ? 0 : n → izq → altSubarbol               ▷ Θ(1)
3  res ← altDer − altIzq                                                 ▷ Θ(1)
```

**Pre** $\equiv$ {caca}

**Complejidad:** $\Theta(1)$

iRotarAIzquierda(**in** $n$ : `puntero(nodo)`)

```
1  if n.padre ≠ NULL then                                                ▷ Θ(1)
2  |  if n.padre → izq = n then                                          ▷ Θ(1)
3  |  |  (n → padre → izq) ← n → der                                     ▷ Θ(1)
4  |  else
5  |  |  (n → padre → der) ← n → der                                     ▷ Θ(1)
6  |  end
7  end
8  (n → der → padre) ← n → padre                                         ▷ Θ(1)
9  n → padre ← n → der                                                   ▷ Θ(1)
10 n → der ← (n → der → izq)                                             ▷ Θ(1)
11 if n → der ≠ NULL then                                                ▷ Θ(1)
12 |  (n → der → padre) ← n                                             ▷ Θ(1)
13 end
14 (n → padre → izq) ← n                                                 ▷ Θ(1)
15 RecalcularAltura(n)                                                   ▷ Θ(1)
16 RecalcularAltura(n → padre)                                           ▷ Θ(1)
```

**Complejidad:** $\Theta(1)$

iRotarADerecha(**in** $n$ : `puntero(nodo)`)

```
1  if n → padre ≠ NULL then                                   ▷ Θ(1)
2  |  if n → padre → izq = n then                             ▷ Θ(1)
3  |  |  (n → padre → izq) ← n → izq                          ▷ Θ(1)
4  |  else
5  |  |  (n → padre → der) ← n → izq                          ▷ Θ(1)
6  |  end
7  end
8  (n → izq → padre) ← n → padre                              ▷ Θ(1)
9  n → padre ← n → izq                                        ▷ Θ(1)
10 n → izq ← (n → izq → der)                                  ▷ Θ(1)
11 if n → izq ≠ NULL then                                     ▷ Θ(1)
12 |  (n → izq → padre) ← n                                   ▷ Θ(1)
13 end
14 (n → padre → der) ← n                                      ▷ Θ(1)
15 RecalcularAltura(n)                                        ▷ Θ(1)
16 RecalcularAltura(n → padre)                                ▷ Θ(1)
```

**Complejidad:** $\Theta(1)$

iRebalancearArbol(**in** $n$ : `puntero(nodo)`)

```
1  puntero(nodo) p ← n                                        ▷ Θ(1)
2  while p ≠ NULL do  ▷ Θ(log(n)) iteraciones
3  |  RecalcularAltura(p)                                     ▷ Θ(1)
4  |  int fdb1 ← FDB(p)                                       ▷ Θ(1)
5  |  if fdb1 = 2 then                                        ▷ Θ(1)
6  |  |  puntero(nodo) q ← (p → der)                         ▷ Θ(1)
7  |  |  int fdb2 ← FDB(q)                                   ▷ Θ(1)
8  |  |  if fdb2 = 1 ∨ fdb2 = 0 then                         ▷ Θ(1)
9  |  |  |  RotarAIzquierda(p)                               ▷ Θ(1)
10 |  |  |  p ← q                                            ▷ Θ(1)
11 |  |  else if fbd2 = −1 then                              ▷ Θ(1)
12 |  |  |  RotarADerecha(q)                                 ▷ Θ(1)
13 |  |  |  RotarAIzquierda(p)                               ▷ Θ(1)
14 |  |  |  p ← (q → padre)                                  ▷ Θ(1)
15 |  |  end
16 |  else if fdb1 = −2 then
17 |  |  puntero(nodo) q ← (p → izq)                         ▷ Θ(1)
18 |  |  int fdb2 ← FDB(q)                                   ▷ Θ(1)
19 |  |  if fdb2 = −1 ∨ fdb2 = 0 then                        ▷ Θ(1)
20 |  |  |  RotarADerecha(p)                                 ▷ Θ(1)
21 |  |  |  p ← q                                            ▷ Θ(1)
22 |  |  else if fbd2 = 1 then                               ▷ Θ(1)
23 |  |  |  RotarAIzquierda(q)                               ▷ Θ(1)
24 |  |  |  RotarADerecha(p)                                 ▷ Θ(1)
25 |  |  |  p ← (q → padre)                                  ▷ Θ(1)
26 |  |  end
27 |  end
28 |  p ← (p → padre)                                         ▷ Θ(1)
29 end
```

**Complejidad:** $\Theta(\log(n))$

# 2. Módulo Heap($\alpha$)

## Interfaz

**parámetros formales**

**géneros** $\alpha$

**función** $\bullet < \bullet$(**in** $a_1 : \alpha$, **in** $a_a : \alpha$) $\rightarrow res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} (a_1 \leq a_2)$}
**Complejidad:** $\Theta(compare(a_1, a_2))$
**Descripción:** función de comparación de menor de $\alpha$.

**función** COPIAR(**in** $a : \alpha$) $\rightarrow res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} a$}
**Complejidad:** $\Theta(copy(a))$
**Descripción:** función de copia de $\alpha$

**se explica con**: COLAPRIOR($\alpha$)

**géneros**: heap($\alpha$)

### Operaciones del heap

VACÍO() $\rightarrow res$ : `heap(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ vacío}
**Complejidad:** $\Theta(1)$
**Descripción:** Constructor por defecto de heap($\alpha$)

ENCOLAR(**in/out** $h$ : `heap(`$\alpha$`)`, **in** $a : \alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ {}
**Complejidad:** $h =_{\text{obs}} h_0$ $h =_{\text{obs}}$ encolar($a$, $h_0$) [$\Theta(log(h.longitud))$] [Agrega un elemento a la cola de prioridades]

VACÍO?(**in** $h$ : `heap(`$\alpha$`)`) $\rightarrow res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$h =_{\text{obs}}$ vacío()}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve true si y solo si h es un heap vacío

PRÓXIMO(**in** $h$ : `heap(`$alpha$`)`) $\rightarrow res$ : $\alpha$
**Pre** $\equiv$ {($\not\exists$vacía?(h))}
**Post** $\equiv$ {$res =_{\text{obs}}$ próximo (h)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve el próximo elemento en el heap

DESENCOLAR(**in/out** $h$ : `heap(`$alpha$`)`)
**Pre** $\equiv$ {($\not\exists$vacía?(h))}
**Post** $\equiv$ {$res =_{\text{obs}}$ desencolar(h)}
**Complejidad:** $\Theta(log(h.longitud))$
**Descripción:** Elimina el próximo elemento en el heap

DESENCOLAR2(**in/out** $h$ : `heap(`$alpha$`)`) $\rightarrow res$ : $\alpha$
**Pre** $\equiv$ {$h =_{\text{obs}} h_0 \wedge$ ($\not\exists$vacía?(h))}
**Post** $\equiv$ {$res =_{\text{obs}}$ próximo($h_0$) $\wedge$ $h =$ desencolar($h_0$)}
**Complejidad:** $\Theta(log(h.longitud))$

**Descripción:** Elimina el próximo elemento en el heap

# Representación

**`heap(α)` se representa con `vector(α)`**

Rep : heap($\alpha$) $\longrightarrow$ bool

Rep($hp$) $\equiv$ true $\Longleftrightarrow$

Abs : vector($\alpha$) $h$ $\longrightarrow$ colaPrior($\alpha$) {Rep($h$)}

Abs($h$) $\equiv$ **if** h.long = 0 **then** vacía() **else** encolar(prim(h), ABS(fin(h))) **fi**

# Algoritmos

iVacío() $\rightarrow$ $res$ : `heap(α)`

```
1 res ← vector<α>
```

iEncolar(**in/out** $hp$: `heap(α)`, **in** $a$: $\alpha$)

```
1 hp.push_back(a)
2 nat i ← hp.longitud - 1
3 nat p ← (i/2) - ((i+1) %2)
4 while hp[p] <hp[i] do
5 |   iSwap(hp[p], hp[i])
6 |   i ← p
7 |   p ← (i/2) - ((i+1) %2)
8 end
```

iVacío?(**in** $hp$: `heap(α)`) $\rightarrow$ $res$ : `bool`

```
1 res ← (hp = vacío())
```

iProximo(**in** $hp$: `heap(α)`) $\rightarrow$ $res$ : $\alpha$

```
1 res ← hp[0]
```

iDesencolar(**in/out** $hp$: `heap`($\alpha$))

```
1  nat i ← hp.longitud - 1
2  iSwap(hp[0], hp[i])
3  hp.pop_back()
4  nat hijo_0 ← 1
5  nat hijo_1 ← 2
6  while ¬estaOrdenado do
7      if hijo_0 <hp.longitud then
8          if hijo_1 <hp.longitud then
9              if hp[hijo_0] ≥ hp[hijo_1] then
10                 if hp[hijo_0] >hp[i] then
11                     iSwap(hp[hijo_0], hp[i])
12                     i ← hijo_0
13                 else
14                     estaOrdenado ← True
15                 end
16             else
17                 if hp[hijo_1] >hp[i] then
18                     iSwap(hp[hijo_1], hp[i])
19                     i ← hijo_1
20                 else
21                     estaOrdenado ← True
22                 end
23             end
24         else
25             if hp[hijo_0] >hp[i] then
26                 iSwap(hp[hijo_0], hp[i])
27                 i ← hijo_0
28             else
29                 estaOrdenado ← True
30             end
31         end
32     else
33         estaOrdenado ← True
34     end
35     hijo_0 ← 2xi+1
36     hijo_1 ← 2xi+2
37 end
```

iDesencolar2(**in/out** $hp$: heap($\alpha$)) $\rightarrow$ $res$ : $\alpha$

```
1  nat i ← hp.longitud - 1
2  iSwap(hp[0], hp[i])
3  res ← hp.pop_back()
4  nat hijo_0 ← 1
5  nat hijo_1 ← 2
6  while ¬estaOrdenado do
7      if hijo_0 <hp.longitud then
8          if hijo_1 <hp.longitud then
9              if hp[hijo_0] ≥ hp[hijo_1] then
10                 if hp[hijo_0] >hp[i] then
11                     iSwap(hp[hijo_0], hp[i])
12                     i ← hijo_0
13                 else
14                     estaOrdenado ← True
15                 end
16             else
17                 if hp[hijo_1] >hp[i] then
18                     iSwap(hp[hijo_1], hp[i])
19                     i ← hijo_1
20                 else
21                     estaOrdenado ← True
22                 end
23             end
24         else
25             if hp[hijo_0] >hp[i] then
26                 iSwap(hp[hijo_0], hp[i])
27                 i ← hijo_0
28             else
29                 estaOrdenado ← True
30             end
31         end
32     else
33         estaOrdenado ← True
34     end
35     hijo_0 ← 2xi+1
36     hijo_1 ← 2xi+2
37 end
```

iSwap(**in/out** $a$: $\alpha$, **in/out** $b$: $\alpha$)

