UNIVERSIDAD
FRANCISCO DE VITORIA
VINCE IN BONO MALUM

Estructuras de Datos y Algoritmos

Tema 4.3. Tipos de datos lineales. Listas

Prof. Dr. P. Javier Herrera

Contenido

- Listas: Conceptos generales
- Operaciones básicas
- Especificación algebraica
- Implementación dinámica
- Extensión de operaciones
- Extensión de la especificación algebraica
- Extensión de la implementación dinámica

Listas: Conceptos generales

- Vivimos rodeados de listas: la lista de la compra, la lista de regalos de boda, la lista de libros para el próximo curso, etc., siendo el denominador común de todas estas listas el establecer una ordenación entre sus elementos.
- Las listas son las **estructuras de datos lineales más flexibles**, puesto que su única característica es imponer un orden entre los elementos almacenados en ellas.
- Según este criterio, las **pilas** y las **colas** serían **casos particulares de listas**, donde se ha restringido la forma de acceso a la estructura.
- No existe una visión unánime de las listas como TAD, con un conjunto básico de operaciones.
- Se trata de un tipo de datos parametrizado, donde el comportamiento de las operaciones sobre los valores del tipo lista es independiente de la naturaleza de los elementos que la componen.

Operaciones básicas

- