UNIVERSIDAD FRANCISCO DE VITORIA
VINCE IN BONO MALUM

Estructuras de Datos y Algoritmos

Tema 3. Tipos de datos básicos

Prof. Dr. P. Javier Herrera

Contenido

1. Pasos en la implementación de un TAD
2. Lenguaje abstracto de programación imperativa
3. Implementación de conjuntos finitos mediante vectores
4. Implementación de los conjuntos finitos con elementos en 1..N
5. Implementación de los multiconjuntos finitos con elementos en 1..N
6. Estructuras lineales enlazadas

1. Pasos en la implementación de un TAD

- Las implementaciones que vamos a realizar seguirán el **paradigma de programación imperativo**, por lo que para las implementaciones y algoritmos usaremos un *lenguaje abstracto de programación imperativa*.

- **Implementar un TAD** consiste en:
  - Representar sus **valores** por medio de valores de tipos de datos más concretos del lenguaje de programación (tipo representante).
    - Esta representación se oculta al usuario del TAD.
    - Existe mas de una representación posible.
  - Simular sus **operaciones** por medio de funciones o procedimientos que actúan sobre dichos tipos más concretos.

1. Pasos en la implementación de un TAD

- Existen dos tipos de representaciones:
  - **Estáticas**: el tamaño de la estructura no cambia durante la ejecución.
    - Desaprovechamiento de la memoria.
    - Desbordamiento.
  - **Dinámicas**:
    - Utilizan memoria dinámica.
    - Proporcionan estructuras mas versátiles ↔ No tienen un tamaño fijo durante la ejecución.
    - Su programación es más compleja.

- Una parte esencial de la programación de cualquier algoritmo es el estudio de su coste en tiempo y en memoria. En general, nos referiremos al **coste en tiempo en el caso peor**.

2. Lenguaje abstracto de programación imperativa

- Para la implementación de TADs usaremos un lenguaje abstracto de programación imperativa al estilo PASCAL.

- Instrucciones que usaremos :
  - Nada:

    nada { instrucción que no hace nada }

  - Asignación:

    $x := E$ { la variable x tiene que ser del mismo tipo que E}

    $\{x_1, x_2, \ldots, x_n\} := \{E_1, E_2, \ldots, E_n\}$ { asignación múltiple }

  - Secuencia:

    $P_1 ; P_2$

2. Lenguaje abstracto de programación imperativa

- Distinción de casos:

casos

$B_1 \rightarrow P_1$

$B_2 \rightarrow P_2$

...

$B_n \rightarrow P_n$

fcasos

- Condicional:

si B **entonces** P_1 **fsi**

si B **entonces** P_1 **si no** P_2 **fsi**

- Bucle:

mientras B **hacer** P **fmientras**

2. Lenguaje abstracto de programación imperativa

- Bucle con contador:

  para *i* = *inicial* **hasta** *final* **paso** *p* **hacer** *P* **fpara**

  Cuando $p = 1$ omitimos la mención del paso en el bucle.

- Entrada:

  leer (*c*) { lee un carácter del dispositivo de entrada }

- Salida:

  imprimir (mensaje) { imprime un mensaje (cadena de caracteres) en el dispositivo de salida }

- Error:

  error (mensaje) { aborta la ejecución del programa e imprime un mensaje de error }

- Comentarios: se escriben entre llaves en el lugar del programa que convenga.

2. Lenguaje abstracto de programación imperativa

- Tenemos los siguientes tipos y construcciones de tipos básicos:
  - Tipo booleano ***bool*** con valores **cierto**, **falso**, y las operaciones booleanas habituales: $\neg$, $\wedge$, $\vee$. En algunas ocasiones se utiliza la versión de estas dos últimas operaciones con cortocircuito: $\wedge_c$, $\vee_c$.
  - Tipos numéricos ***nat*** (naturales), ***ent*** (enteros) y ***real*** (reales).
  - Tipo de caracteres ***car***.
  - Tipos enumerados $\{\text{valor}_1, \ldots, \text{valor}_k\}$, con $k \geq 1$.
  - Rangos $i..j$ donde i y j son números naturales.
  - Vectores
    - Declaración: $V[i..j]$ **de** $tipo$
    - Asignar todas las posiciones del vector: $V[i..j] := [\text{valor}]$
  - Registros
    - Declaración: **reg** $campo_1 : tipo_1, \ldots, campo_n : tipo_n$ **freg**
    - Acceso y modificación: $R.campo_1$

2. Lenguaje abstracto de programación imperativa

– Punteros

tipos
enlace = **puntero a** *nodo*
nodo = **reg**
valor : *elemento*
sig : *enlace*
freg
estructura = *enlace*
ftipos

- Definición: **puntero a** *nombre-tipo*
- Acceso: siendo p de tipo puntero, p↑.
- Reservar memoria: **reservar** (p)
- Liberar memoria: **liberar** (p)

2. Lenguaje abstracto de programación imperativa

– Funciones: una función se declara con varios parámetros de entrada (o ninguno cuando la función es constante) y al menos un parámetro de salida, cada uno de ellos con su tipo correspondiente.

fun nombre-fun($e_1 : tipo_1, \ldots, e_n : tipo_n$) **dev** $s : tipo'$
var $x_1 : tipo''_1, \ldots, x_k : tipo''_k$
P
ffun

Cuando las funciones tienen más de un parámetro de salida, estos se declaran de la forma: $\{ s_1 : tipo_1, \ldots, s_m : tipo_m \}$, con $m > 1$

En el cuerpo P se pueden usar variables auxiliares locales declaradas tras la cabecera. En general, no hacemos explícitas las declaraciones de variables auxiliares de tipos básicos, tales como, por ejemplo, las variables usadas como contadores en bucles con contador. Pero hay que tener en cuenta que todas las variables que no son parámetros de entrada o salida son variables auxiliares locales, y **nunca hay variables globales**.

2. Lenguaje abstracto de programación imperativa

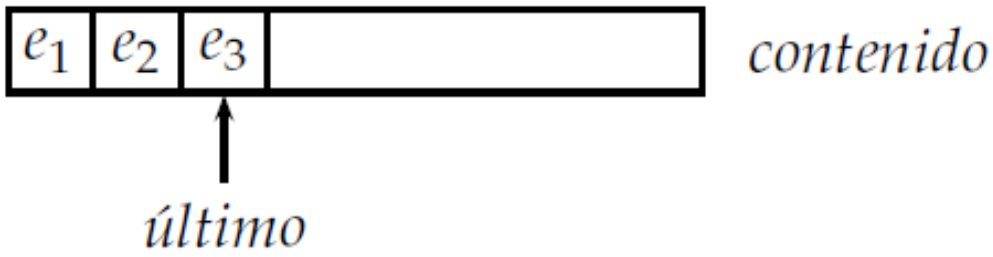
– Procedimientos: pueden tener parámetros de entrada/salida, cuyo valor se puede modificar a lo largo de la ejecución, y parámetros de entrada que no cambian y que se declaran precediéndolos con el símbolo **e**.

proc nombre-proc(**e** $e_1 : tipo_1, \ldots,$ **e** $e_n : tipo_n, es_1 : tipo'_1, \ldots, es_m : tipo'_m$)
var $x_1 : tipo''_1, \ldots, x_k : tipo''_k$
P
fproc

En algunas ocasiones usamos la notación de *precondición* y *poscondición* para dar información sobre la entrada y la salida de un algoritmo.

- La **precondición** es una expresión booleana que expresa las condiciones sobre los parámetros de entrada de un algoritmo que garantizan que la aplicación del algoritmo tiene sentido, además de los tipos.
- La **poscondición** es una expresión booleana que relaciona los parámetros de salida con los de entrada, indicando de esta forma qué cálculo o proceso realiza el algoritmo sobre los datos de entrada.