```
1  α c
2  c ← a
3  a ← b
4  b ← c
```

# 3. Módulo Red

## Interfaz

**se explica con**: RED, ITERADOR UNIDIRECCIONAL(COMPU)

**géneros**: `red`, `itRed`

### Operaciones básicas de red

INICIARRED() $\rightarrow$ *res* : `Red`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ iniciarRed()}
**Complejidad:** $\Theta(1234)$
**Descripción:** Genera una nueva red sin ninguna computadora.

AGREGARCOMPU(**in/out** $r$ : `Red`, **in** $c$ : `compu`)
**Pre** $\equiv$ {$r =_{\text{obs}} r_0 \wedge (\forall c'$: compu)($c' \in$ computadoras($r$) $\rightarrow$ ip($c$) $\neq$ ip($c'$))}
**Post** $\equiv$ {$r =_{\text{obs}}$ agregarCompu($r_0$, c)}
**Complejidad:** $\Theta(1324)$
**Descripción:** Agrega una nueva pc a una red.

CONECTAR(**in/out** $r$ : `Red`, **in** $c_0$ : `compu`, **in** $i_0$ : `interfaz`, **in** $c_1$ : `compu`, **in** $i_1$ : `interfaz`) $\rightarrow$ *res* : `Red`
**Pre** $\equiv$ {$r =_{\text{obs}} r_0 \wedge c_1 \in$ computadoras($r$) $\wedge$ $c_2 \in$ computadoras($r$) $\wedge$ ip($c_0$) $\neq$ ip($c_1$) $\wedge$ $\neg$conectadas?($r$, $c_0$ , $c_1$) $\wedge$ $\neg$usaInterfaz?($r$, $c_0$, $i_0$) $\wedge$ $\neg$usaInterfaz?($r$, $c_1$, $i_1$)}
**Post** $\equiv$ {$r =_{\text{obs}}$ conectar($r_0$, $c_0$, $i_0$, $c_1$, $i_1$)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Conecta la pc $c_0$ con la pc $c_1$ a través de las interfaces $i_0$ y $i_1$ respectivamente.

COMPUTADORAS(**in** $r$ : `Red`) $\rightarrow$ *res* : `conj(compu)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ computadoras(r)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve todas las computadoras de la red.

CONECTADAS?(**in** $r$ : `Red`, **in** $c_0$ : `compu`, **in** $c_1$ : `compu`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {$c_0 \in$ computadoras($r$) $\wedge$ $c_1 \in$ computadoras($r$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ conectadas?($r$, $c_0$, $c_1$)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve true si y solo si la pc $c_0$ esta conectada a la pc $c_1$

INTERFAZUSADA(**in** $r$ : `Red`, **in** $c_0$ : `compu`, **in** $c_1$ : `compu`) $\rightarrow$ *res* : `interfaz`
**Pre** $\equiv$ {$c_0 \in$ computadoras($r$) $\wedge$ $c_1 \in$ computadoras($r$) $\wedge_{\text{L}}$ conectadas?($r$, $c_0$, $c_1$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ interfazUsada($r$, $c_0$, $c_1$)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve la interfaz usada por $c_0$ para conectarse a $c_1$

VECINOS(**in** $r$ : `Red`, **in** $c$ : `compu`) $\rightarrow$ *res* : `conj(compu)`
**Pre** $\equiv$ {$c \in$ computadoras($r$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ vecinos($r$, $c$)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve el conjunto de vecinos de la pc $c$

USAINTERFAZ?(**in** $r$ : `Red`, **in** $c$ : `compu`, **in** $i$ : `interfaz`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {$c \in$ computadoras($r$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ usaInterfaz?($r$, $c$, $i$)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve true si y solo si la pc $c$ esta usando la interfaz $i$.

CAMINOSMINIMOS(**in** $r$ : `Red`, **in** $c_0$ : `compu`, **in** $c_1$ : `compu`) $\rightarrow$ *res* : `conj(secu(compu))`
**Pre** $\equiv$ {$c_0 \in$ computadoras($r$) $\wedge$ $c_1 \in$ computadoras($r$)}

**Post** $\equiv \{res =_{\text{obs}} \text{caminosMinimos}(r, c_0, c_1)\}$
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve todos los caminos mńimos posibles entre $c_0$ y $c_1$. De no haber ninguno, devuelve $\emptyset$.

HAYCAMINO?(**in** $r$ : `Red`, **in** $c_0$ : `compu`, **in** $c_1$ : `compu`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv \{c_0 \in \text{computadoras}(r) \land c_1 \in \text{computadoras}(r)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{hayCamino?}(r, c_0, c_1)\}$
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve true si y solo si hay algún camino posible entre $c_0$ y $c_1$.

### Operaciones básicas del iterador de red

CREARIT(**in** $r$ : `Red`) $\rightarrow$ $res$ : `itRed`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{crearItUni}(r.\text{compus})\}$
**Complejidad:** $\Theta(1)$????????.
**Aliasing:** Crea un iterador de red.

HAYSIGUIENTE?(**in** $it$ : `itRed`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{hayMas?}(it)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve true si y solo si $it$ tiene siguiente.

SIGUIENTE(**in** $it$ : `itRed`) $\rightarrow$ $res$ : `compu`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{actual}(it)\}$
**Complejidad:** $\Theta(1)$????????.
**Aliasing:** $res$ se devuelve por referencia

AVANZAR(**in/out** $it$ : `itRed`)
**Pre** $\equiv \{it =_{\text{obs}} it_0 \land \text{haySiguiente?}(it)\}$
**Post** $\equiv \{it =_{\text{obs}} \text{avanzar}(it_0)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Avanza el iterador a la siguiente posicion.

## Representación

**`Red` se representa con `estrRed`**

donde `estrRed` es `tupla`(*compus*: `lista(estrCompu)` , *cantidadCompus*: `nat` )

donde `estrCompu` es `tupla`(*IP*: `string` , *conexiones*: `lista(tupla(interfaz, itLista(estrCompu)))` )

**`itRed` se representa con `itLista(estrCompu)`**

Rep : Red $\longrightarrow$ bool

Rep($e$) $\equiv$ ($\forall c$: compu)(c $\in$ ArmarComputadoras(e.conexiones) $\Rightarrow_{\text{L}}$ $\neg$ Pertenece?(e.conexiones, c, c))$\land$
#ArmarComputadoras(e.conexiones) = e.cantidadCompus $\land$
($\forall c_1$: compu)(($\forall c_2$: compu) ($c_1 \in$ ArmarComputadoras(e.conexiones) $\land$ $c_2 \in$ ArmarComputadoras(e.conexiones) $\Rightarrow_{\text{L}}$ Pertenece?(e.conexiones, $c_1$, $c_2$) $\Leftrightarrow$ Pertenece?(e.conexiones, $c_2$, $c_1$))) $\land$
($\forall c_1$: compu)($c_1 \in$ ArmarComputadoras(e.conexiones) $\Rightarrow_{\text{L}}$ ($\forall c_2$: compu) (Pertenece?(e.conexiones, $c_1$, $c_2$) $\Rightarrow$ $c_2 \in$ ArmarComputadoras(e.conexiones))) $\land$
sinRepetidos(ArmarSecuencia(e.conexiones))

Abs : estrRed $e$ $\longrightarrow$ Red {Rep($e$)}

Abs($e$) $\equiv$ (r: Red | computadoras(r) = ArmarComputadoras(e.conexiones) $\wedge$
($\forall c_1$: compu)(($\forall c_2$: compu) conectados?(r, $c_1$, $c_2$) = Pertenece?(e.conexiones, $c_1$, $c_2$) $\wedge$
InterfazUsada(r, $c_1$, $c_2$) = DevolverInterfaz(e.conexiones, $c_1$, $c_2$)))

Rep : itRed $\longrightarrow$ bool

Rep($it$) $\equiv$ true

Abs : itRed $itl$ $\longrightarrow$ itUni(estrCompu) {Rep($itl$)}

Abs($itl$) $\equiv$ $itr$: itUni(estrCompu)| siguientes($itr$) $=_{\text{obs}}$ armarCompus(siguiente($itl$))

ArmarComputadoras : lista(tupla<string $\times$ lista(tupla<Interfaz $\times$ ItRed>)>) $\longrightarrow$ conj(compu)

ArmarComputadoras($l$) $\equiv$ **if** vacia?(l) **then**
$\emptyset$
**else**
Ag(<$\pi_1$(prim($l$)), GenerarInterfaces($\pi_2$(prim($l$)))>, ArmarComputadoras(fin($l$)))
**fi**

GenerarInterfaces : lista(tupla<Interfaz $\times$ ItLista(estrCompu)>) $\longrightarrow$ conj(Interfaz)

GenerarInterfaces($l$) $\equiv$ **if** vacia?($l$) **then** $\emptyset$ **else** Ag($\pi_1$(prim($l$)), GenerarInterfaces(fin($l$))) **fi**

Pertenece? : lista(tupla<string $\times$ lista(tupla<Interfaz $\times$ ItRed>)>) $l$ $\times$ compu $c_1$ $\times$ compu $c_2$ $\longrightarrow$ bool

Pertenece?($l, c_1, c_2$) $\equiv$ **if** ($\pi_1$(prim($l$) = $\pi_1$($c_1$))) **then**
$\pi_1(c_2) \in$ GenerarCompus($\pi_2$(prim($l$)))
**else**
Pertenece?(fin($l$), $c_1$, $c_2$)
**fi**

GenerarCompus : lista(tupla<Interfaz $\times$ ItLista(estrCompu)>) $\longrightarrow$ conj(string)

GenerarCompus($l$) $\equiv$ **if** vacia?($l$) **then** $\emptyset$ **else** Ag($\pi_1$(siguiente($\pi_2$(prim($l$)))), GenerarCompus(fin($l$))) **fi**

DevolverInterfaz : lista(tupla<string $\times$ lista(tupla<Interfaz $\times$ ItRed>)>) $l$ $\times$ compu $c_1$ $\times$ compu $c_2$ $\longrightarrow$ Interfaz

DevolverInterfaz($l, c_1, c_2$) $\equiv$ **if** ($\pi_1$(prim($l$)) = $\pi_1$($c_1$)) **then**
DevolverInterfazAux($\pi_2$(prim($l$), $c_2$))
**else**
DevolverInterfaz(fin($l$, $c_1$, $c_2$))
**fi**

DevolverInterfazAux : lista(tupla<Interfaz $\times$ ItRed>) $l$ $\times$ compu $c$ $\longrightarrow$ Interfaz

DevolverInterfaz($l, c$) $\equiv$ **if** ($\pi_1(c_2) = \pi_1$(siguiente($\pi_2$(prim($l$))))) **then**
$\pi_1$(prim($l$))
**else**
DevolverInterfazAux(fin($l$, c))
**fi**

armarCompus : lista(estrCompu) $l$ $\longrightarrow$ secu(compu)

armarCompus($es$) $\equiv$ **if** vacía(es) **then** <> **else** armarCompu(prim($es$)) $\bullet$ armarCompus(fin($es$)) **fi**

armarCompu : estrCompu $e$ $\longrightarrow$ compu

armarCompus($e$) $\equiv$ $\langle e$.IP, dame$\Pi_1(e$.conecciones)$\rangle$

dameΠ$_1$ : secu(tupla⟨*inter*:interfaz × *itCompu*:itLista(estrCompu)⟩) $l$ $\longrightarrow$ conj(interfaz)
dameΠ$_1(l)$ $\equiv$ **if** vacía($l$) **then** $\emptyset$ **else** ag($Pi_1$(prim($l$)), dameΠ$_1$(fin($l$))) **fi**

# Algoritmos

**`red` se representa con `estrRed`**

donde `estrRed` es `tupla`(*compus*: `lista(estrCompu)` , *cantidadCompus*: `nat` )

donde `estrCompu` es `tupla`(*IP*: `string` , *conexiones*: `lista(tupla(`*inter*`: interfaz,` *itCompu*`: itLista(estrCompu)))` )

iVacia() → *res* : `estrRed`

```
1 res ← ⟨<>, 0⟩                                                    ▷ Θ(1)
```

iAgregarCompu(**in/out** $r$ : `estrRed`, **in** $c$ : `estrCompu`)

```
1 agregarAtras(r.compus, ⟨c.IP, iArmarLista(c.interfaces)⟩)
2 r.cantidadCompus ← r.cantidadCompus + 1                          ▷ Θ(1)
```

iArmarLista(**in** *ci* : `conj(interfaz)`) → *res* : `lista(⟨Interfaz, itLista(estrComp)⟩)`

```
1 res ← vacia()
2 itConj(interfaz) it ← crearIt(ci)
3 while haySiguiente(it) do
4 |   agregarAtras(res, ⟨siguiente(it), NULL⟩)
5 |   avanzar(it)
6 end
```

iConectar(**in/out** $r$ : `estrRed`), **in** $c_0$ : `estrCompu`, **in** $i_0$ : `interfaz`, **in** $c_1$ : `estrCompu`, **in** $i_1$ : `interfaz`)

```
1  itLista(estrComp) it_0 ← crearIt(r.compus)
2  itLista(estrComp) it_1 ← crearIt(r.compus)
3  while siguiente(it_0.IP ≠ c_0.IP) do
4  |   avanzar(it_0)
5  end
6  while siguiente(it_1.IP ≠ c_1.IP) do
7  |   avanzar(it_1)
8  end
9  itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it_2 ← crearIt(siguiente(it_0.conexiones))
10 itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it_3 ← crearIt(siguiente(it_1.conexiones))
11 while siguiente(it_2.inter ≠ i_0) do
12 |   avanzar(it_2)
13 end
14 while siguiente(it_3.inter ≠ i_1) do
15 |   avanzar(it_3)
16 end
17 siguiente(it_2).itCompu ← i_0 siguiente(it_3).itCompu ← i_1
```

iCrearConjuntoDeInterfaces(**in** $l$: `lista(tupla(interfaz, itLista(estrCompu)))`) $\rightarrow$ $res$ : `conj(estrInterfaz)`

```
1 nat n ← 0
2 res ← vacio()
3 while n <longitud(l) do
4 |  agregar(res, Π1(l[n]))
5 |  n ← n + 1
6 end
```

iConectadas?(**in** $r$ : `estrRed`, **in** $c_0$ : `estrCompu`, **in** $c_1$ : `estrCompu`) $\rightarrow$ $res$ : `bool`

```
1 itLista(estrComp) it0 ← crearIt(r.compus)
2 while siguiente(it0).IP ≠ c0.IP do
3 |  avanzar(it0)
4 end
5 itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it1 ← crearIt(siguiente(it0.conexiones))
6 while haySiguiente(it1) ∧L siguiente(siguiente(it1).itCompu).IP ≠ c1.IP do
7 |  avanzar(it1)
8 end
9 res ← (siguiente(siguiente(it1).itCompu).IP = c1.IP)
```

iInterfazUsada(**in** $r$ : `estrRed`, **in** $c_0$ : `estrCompu`, **in** $c_1$ : `estrCompu`) $\rightarrow$ $res$ : `interfaz`

```
1 itLista(estrComp) it0 ← crearIt(r.compus)
2 while siguiente(it0).IP ≠ c0.IP do
3 |  avanzar(it0)
4 end
5 itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it1 ← crearIt(siguiente(it0.conexiones))
6 while haySiguiente(it1) ∧L siguiente(siguiente(it1).itCompu).IP ≠ c1.IP do
7 |  avanzar(it1)
8 end
9 res ← siguiente(it1).inter
```

iVecinos(**in** $r$ : `estrRed`, **in** $c$: `estrCompu`) $\rightarrow$ $res$ : `conj(compu)`

```
1  res ← vacio()
2  itLista(estrComp) it0 ← crearIt(r.compus)
3  while siguiente(it0).IP ≠ c.IP do
4  |  avanzar(it0)
5  end
6  itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it1 ← crearIt(siguiente(it0.conexiones))
7  while haySiguiente?(it1) do
8  |  if haySiguiente?(siguiente(it1).itCompu) then
9  |  |  agregar(res, siguiente(siguiente(it1).itCompu))
10 |  end
11 |  avanzar(it1)
12 end
```

iUsaInterfaz?(**in** $r$ : `estrRed`, **in** $c$ : `estrCompu`, **in** $i$ : `interfaz`) → $res$ : `bool`

```
1  itLista(estrComp) it_0 ← crearIt(r.compus)
2  while siguiente(it_0).IP ≠ c.IP do
3  |  avanzar(it_0)
4  end
5  itLista(tupla⟨interfaz, itLista(estrCompu)⟩) it_1 ← crearIt(siguiente(it_0.conexiones))
6  while siguiente(it_1).inter ≠ i do
7  |  avanzar(it_1)
8  end
9  res ← haySiguiente?(siguiente(it_1).itCompu)
```

iCaminosMinimos(**in** $r$ : `estrRed`, **in** $c_0$ : `estrCompu`, **in** $c_1$ : `estrCompu`) → $res$ : `conj(lista(estrCompu))`

```
1  res ← vacio()
2  if pertenece(c_1, vecinos(r, c_1)) then
3  |  agregar(res, agregarAtras(agregarAtras(<>,c_0), c_1))
4  else
5  |  res ← dameMinimos(iCaminos(r, c_0, c_1, agregarAtras(<>,c_0), pasarConjASecu(vecinos(r, c_0))))
6  end
```

iHayCaminos?(**in** $r$ : `estrRed`, **in** $c_0$ : `estrCompu`, **in** $c_1$ : `estrCompu`) → $res$ : `bool`

```
1  res ← esVacio?(iCaminosMinimos(r, c_0, c_1))
```

iCaminos(**in** $r$ : `estrRed`, **in** $c_0$ : `estrCompu`, **in** $c_1$ : `estrCompu`, **in** $l$ : `lista(estrCompu)`, **in** $vec$ : `lista(estrCompu)`) → $res$ : `conj(lista(estrCompu))`

```
1  if vacia?(vec) then
2  |  res ← vacia()
3  else
4  |  if último(l) = c_1 then
5  |  |  res ← agregar(l, vacia())
6  |  else
7  |  |  if ¢stá?(primero(vec, l)) then
8  |  |  |  res ← unión(caminos(r, c_0, c_1, agregarAtras(l, primero(vec)), iVecinos(r, primeros(vec))),
   |  |  |  caminos(r, c_0, c_1, l, fin(vec)))
9  |  |  else
10 |  |  |  res ← caminos(r, c_0, c_1, l, fin(vec))
11 |  |  end
12 |  end
13 end
```

iComputadoras(**in** $r$ : `estrRed`) → $res$ : `conj(estrCompu)`

```
1  res ← vacio()
2  itRed it ← crearItRed()
3  while haySiguiente?(it) do
4  |  agregar(res, siguiente(it))
5  |  avanzar(it)
6  end
```

iHaySiguiente?(**in** $it$ : `itLista(estrCompu)`) → $res$ : `bool`

```
1  res ← haySiguiente?(it)
```

iSiguiente(**in** $it$ : `itLista(estrCompu)`) $\rightarrow$ $res$ : `compu`

```
1 estrCompu e ← siguiente(it)
2 res.IP ← e.IP
3 conoj(interfaz) interfaces ← vacío()
4 itLista(tupla⟨ inter: interfaz, itCompu: itLista(estrCompu)⟩) itInterfaces ←
  crearIt(e.conexiones)
5 while haySiguiente?(itInterfaces) do
6 |   agregar(interfaces, siguiente(itInterfaces).inter)
7 |   avanzar(itInterfaces)
8 end
9 res.Interfaces ← e.Interfaces
```

iCrearIt(**in** $e$ : `estrRed`)

```
1 res ← crearIt(e.compus)
```

iAvanzar(**in**/**out** $it$ : `itLista(estrCompu)`)

```
1 it ← avanzar(it)
```

# 4. Módulo DCNet

## Interfaz

**se explica con**: DCNet

**géneros**: `dcnet`

### Operaciones básicas de lista

INICIARDCNET(**in** $r$ : `Red`) $\rightarrow$ $res$ : `DCNet`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ iniciarDCNet(r)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Genera un nuevo DCNet sin paquetes. "vacía"

CREARPAQUETE(**in/out** $D$ : `DCNet`, **in** $p$ : `paquete`)
**Pre** $\equiv$ {D $=_{obs}$ $D_0$ $\wedge$ ($\nexists p'$: paquete)(paqueteEnTransito?(D,p') $\wedge$ id(p') = id(p)) $\wedge$ origen(p) $\in$ computadoras(red(D)) $\wedge_{\text{L}}$ destino(p) $\in$ computadoras(red(D)) $\wedge_{\text{L}}$ hayCamino?(red(D), origen(p), destino(p)))}
**Post** $\equiv$ {D $=_{obs}$ crearPaquete($D_0$, p)}
**Complejidad:** $\Theta(l + log(k))$
**Descripción:** Crea un nuevo paquete que no exite en el DCNet anterior.

AVANZARSEGUNDO(**in/out** $D$ : `DCNet`)
**Pre** $\equiv$ {D $=_{obs}$ $D_0$}
**Post** $\equiv$ {D $=_{obs}$ avanzarSegundo($D_0$)}
**Complejidad:** $\Theta(n.(L + log(n) + log(k)))$
**Descripción:** Avanza un segundo en el DCNet, moviendo todos los paquetes correspondientes.

RED(**in** $D$ : `DCNet`) $\rightarrow$ $res$ : `red`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ red(D)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve la red donde esta funcionando el DCNet.

CAMINORECORIDO(**in** $D$ : `DCNet`, **in** $p$ : `paquete`) $\rightarrow$ $res$ : `secu(compu)`
**Pre** $\equiv$ {paqueteEnTrancito?(D, p)}
**Post** $\equiv$ {res $=_{obs}$ caminoRecorrido(D, p)}
**Complejidad:** $\Theta(n.log(max(n, k)))$
**Descripción:** Devuelve la secuencia que contiene de forma ordenada todas las computadoras por las que fue pasando.

CANTIDADENVIADOS(**in** $D$ : `DCNet`, **in** $c$ : `compu`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {c $\in$ computadoras(red(D))}
**Post** $\equiv$ {res $=_{obs}$ cantidadEnviados(D, c)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve la cantidad de paquetes que envió la computadora "c".

ENESPERA(**in** $D$ : `DCNet`, **in** $c$ : `compu`) $\rightarrow$ $res$ : `conj(paquete)`
**Pre** $\equiv$ {c $\in$ computadoras(red(D))}
**Post** $\equiv$ {res $=_{obs}$ enEspera(D, c)}
**Complejidad:** $\Theta(L)$
**Descripción:** Devuelve los paquetes que tiene en espera la compu "c".

PAQUETESENTRÁNSITO?(**in** $D$ : `DCNet`, **in** $p$ : `paquete`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ paquetesEnTrancito?(D, p)}
**Complejidad:** $\Theta(1234)$
**Descripción:** Devuelve "True" si y solo si el paquete esta en los paquetes en espera de alguna computadora.

LAQUEMÁSENVIÓ(**in** $D$ : `DCNet`) $\rightarrow$ $res$ : `compu`

**Pre** ≡ {true}
**Post** ≡ {res =$_{obs}$ laQueMásEnvió(D)}
**Complejidad:** Θ(1)
**Descripción:** Devuelve una de las computadoras con mas paquetes enviados

# Representación

**`dcnet` se representa con `estrDCNet`**

donde `estrDCNet` es `tupla`(*red*: `red` , *IDsCompusPorIP*: `dicc_trie(string, nat)` , *siguientesCompus*: `ad(ad(nat))` , *paquetesEnEspera*: `ad(tupla(enConjunto: conj(paquete), porID: dicc_AVL(ID, tupla(iPaquete: itConj(paquete),codOrigen: nat, codDestino: nat), porPrioridad: heap(tupla(prioridad, itConjunto(paquete))))))` , *#PaqEnviados*: `ad(nat)` , *laQueMásEnvió*: `itRed` , *IPsCompuPorID*: `ad(itRed)` )

Rep : e $\longrightarrow$ bool

Rep($l$) ≡ true $\iff$ ($\forall e$: estr)(
(tam(e.paquetesEnEspera) = tam(e.IPcompusXID) = tam(e.siguienteCompu) = tam (e.cantPaquetesEnviados) = #(computaodras(e.estrRed))) ∧ ($\forall n$: nat)(definido?(e.siguienteCompu,n) $\Rightarrow_L$ tam(e.siguienteCompu[n]) = #(computadoras(e.estrRed))) ∧
Maximo(e.cantPaqEnviados) = e.cantidadEnviados[obtener($\pi_1$(siguiente(e.laQueMásEnvió)),e.IDcompusPorIP)]
∧ ($\forall c$: compu)(c ∈ computadoras(e.estRed) $\Rightarrow_L$ obtener($\pi_1$(c),e.IDcompusPorIP) <#(computadoras(e.estrRed))) ∧ (($\forall c_1$, $c_2$: compu)(($c_1$ ∈ computadoras(e.estRed) ∧ ($c_2$ ∈ computadoras(e.estrRed)) ∧ ($c_1 \neq c_2$)) $\Rightarrow_L$ ((obtener($\pi_1(c_1)$, e.IDcompusPorIP) ≠ obtener($\pi_1(c_2)$, e.IDcompusPorIP))))) ∧
(dameNombres(computadoras(e.estrRed)) = claus(IDcompusPorIP)) ∧
($\forall L$: nat)( 0 ≤ L <tam(e.paqEnEspera) $\Rightarrow_L$ (
($\forall$ $it_1$: ItConj(paquete)) $it$ ∈ dame$\pi_1$(juntarSignificado($\pi_2$(e.paquetesEnEspera[L]))) $\Rightarrow_L$ haySiguiente?($it_1$)
∧ ($\forall$ $it_2$: ITconj(pauqete)) $it_2$ ∈ dame$pi_2$(juntarColaEnConjunto($\pi_3$(e.paquetesEnEspera[L]))) $\Rightarrow_L$ haySiguiente?($it_2$) $\wedge_L$
($\forall i$: ItConj(paquete))(siguiente(i) ∈ (dameSiguientes(dame$\pi_1$($juntarSignificados_1$($\pi_2$(e.paquetesEnEspera[L])))= siguiente(i) ∈ $\pi_1$(e.paquetesEnEspera[L])))) ∧ ($\forall c$: ID)(c∈ claves($\pi_2$(e.paquetesEnEspera[L])) $\Rightarrow_L$ $\pi_1$(siguiente($\pi_1$(obtener($\pi_2$(c), e.paquetesEnEspera[L])))) = c) ∧ ($\forall$ $it$:ItConj(paquete)) siguiente($it$) ∈ dameSiguientes(dame$\pi_2$(juntarColaPrioriEnConj($\pi_3$(e.paquetesEnEspera[L])))) $\Rightarrow_L$ siguiente($it$) ∈ $\pi_1$(e.paquetesEnEspera[L]) ($\forall$ $t$: tupla <prioridad,ItConj(paquete)>) $t$ ∈ juntarColaPrioriEnConj($\pi_3$(e.paquetesEnEspera[L])) → siguiente($\pi_2(t)$).prioridad = $\pi_1(t)$
($\forall x$, $z$: nat)((0 ≤ x <tam(e.paquetesEnEspera) ∧ 0 ≤ z <tam(e.paquetesEnEspera) ∧ x ≠ z) $\Rightarrow_L$ ($\pi_1$(e.paquetesEnEspera[x]) ∩ $\pi_2$(e.paquetesEnEspera[z])) = ∅) ∧
($\forall i$: nat)(0 ≤ i <#(computadoras(e.estrRed)) ∧ obtener($\pi_1$(siguiente(e.IPcompusPorID[i])), e.IDcompusPorID) = i) ∧
($\forall n$, $m$: nat)(0 ≤ n <#(computadoras(e.estrRed)) ∧ 0 ≤ n <#(computadoras(e.estrRed)) $\Rightarrow_L$ ($\exists$ $x$: (secu(compu))) $x$ ∈ caminosminimos(e.estrRed, siguiente(e.IPcompusPorID[n]), siguiente(e.IPcompusPorID[m])) ∧ prim($x$) = e.siguienteCompu[n][m]
($\forall i$: nat)(0 ≤ i ≤ #computadoras(e.estrRed) $\Rightarrow_L$ siguiente(e.IPcompusPorID[i]) ∈ compitadoras(e.estrRed))
∧
($\forall x$, $y$: nat)((0 ≤ x <#computadoras(e.estrRed) ∧ 0 ≤ y <#computadoras(e.estrRed) ∧ x ≠ y) $\Rightarrow_L$ (siguiente(e.IPcompusPorID[x]) ≠ siguiente(e.IPcompusPorID[y])))) ∧
#($\pi_1$(e.paquetesEnEspera)) = #claves($\pi_2$(e.paquetesEnEspera)) ∧ = #(juntarSecuenciasEnConj(juntarSignificad
∧
($\forall$ $it_1$: ItConj(paquete)) ($\forall$ $it_2$: ItConj(paquete)) $it_1$ ∈ dame$\pi_2$(juntarColaPriorEnConj($\pi_3$(e.paquetesEnEspera[L])
∧ $it_2$ ∈ dame$\pi_2$(juntarColaPriorEnConj($\pi_3$(e.paquetesEnEspera[L]))) $it_1 \neq it_2$ $\Rightarrow_L$ siguiente($it_1$) ≠ siguiente($it_2$)

Abs : estrDCNet $e$ $\longrightarrow$ DCNet {Rep($e$)}

Abs($e$) $\equiv$ red($d$) = e.estrRed $\wedge$

($\forall p$: paquete) paqueteEnTránsito?($d$, $p$) $\Rightarrow_{\text{L}}$ caminoRecorrido($d$, $p$) = caminoDelPaquete(e.siguienteCompu, e.IPsCompusPorID, e.PaquetesEnEspera, $p$) $\wedge$

($\forall c$: compu) $c \in$ computadoras(red($d$)) $\Rightarrow_{\text{L}}$ cantidadEnviados($d$, $c$) = e.#PaqEnviados[obtener($c$, e.IDsCompusPorIP)] $\wedge$

($\forall c$: compu) $c \in$ computadoras(red($d$)) $\Rightarrow_{\text{L}}$ enEspera($d$, $c$) = enConjunto(e.paquetesEnEspera[obtener($c$, e.IDsCompusPorIP)])

caminoDelPaquete : ad(ad(nat)) $\times$ ad(ItRed) $\times$ ad(tupla(enConjunto:conj(paquete) $\times$ porID:dicc$_{\text{AVL}}$(ID tupla(iPaquete:i

caminoDelPaquete($t, CsxID, ps, p, Psr$) $\equiv$ **if** def?(ID($p$), porID($ps$[0])) **then**
caminoDelPaquete$_{\text{aux}}$($t$, $CsxID$, $ps$, $p$, codOrigen(obtener(ID($p$), porId($ps$[0]))), codDestino(obtener(ID($p$), porId($ps$[0]))))
**else**
caminoDelPaquete($t$, $CsxID$, $ps$, $p$, finAd(psr))
**fi**

caminoDelPaquete$_{\text{aux}}$ : ad(ad(nat)) $\times$ ad(ItRed) $\times$ ad(tupla(enConjunto:conj(paquete) porID:dicc$_{\text{AVL}}$ $\times$ tupla(iPaquete:i

caminoDelPaquete$_{\text{aux}}$($t, CsxID, ps, p, Psr, compuActual, d$) $\equiv$ **if** def?(ID($p$), porID($ps$[$compuActual$])) **then**
siguiente($CsxID$[$compuActual$]) $\bullet$ <>
**else**
siguiente($CsxID$[$compuActual$]) $\bullet$
caminoDelPaquete$_{\text{aux}}$($t$, $CsxID$, $ps$, $p$, $Psr$, $t$[$compuActual$][$d$], $d$)
**fi**

finAd : ad(tupla(enConjunto:conj(paquete) $\times$ porID:dicc$_{\text{AVL}}$(ID tupla(iPaquete:itConj(paquete) codOrigen:nat codDestino

finAd($a$) $\equiv$ **if** (tam(a) $\leq$ 1) **then** crearArreglo(0) **else** finAd$_{\text{aux}}$(a, crearArreglo(tam(a)-1), tam(a)-2) **fi**

finAd$_{\text{aux}}$ : ad(tupla(enConjunto:conj(paquete) $\times$ porID:dicc$_{\text{AVL}}$(ID tupla(iPaquete:itConj(paquete) codOrigen:nat codDest
{tam(a)>1 $\wedge$ tam(b)>0}

finAd$_{\text{aux}}$($a, b, n$) $\equiv$ **if** ($n$=0) **then** $b$[0] $\leftarrow$ $a$[1] **else** finAd$_{\text{aux}}$($a$, $b$[n]$\leftarrow$ $a$[$n$+1], $n$ -1) **fi**

juntarColaPriorEnConj : colaPrior(tupla<prioridad,ItConj(paquete)>) $\longrightarrow$ conj(tupla<prioridad, ItConj(paquete)>)

juntarColaPriorEnConj(c) $\equiv$ **if** vacia?(c) **then** $\emptyset$ **else** ag(proximo(c), juntarColaPriorEnConj(desencolar(c))) **fi**

dame$\pi_2$ : conj(tupla<prioridad $\times$ ItConj(paquete)>) $\longrightarrow$ conj(ItConj(paquete))

dame$\pi_2$(c) $\equiv$ **if** $\emptyset$?(c) **then** $\emptyset$ **else** ag($\pi_2$(dameuno(c)), dame$\pi_2$(sinuno(c))) **fi**

dameSiguientes : conj(ItConj(paquete)) $\longrightarrow$ conj(paquete) {restricciones}

dameSiguientes(c) ≡ **if** ∅?(c) **then** ∅ **else** ag(siguiente(dameuno(c)), dameSiguientes(sinuno(c))) **fi**

# Algoritmos

iIniciarDCNet(**in** $r$ : `Red`) → $res$ : `DCNet`

```
1  res ← iCopiar(r)
2  res.#PaqEnviados ← crearArreglo(cantCompus(res.red))
3  res.IPsCompuPorID ← crearArreglo(cantCompus(res.red))
4  res.siguientesCompus ← crearArreglo(cantCompus(res.red))
5  res.paquetesEnEspera ← crearArreglo(cantCompus(res.red))
6  itRed it_0 ← crearItRed(res.red)
7  nat j ← 0
8  while j <iCardinal(iComputadoras(res.red)) do
9  |  res.siguientesCompus[j] ← crearArreglo(cantCompus(res.red))
10 |  res.#PaqEnviados[j] ← 0
11 |  res.paquetesEnEspera[j] ← ⟨vacío(),vacío(),vacío()⟩
12 |  definir(siguiente(it_0).IP, j, res.IDsCompusPorIP)
13 |  res.IPsCompusPorID[j] ← it_0
14 |  j ← j + 1
15 |  avanzar(it_0)
16 end
17 nat k ← 0
18 j ← 0
19 while j <iCardinal(iComputadoras(res.red)) do
20 |  while k <iCardinal(iComputadoras(res.red)) do
21 |  |  if conectadas?(res.red, siguiente(res.IPsCompusPorID[j]), siguiente(res.IPsCompusPorID[k])) then
22 |  |  |  itConj it_0 ← crearIt(caminosMinimos(res.red, siguiente(res.IPsCompusPorID[j]),
   |  |  |  siguiente(res.IPsCompusPorID[k])))
23 |  |  |  res.siguientesCompus[j][k] ← prim(fin(siguiente(it_1)))
24 |  |  end
25 |  |  k ← k + 1
26 |  end
27 |  j ← j + 1
28 end
```

iCrearPaquete(**in/out** $d$ : `DCNet`,**in** $p$ : `paquete`)

```
1 nat o ← iObtener(p.origen, d.IDsCompusPorIP)
2 nat dest ← iObtener(p.destino, d.IDsCompusPorIP)
3 it ← CrearIt((d.paquetesEnEspera[o]).enConjunto)
4 it ← iAgregar((d.paquetesEnEspeta[o]).enConjunto, p)
5 iDefinir(d.paquetesEnEspera[o].porID, p.ID, ⟨ it, o, dest, ⟩)
6 iAgregarHeap(d.paquetesEnEspera[o].porPrioridad, p.prioridad, it)
```

iAvanzarSegundo(**in/out** $d$: `DCNet`)

```
1  nat j ← 0
2  nat o
3  nat dest
4  paquete paq
5  while j <iCardinal(iComputadoras(d.red)) do
6  |  if !(iEsVacio?(d.paquetesEnEspera[j]).enConjunto) then
7  |  |  paq ← iSiguiente(iDameElDeMayorPrioridad((d.paquetesEnEspera[j]).porPrioridad))
8  |  |  iBorrarElDeMaxPrioridad(d.paquetesEnEspera[j].porPrioridad)
9  |  |  o ← (iObtener((d.paquetesEnEspera[j]).porID, paq.ID)).codOrigen
10 |  |  dest ← (iObtener((d.paquetesEnEspera[j]).porID, paq.ID)).codDestino
11 |  |  iBorrar((d.paquetesEnEspera[j]).porID, paq.ID)
12 |  |  d.#paqEnviados[j]++
13 |  |  if !(d.siquienteCompu[j][dest] = dest) then
14 |  |  |  it ← crearIt(d.paquetesEnEspera[d.siguienteCompu[j][dest]])
15 |  |  |  it ← iAgregar((d.paquetesEnEspera[d.siguienteCompu[j][dest]]).enConjunto, p)
16 |  |  |  iDefinir(d.paquetesEnEspera[d.siguienteCompu[j][dest]].porID, p.ID, ⟨ it,
   |  |  |  d.siguienteCompu[j][dest], dest, ⟩)
17 |  |  |  iAgregarHeap(d.paquetesEnEspera[d.siguienteCompu[j][dest]].porPrioridad, paq.prioridad, it)
18 |  |  end
19 |  end
20 end
21 nat k ← 0
22 nat h ← 0
23 if iCardinal(iComputadoras(d.red))>0 then
24 |  while k<iCardinal(iComputadoras(d.red))>0 do
25 |  |  if d.#paqEnviados[k]>d.#paqEnviados[h] then
26 |  |  |  h ← k
27 |  |  |  k++
28 |  |  end
29 |  end
30 end
31 d.laQueMásEnvió ← d.IPsCompusPorID[h]
```

iCaminoRecorrido(**in** $d$: `DCNet`, **in** $p$: `paquete`) → $res$ : `lista(compu)`

```
1  nat j ← 0
2  res ← iVacia()
3  while !(iDefinido?((d.paquetesEnEspera[j]).porID), p.ID) do
4  |  j++
5  end
6  nat o
7  o ← (iObtener((d.paquetesEnEspera[j]).porID, p.ID)).codOrigen
8  nat dest
9  dest ← (iObtener((d.paquetesEnEspera[o]).porID, p.ID)).codDestino
10 while !(iDefinido?((d.paquetesEnEspera[o]).porID), p.ID) do
11 |  iAgregarAtras(res, siguiente(d.IPsCompusPorID[0]))
12 |  o ← d.siguientesCompus[o][dest]
13 end
14 iAgregarAtras(res, siguiente(d.IPsCompusPorID[o]))
```

iCantidadEnviados(**in** $d$: `DCNet`, **in** $c$: `compu`) → $res$ : `nat`

```
1  nat i ← iObtener(d.IDsCompusPorIP, c.IP)
2  res ← d.#paqEnviados[i]
```

iEnEspera(**in** $d$: `DCNet`, **in** $c$: `compu`) $\rightarrow$ $res$ : `conj(Paquete)`

```
1 nat i ← iObtener(d.IDsCompusPorIP, c.IP)
2 res ← (d.paquetesEnEspera[i]).enConjunto
```