Índice

1. Workflow

1.1. Interfaz

interfaz: WORKFLOW
usa: NAT, MULTICONJUNTO(α), CONJ(α)
se explica con: WORKFLOW
géneros: WORKFLOW

Operaciones:

tareas(**in** w: workflow) $\rightarrow$ res: conj(tarea)
{P: true }
{Q: res $=_{\text{obs}}$ tareas (w) }

consumo(**in** w: workflow, **in** t: tarea) $\rightarrow$ res: recursos O(1)
{P: t < # (tareas (w)) }
{Q: res $=_{\text{obs}}$ consumo (w,t) }

predecesoras(**in** w: workflow, **in** t: tarea) $\rightarrow$ res: conj(tarea) O(1)
{P: t < #(tareas (w)) }
{Q: res $=_{\text{obs}}$ predecesoras (w,t) }

prioridad(**in** w: workflow, **in** t: tarea) $\rightarrow$ res: nat O(1)
{P: t < #(tareas (w)) }
{Q: res $=_{\text{obs}}$ prioridad (w,t) }

nuevo(**in** p: nat, **in** r: recursos) $\rightarrow$ res: workflow O()
{P: true }
{Q: res $=_{\text{obs}}$ nuevo(p,r)}

agTarea(**inout** w: workflow, **in** p:prioridad, **in** r:recursos, **in** ps:conj(tarea)) O()
{P: w = w_0 $\wedge$ p < prioridad(0,w) $\wedge$ $\emptyset \subset$ ps $\subseteq$ tareas (w) $\wedge$ $\forall$(t:tarea) (t $\in$ tareas (w) $\Rightarrow_{\text{L}}$ p $\neq$ prioridad (t,w)}
{Q: w $=_{\text{obs}}$ agTarea(w_0,p,r,ps))}

finales(**in** w: workflow) $\rightarrow$ res: conj (tarea) O()
{P: true }
{Q: res $=_{\text{obs}}$ finales (w)}

sucesoras(**in** w: workflow, **in** t: tarea) $\rightarrow$ res: conj (tarea) O()
{P: t $\in$ tareas (w) }
{Q: res $=_{\text{obs}}$ sucesoras (t,w) }

cantPredecesoras(**in** w: workflow, **in** t: tarea) $\rightarrow$ res: nat O(1)
{P: t < #(tareas (w)) }
{Q: res $=_{\text{obs}}$ #(predecesoras (w,t)) }

cantTareas(**in** w: workflow) $\rightarrow$ res: nat O(1)
{P: t < #(tareas (w)) }
{Q: res $=_{\text{obs}}$ #(tareas(w)) }

1.2. Estructura

ESTRWORKFLOW SE REPRESENTA CON TUPLA< tareas: arregloDimensionable (<prioridad:nat × predecesoras: conj (tarea) × cantPred: nat × recursos: recursos>) × cantTareas: nat>

1.3. Invariante de Representacion

Rep: $estr\widehat{Workflow} \to$ bool

($\forall$ e : estrWorkflow) Rep(e) = cantTareasCorrecta (e.tareas, e.cantTareas) $\wedge$ primeraCorrecta (e.tareas[0]) $\wedge_L$ prioridadesCorrectas (e.tareas, e.cantTareas, $\emptyset$, 1, prioridad ((e.tareas)[0])) $\wedge$ predecesoresCorrectos (e.tareas, e.cantTareas, 1)

cantTareasCorrecta: ad(<nat, conj(tarea), recursos>) × nat $\to$ bool
cantTareasCorrecta (a, n) = (tam (a) $\geq$ n $\geq$ 1)

primeraCorrecta: tupla(<nat, conj(tarea), nat,recursos>) $\to$ bool
primeraCorrecta (t) = $\emptyset$? predecesoras (t) $\wedge$ cantPred (t) == 0

prioridadesCorrectas: ad(<nat, conj(tarea), recursos>) × nat × conj(nat) × nat × nat $\to$ bool
prioridadesCorrectas (a,cant,c,n,p) = n $\geq$ cant $\vee_L$ (prioridad(a[n]) < p $\wedge$ prioridad(a[n]) $\notin$ c $\wedge_L$ (prioridadesCorrectas(a,cant,Ag(prioridad(a[n]), c), n+1, p)))

predecesoresCorrectos: ad(<nat, conj(tarea), recursos>) × nat × nat $\to$ bool
predecesoresCorrectos (a,cant,n) = n $\geq$ cant $\vee_L$ ((cantPred(a[n]) == long (predecesores(a[n]) $\wedge$ ($\neg$ $\emptyset$?(predecesores(a[n])) $\wedge$ sonMenores(predecesores(a[n]), n)) $\wedge_L$ (predecesoresCorrectos(a,cant, n+1)))))

sonMenores: conj(nat) × nat $\to$ bool
sonMenores (c, n) = $\emptyset$?(c) $\vee_L$ dameUno (c) < n $\wedge$ sonMenores (sinUno(c), n)

1.4. Funcion de abstraccion

Abs : estrWorkflow e $\to$ workflow (Rep(e))

($\forall$ e: estrWorkflow) Abs(e) = w: workflow / e.cantTareas = # tareas(w) $\wedge_L$ ($\forall$ n: nat) n < cantTareas $\Rightarrow_L$ prioridad (e.tareas[n]) == prioridad (w,n) $\wedge$ recursos (e.tareas[n]) == recursos (w,n) $\wedge$ predecesoras (e.tareas[n]) == predecesoras (w,n) $\wedge$ cantPred (e.tareas[n]) == long (predecesoras(w,n))

1.5. Algoritmos

Algoritmo 1

iTareas(in w: estrWorkflow) → *res*: conj(tarea)
    var n: nat = 0   O(1)
    var c: conj (tarea) = vacio   O(1)
    while (n < w.cantTareas)   O(t)
        agregar (w.tareas[n], c)   O(1)
        n++   O(1)
    endWhile   O(1)
    res ← c   O(1)
    devolver res
end Function

Creamos un conjunto vacio en O(1), la condicion del while se evalúa en O(1) porque es un observador de la tupla, dentro del while el agregar tarda O(1) de acuerdo a la implementacion que hicimos en el módulo conjunto, este while se hace n veces, siendo n la cantidad de tareas del workflow, como hace n veces O(1), la operación tarda O(n)

Algoritmo 2

iConsumo(in w: estrWorkflow, in t: tarea) → *res*: recursos
    res ← recursos(w.tareas[t])   O(1)
    devolver res
end Function

Como simplemente tenemos que acceder a un observador de una tupla que se accede sobre una posición del arreglo (lo cual es inmediato) y dicho arreglo es un observador de la tupla del workflow, esto tarda O(1)

Algoritmo 3

iPrioridad(in w: estrWorkflow, in t: tarea) → *res*: nat
    res ← prioridad(w.tareas[t])   O(1)
    devolver res
end Function

Como simplemente tenemos que acceder a un observador de una tupla que se accede sobre una posición del arreglo (lo cual es inmediato) y dicho arreglo es un observador de la tupla del workflow, esto tarda O(1)

Algoritmo 4

iPredecesoras(in w: estrWorkflow, in t: tarea) $\rightarrow$ *res*: conj(tarea)
    res $\leftarrow$ predecesoras(w.tareas[t])   O(1)
    devolver res
end Function

Como simplemente tenemos que acceder a un observador de una tupla que se accede sobre una posición del arreglo (lo cual es inmediato) y dicho arreglo es un observador de la tupla del workflow, esto tarda O(1)

Algoritmo 5

iNuevo(in p: nat, in r: recursos) $\rightarrow$ *res*: workflow
    res.cantTareas = 1   O(1)
    var a: arregloDimensionable (¡nat, conj(tarea), recursos) = crearArreglo (1)   O(1)
    res.tareas = a   O(1)
    prioridad(res.tareas[0]) = p   O(1)
    recursos(res.tareas[0]) = r   O(1)
    predecesoras(res.tareas[0]) = vacio   O(1)
    cantPred(res.tareas[0]) = 0   O(1)
    devolver res
end Function

Crear un arreglo tiene siempre costo constante, después sólo debemos asignar a los observadores de la tupla ya sea los parámetros pasados o en el caso de predecesoras, un conjunto vacio que se crea en tiempo constante (ver módulo conjunto), por lo tanto, el algoritmo cuesta O(1)

Algoritmo 6

iAgTarea(inout w: estrWorkflow, in p:prioridad, in r:recursos, in ps:conj(tarea)) $\rightarrow$ *res*: conj(tarea)
    if (w.cantTareas $>$ 1 $\wedge$ vacio? (predecesoras(w.tareas [cantTareas -1])))
        var dim: nat = w.cantTareas * 2   O(1)
        var a: arregloDimensionable (<nat, conj(tarea), recursos)> = crearArreglo (dim)   O(1)
        var n: nat = 0   O(1)
        while (n $<$ w.cantTareas)
            a[n] = w.tareas[n]   O(1)
            n++   O(1)
        endWhile
        π_1(a[n]) = p   O(1)
        π_2 (a[n]) = ps   O(1)
        π_3 (a[n]) = long (ps)   O(ps)
        π_4 (a[n]) = r   O(1)
        w.cantTareas = dim   O(1)
        w.tareas = a   O(1)
    else
        prioridad (a[n]) = p   O(1)
        recursos (a[n]) = r   O(1)
        predecesoras (a[n]) = ps   O(1)
        cantPred(a[n]) = long(ps)   O(ps)
    endIf
end Function

El primer if lo hacemos en tiempo constante para evitar pagar el O(log n) que tardaría si evaluáramos si n es potencia de dos, costo que no podemos pagar si n no lo fuese, por lo tanto, sólo vemos si el último conjunto de predecesores es vacio, esto se supone que es verdad si todavía no se le agregó una tarea (ya que deben tener predecesores si no es la tarea inicial). En caso de que el if sea falso, simplemente se asignan a la posicion del arreglo los parámetros pasados, esto tiene costo constante ya que es una acceso a un array. En caso que el if sea true, debemos crear un arreglo nuevo (tiempo constante) de tamaño 2 $*$ tamaño anterior, y despues asignarle a las n-1 posiciones previas los valores que tenia el arreglo anterior, cada asignación tarda O(1), por lo tanto en total el while tarda O(n) con n el tamaño del arreglo previo (o sea, la cantidad de tareas a reasignar). Luego asignamos la nueva tarea de la misma manera en O (1).