3. Implementación de conjuntos finitos mediante vectores



tipos

conjunto = **reg**

contenido[1..N] de elemento

último : 0..N

freg

ftipos

- Nótese que un conjunto es vacío si y sólo si el índice *último* vale cero, y que se ignora la información que pueda haber en el vector entre las posiciones *último* + 1 y N.

3. Implementación de conjuntos finitos mediante vectores

- Implementación de las operaciones cjto-vacío, unit y es-cjto-vacío?:

fun cjto-vacío() **dev** *x :* conjunto { $O(1)$ }
    x. último := 0
ffun

fun unit(*e :* elemento) **dev** *x :* conjunto { $O(1)$ }
    x. último := 1
    x.contenido[1] := *e*
ffun

fun es-cjto-vacío?(*x :* conjunto*)* **dev** *b :* bool { $O(1)$ }
    b := (x. último = 0*)*
ffun

3. Implementación de conjuntos finitos mediante vectores

- Implementación de la operación está?.

fun está?(*e :* elemento*, x :* conjunto*)* **dev** *b :* bool { $O(x.último)$ }

b := falso

i := 1

mientras $i \leq x.\ último \wedge \neg b$ **hacer**

b := *(x.contenido*[*i*] = *e)*

i := *i* + 1

fmientras

ffun

- El coste es lineal con respecto al tamaño de la parte ocupada del vector.

Implementación (a): vector con posibles repeticiones

- Implementación de la operación añadir en un vector con posibles repeticiones. Aquí basta con añadir el elemento en la primera posición libre del vector.

proc añadir-a(**e** *e :* elemento*, x :* conjunto*)* { $O(1)$ }
    si *x. último* = *N* ***entonces*** error(Espacio insuficiente)
    si no
        x. último := *x. último* + 1 ;
        x.contenido[*x. último*] := *e*
    fsi
fproc

Implementación (b): vector sin repeticiones

- Implementación de la operación añadir en un vector sin repeticiones. En este caso se añade el elemento en la primera posición libre, pero sólo si el elemento no está ya en el vector.

proc añadir-b(**e** *e :* elemento*, x :* conjunto*)* { $O(x.último)$ }
    si ¬está?(*e, x)* **entonces**
        si *x.último* = *N* **entonces** error(Espacio insuficiente)
        si no
            x.último := *x.último* + 1 ;
            x.contenido[*x.último*] := *e*
        fsi
    fsi
fproc

- El coste es lineal debido a la búsqueda.

Implementación (c): vector ordenado sin repeticiones

- Implementación de la operación añadir en un vector de elementos sin repeticiones y ordenados. Suponemos que el tipo de los elementos sobre el que se construyen los conjuntos admite una **relación de orden total**.

- En este caso, si no está el elemento, se añade en la posición adecuada, para lo cual utilizamos la función búsqueda-binaria y un procedimiento auxiliar desplazar-der que desplaza elementos hacia la derecha del vector.

Implementación (c): vector ordenado sin repeticiones

```
proc añadir-c(e e : elemento, x : conjunto) { O(x.último) }
    {b, n} := búsqueda-binaria(x.contenido, e, 1, x.último)
    si ¬b entonces
        si x.último = N entonces error(Espacio insuficiente)
        si no
            desplazar-der(x.contenido, n, x.último)
            x.contenido[n] := e
            x.último := x.último + 1
        fsi
    fsi
fproc
```

- El coste total es lineal con respecto a *x.último* (logarítmico por la búsqueda y lineal por el desplazamiento)

Implementación (c): vector ordenado sin repeticiones

$\{\ 1 \leq c \leq f+1 < N+1\ \}$
proc desplazar-der($V[1..N]$ **de** elemento, **e** c, f: nat) { $O(f-c+1)$ }
    para $i = f+1$ **hasta** $c+1$ **paso** -1 **hacer**
        $V[i] := V[i-1]$
    fpara
fproc

$\{\ 1 < c \leq f+1 \leq N+1\ \}$
proc desplazar-izq($V[1..N]$ **de** elemento, **e** c, f: nat) { $O(f-c+1)$ }
    para $i = c$ **hasta** f **hacer**
        $V[i-1] := V[i]$
    fpara
fproc

Implementación (a): vector con posibles repeticiones

- Implementación de la operación quitar en un vector con posibles repeticiones. En este caso hay que recorrer el vector siempre hasta el final para asegurarse de quitar todas las posibles repeticiones. Para llenar el hueco que se deja al quitar un elemento, se coloca allí el último elemento.

proc quitar-a(**e** *e :* elemento*, x :* conjunto*)* { $O(x.último)$ }
    $i := 1$
    mientras $i \leq x.último$ **hacer**
        si $x.contenido[i] = e$ **entonces**
            $x.contenido[i] := x.contenido[x.último]$
            $x.último := x.último - 1$
        si no
            $i := i + 1$
        fsi
    fmientras
fproc

Implementación (b): vector sin repeticiones

- Implementación de la operación quitar en un vector sin repeticiones. En este caso podemos parar cuando se encuentra el elemento (o cuando se llega al final si no está). Para llenar el hueco que se deja al quitar un elemento, se coloca allí el último elemento.

proc quitar-b(**e** *e :* elemento*, x :* conjunto*)* { $O(x.último)$ }

$i := 1$

mientras $i \leq x.último \wedge_c x.contenido[i] \neq e$ **hacer**

$i := i + 1$

fmientras

si $i \leq x.último$ **entonces**

$x.contenido[i] := x.contenido[x.último]$;

$x.último := x.último - 1$

fsi

fproc

Implementación (c): vector ordenado sin repeticiones

- Implementación de la operación quitar en un vector de elementos sin repeticiones y ordenados. Invocamos en primer lugar a la función búsqueda-binaria. Cuando el elemento *e* no se encuentra en el vector, no hay que hacer nada. Si elemento *e* se encuentra en la posición *n* indicada por la búsqueda, se elimina llamando a un procedimiento auxiliar desplazar-izq.

proc quitar-c(**e** *e :* elemento*, x :* conjunto*)* { $O(x.último)$ }
    {*b, n*} := búsqueda-binaria(*x.contenido, e,* 1*, x.último*)
    si *b* **entonces**
        desplazar-izq(*x.contenido, n* + 1*, x.último)*
        x.último := *x.último* − 1
    fsi
fproc

- El coste total es lineal debido al desplazamiento que hay que realizar.

Unión, intersección y diferencia

- Una interesante posibilidad para implementar las operaciones unión, intersección y diferencia es hacerlo en términos de cjto-vacío, está? y añadir, de manera que los algoritmos sean comunes para todas las representaciones, aunque el coste en tiempo dependerá del coste de las funciones y procedimientos invocados para cada representación.

```
fun unión(x, y : conjunto) dev z : conjunto
    z := cjto-vacío()
    para i = 1 hasta x.último hacer
        añadir(x.contenido[i], z)
    fpara
    para i = 1 hasta y.último hacer
        añadir(y.contenido[i], z)
    fpara
ffun
```

Unión, intersección y diferencia

```
fun intersección(x, y : conjunto) dev z : conjunto
    z := cjto-vacío()
    para i = 1 hasta x.último hacer
        si está?(x.contenido[i], y) entonces añadir(x.contenido[i], z) fsi
    fpara
ffun

fun diferencia(x, y : conjunto) dev z : conjunto
    z := cjto-vacío()
    para i = 1 hasta x.último hacer
        si ¬está?(x.contenido[i], y) entonces añadir(x.contenido[i], z) fsi
    fpara
ffun
```

Unión, intersección y diferencia

- Implementación (a): **vector con repeticiones**
  - Unión $O(\max(x.último, y.último))$
  - Intersección $O(x.último \cdot y.último)$
  - Diferencia $O(x.último \cdot y.último)$

- Implementación (b): **vector sin repeticiones**
  - Unión $O((x.último + y.último)^2)$
  - Intersección $O(x.último(y.último + x.último))$
  - Diferencia $O(x.último(y.último + x.último))$

- Implementación (c): **vector ordenado sin repeticiones**
  - Unión $O((x.último + y.último)^2)$
  - Intersección $O(x.último(\log(y.último) + x.último))$
  - Diferencia $O(x.último(\log(y.último) + x.último))$

Cardinal de un conjunto

- Calcular el cardinal en la segunda y tercera representaciones es inmediato. En la primera es más complicado debido a las posibles repeticiones, por lo que el coste es cuadrático.

fun cardinal-b/c(x : conjunto) **dev** n : nat { $O(1)$ }
    $n := x.último$
ffun

fun cardinal-a1(x : conjunto) **dev** n : nat { $O(x.último^2)$ }
    $n := 0$
    para $i = 1$ **hasta** $x.último$ **hacer**
        $j := i + 1$
        mientras $j \leq x.último \wedge_c x.contenido[j] \neq x.contenido[i]$ **hacer**
            $j := j + 1$
        fmientras
        si $j > x.último$ **entonces** $n := n + 1$ **fsi**
    fpara
ffun

Cardinal de un conjunto

- Otra posibilidad para la primera representación es utilizar un conjunto auxiliar y en el que solamente guardamos los elementos diferentes.

proc cardinal-a2(x : conjunto, n : nat) { $O(x.último^2)$ }
var y : conjunto
    y := cjto-vacío()
    para i = 1 **hasta** $x.último$ **hacer**
        si ¬está?($x.contenido[i]$, y) **entonces**
            añadir-a($x.contenido[i]$, y)
        fsi
    fpara
    $n := y.último$
    $x := y$
fproc

- Sin embargo, el coste en el caso peor permanece cuadrático.

4. Implementación de los conjuntos finitos con elementos en 1..N

- Al imponer requisitos muy exigentes sobre los elementos, podemos simplificar considerablemente la representación general de los conjuntos.

tipos

conjunto : **vector**[1..N] **de** bool

elemento: 1..N

ftipos

- Para crear el conjunto vacío, todas las posiciones del vector se rellenar con el valor falso.

fun cjto-vacio() **dev** x: conjunto { $O(N)$ }

x[1..N] := [falso]

ffun

4. Implementación de los conjuntos finitos con elementos en 1..N

- Para añadir un elemento, la posición del vector correspondiente a ese elemento se rellena con cierto (si el elemento ya estaba, la información no ha cambiado) y las demás posiciones se dejan igual.

proc añadir(**e** *e*: elemento, *x*: conjunto) { $O(1)$ }
    $x[e]$:= cierto
fproc

- Para quitar un elemento se aplica la misma idea con falso.

proc quitar(**e** *e*: elemento, *x*: conjunto) { $O(1)$ }
    $x[e]$:= falso
fproc

4. Implementación de los conjuntos finitos con elementos en 1..N

- Para construir un vector unitario con el elemento e, podemos hacer x := cjto-vacío(); añadir (e,x). Expandiendo ambos algoritmos se obtiene el siguiente:

fun unit(e: elemento) **dev** x : conjunto { $O(N)$ }
    $x[1..N]$:= [falso]
    $x[e]$:= cierto
ffun

- En esta representación las búsquedas son inmediatas, pues basta acceder directamente a la posición correspondiente del vector.

fun está?(e: elemento, x: conjunto) **dev** b: bool { $O(1)$ }
    $b := x[e]$
ffun

4. Implementación de los conjuntos finitos con elementos en 1..N

- Para reconocer el conjunto vacío ahora es necesario recorrer todo el vector, puesto que la representación no contiene ningún contador que facilite la información relacionada con la cardinalidad del conjunto representado.

fun es-cjto-vacio?(x: conjunto) **dev** b: bool $\quad \{ \ O(N) \ \}$

    i := 1
    b := cierto
    mientras $i \leq N \wedge b$ **hacer**
        $b := \neg\, x[i]$
        $i := i + 1$
    fmientras

ffun

- En el caso mejor el coste es constante, pero en el caso peor el coste es lineal.