Por lo tanto, esta operación tarda O(1) si la tarea a agregar no es potencia de 2, y O(n) si lo es

Algoritmo 7

iFinales(in w: estrWorkflow) → *res*: conj(tarea)
    res ← tareas (w)   O(1)
    var cp: conj (tarea) = vacio   O(1)
    var i: nat = 0   O(1)
    var t: tarea = 0   O(1)
    while (i < w.cantTareas)
        union (cp, predecesoras(w[i]))   O(n^2)
        i++   O(1)
    endWhile
    var it: conj(tarea) ← crearIt (cp)
    while (hayMas (it))
        t = actual (it)   O(1)
        sacar (res, t)   O(n)
        borrarActual (it)   O(1)
    endWhile
    devolver res
end Function

La función union de conjunto tarda O(n^2) según el módulo conjunto, esta operación la ejecutamos n veces siendo n la cantidad de tareas en el workflow, por lo tanto ese while tiene costo O(n^3). Para el segundo while, la operación sacar sabemos que tiene costo O(n), como nuevamente se ejecuta n veces, el costo del segundo while es O(n^2). Como lo demás son operaciones que tardan tiempo constante, el costo total es O(n^3) + O(n^2) = O(n^3)

Algoritmo 8

iSucesoras(in w: estrWorkflow, in t: tarea) → *res*: conj(tarea)
    var n: nat = 0   O(1)
    while (n < w.cantTareas)
        if (pertenece (t, predecesoras(w.tareas[n])) O(p)
            agregar (n, res)   O(1)
        n++   O(1)
    endWhile
    devolver res
end Function

En esta operación, recorremos todo el arreglo de tareas, y en cada pasada nos fijamos si la tarea actual pertenece a las predecesoras de la tarea pasada por parámetro. Sabemos que la operación pertenece del conjunto tiene costo O(p), con p la cantidad de elementos del conjunto (en este caso, la cantidad de predecesores). Además, el agregar insume tiempo constante. Por lo tanto, la operación tarda O(n∗p)

Algoritmo 9

iCantTareas(in w: estrWorkflow) → *res*: nat
    res ← w.cantTareas
    devolver res
end Function

En esta operación simplemente devolvemos un observador de la tupla, por lo tanto es O(1)

2. UsoRecursos

2.1. Interfaz

interfaz: UsoRecursos
usa: Nat,Multiconjunto(α),Conj(α)
se explica con: UsoRecursos
géneros: UsoRecursos

Operaciones:

tiposDeRecurso (**in** u: UsoRecursos) $\rightarrow$ res: nat
{P: true }
{Q: res $=_{\text{obs}}$ tiposDeRecursos (u) }

verTotal (**in** u: UsoRecursos, r: recurso) $\rightarrow$ res: nat
{P: r < tiposDeRecurso (u) }
{Q: res $=_{\text{obs}}$ verTotal (u,r) }

disponible (**in** u: UsoRecursos, r: recurso) $\rightarrow$ res: nat
{P: r < tiposDeRecurso (u) }
{Q: res $=_{\text{obs}}$ disponibilidad (u,r) }

generar (**in** rs: recursos) $\rightarrow$ res: usoRecursos
{P: true }
{Q: res $=_{\text{obs}}$ generar(rs) }

nuevoConsumo (**inout** u: UsoRecursos, r: recurso, n: nat)
{P: u = u_0 $\wedge$ r < tiposDeRecurso (u) $\wedge$ n < verTotal (u,r) }
{Q: u $=_{\text{obs}}$ nuevoConsumo (u_0, r, n) }

menorConsumo (**in** u: UsoRecursos) $\rightarrow$ res: recurso
{P: tiposDeRecurso (u) > 0 }
{Q: res $=_{\text{obs}}$ menorConsumo (u) }

actualizarConsumo (**inout** u: UsoRecursos, **in** $mconj$: recursos)
{P: u = u_0 $\wedge$ ($\forall$ r: recurso) r $\in$ mconj $\Rightarrow_{\text{L}}$ (r < tiposDeRecurso $\wedge_{\text{L}}$ # (r, mconj) < verTotal (u, i)) }
{Q: u $=_{\text{obs}}$ actualizarConsumo (u_0, mconj) }

actualizarUso (**inout** u: UsoRecursos, ar: arregloDimensionable (nat))
{P: u = u_0 $\wedge$ tiposDeRecurso (u) == tam (ar) $\wedge$ ($\forall$ i: nat) i < tam (ar) $\Rightarrow_{\text{L}}$ ar[i] < verTotal (u, i) }
{Q: u $=_{\text{obs}}$ actualizarUso (u_0, ar) }

alcanzanLosRecursos (**in** u: UsoRecursos, **in** $mconj$: recursos) $\rightarrow$ res: bool
{P: ($\forall$ r: recurso) r $\in$ mconj $\Rightarrow_{\text{L}}$ r < tiposDeRecurso (u) }
{Q: res $=_{\text{obs}}$ alcanzanLosRecursos (u, mconj) }

multiconjuntizar (**in** u: UsoRecursos) $\rightarrow$ res: recursos
{P: true }
{Q: res $=_{\text{obs}}$ multiconjuntizar (u) }

disponiblesEnMulticonj (**in** u: UsoRecursos) $\rightarrow$ res: recursos
{P: true }
{Q: res $=_{\text{obs}}$ disponiblesEnMulticonj (u) }

2.2. Estructura

ESTRUSORECURSOS SE REPRESENTA TUPLA <status: arregloDimensionable (<total:nat × disponible: nat × orden: nat> × ordenesConsumo: arregloDimensionable (recurso: nat × disponible: nat)>) × cantRecursos: nat>

2.3. Invariante de Representación

Rep: $\widehat{estrUsoRecursos}$ → bool

(∀ e : estrUsoRecursos) Rep(e) = cantidadRecursosCorrecta (u.status, u.ordenesConsumo, u. cantRecursos) $\wedge_L$ disponiblesCorrectos(u.status, 0, u.cantRecursos) ∧ seCorrespondenLosArreglos (u.status, u.ordenesConsumo, 0, u.cantRecursos) ∧ esUnHeap (u.ordenesConsumo, u.cantRecursos, u.cantRecursos)

cantidadRecursosCorrecta: ad(<nat, nat, nat>) × ad(<nat, nat>) × nat → bool
cantTareasCorrecta (ar, ao, n) = (tam (ar) == tam (ao) == n)

disponiblesCorrectos: ad(<nat, nat, nat>) × nat × nat → bool
disponiblesCorrectos (ar, i, n) = i == n $\vee_L$ (total(a[i]) ≥ disponible(a[i]) $\wedge_L$ (disponiblesCorrectos (ar, i+1, n)))

seCorrespondenLosArreglos: ad(<nat, nat, nat>) × ad(<nat, nat>) × nat × nat → bool
seCorrespondenLosArreglos (ar, ao, i, n) = i == n $\vee_L$ (recurso (ao[(orden (ar[i]))]) == i ∧ disponible(ar[i]) == disponible (ao[(orden (ar[i]))]) $\wedge_L$ (disponiblesCorrectos (ar, ao, i+1, n)))

esUnHeap: ad(<nat, nat>) × nat × nat→ bool
esUnHeap (ao, i, n) = **if** i == 0 **then** nodoValido (ao, i, n) **else** nodoValido (ao, i, n) ∧ esUnHeap (ao, i-1, n) **fi**

nodoValido: ad(<nat, nat>) × nat × nat→ bool
nodoValido (ao, i, n) = i ≤ n/2 -1 $\Rightarrow_L$ **if** i ≤ n/2 - 2 **then** disponible(ao[i]) ≥ disponible(ao [i*2+1]) ∧ disponible(ao[i]) ≥ disponible(ao [i*2+2]) **else** disponible(ao[i]) ≥ disponible(ao [i*2+1]) **fi**

2.4. Función de Abstracción

Abs : estrUsoRecursos e → usoRecursos (Rep(e))

(∀ e: estrUsoRecursos) Abs(e) = u: usoRecursos / e.cantRecursos = # tiposDeRecurso(w) $\wedge_L$ (∀ n: nat) n < cantRecursos $\Rightarrow_L$ total (e.status[n]) == verTotal (u,n) ∧ disponible (e.status[n]) == disponible (u,n)

2.5. Algoritmos

Algoritmo 10

iTiposDeRecurso(in u: estrUsoRecursos) → *res*: conj(tarea)
    res ← u.cantRecursos   O(1)
    devolver res
end Function

Acá sólo debemos devolver un observador de la tupla, entonces demora O(1)

Algoritmo 11

iVerTotal(in u: estrUsoRecursos, in r: recurso) → *res*: nat
    res ← total(u.status[r])   O(1)
    devolver res
end Function

Acá sólo debemos acceder a una posición del arreglo de un observador de la tupla, lo cual demora O(1)

Algoritmo 12

iDisponible(in u: estrUsoRecursos, in r: recurso) → *res*: nat
    res ← disponible(u.status[r])   O(1)
    devolver res
end Function

Acá sólo debemos acceder a una posición del arreglo de un observador de la tupla, lo cual demora O(1)

Algoritmo 13

iGenerar(in r: recursos) $\rightarrow$ res: usoRecursos
    var n: nat = cantElemDistintos (n)   O(r)
    var it: recursos $\leftarrow$ crearIt (r)   O(1)
    var ar: arregloDimensionable (nat,nat,nat) = ad(n)   O(1)
    var temp: nat = 0   O(1)
    while (hayMas (it))   O(r)
        temp = actual (it)   O(1)
        total(ar[temp] = cardinal (temp,r))   O(1)
        disponible(ar[temp]) = total(ar[temp])   O(1)
        borrarActual (it)   O(1)
    endWhile   O(1)
    var ao: arregloDimensionable (nat, nat) = ad (n)   O(1)
    i = 0   O(1)
    while (i < n)   O(n)
        recurso (ao [i]) = i   O(1)
        disponible (ao [i]) = disponible (ar [i])   O(1)
        orden (ar [i]) = i   O(1)
        i++   O(1)
    endWhile   O(1)
    i = n/2 - 1   O(1)
    while (i $\geq$ 0)
        heapify (ao, i, ar, n)   O(...)
        i−−   O(1)
    endWhile   O(1)
    res.status $\leftarrow$ ar   O(1)
    res.ordenesConsumo $\leftarrow$ ao   O(1)
    res.cantRecuros $\leftarrow$ n   O(1)
    devolver res
end Function

La creación de las variables insume tiempo constante. Luego, debemos recorrer el multiconjunto, lo cual tarda O(r) siendo r la cantidad de elementos del mismo, en cada pasada asignamos variables al arreglo lo cual es constante (teniendo en cuenta que el cardinal insume tiempo O(1) si el elemento a buscar es el primero, lo cual está garantizado por la forma de recorrer el multiconjunto que hacemos con el iterador), entonces, ese while insume O(r). Luego, recorremos el nuevo arreglo asignando variables ya creadas, eso demora O(r) también. Cuando tenemos que heapificar, el heapify sabemos que demora O(log r) para una llamada, pero nosotros seguimos el algoritmo de Floyd, que nos garantiza que recorriendo el arreglo desde abajo y heapificando desde ahí se reducen la cantidad de swaps que se deben hacer, y en total demora O (r). Finalmente, simplemente debemos asignar las variables a los observaadores de la tupla, por lo tanto la operación demora O (r) + O (r) + O (r) = O (r)

Algoritmo 14

```
heapify(inout ao: arregloDimensionable (nat, nat), in i: nat, inout ar: arregloDimensionable (nat, nat, nat), in n: nat)
    if (i ≤ n/2 -1)
        if (i ≤ n/2 - 2)
            if (disponible(ao[i*2+1]) > disponible (ao[i]) || (disponible(ao[i*2+2]) > disponible (ao[i])))
                if (disponible(ao[i*2+1]) > (disponible(ao[i*2+2])))
                    swap (ao[i], ao[i*2+1])    O(1)
                    orden (ar[recurso(ao[i])]) = i    O(1)
                    orden (ar[recurso(ao[i*2+1])]) = i*2+1    O(1)
                    heapify (ao, i*2+1, ar, n)    O(log r)
                else
                    swap (ao[i], ao[i*2+2])    O(1)
                    orden (ar[recurso(ao[i])]) = i    O(1)
                    orden (ar[recurso(ao[i*2+2])]) = i*2+2    O(1)
                    heapify (ao, i*2+2, ar, n)    O(...)
                endIf
            endIf
        else
            if (disponible(ao[i*2+1]) > disponible (ao[i]))
                swap (ao[i], ao[i*2+1])    O(1)
                orden (ar[recurso(ao[i])]) = i    O(1)
                orden (ar[recurso(ao[i*2+1])]) = i*2+1    O(1)
            endIf
        endIf
    endIf
end Function
```

Una llamada al heapify (sin contar la llamada recursiva) insume tiempo lineal, ya que se puede ver que se acceden a posiciones de un arreglo y se cambian los valores o swapea, lo que tiene costo O (1) al igual que las comparaciones del if. Por lo tanto, el costo del heapify se reduce a la cantidad de veces que se llama recursivamente, se puede ver que cada llamada duplica el i, por lo tanto, este crece exponencialmente y deja de llamar recursivamente cuando esa operación daría mayor que el tamaño del arreglo, por lo tanto, se puede llamar como mucho log n veces, siendo n el tamaño del arreglo. Entonces, el heapify tiene costo O(log n) en peor caso

Algoritmo 15

```
iNuevoConsumo(inout u: estrUsoRecursos, in r: recurso, in n: nat)
    disponible(u.status[r]) = n    O(1)
    disponible(u.ordenesConsumo[(orden(u.status[r]))]) = n    O(1)
    nat i = orden [(u.status[r])] = n    O(1)
    while i ≥ 0
        heapify (u.ordenesConsumo, i, u.status, u.cantRecursos)    O(...)
        i = i/2 -1    O(1)
    endWhile    O(1)
end Function
```

Esta operación cuesta primero asignar valores a posiciones de un array, cuyo acceso es inmediato. Luego, se debe heapificar, y el heapify tiene costo O(log n), con n el tamaño del arreglo, se puede ver que la cantidad de veces que se llama a la función también cambia exponecialmente sobre el tamaño del arreglo. Es más, puede verse que mientras más chico sea el i inicial, menos veces se va a llamar al heapify, y al revés, por lo tanto, la operación tarda O(log n)

Algoritmo 16

iMenorConsumo(in u: estrUsoRecursos) → res: nat
  var n: nat = recurso (u.ordenesConsumo[0])   O(1)
  res ← total(u.status[n]) - disponible(u.status[n])   O(1)
  devolver res
end Function

Esta función tan sólo debe acceder a la primera posición del arreglo de órdenes por consumo, y con ese dato, buscar en el otro arreglo (status) su total y su disponibilidad, y hacer la resta, por lo tanto al ser todas operaciones inmediatas es O(1)

Algoritmo 17

iActualizarConsumo (inout u: estrUsoRecursos, in mconj: recursos)
  var n: nat = u.cantRecursos   O(1)
  var i: nat = 0   O(1)
  var it: recursos ← crearIt (mconj)   O(1)
  while (hayMas (it))   O(1)
    var rtemp: recurso = actual(it)
    disponible (u.status [rtemp]) += cardinal (rtemp, mconj)   O(1)
    disponible (u.ordenesConsumo [orden(u.status[rtemp])]) += cardinal (rtemp, mconj)   O(1)
    heapify (u.ordenesConsumo, rtemp, u.status, n)   O(log r)
    borrarActual (it)   O(1)
  endWhile   O(1)
  devolver res
end Function

Aquí nuevamente debemos iterar sobre un multiconjunto y pedir el cardinal del elemento actual, iterar sobre el multiconjunto, sabiendo que la cantidad de recursos utilizada por una tarea es siempre O(1), se puede hacer en O(1), además sabemos que el cardinal que pedimos es siempre el primero por lo tanto esta operación también siempre demora O(1), dentro del while debemos heapificar, lo cual se sabe que tiene un costo de O(log r) ya justificado; por lo tanto hacer O(1) veces O(log r) resulta en una complejidad para la función de O(log r)

Algoritmo 18

iActualizarUso (inout u: estrUsoRecursos, in ar: arregloDimensionable (nat))
  var n: nat = u.cantRecursos   O(1)
  var i: nat = 0   O(1)
  while ($i < n$)   O(r)
    disponible (u.status [i]) = ar[i]
    disponible (u.ordenesConsumo [orden(u.status[i])] = ar[i]   O(1)
  endWhile
  i = n/2 -1   O(1)
  while ($i \geq 0$)   O(r)
    heapify (u.ordenesConsumo, i, u.status, n)   O(log r)
    i −−   O(1)
  endWhile
end Function

Esta función es similar a la heapificación del generar, primero se actualizan todos los disponibles recorriendo todo el arreglo (con un costo de O(r)); luego se heapifica según el algoritmo de Floyd, que ya justificamos que tiene costo O(r), por lo tanto la función es O(r)

Algoritmo 19

iAlcanzanLosRecursos (in u: estrUsoRecursos, in mconj: recursos) → *res*: bool
  res ← true O(1)
  var n: nat = u.cantRecursos O(1)
  var i: nat = 0 O(1)
  var it: recursos ← crearIt (mconj) O(1)
  while (hayMas (it) ∧ res) O(1)
    var rtemp: recurso = actual(it)
    res ← (disponible (u.status [rtemp] ≥ cardinal (rtemp, mconj)))) O(1)
    borrarActual (it) O(1)
  endWhile O(1)
  devolver res
end Function

Nuevamente, el multiconjunto sobre el cual iteramos nos demora O(1) porque está acotado al ser los consumos de una tarea; el cardinal, como siempre, demora O(1) porque nuestra manera de iterar nos asegura que el cardinal pedido siempre es el primero, por lo tanto, la operación se resuelve en O(1)

Algoritmo 20

iMulticonjuntizar (in u: estrUsoRecursos) → *res*: recursos
  res ← vacio O(1)
  var n: nat = u.cantRecursos O(1)
  var i: nat = 0 O(1)
  while (i < n) O(1)
    var cant: recurso = total (u.status [i])
    while (cant > 0) (agregar (i, res)) O(1)
      cant−− O(1)
    endWhile
    i++
  endWhile O(1)
  devolver res
end Function

Esta función itera sobre el arreglo de recursos, que tiene tamaño n; dentro del while hay otro que itera sobre la cantidad de un mismo recurso existente, esto lo consideramos acotado, pero suponiendo que no lo fuera, si la cantidad máxima de recursos fuera m, la compĺejidad total, sabiendo que el agregar es O(r) porque debe recorrer todo el multiconjunto para verificar si este se encuentra en el mismo, resulta O(n^2∗m)

Algoritmo 21

iDisponiblesEnMulticonj (in u: estrUsoRecursos) → *res*: recursos
  res ← vacio O(1)
  res ← vacio O(1)
  var n: nat = u.cantRecursos O(1)
  var i: nat = 0 O(1)
  while (i < n) O(1)
    var cant: recurso = disponible (u.status [i])
    while (cant > 0) (agregar (i, res)) O(1)
      cant−− O(1)
    endWhile
    i++
  endWhile O(1)
  devolver res
end Function

Esta función itera sobre el arreglo de recursos, que tiene tamaño n; dentro del while hay otro que itera sobre la cantidad de un mismo recurso existente, esto lo consideramos acotado, pero suponiendo que no lo fuera, si la cantidad máxima de recursos fuera m, la compĺejidad total, sabiendo que el agregar es O(r) porque debe recorrer todo el multiconjunto para verificar si este se encuentra en el mismo, resulta O(n^2∗m)

3. Planta

3.1. Interfaz

interfaz: PLANTA
usa:NAT,MULTICONJUNTO(α),CONJ(α),SECU(α),WORKFLOW,USORECURSOS
se explica con: PLANTA DE PRODUCCION
géneros: PLANTA

Operaciones:

workflow(**in** p: planta) $\rightarrow$ res: workflow
{P: true }
{Q: res $=_{\text{obs}}$ workflow (p) }

recursos(**in** p: planta) $\rightarrow$ res: recursos
{P: true }
{Q: res $=_{\text{obs}}$ recursos (p) }

enEspera(**in** p: planta) $\rightarrow$ res: cola
{P: true }
{Q: res $=_{\text{obs}}$ enEspera (p) }

enEjecucion(**in** p: planta) $\rightarrow$ res: actividades
{P: true }
{Q: res $=_{\text{obs}}$ enEjecucion (p) }

finalizadas(**in** p: planta) $\rightarrow$ res: actividades
{P: true }
{Q: res $=_{\text{obs}}$ finalizadas (p) }

crear(**in** w: workflow, **in** r: recursos) $\rightarrow$ res: planta
{P: # finales (w) = 1 }
{Q: res $=_{\text{obs}}$ crear (w,r) }

agOrden(**in** o: orden, **inout** p: planta)
{P: p = p_0 $\wedge$ o $\notin$ ordenes (p) = 1 }
{Q: p $=_{\text{obs}}$ agOrden (o,p_0) }

terminar(**in** a: actividad, **inout** p: planta)
{P: p = p_0 $\wedge$ esta? (a, enEjecucion(p)) }
{Q: p $=_{\text{obs}}$ agOrden (o,p_0) }

actividades(**in** p: planta) $\rightarrow$ res: actividades
{P: true }
{Q: res $=_{\text{obs}}$ actividades (p) }

ordenes(**in** p: planta) $\rightarrow$ res: conj(orden)
{P: true }
{Q: res $=_{\text{obs}}$ ordenes (p) }

ordenesFinalizadas(**in** p: planta) $\rightarrow$ res: conj(orden)
{P: true }
{Q: res $=_{\text{obs}}$ ordenesFinalizadas(p) }

disponibles(**in** p: planta) $\rightarrow$ res: recursos
{P: true }
{Q: res $=_{\text{obs}}$ disponibles(p) }

consumoDeRecurso(**in** p: planta, **in** r: recurso) $\rightarrow$ res: nat
{P: r $\in$ recursos(p) }
{Q: res $=_{\text{obs}}$ consumoDeRecurso (r,p) }

menorConsumo(**in** p: planta) $\rightarrow$ res: nat
{P: true }
{Q: res $=_{\text{obs}}$ menorConsumo (p) }

3.2. Estructura

estrPlanta se representa tupla <recursos: usoRecursos $\times$ workflow: workflow $\times$ terminadas: secu(< orden: nat $\times$ tareas:arregloDimensionable(<termino:bool $\times$ predPendientes: nat>)) $\times$ enEspera: cola $\times$ sucesores: arregloDimensionable (secu (nat))

3.3. Invariante de Representacion

Rep: $\widehat{estrPlanta} \rightarrow$ bool

($\forall$ e : estrPlanta) Rep(e) = workflowValido (e.workflow) $\wedge$ tamSucCorrecto (e.workflow, e.sucesores) $\wedge$ tamTermCorrecto (e.workflow, e.terminadas) $\wedge$ noHayRecursosInvalidos (tiposDeRecurso (e.recursos), e.workflow, 0) $\wedge_{\text{L}}$ (sucesoresCorrectos (e.workflow, e.sucesores, 0) $\wedge$ ordenesCorrectas (e.terminadas, e.enEspera $\wedge$ prioridadesCorrectasEnEspera (e.enEspera, e.workflow) $\wedge_{\text{L}}$ ((predPendientesCorrectas (e.terminadas, e.workflow) $\wedge$ puedenHaberTerminado (e.terminadas, e.workflow) $\wedge$ puedenEstarEnEspera (e.enEspera, e.terminadas, e.workflow) $\wedge$ noHayRecursosParaTareasEnEspera (e.enEspera, e.recursos, e.workflow))))

workflowValido: workflow $\rightarrow$ bool
workflowValido (w) = # finales (w) == 1

noHayRecursosInvalidos: nat $\times$ workflow $\times$ nat $\rightarrow$ bool
noHayRecursosInvalidos (n,w,i) = i == # (tareas(w)) $\vee_{\text{L}}$ (multiconjMenor (consumo(i, w), n) $\wedge_{\text{L}}$noHayRecursosInvalidos (n,w,i+1))

multiconjMenor: recursos $\times$ nat $\rightarrow$ bool
multiconjMenor (r, n) = $\emptyset$? (r) $\vee_{\text{L}}$ (dameUno (r) < n $\wedge_{\text{L}}$ multiconjMenor (r / {dameUno(r)}, n))

tamSucCorrecto: workflow $\times$ arregloDimensionable (secu (nat)) $\rightarrow$ bool
tamSucCorrecto (w,as) = # (tareas(w)) == tam (as)

tamTermCorrecto: workflow $\times$ secu(< nat $\times$ arregloDimensionable(<bool $\times$ nat>)>) $\rightarrow$ bool
tamTermCorrecto (w,s) = vacia? (s) $\vee_{\text{L}}$ tamArrCorrecto (tareas(prim (s))) $\wedge_{\text{L}}$ tamTermCorrecto (w, fin(s))

tamArrCorrecto: workflow $\times$ arregloDimensionable(<bool $\times$ nat>) $\rightarrow$ bool
tamArrCorrecto (w,at) = # (tareas(w)) == tam (at)

sucesoresCorrectos: workflow $\times$ arregloDimensionable (secu (nat)) $\times$ nat $\rightarrow$ bool
sucesoresCorrectos (w,as,i) = i == # (tareas(w)) $\vee_{\text{L}}$ (sucesoresTareaCorrectos(sucesoras(i, w),as[i]) $\wedge_{\text{L}}$sucesoresCorrectos (w,as,i+1))

sucesoresTareaCorrectos: conj(tarea) $\times$ secu (nat) $\rightarrow$ bool
sucesoresTareaCorrectos (c,s) = # (c) == long (s) $\wedge_{\text{L}}$ mismosElems (c,s)

mismosElems: conj(tarea) $\times$ secu (nat) $\rightarrow$ bool
mismosElems (c,s) = vacia? (s) $\vee_{\text{L}}$ (prim (s) $\in$ c $\wedge_{\text{L}}$ mismosElems (c, fin(s))

ordenesCorrectas: secu(< nat $\times$ arregloDimensionable(<bool $\times$ nat>)>) $\times$ cola $\rightarrow$ bool
ordenesCorrectas (s,c) = noHayOrdenesRepetidas (s) $\wedge_{\text{L}}$ordenesColaEnOrdenes (s,c) $\wedge_{\text{L}}$ ordenadasEnEspera (s,c)

ordenadasEnEspera: secu(< nat × arregloDimensionable(<bool × nat>) × cola) → bool
ordenadasEnEspera (s,c) = vacia? (c) $\vee_{L}$ordenadas (ordenes (proximo (c)), dameOrdenes(s)) $\wedge_{L}$ordenadasEnEspera (s, sacarOrden (c, tarea (proximo (c)), prim (ordenes (proximo (c)))))

dameOrdenes: secu(< nat × arregloDimensionable(<bool × nat>)) → secu (nat)
dameOrdenes (s) = **if** vacia? (s) **then** vacia **else** orden (prim (s) • dameOrdenes (fin (s))

ordenadas: secu(nat) × secu (nat)) → bool
ordenadas (s1,s2) = long (s1) $\leq$ 1 $\vee_{L}$ iesimo (prim (s1), s2) < iesimo (prim (fin (s1), s2) $\wedge_{L}$ ordenadas (fin (s1), s2)

iesimo: nat × secu (nat)) → nat
iesimo (n,s) = **if** prim (s) == n **then** 0 **else** 1 + (iesimo (n, fin (s)))

noHayOrdenesRepetidas: secu(< nat × arregloDimensionable(<bool × nat>)>) → bool
noHayOrdenesRepetidas (s) = vacia? (s) $\vee_{L}$ noEstaEnResto (orden(prim (s), fin(s)) $\wedge_{L}$ noHayOrdenesRepetidas (fin(s))

noEstaEnResto: nat × secu(< nat × arregloDimensionable(<bool × nat>)>) → bool
noEstaEnResto (n, s) = vacia? (s) $\vee_{L}$ (n $\neq$ (orden(prim (s))) $\wedge_{L}$ noEstaEnResto (n, fin(s)))

prioridadesCorrectasEnEspera: cola × workflow → bool
prioridadesCorrectasEnEspera (c,w) = vacia? (c) $\vee_{L}$ (prioridad (proxima (c)) == prioridad (tarea(proxima (c)), w) $\wedge_{L}$ prioridadesCorrectasEnEspera sacarOrden (c, tarea (proxima (c)), prim (ordenes (proxima(c))))

ordenesColaEnOrdenes: secu(< nat × arregloDimensionable(<bool × nat>)>) × cola → bool
ordenesColaEnOrdenes (s,c) = vacia? (c) $\vee_{L}$ esOrden (orden(proxima(c))) $\wedge_{L}$ ordenesColaEnOrdenes (sacarOrden (c, tarea (proxima (c)), prim (ordenes (proxima(c))), s)

esOrden: nat × secu(< nat × arregloDimensionable(<bool × nat>)>) → bool
esOrden (n, s) = **if** vacia? (s) **then** false **else** (n == orden(prim (s)) $\vee_{L}$ esOrden (n, fin(s)))

predPendientesCorrectas: secu(< nat × arregloDimensionable(<bool × nat>)>) × workflow → bool
predPendientesCorrectas (s,w) = vacia? (s) $\vee_{L}$ (predPendCorrEnArreglo (tareas(prim (s), w, 0)) $\wedge_{L}$ predPendientesCorrectas (fin(s), w))

predPendCorrEnArreglo: arregloDimensionable(<bool × nat>) × workflow × nat → bool
predPendCorrEnArreglo (a,w,i) = i == # (tareas(w)) $\vee_{L}$ cantPredCorrecta (predecesoras (i), a) $\wedge_{L}$ predPendCorrEnArreglo (a,w,i+1)

cantPredCorrecta: conj (tarea) × arregloDimensionable(<bool × nat>) × workflow × nst → bool
cantPredCorrecta (c,a,i) = # (c) - cantPredTerminaron (c,a) == predPendientes (a[i])

cantPredTerminaron: conj (tarea) × arregloDimensionable(<bool × nat>) → bool
cantPredTerminaron (c,a) = **if** ∅? (c) **then** 0 **else if** termino (a[dameUno(c)]) **then** 1 **else** 0 **fi** + cantPredTerminaron (sinUno (c),a) **fi**

puedenHaberTerminado: secu(< nat × arregloDimensionable(<bool × nat>)>) × workflow → bool
puedenHaberTerminado (s,w) = vacia? (s) $\vee_{L}$ (puedenHabTermEnArreglo (tareas(prim (s), w, 0)) $\wedge_{L}$ puedenHaberTerminado (fin(s), w))

puedenHabTermEnArreglo: arregloDimensionable(<bool × nat>) × workflow × nat → bool
puedenHabTermEnArreglo (a,w,i) = (i == # (tareas(w))) $\vee_{L}$ (termino (a[i]) $\Rightarrow_{L}$terminaronTodosPred (predecesores (i,w), a))

terminaronTodosPred: conj (tarea) × arregloDimensionable(<bool × nat>) → bool
terminaronTodosPred (c,a) = ∅? (c) $\vee_{L}$(termino (a[dameUno(c)]) $\wedge_{L}$terminaronTodosPred (sinUno(c),a))

puedenEstarEnEspera: cola × secu(< nat × arregloDimensionable(<bool × nat>)>) × workflow → bool
puedenEstarEnEspera (c,s,w) = vacia? (c) $\vee_L$ (activPuedeEstar(proxima(c),s,w) $\wedge_L$ puedenEstarEnEspera (sacarOrden (c, tarea (proxima (c)), prim (ordenes (proxima(c))), w))

activPuedeEstar: actividad × secu(< nat × arregloDimensionable(<bool × nat>)>) × workflow → bool
activPuedeEstar (a,s,w) = **if** orden (a) == orden (prim(s)) **then** puedeEstarTarea (tarea(a), tareas(prim(s)), w) **else** activPuedeEstar(a,fin (s),w)

activPuedeEstar: tarea × arregloDimensionable(<bool × nat>) × workflow → bool
activPuedeEstar (t,a,w) = terminaronTodosPred (predecesores (t, w), a)

noHayRecursosParaTareasEnEspera: cola × usoRecursos × workflow → bool
noHayRecursosParaTareasEnEspera (c,u,w) = vacia? (c) $\vee_L$ noHayRecParaTarea(tarea(proxima(c)),s,w) $\wedge_L$ noHayRecursosParaTareasEnEspera (sacarOrden (c, tarea (proxima (c)), prim (ordenes(proxima(c)), w))

noHayRecParaTarea: tarea × usoRecursos × workflow → bool
noHayRecParaTarea (t,u,w) = ¬ alcanzanLosRecursos (u, consumo (t,w))

3.4. Funcion de Abstraccion

Abs : estrPlanta e → planta (Rep(e))

(∀ e: estrPlanta) Abs(e) = p: planta / multiconjuntizar (e.recursos) == (recursos (p)) ∧ e.workflow == workflow (p) ∧ e.enEspera == enEspera (p) ∧ e.enEjecucion == enEjecuion (p) ∧ e.finalizadas == finalizadas (p)

3.5. Algoritmos

Algoritmo 22

iWorfklow(in p: estrPlanta) → *res*: workflow
  res ← p.workflow O(1)
  devolver res
end Function

Acá solo se devuelve el observador de una tupla, lo cual demora O(1)

Algoritmo 23

iRecursos(in p: estrPlanta) → *res*: recursos
  res ← multiconjuntizar(p.recursos) O(r2*m)
  $devolverres$
$endFunction$

Tomar el observador de la tupla demora O(1), la función multiconjuntizar sabemos que demora O(r^2*m) como mucho, siendo r la cantidad de recursos diferentes y m el máximo presente de un recurso dado, por lo tanto esta función es O(r^2*m)

Algoritmo 24

iEnEspera(in p: estrPlanta) → *res*: cola
  res ← (p.enEspera) O(e)
  devolver res
end Function

Acá solo se devuelve el observador de una tupla, lo cual demora O(1). Nota: En este caso cambiamos lo que devolvía la función, en lugar de devolver una ColaEspera devuelve nuestro propio TAD Cola, el cual contiene no actividades sino las ordenes dada una tarea en particular ordenadas en una secuencia. Dado que pasar de uno a otro no es díficil algorítmicamente hablando, pero sí nos requeriría hacer un nuevo módulo sólo para esa operación, preferimos devolver simplemente algo que es suficientemente similar, asegurándonos que la funcionalidad básica y la forma de ser consultada de la misma sigue preservada.

Algoritmo 25

```
iEnEjecucion(in p: estrPlanta) → res: actividades
    res ← vacio    O(1)
    var it: secu (nat) ← crearIt(p.terminadas)    O(1)
    var i: nat = 0    O(1)
    var n: nat = cantTareas (p.workflow)    O(1)
    while (hayMas (it))
        i = 0    O(1)
        while (i < n)    O(1)
            if (¬ termino (tareas(actual(it)) [i]) ∧ predPendientes (tareas(actual(it)) [i]) == 0)    O(1)
                if (¬ esta? (actividad(orden(actual(it)), i), p.enEspera))    O(e)
                    Ag (actividad(orden(actual(it)), i), res)    O(1)
                endIf    O(e)
            endIf    O(e)
            i++    O(1)
        endWhile
        avanzar (it)    O(1)
    endWhile
    devolver res
end Function
```

Acá iteramos sobre todas las órdenes, y luego sobre las tareas de dicha orden (ver lo que contiene dicha tarea es O(1)). Pero a su vez, sobre estas iteramos siempre sobre la cola de enEspera, para buscar si existe en la misma, teniendo en cuenta que para cada tarea llamamos a la función esta?, que debe recorrer toda la secu de órdenes y en caso contrario, agregarlo (en O(1)) a un conjunto; por lo tanto, esta operación demora $O(o*t*(te*o))$, o sea, $O(o^2*t*te)$, siendo o la cantidad de órdenes en la planta, t la cantidad de tareas y te la cantidad de tareas en espera.

Algoritmo 26

iFinalizadas(in p: estrPlanta) → *res*: actividades
    res ← vacio   O(1)
    var it: secu (nat) ← crearIt(p.terminadas)   O(1)
    var i: nat = 0   O(1)
    var n: nat = cantTareas (p.workflow)   O(1)
    while (hayMas (it))
        i = 0   O(1)
        while (i < n)   O(1)
            if (termino (tareas(actual(it)) [i]))   O(1)
                Ag (actividad(orden(actual(it)), i), res)   O(1)
            endIf   O(e)
            i++   O(1)
        endWhile
        avanzar (it)   O(1)
    endWhile
    devolver res
end Function

Esta función es similar a la anterior, primero itera sobre las órdenes de la planta, y para cada una sobre las cantidad de tareas en el workflow, en estas para cada una se accede al arreglo (O(1)) y se agregan si satisfacen al conjunto (también tiempo constante). Por lo tanto, la operación tiene complejidad O(o∗t), siendo o la cantidad de órdenes y t la cantidad de tareas en la planta

Algoritmo 27

```
iCrear(in w: workflow, in r: recursos) → res: planta
    res.recursos ← generar (r)    O(r)
    res.workflow ← w    O(1)
    res.enEspera ← vacia    O(1)
    res.terminadas ← vacia    O(1)
    var i: nat = 0    O(1)
    var n: nat = cantTareas (p.workflow)    O(1)
    var ap: arregloDimensionable (<nat, nat>) = ad(n)    O(1)
    while (i < n)
        π1 (ap[i]) = i    O(1)
        π2 (ap[i]) = prioridad (i,w)    O(1)
        i++    O(1)
    endWhile
    i = n/2 - 1    O(1)
    while (i ≥ 0)
        heapify (ap, i, n)    O(...)
        i−−    O(1)
    endWhile    O(1)
    i = 0    O(1)
    var s: secu (tarea) ← vacia    O(1)
    var it: secu (nat,nat) ← crearIt(s)    O(1)
    while (i < n)
        agregarAtrasDeIt (it, ap[0])
        π2 (ap[0]) = 0    O(1)
        heapify (ap, 0, n)    O(log n)
        i++    O(1)
    endWhile    O(1)
    retrocederAlPrincipio (it)    O(1)
    var asuc: arregloDimensionable (secu (nat)) = ad(n)    O(1)
    while (hayMas (it))
        i = π1 (actual(it))    O(1)
        var itpred: conj(tarea) ← crearIt(predecesores(w,i))    O(1)
        while (hayMas(itpred))    O(1)
            agregarAtras (asuc[actual(itpred)], i)    O(1)
            avanzar (itpred)    O(1)
        endWhile
        avanzar (it)    O(1)
    endWhile
    res.sucesores ← asuc    O(1)
    devolver res
end Function
```

Al crear la planta, primero debemos generarnos el usoRecursos, que sabemos demora O(r) siendo r la cantidad de recursos; a continuación debemos recorrer el arreglo (creado en O(1)) y asignarle a cada posición la tarea correspondiente, con costo total O(n), siendo n la cantidad de tareas del workflow. Luego, procedemos a heapificar este arreglo según el algoritmo de Floyd, de manera muy similar a como lo hicimos en el módulo usoRecursos, sabemos que dicha complejidad es O(n). Después debemos completar el heapsort, para esto tomamos el primer elemento y heapificamos nuevamente (reemplazando ese elemento por uno con prioridad 0, que necesariamente va a bajar con el heapify), como esto se hace n veces y el heapify cuesta O(log n), el heapsort en total nos cuesta O(n log n). Con el heapsort hecho, podemos proceder a obtener los sucesores, los cuales vamos a necesitar que estén de manera ordenada para poder satisfacer complejidades posteriores; al acceder a las tareas de manera ordenada, podemos asegurar que podemos colocarlos ordenados sin problema. Por lo tanto, recorremos los sucesores de cada tarea, en orden de prioridad, y los agregamos en O (1) a la tarea correspondiente (la cantidad de sucesores es dependiente de la cantidad de tareas, por lo tanto al recorrer los sucesores estamos pagando el coste de recorrer las tareas, ya que para que el workflow sea válido la cantidad de sucesores debe ser cuanto menos muy similar a la cantidad de tareas). Si la cantidad de sucesores es m, sabemos que cada uno lo recorremos en O(1), y esto es independiente de las operaciones anteriores, por lo tanto la complejidad de la función es O(r + n log n + m)

Algoritmo 28

heapify(inout ap: arregloDimensionable (nat, nat), in i: nat, in n: nat)
if (i $\leq$ n/2 -1)
if (i $\leq$ n/2 - 2)
if (π_2(ao[i*2+1]) $>$ π_2 (ao[i]) || (π_2(ao[i*2+2]) $>$ π_2 (ao[i])))
if (π_2(ao[i*2+1]) $\geq$ (π_2(ao[i*2+2])))
swap (ao[i], ao[i*2+1]) O(1)

heapify (ao, i*2+1, ar, n) O(..)
else
swap (ao[i], ao[i*2+2]) O(1)
heapify (ao, i*2+2, ar, n) O(...)
endIf
endIf
else
if (π_2(ao[i*2+1]) $\geq$ π_2 (ao[i]))
swap (ao[i], ao[i*2+1]) O(1)
endIf
endIf
endIf
end Function

Esta función es muy similar a la homónima del módulo usoRecursos, la complejidad es idéntica, pot lo tanto demora O(log n), siendo n el tamaño del arreglo a heapificar

Algoritmo 29

iAgOrden(in o: orden, inout p: estrPlanta)
var n: nat = cantTareas (p.workflow)) O(1)
var at: arregloDimensionable (<bool, nat>) = ad(n) O(1)
var i: nat = 0 O(t)
while (i < n)
at[i] = <false, cantPredecesoras (p.worfklow, i)> O(1)
i++ O(1)
endWhile
if (alcanzanLosRecursos (p.recursos, consumo (p.workflow, 0)))
actualizarConsumo (p.recursos, consumo (p.workflow, 0)) O(log r)
else
encolar (p.enEspera, 0, prioridad (p.workflow, 0), o) O(1)
endIf
agregarAlFinal (p.terminadas, <o, at>) O(1)
devolver res
end Function

Al ejecutar esta función, se debe agregar un nuevo arreglo para la nueva orden, para esto lo creamos y le asignamos a cada posición los datos correspondientes (con costo total O(n), siendo n la cantidad de tareas del workflow). Luego de hecho esto, podemos verificar si la primera tarea puede ejecutarse (sabemos que no tiene predecesores), si puede, actualizamos los recursos con la función actualizarConsumo, con costo O(log r) según el módulo usoRecursos (r siendo la cantidad de recursos distintos, o sea la longitud de dicho arreglo); en caso contrario, debemos ponerla en cola, como sabemos que la primera tarea siempre tiene máxima prioridad, el módulo cola se ocupa de agregarla en O(1). Por lo tanto, la complejidad en peor caso es O (n + log r)

Algoritmo 30

```
iTerminar (in actividad a, inout p: estrPlanta)
    var rec: arregloDimensionable(nat) = recursosAArreglo (p)    O(r)
    var it: secu (<nat,arregloDimensionable(<bool, nat>)>) ← crearIt(p.terminadas)    O(1)
    while (orden(actual (it)) ≠ orden (a))
        avanzar (it)    O(1)
    endWhile
    termino (tareas(actual(it))[tarea(a)]) = true    O(1)
    var mulrec: recursos = recursos (tarea(a), p.workflow)    O(1)
    actualizarArreglo (rec, mulrec, true)    O(1)
    var suc: secu (nat) = p.sucesores [tarea(a)]    O(1)
    var aencolar: secu (<actividad, nat>) = vacia    O(1)
    var itsuc: secu (nat) ← crearIt(suc)    O(1)
    while (hayMas (it))
        predPendientes (tareas(actual(it))[actual (itsuc)]) -= 1    O(1)
        if (predPendientes (tareas(actual(it))[actual (itsuc)]) == 0)    O(1)
            agregarAlFinal (aencolar, <actividad (orden(actual(it), actual (itsuc))), prioridad (p.worfklow,actual
(itsuc)))>)    O(1)
        endIf
        avanzar (it)    O(1)
    endWhile
    encolarSecuOrdenada (p.enEspera, aencolar, p.workflow)    O(e+t)
    var itEspera: colaConPrioridades ← crearIt(p.enEspera)    O(1)
    while (hayMas (itEspera))
        mulrec = recursos (tarea(actual(itEspera)), p.workflow)    O(1)
        if (mconjIncluidoEnArreglo(rec, mulrec))    O(1)
            var itOrden: secu(nat) ← crearIt(orden(actual(itEspera)))    O(1)
            while (hayMas (itOrden) ∧ (mconjIncluidoEnArreglo(rec, mulrec)))
                borrarActual (itOrden)    O(1)
                actualizarArreglo (rec, mulrec, false)    O(1)
            endWhile
            if (vacia? (ordenes(actual(itEspera))))
                borrarActual (itEspera)    O(1)
            endIf
        endIf
        avanzar (itEspera)    O(1)
    endWhile
    actualizarUso (p.recursos, rec)    O(r)
end Function
```

Al ejecutar esta función, primero bajamos el usoRecursos a un arreglo, para poder trabajar con esto mismo sin necesidad de que al actualizarlos se re-heapifique el consumo (ya que es posible que necesitemos actualizar el consumo varias veces dadas diferentes tareas que podrían entrar en ejecución), esta operación tiene costo O(r), con r la cantidad de recursos distintos, o sea, la longitud del arreglo que estamos creando; luego recorremos la secuencia de órdenes hasta encontrar la de la tarea que terminó con el fin de marcarla como terminada, esto tiene costo O(o), siendo o la cantidad de órdenes en la planta. Después, podemos actualizar los recursos liberados por la tarea recién finalizada, esta operación tiene costo O(1) ya que el multiconjunto está acotado. A continuación, debemos ver los sucesores de esta tarea, iterar sobre esta secuencia tiene costo O(s) siendo o los sucesores de ésta; al iterar sólo actualizamos los predecesores pendientes y vemos si este número es 0, hacer esto es inmediato puesto que es una posición del arreglo al cual ya habíamos accedido cuando iteramos sobre orden; en caso de que esa tarea tenga todos sus predecesores terminados, la agregamos al final de la secuencia (recordar que estamos accediendo a una secuencia ordenada por prioridad) en O(1). Una vez lista la secuencia, la encolamos en la cola enEspera, esta operacion tiene costo O(t+s) tal como lo dice el módulo cola, siendo t la cantidad de tareas en la cola y s la de la secuencia de sucesores listos para ejecutar. Hecho esto, debemos iterar sobre la cola para ver qué tareas entran en ejecución, iterar sobre la cola tiene un costo O(e), en cada tarea se verifica si alcanzan los recursos (en O(1) con la operacion mconjIncluidoEnArreglo), en caso de que se pueda ejecutar, se saca la primera orden y se actualiza el arreglo con los nuevos consumos en tiempo constante, luego se vuelve a verificar hasta terminar la cola. Por lo tanto, esta última operación solo se ejecuta una cantidad de veces determinada por cuántas operaciones se entren a ejecutar, si esta cantidad es j, esta operación es O(j), sabiendo que el costo O(t) ya estaba pagado. En resumen, la operación tiene un costo total O(r+o+s+e+j), siendo cada letra lo ya indicado

Algoritmo 31

recursosAArreglo (in p: estrPlanta) $\rightarrow$ *res*: arregloDimensionable (nat)
  var n: nat = tiposDeRecurso (p.recursos)   O(1)
  var ar: arregloDimensionable (nat) = ad(n)   O(1)
  var i: nat = 0   O(1)
  while (i < n)   O(r)
    ar[i] = disponible (p.recursos, i)   O(1)
    i++   O(1)
  endWhile   O(1)
  devolver res
end Function

Para bajar el usoRecursos a un arreglo, simplemente debemos crear un arreglo con la cantidad de recursos diferentes, y mover cada disponible al nuevo arreglo. Cada una de estas operaciones tiene costo O(1), por lo tanto la operación tiene costo O(r), siendo r la cantidad de recursos diferentes en usoRecursos

Algoritmo 32

mconjIncluidoEnArreglo(in r: arregloDimensionable(nat), in mconj: recursos) $\rightarrow$ *res*: bool)
  res $\leftarrow$ true   O(1)
  var it: recursos $\leftarrow$ crearIt (mconj)   O(1)
  while (hayMas (it) $\wedge$ res)
    rtemp = actual (it)   O(1)
    res $\leftarrow$ (r[rtemp] $\geq$ cardinal (mconj, rtemp))   O(1)
    borrarActual(it)   O(1)
  endWhile   O(1)
  devolver res
end Function

Esta función siempre se llama con un multiconjunto de recursos proveniente de los recursos requeridos por una tarea, los cuales sabemos que estan acotados, por lo tanto iterar sobre ellos es siempre O(1), por lo tanto la operación es O(1)

Algoritmo 33

```
actualizarArreglo(inout r: arregloDimensionable(nat), in mconj: recursos, in sumar: bool)
    var rtemp: recurso = 0
    var it: recursos ← crearIt (mconj)    O(1)
    if (sumar)
        while (hayMas (it))
            rtemp = actual (it)    O(1)
            r[rtemp] += cardinal (mconj, rtemp)    O(1)
            borrarActual (it)    O(1)
        endWhile    O(1)
    else
        while (hayMas (it))
            rtemp = actual (it)    O(1)
            r[rtemp] -= cardinal (mconj, rtemp)    O(1)
            borrarActual (it)    O(1)
        endWhile    O(1)
    endIf
end Function
```

Nuevamente, debemos iterar sobre un multiconjunto proveniente de un requerimiento de tarea, entonces la iteración sobre el mismo tiene costo O(1), y obviamente acceder al arreglo conociendo las posiciones a actualizar es constante, entonces la operación tarda O(1)

Algoritmo 34

```
iActividades(in p: estrPlanta) → res: conj (actividad)
    res ← vacio    O(1)
    var i: nat = 0    O(1)
    var itord: secu (<nat,arregloDimensionable(<bool, nat>)>) ← crearIt(p.terminadas)    O(1)
    while (hayMas (itord))
        while i < cantTareas(p.workflow)    O(1)
            agregar (actividad (orden (actual(itord)), i))    O(1)
            i++)    O(1)
        endWhile    O(1)
        avanzar (it)    O(1)
        i = 0    O(1)
    endWhile
    devolver res
end Function
```

Esta operación en principio itera sobre la secuencia de órdenes, por lo tanto implica un costo O(o), con o la cantidad de órdenes de la planta, y luego en cada orden sobre la cantidad de tareas, como el agregar insume tiempo constante, la operacion en total tiene un costo O(o∗t), siendo t la cantidad de tareas

Algoritmo 35

iOrdenes(in p: estrPlanta) $\rightarrow$ *res*: conj (orden)
  res ← vacio O(1)
  var it: secu (<nat,arregloDimensionable(<bool, nat>)>) ← crearIt(p.terminadas) O(1)
  while (hayMas (it))
    Ag (π_1 (actual(it), res)) O(1)
    avanzar (it) O(1)
  endWhile
  devolver res
end Function

Esta operación itera sobre la secuencia de órdenes, por lo tanto implica un costo O(o), como el agregar insume tiempo constante, la operacion en total tiene un costo O(o)

Algoritmo 36

iOrdenesFinalizadas(in p: estrPlanta) $\rightarrow$ *res*: conj (orden)
  res ← vacio O(1)
  var it: secu (<nat,arregloDimensionable(<bool, nat>)>) ← crearIt(p.terminadas) O(1)
  while (hayMas (it))
    if (terminoOrden (tareas (it)), cantTareas (p.workflow)) O(t)
      Ag (π_1 (actual(it), res)) O(1)
    endIf O(1)
    avanzar (it) O(1)
  endWhile
  devolver res
end Function

Esta operación en principio itera sobre la secuencia de órdenes, por lo tanto implica un costo O(o), con o la cantidad de órdenes de la planta, y luego en cada orden llamar la funcion terminoOrden con costo O(t) como el agregar insume tiempo constante, la operacion en total tiene un costo O(o∗t), siendo t la cantidad de tareas

Algoritmo 37

terminoOrden (in at:arregloDimensionable(<bool, nat>), in n: nat) $\rightarrow$ *res*: bool
  res ← true O(1)
  var i: nat = 0 O(1)
  while (i< n $\wedge$ res)
    if ($\neg$ π_1 (at[i]) O(1)
      res ← false O(1)
    endIf O(1)
    i++ O(1)
  endWhile
  devolver res
end Function

Aquí debemos recorrer todo el arreglo para verificar si los bool de terminoTarea son true, esto tiene un costo de O(t) siendo t la cantidad de tareas, por lo tanto al final el costo total es O(t)

Algoritmo 38

iDisponibles (in at:arregloDimensionable(<bool, nat>), in n: nat) → *res*: bool
res ← disponiblesEnMulticonj (p.recursos) O(r^2∗m)
devolver res
end Function

Esta función se limita a llamar a disponiblesEnMulticonj, por lo tanto su costo es idéntico, esto es O(r^2∗m)), siendo r la cantidad de recursos diferentes y m el máximo existente de un recurso

Algoritmo 39

iConsumo(in p: estrPlanta, in r: recurso) → *res*: nat
res ← total(p.recursos, r) - disponible (p.recursos, r) O(r)
devolver res
end Function

Esta funcion dado un recurso simplemente llama a las funciones de usoRecursos con el mismo, sabemos que ambas tardan O(1) por lo tanto esta función también es O(1)

Algoritmo 40

iMenorConsumo(in p: estrPlanta) → *res*: nat
res ← menorConsumo (p.recursos) O(r)
devolver res
end Function

Esta funcion dado un simplemente llama a la de usoRecursos, que tiene un costo O(1) por lo tanto esta función también es O(1)

4. Secuencia con Iteradores

4.1. Interfaz

interfaz: SECUENCIA CON ITERADORES
usa: NAT
se explica con: SECUENCIA (α), ITERADOR (α)
géneros: SECU α, ITERADOR (α)

Operaciones:

vacia?(**in** s: secu(α)) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ vacia? (s) }

esta?(**in** s: secu(α), n: α) $\rightarrow$ res: α
{P: true) }
{Q: res $=_{\text{obs}}$ esta? (s,n) }

prim(**in** s: secu(α)) $\rightarrow$ res: α
{P: $\neg$ vacia? (s) }
{Q: res $=_{\text{obs}}$ prim (s) }

fin(**inout** s: secu(α))
{P: s = s_0 $\wedge$ $\neg$ vacia? (s) }
{Q: s $=_{\text{obs}}$ fin (s_0) }

vacia() $\rightarrow$ res: secu(α)
{P: true }
{Q: res $=_{\text{obs}}$ <> }

agregarAdelante(**in** s: secu(α), n: α) $\rightarrow$ res: secu(α)
{P: true }
{Q: res $=_{\text{obs}}$ n $\bullet$ s }

agregarAtras(**in** s: secu(α), n: α) $\rightarrow$ res: secu(α)
{P: true }
{Q: res $=_{\text{obs}}$ n $\circ$ s }

crearIt(**in** s: secu(α)) $\rightarrow$ res: iterador(α)
{P: true }
{Q: res $=_{\text{obs}}$ crearIt(<>,s) }

hayMas(**in** it: iterador(α)) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ hayMas(it) }

actual(**in** it: iterador(α)) $\rightarrow$ res: α
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ actual(it) }

avanzar(**inout** it: iterador(α))
{P: it = it_0 $\wedge$ hayMas (it) }
{Q: it $=_{\text{obs}}$ avanzar(it_0) }

agregarAtrasDeIt(**inout** it: iterador(α), n: α)
{P: it = it_0 }
{Q: it $=_{\text{obs}}$ agregarAtrasDeIt (it_0, n) }

agregarAdelanteDeIt(**inout** *it*: iterador(α), n: α)
{P: it = it_0 }
{Q: it $=_{obs}$ agregarAdelanteDeIt (it_0, n) }

retrocederAlPrincipio(**inout** *it*: iterador(α))
{P: it = it_0 }
{Q: it $=_{obs}$ retrocederAlPrincipio (it_0) }

borrarActual(**inout** *it*: iterador(α))
{P: it = it_0 $\wedge$ hayMas (it) }
{Q: it $=_{obs}$ borrarActual (it_0, n) }

Es importante remarcar que las operaciones actual, agregarAtrasDeIt, agregarAdelanteDeIt y borrarActual sufren de aliasing, que es lo que nosotros queremos para poder usar el iterador correctamente, cualquier cambio realizado al iterador con alguna de esas funciones hará que los cambios se vean reflejados en la secuencia que está siendo iterada, ya que el iterador comparte la estructura interna con la secuencia. Análogamente, cualquier cambio que ocurra en la secuencia iterada que no tenga que ver con el iterador lo afectará al mismo. Esto hace fácil eliminar un elemento del medio de la secuencia, agregarlo, o cambiarlo, y para esto usaremos dichas operaciones con el fin de facilitarnos el manejo de las secuencias. Obviamente, el aliasing de este iterador se aplica también en los otros módulos que usan el mismo iterador

4.2. Estructura

ESTRSECU SE REPRESENTA TUPLA $<$ primero: puntero (nodo) $\times$ ultimo: puntero (nodo) $>$, donde nodo es tupla $<$dato: α $\times$ prox: puntero (nodo) $\times$ ant: puntero (nodo)$>$
ESTRITERADOR ES TUPLA $<$ sec: puntero (estrSecu) $\times$ posic: puntero (nodo) $>$

4.3. Invariante de Representacion

Rep: $\widehat{estrSecu} \rightarrow$ bool

($\forall$ e : estrSecu) Rep(e) = true $\leftrightarrow$ no hay ciclos en la lista encadenada

4.4. Funcion de Abstraccion

Abs : estrSecu e $\rightarrow$ secu(α) (Rep(e))

($\forall$ e: estrSecu) Abs(e) = s: secu(α) / **if** e.primero = nil **then** $<>$ **else** e.dato $\bullet$ Abs (e.prox) **fi**

AbsIt : estrSecu e $\rightarrow$ iterador(α) (Rep(e))

($\forall$ e: estrSecu) AbsIt(e) = s: iterador(α) / crearIt(Abs(e))

4.5. Algoritmos

Algoritmo 41

iVacia?(in s: estrSecu(α)) $\rightarrow$ *res*: bool
  res $\leftarrow$ (s.primero == nil) O(1)
end Function

Algoritmo 42

iPrim(in s: estrSecu(α)) $\rightarrow$ *res*: α
  res $\leftarrow$ dato (s.primero) O(1)
end Function

Algoritmo 43

iFin(inout s: estrSecu(α))
  s.primero = prox (s.primero) O(1)
  ant (s.primero) $\rightarrow$ nil O(1)

end Function

Algoritmo 44

iVacia() $\rightarrow$ *res*: α
  res.primero = res.ultimo = nil O(1)
end Function

Algoritmo 45

iAgregarAdelante(inout s: estrSecu(α), in n: α)
  var new: nodo O(1)
  new.prox $\leftarrow$ (s.primero) O(1)
  new.ant $\leftarrow$ nil O(1)
  new.dato $\leftarrow$ n O(1)
  s.primero $\leftarrow$ new O(1)
end Function

Algoritmo 46

iAgregarAtras(inout s: estrSecu(α), in n: α)
  var new: nodo O(1)
  new.prox $\leftarrow$ nil O(1)
  new.dato $\leftarrow$ n O(1)
  new.ant $\leftarrow$ s.ultimo O(1)
  proximo (s.ultimo) $\leftarrow$ new O(1)
  s.ultimo $\leftarrow$ new O(1)
end Function

Algoritmo 47

iCrearIt(in s: estrSecu(α)) $\rightarrow$ *res*: estrIterador(α)
  res.sec $\leftarrow$ s O(1)
  res.posic $\leftarrow$ s.primero O(1)
end Function

Algoritmo 48

iHayMas(in it: estrIterador(α)) $\rightarrow$ *res*: bool
  res $\leftarrow$ (it.posic $\neq$ nil) O(1)
end Function

Algoritmo 49

iActual(in it: estrIterador(α)) $\rightarrow$ *res*: α
  res $\leftarrow$ dato (it.posic) O(1)
end Function

Algoritmo 50

iAvanzar(inout it: estrIterador(α))
  it.posic $\leftarrow$ prox (it.posic) O(1)
end Function

Algoritmo 51

```
iAgregarAtrasDeIt(inout it: estrIterador(α), in n: α)
    var new: nodo    O(1)
    new.dato ← n    O(1)
    if (primero (it.sec) → nil)    O(1)
        primero (it.sec) = ultimo (it.sec) = new    O(1)
        new.prox ← nil    O(1)
        new.ant ← nil    O(1)
        it.posic ← new    O(1)
    else if (it.posic = ultimo (it.sec))
        new.ant ← ultimo (it.sec)    O(1)
        ultimo (it.sec) = new    O(1)
        new.prox ← nil    O(1)
        it.posic ← new    O(1)
    else
        new.prox ← prox (it.posic)    O(1)
        new.ant ← ant (it.posic)    O(1)
        ant(new.prox) ← new    O(1)
        prox (it.posic) ← new    O(1)
        it.posic ← new    O(1)
    endIf
end Function
```

Algoritmo 52

```
iAgregarAdelanteDeIt(inout it: estrIterador(α), in n: α)
    var new: nodo    O(1)
    new.dato ← n    O(1)
    if (primero (it.sec) → nil)    O(1)
        primero (it.sec) = ultimo (it.sec) = new    O(1)
        new.prox ← nil    O(1)
        new.ant ← nil    O(1)
        it.posic ← new    O(1)
    else if (it.posic = primero (it.sec))
        new.prox ← primero (it.sec)    O(1)
        primero (it.sec) = new    O(1)
        new.ant ← nil    O(1)
        it.posic ← new    O(1)
    else
        new.prox ← it.posic    O(1)
        new.ant ← ant (it.posic)    O(1)
        ant (it.posic) ← new    O(1)
        prox (new.ant) ← new    O(1)
    endIf
end Function
```

Algoritmo 53

```
iRetrocederAlPrincipio(inout it: estrIterador(α))
    it.posic ← primero (it.sec)    O(1)
end Function
```

Algoritmo 54

```
iBorrarActual(inout it: estrIterador(α))
    if (it.posic = prim (it.sec))
        prim (it.sec) ← prox (it.posic)    O(1)
        if (primero (it.sec) → nil)    O(1)
            it.posic← nil    O(1)
            ultimo (it.sec)← nil    O(1)
        else    O(1)
            ant (prox (it.posic)) ← nil    O(1)
            it.posic ← prox (it.posic)    O(1)
        endIf    O(1)
    else if (it.posic = ultimo (it.sec))
        ultimo (it.sec) ← ant (it.posic)    O(1)
        prox (ant (it.posic)) ← nil    O(1)
        it.posic ← nil    O(1)
    else
        ant (prox (it.posic) ← ant (it.posic)    O(1)
        prox (ant (it.posic) ← prox (it.posic)    O(1)
        it.posic ← prox (it.posic)    O(1)
    endIf
end Function
```

5. Conjunto con Iteradores

5.1. Interfaz

interfaz: Conjunto con Iteradores
usa: Secuencia con Iteradores (α)
se explica con: Conjunto (α), iterador (α)
géneros: conj α, iterador (α)

Operaciones:

pertenece(**in** *a*: α, **in** *c*: conj(α)) $\rightarrow$ *res*: bool
{P: true }
{Q: res $=_{\text{obs}}$ a $\in$ c }

vacio() $\rightarrow$ *res*: conj(α)
{P: true }
{Q: res $=_{\text{obs}}$ $\emptyset$ }

vacio?(**in** *c*: conj(α)) $\rightarrow$ *res*: bool
{P: true }
{Q: res $=_{\text{obs}}$ $\emptyset$? (c) }

agregar(**in** *a*: α, **inout** *c*: conj(α))
{P: c = c_0 }
{Q: c $=_{\text{obs}}$ Ag (a,c_0) }

sacar(**in** *a*: α, **inout** *c*: conj(α))
{P: c = c_0 }
{Q: c $=_{\text{obs}}$ c_0 - {a} }

union(**inout** *c*: conj(α), **in** *otroc*: conj(α))
{P: c = c_0 }
{Q: c $=_{\text{obs}}$ c_0 $\cup$ otroc }

crearIt(**in** *c*: conj(α)) $\rightarrow$ *res*: iterador(α)
{P: true }
{Q: res $=_{\text{obs}}$ crearIt(c) }

hayMas(**in** *it*: iterador(α)) $\rightarrow$ *res*: bool
{P: true }
{Q: res $=_{\text{obs}}$ hayMas(it) }

actual(**in** *it*: iterador (α)) $\rightarrow$ *res*: α
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ actual(it) }

avanzar(**inout** *it*: iterador (α))
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ avanzar(it) }

borrarActual(**inout** *it*: iterador(α))
{P: it = it_0 $\wedge$ hayMas (it) }
{Q: it $=_{\text{obs}}$ borrarActual (it_0, n) }

5.2. Estructura

estrConj se representa con Secuencia(α)

5.3. Invariante de Representacion

Rep: $\widehat{estrConj} \rightarrow$ bool

($\forall$ e : estrConj) Rep(e) = noHayRepetidos (e)

noHayRepetidos: secu(α) $\rightarrow$ bool
noHayRepetidos(s) = $\neg$ esta?(prim(s), noHayRepetidos(fin(s)))

5.4. Funcion de Abstraccion

5.5. Algoritmos

Algoritmo 55

iPertenece(in a: α, in c: estrConj(α)) $\rightarrow$ res: bool
  res $\leftarrow$ esta?(c,a) O(1)
end Function

Algoritmo 56

iVacio() $\rightarrow$ res: estrConj(α)
  res $\leftarrow$ vacia O(1)
end Function

Algoritmo 57

iVacio? (in c: estrConj(α)) $\rightarrow$ res: bool
  res $\leftarrow$ vacia? (c) O(1)
end Function

Algoritmo 58

iAgregar(in a: α, inout c: estrConj(α))
    if ($\neg$ pertenece (a,c))   O(1)
        agregarAdelante (c,a)   O(1)
    endIf   O(1)
end Function

Como la funcion debe llamar a pertenece, que tiene un costo O(n), y el agregarAdelante es O(1), en total el costo es O(n)

Algoritmo 59

iSacar(in a: α, inout c: estrConj(α))
    var it: secu(α) $\leftarrow$ crearIt(c)   O(1)
    var bool: queda = true   O(1)
    while (hayMas (it) $\wedge$ queda)   O(n)
        if (actual (it) == a)   O(1)
            borrarActual (it)   O(1)
            queda = false   O(1)
        endIf   O(1)
        avanzar (it)   O(1)
    endWhile
end Function

Algoritmo 60

iUnion(inout c: estrConj(α), in otroc: estrConj(α))
    var it: secu(α) $\leftarrow$ crearIt(otroc)   O(1)
    while (hayMas (it))   O(# otroc)
        if ($\neg$ esta? (c, actual (it))   O(# c)
            agregarAdelante (c, actual(it))   O(1)
        endIf   O(1)
        avanzar (it)   O(1)
    endWhile
end Function

Algoritmo 61

iCrearIt(in c: estrConj(α)) $\rightarrow$ res: iterador(α)
    res $\leftarrow$ crearIt (c)   O(1)
end Function

Algoritmo 62

iHayMas(in it: iterador(α)) $\rightarrow$ res: bool
    res $\leftarrow$ hayMas (it)   O(1)
end Function

Algoritmo 63

iActual(in it: iterador(α)) $\rightarrow$ res: α
    res $\leftarrow$ actual (it)   O(1)
end Function

Algoritmo 64

iAvanzar(inout it: iterador(α))
    avanzar(it)   O(1)
end Function

Algoritmo 65

iBorrarActual(inout it: iterador(α))
    res $\leftarrow$ borrarActual (it)   O(1)
end Function

6. Multiconjunto con Iteradores

6.1. Interfaz

interfaz: MULTICONJUNTO CON ITERADORES
usa:SECUENCIA CON ITERADORES (α)
se explica con: MULTICONJUNTO (α), ITERADOR (α)
géneros: MCONJ α, ITERADOR (α)

Operaciones:

cardinal (**in** a: α, **in** c: mconj(α)) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ # (a,c) }

vacio() $\rightarrow$ res: mconj(α)
{P: true }
{Q: res $=_{\text{obs}}$ $\emptyset$ }

agregar(**in** a: α, **inout** c: mconj(α))
{P: c = c_0 }
{Q: c $=_{\text{obs}}$ Ag (a,c_0) }

cantElemDistintos(**inout** c: mconj(α) $\rightarrow$ res: bool)
{P: true }
{Q: res $=_{\text{obs}}$ (cantElemDistintos (c)) }

crearIt(**in** c: mconj(α)) $\rightarrow$ res: iterador(α)
{P: true }
{Q: res $=_{\text{obs}}$ crearIt(c) }

hayMas(**in** it: iterador(α)) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ hayMas(it) }

actual(**in** it: iterador (α)) $\rightarrow$ res: α
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ actual(it) }

avanzar(**inout** it: iterador (α))
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ avanzar(it) }

borrarActual(**inout** it: iterador(α))
{P: it = it_0 $\wedge$ hayMas (it) }
{Q: it $=_{\text{obs}}$ borrarActual (it_0, n) }

6.2. Estructura

estrMConj se representa con Secuencia(<cant: nat, elem: α >)

6.3. Invariante de Representacion

Rep: $\widehat{estrMConj}$ → bool

($\forall$ e : estrMConj) Rep(e) = noHayRepetidos (e) ∧ cantidadCorrecta(e)

noHayRepetidos: secu(α) → bool
noHayRepetidos(s) = ¬ esta?(prim(s), noHayRepetidos(fin(s)))

cantidadCorrecta: secu(α) → bool
cantidadCorrecta(s) = cant(elem) ≥ 0

6.4. Funcion de Abstraccion

6.5. Algoritmos

Algoritmo 66

iCardinal(in a: α, in c: estrConj(α)) → res: nat
  var it: secu (α) ← crearIt(c) O(1)
  var noencontro: bool = true O(1)
  res ← 0 O(1)
  while (hayMas (it) ∧ noecontro) O(# otroc)
    if (elem(actual (it) == a) O(# c)
      res ← cant(actual(it)) O(1)
      noencontro = false
O(1)
    endIf
    avanzar (it) O(1)
  endWhile
end Function

La función cardinal es fácil ver que su complejidad en peor caso es O(n) siendo n la cantidad de tuplas en la secuencia, pero hay que tener en cuenta cada vez que la usemos lo haremos de manera que el cardinal que se pida sea el del primer elemento, por lo que en esos casos va a ser O(1) ya que es inmediato

Algoritmo 67

iVacio() → res: estrConj(α)
  res ← vacia O(1)
end Function

La operacion vacia? de secu es O(1), por lo tanto esta operación lo es también

Algoritmo 68

iAgregar(in a: α, inout c: estrMConj(α))
    var it: secu (α) $\leftarrow$ crearIt(c)   O(1)
    var noencontro: bool = true   O(1)
    res $\leftarrow$ 0   O(1)
    while (hayMas (it) $\wedge$ noecontro)   O(# otroc)
        if (elem(actual (it) == a)   O(# c)
            actual (it) = actual(it) +1
O(1)
            noencontro = false   O(1)
        endIf
        avanzar (it)   O(1)
    endWhile
    if ($\neg$ noencontro)
O(1)   agregarAlFinal (<1,a>)
    endWhile
end Function

La funcion debe recorrer toda la secuencia hasta encontrar el elemento en la tupla (si lo hubiere), esto cuesta O(n), y el agregarAlFinal es O(1), en total el costo es O(n), con n la longitud de la secuencia

Algoritmo 69

iCantElemDistintos(in c: estrMConj(α))
    var it: secu (α) $\leftarrow$ crearIt(c)   O(1)
    res $\leftarrow$ 0   O(1)
    while (hayMas (it))   O(long(c))
        res ++   O(1)
        avanzar (it)   O(1)
    endWhile
    devolver res
O(1) end Function

Esta función debe iterar sobre toda la secuencia, es decir, sobre la cantidad de tuplas, en total tiene complejidad O(n), con n la longitud de la secuencia

Algoritmo 70

iCrearIt(in c: estrMConj(α)) $\rightarrow$ res: iterador(α)
    res $\leftarrow$ crearIt (c)   O(1)
end Function

Algoritmo 71

iHayMas(in it: iterador(α)) $\rightarrow$ res: bool
    res $\leftarrow$ hayMas (it)   O(1)
end Function

Algoritmo 72

iActual(in it: iterador(α)) $\rightarrow$ res: α
    res $\leftarrow$ actual (it)   O(1)
end Function

Algoritmo 73

iAvanzar(inout it: iterador(α))
    avanzar(it)   O(1)
end Function

Algoritmo 74

iBorrarActual(inout it: iterador(α))
    res $\leftarrow$ borrarActual (it)   O(1)
end Function

7. Cola

7.1. Interfaz

interfaz: COLA
usa: SECUENCIA CON ITERADORES (α)
se explica con: COLA, ITERADOR (α)
géneros: COLA, ITERADOR (α)

Operaciones:

vacia? (**in** c: cola) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ vacia? (c) }

proxima (**in** c: cola) $\rightarrow$ res: actividad
{P: $\neg$ (vacia?(c)) }
{Q: res $=_{\text{obs}}$ proximo (c) }

prioridad (**in** c: cola) $\rightarrow$ res: nat
{P: $\neg$ (vacia?(c)) }
{Q: res $=_{\text{obs}}$ prioridad (c) }

ordenes (**in** c: cola, **in** t: tarea) $\rightarrow$ res: secu (orden)
{P: (esta? (c,t)) }
{Q: res $=_{\text{obs}}$ ordenes (c,t) }

esta? (**in** c: cola, **in** t: tarea) $\rightarrow$ res: true
{P: true }
{Q: res $=_{\text{obs}}$ esta? (c,t) }

sacarOrden (**inout** c: cola, **in** t: tarea, **in** o: nat)
{P: c = c_0 $\wedge$ esta? (c,t) $\wedge_L$ esta? (o, ordenes (c,t)) }
{Q: c $=_{\text{obs}}$ desencolar (c_0,o) }

vacia () $\rightarrow$ res: c: cola
{P: true) }
{Q: c $=_{\text{obs}}$ vacia () }

encolar (**inout** c: cola, **in** t: tarea, **in** p: prioridad, **in** o: nat)
{P: c = c_0 $\wedge$) }
{Q: c $=_{\text{obs}}$ encolar (c_0, t, p, o) }

encolarSecuOrdenada (**inout** c: cola, **in** s: secu (<actividad, nat>))
{P: c = c_0 $\wedge$) }
{Q: c $=_{\text{obs}}$ encolarSecuOrdenada (c, s) }

crearIt(**in** c: conj(α)) $\rightarrow$ res: iterador(α)
{P: true }
{Q: res $=_{\text{obs}}$ crearIt(c) }

hayMas(**in** it: iterador(α)) $\rightarrow$ res: bool
{P: true }
{Q: res $=_{\text{obs}}$ hayMas(it) }

actual(**in** it: iterador (α)) $\rightarrow$ res: α
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ actual(it) }

avanzar(**inout** *it*: iterador (α))
{P: hayMas (it) }
{Q: res $=_{\text{obs}}$ avanzar(it) }

borrarActual(**inout** *it*: iterador(α))
{P: it = it_0 $\wedge$ hayMas (it) }
{Q: it $=_{\text{obs}}$ borrarActual (it_0, n) }

7.2. Estructura

ESTRCOLA SE REPRESENTA CON SECUENCIA(PRIORACT), con PriorAct un renombre de la tupla <tarea: nat, prioridad: nat, ordenes: secu (orden)>

7.3. Invariante de Representacion

Rep: $\widehat{estrCola} \to$ bool

($\forall$ e : estrMConj) Rep(e) = estanOrdenados (e) $\wedge$ noHayTareasRepetidas (e) $\wedge$ noHayPrioridadesRepetidas (e) noHayOrdenesRepeEnSecu (e)

estanOrdenados: secu(<nat, nat, secu (nat)) $\to$ bool
estanOrdenados(s) = vacia (s) $\vee_L$ (esMayor (prim (s), fin (s)) $\wedge_L$ estanOrdenados (fin (s))

esMayor: <nat, nat, secu (nat), secu(<nat, nat, secu (nat)) $\to$ bool
esMayor(t,s) = vacia (s) $\vee_L$ (prioridad (t) < prioridad (fin(s)) $\wedge_L$ esMayor (t, fin(s)))

noHayTareasRepetidas: secu(<nat, nat, secu (nat)) $\to$ bool
noHayTareasRepetidas(s) = sinRepe (dameTareas (s))

noHayPrioridadesRepetidas: secu(<nat, nat, secu (nat)) $\to$ bool
noHayPrioridadesRepetidas(s) = sinRepe (damePrio (s))

noHayOrdenesRepeEnSecu: secu(<nat, nat, secu (nat)) $\to$ bool
noHayOrdenesRepeEnSecu(s) = vacia (s) $\vee_L$ (sinRepe (ordenes(prim(s))) $\wedge_L$ noHayOrdenesRepeEnSecu (fin(s)))

dameTareas: secu(<nat, nat, secu (nat)) $\to$ secu (nat)
dameTareas(s) = **if** vacia? (s) **then** vacia **else** tarea (prim(s)) $\bullet$ dameTareas (fin (s))

damePrio: secu(<nat, nat, secu (nat)) $\to$ secu (nat)
damePrio(s) = **if** vacia? (s) **then** vacia **else** prioridad (prim(s)) $\bullet$ damePrio (fin (s))

sinRepe: secu(α) $\to$ bool
sinRepe(s) = vacia (s) $\vee_L$ $\neg$ esta? (prim (s), fin (s)) $\wedge_L$ sinRepe (fin(s)))

7.4. Funcion de Abstraccion

Abs : estrCola e $\to$ cola (Rep(e))

($\forall$ e: estrCola) Abs(e) = c: cola) / **if** vacia? (e) **then** vacia (c) **else** tarea (prim (e)) = proximo (c) $\wedge$ prioridad (prim(e)) = prioridad (c) $\wedge$ prim (ordenes (prim (e))) = prim (ordenes (c, proximo (c))) $\wedge_L$ sacarOrden (e) = sacarOrden (c)

7.5. Algoritmos

Algoritmo 75

iVacia?(in c: estrCola) → *res*: bool
    res ← vacia?(c)   O(1)
end Function

La operacion vacia? de secu es O(1), por lo tanto esta operación lo es también

Algoritmo 76

iEsta?(in c: estrCola, in t: tarea) → *res*: bool
    var it: secu (priorAct) ← crearIt(c)   O(1)
    res ← false   O(1)
    while (hayMas (it) ∧ ¬ res)   O(c)
        if (tarea(actual (it) == t))   O(1)
            res ← true   O(1)
        endIf
        avanzar (it)   O(1)
    endWhile
end Function

Esta operacion debe recorrer todas las tuplas hasta encontrar la buscada, por lo tanto es O(c) siendo c la longitud de la secuencia

Algoritmo 77

iProxima(in c: estrCola) → *res*: actividad
    res ← actividad(tarea (prim (c), prim (ordenes (prim(c)))))   O(1)
end Function

El acceso al primer elemento de la secuencia es inmediato, por lo tanto es O(1)

Algoritmo 78

iPrioridad(in c: estrCola) → *res*: nat
    res ← prioridad (prim (c))   O(1)
end Function

El acceso al primer elemento de la secuencia es inmediato, por lo tanto es O(1)

Algoritmo 79

iOrdenes(in c: estrCola) → *res*: secu(orden)
    res ← ordenes (prim (c))   O(1)
end Function

El acceso al primer elemento de la secuencia es inmediato, por lo tanto es O(1)

Algoritmo 80

iSacarOrden(inout c: estrCola, in t:tarea, in o :orden)
    var it: secu (priorAct) ← crearIt(c)   O(1)
    while (tarea (actual (it)) ≠ t)   O(c)
        avanzar (it)   O(1)
    endWhile
    var itord: secu (nat) ← crearIt(ordenes(actual))
    while (actual (itord)) ≠ t)   O(c)
        avanzar (it)   O(1)
    endWhile
    borrarActual (itord)   O(c)
    if (vacia? (ordenes(actual(it))))   O(c)
        borrarActual (it)   O(1)
    endIf
end Function

Esta función recorre toda la secuencia de tuplas hasta encontrar la tarea buscada, y una vez encontrada recorre toda la secuencia de ordenes hasta encontrar la orden pasada, en caso de que luego de borrar esa orden la secuencia quede vacia el borrado de la tupla se hace en orden constante, por lo tanto la complejidad total es O(e+o), siendo e la longitud de la cola y o la cantidad de ordenes que contiene la tarea pasada por parámetro

Algoritmo 81

iVacia() → *res*: estrCola
    res ← vacia   O(1)
end Function

La operacion vacia de secu es O(1), por lo tanto esta operación lo es también

Algoritmo 82

iEncolar(inout c: estrCola, in t:tarea, in p:prioridad, in o :orden)
  var it: secu (priorAct) ← crearIt(c) O(1)
  while (hayMas(it) ∧ prioridad(actual (it)) < p) O(c)
    avanzar (it) O(1)
  endWhile
  if (hayMas(it) ∧ prioridad (actual (it)) == p)
    agregarAtras (ordenes (actual (it)), o) O(1)
  else
    var secord: secu (nat) ← vacia O(1)
    agregarAdelanteDeIt (<t, p, agregarAtras (secord, o)>) O(1)
  endIf
end Function

Esta función recorre toda la secuencia de tuplas hasta encontrar una tarea con menor o igual prioridad, en caso de que encuentre una con menor, significa que debe crear un nuevo nodo en O (1) y agregarlo también en tiempo constante, en caso contrario debe agregarAtras sobre la secuencia de órdenes, también O(1), por lo tanto la complejidad es O(c) siendo c la longitud de la cola

Algoritmo 83

iEncolarSecuOrdenada(inout c: estrCola, in s:secu (<actvidad, prioridad>))
  var itsecu: secu (<actvidad, prioridad>) ← crearIt(s) O(1)
  var itcola: secu (priorAct) ← crearIt(c) O(1)
  while (hayMas(itsecu)) < p) O(c)
    while (hayMas(itcola) ∧ prioridad(actual (it)) < π_2 (actual (itsecu))) O(c)
      avanzar (itcola) O(1)
    endWhile
    if (hayMas(itcola) ∧ prioridad (actual (itcola)) == π_2 (actual (itsecu)))
      agregarAtras (ordenes (actual (itcola)), o) O(1)
    else
      var secord: secu (nat) ← vacia O(1)
      agregarAdelanteDeIt (<t, p, agregarAtras (secord, o)>) O(1)
    endIf
  endWhile
end Function

Como sabemos que ambas secuencias estan ordenadas, agregar un elemento a la otra es simplemente recorrer hasta encontrar la posición, y luego es posible seguir agregando el siguiente elemento desde la misma sin necesidad de volver hacia atrás, esto garantiza que se puede hacer la operación con una sola pasada por cada secuencia, como las operaciones de agregar y la creación de iteradores se hace en tiempo constante, en total la operación cuesta O(c+s), siendo c la cola y s la secuencia ordenada a agregar

Algoritmo 84

iCrearIt(in c: estrCola(α)) → res: iterador(α)
  res ← crearIt (c) O(1)
end Function

Algoritmo 85

iHayMas(in it: iterador(α)) → res: bool
  res ← hayMas (it) O(1)
end Function

Algoritmo 86

iActual(in it: iterador(α)) $\rightarrow$ res: α
  res $\leftarrow$ actual (it) O(1)
end Function

Algoritmo 87

iAvanzar(inout it: iterador(α))
  avanzar(it) O(1)
end Function

Algoritmo 88

iBorrarActual(inout it: iterador(α))
  res $\leftarrow$ borrarActual (it) O(1)
end Function