4. Implementación de los conjuntos finitos con elementos en 1..N

- Las operaciones de unión, intersección y diferencia tienen una traducción inmediata en términos de operaciones booleanas.

fun unión(x, y : conjunto) **dev** z : conjunto $\{ O(N) \}$
    para $i = 1$ **hasta** N **hacer**
        $z[i] := x[i] \vee y[i]$
    fpara
ffun

fun intersección(x, y : conjunto) **dev** z : conjunto $\{ O(N) \}$
    para $i = 1$ **hasta** N **hacer**
        $z[i] := x[i] \wedge y[i]$
    fpara
ffun

4. Implementación de los conjuntos finitos con elementos en 1..N

fun diferencia(x, y : conjunto) **dev** z : conjunto { $O(N)$ }

    para $i = 1$ **hasta** N **hacer**

        $z[i] := x[i] \wedge \neg y[i]$

    fpara

ffun

4. Implementación de los conjuntos finitos con elementos en 1..N

- Como la representación no contienen ningún contador explícito, para calcular el cardinal del conjunto representado hay que recorrer todo el vector, contando el número de posiciones cuyo valor es cierto.

```
fun cardinal(x : conjunto) dev n: nat      { O(N)}
    n := 0
    para i = 1 hasta N hacer
       si x[i] entonces
          n := n + 1
       fsi
    fpara
ffun
```

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Ahora no interesa saber sólo si un elemento está o no, sino el número de veces que aparece. La representación de un multiconjunto x es un vector $x[1..N]$ de números naturales de forma que $x[i]$ indica la multiplicidad en x del elemento i.

tipos

multiconjunto : **vector**$[1..N]$ **de** nat

elemento : $1..N$

ftipos

- Para crear al multiconjunto vacío, todas las posiciones del vector se rellenan con cero.

fun mcjto-vacio() **dev** x : multiconjunto $\{O(N)\}$

$x[1..N] := [0]$

ffun

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Para añadir un elemento, se incrementa la posición correspondiente.

proc añadir(**e** e : elemento, x : multiconjunto) $\{\ O(1)\ \}$

$x[e] := x[e] + 1$

fproc

- Para calcular la multiplicidad de un elemento dado, basta acceder directamente a la posición correspondiente del vector, pues la representación se basa precisamente en esa idea.

fun multip(e : elemento, x : multiconjunto) **dev** n : nat $\{\ O(1)\ \}$

$n := x[e]$

ffun

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- En cambio, para reconocer el multiconjunto vacío hay que recorrer todo el vector, comprobando que todas las posiciones valen cero.

fun es-mcjto-vacio?(x : multiconjunto) **dev** b : bool $\quad \{ O(N) \}$

$i := 1$

$b :=$ cierto

mientras $i \leq N \wedge b$ **hacer**

$b := (x[i] = 0)$

$i := i + 1$

fmientras

ffun

- En el caso peor el coste es lineal, aunque en el caso mejor es constante.

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Para quitar una aparición de un elemento, basta decrementar el valor de la posición correspondiente, si el elemento está en el multiconjunto, es decir, su multiplicidad es mayor que cero.

proc quitar(**e** e : elemento, x : multiconjunto) $\{ O(1) \}$
    si $x[e] > 0$ **entonces** $x[e] := x[e] - 1$ **fsi**
fproc

- Para borrar todas las apariciones de un elemento en un multiconjunto, basta anular su posición correspondiente.

proc borrar(**e** e : elemento, x : multiconjunto) $\{ O(1) \}$
    $x[e] := 0$
fproc

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Las operaciones de unión, intersección y diferencia tienen una traducción inmediata en operaciones aritméticas.

```
fun unión(x, y : multiconjunto) dev z : multiconjunto          { O(N) }
    para i = 1 hasta N hacer
        z[i] := x[i] + y[i]
    fpara
ffun
```

```
fun intersección(x, y : multiconjunto) dev z : multiconjunto { O(N) }
    para i = 1 hasta N hacer
        z[i] := mín(x[i], y[i])
    fpara
ffun
```

5. Implementación de los multiconjuntos finitos con elementos en 1..N

```
fun diferencia(x, y : multiconjunto) dev z : multiconjunto        { O(N) }
    para i = 1 hasta N hacer
        si x[i] > y[i] entonces
            z[i] := x[i] – y[i]
        si no
            z[i] := 0
    fpara
ffun
```

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Para calcular el cardinal, teniendo en cuenta las multiplicidades de los elementos, se recorre el vector sumando todas las multiplicidades que se van encontrando.

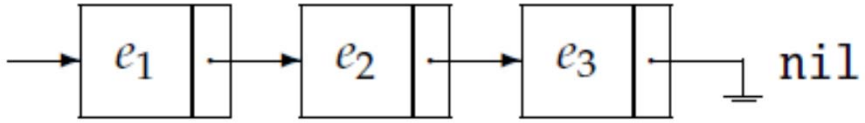
```
fun cardinal(x : multiconjunto) dev z : nat          { O(N) }
    n := 0
    para i = 1 hasta N hacer
        n := n + x[i]
    fpara
ffun
```

5. Implementación de los multiconjuntos finitos con elementos en 1..N

- Para contar solamente los elementos distintos, también se recorre todo el vector, contando ahora el número de posiciones cuyo valor es un número mayor que cero.

fun cardinal-dist(x : multiconjunto) **dev** z : nat { $O(N)$ }
    $n := 0$
    para $i = 1$ **hasta** N **hacer**
        si $x[i] > 0$ **entonces**
            $n := n + 1$
        fsi
    fpara
ffun

6. Estructuras lineales enlazadas



tipos

enlace = **puntero a** nodo

nodo = **reg**

valor : elemento

sig : enlace

freg

estructura = enlace

ftipos

Copia de una estructura lineal enlazada

- La idea del algoritmo copiar es que se construya en q una copia idéntica de la estructura p dada como argumento. Una asignación $q := p$ no tendría este efecto, ya que sólo se haría una copia de punteros, es decir los punteros p y q apuntarían a la misma estructura, pero no habría una duplicación de la memoria ocupada, de forma que q apunte a una estructura idéntica a la que es apuntada por p pero en posiciones de memoria diferentes.

```
fun copiar(p : estructura) dev q : estructura          { Versión recursiva del algoritmo }
var r : estructura
    si p = nil entonces
        q := nil
    si no
        r := copiar(p↑.sig)          { copiar recursivamente el resto de la estructura }
        reservar(q)
        q↑.valor := copiar-elem(p↑.valor)      { copiar elementos }
        q↑.sig := r
    fsi
ffun
```

Copia de una estructura lineal enlazada

```
fun copiar-it(p : estructura) dev q : estructura          { Versión iterativa del algoritmo }
var r, s, t : enlace
    si p = nil entonces q := nil
    si no
        r := p
        reservar(q)
        q↑.valor := copiar-elem(r↑.valor)        { copiar elementos }
        s := q
        mientras r↑.sig ≠ nil hacer
            r := r↑.sig
            reservar(t)                  { s apunta al nodo anterior de t }
            t↑.valor := copiar-elem(r↑.valor)                { copiar elementos }
            s↑.sig := t
            s := t
        fmientras
        s↑.sig := nil
    fsi
ffun
```

Liberación de una estructura lineal enlazada

- El procedimiento anular recorre las posiciones de la estructura liberando uno a uno cada nodo. Por si se diera el caso de que los elementos que se guardan en cada nodo necesitaran ser anulados de forma semejante, se invoca al procedimiento anular-elem que libera la memoria correspondiente.

```
proc anular(p : estructura)
var q : enlace
    mientras p ≠ nil hacer
        q := p
        p := p↑.sig
        anular-elem(q↑.valor) { anular elementos }
        liberar(q)
    fmientras
fproc
```

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003. Capítulo 2

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. Sección 5.8

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Clara Segura, Alberto Verdejo y Yolanda García de la UCM, y la bibliografía anterior)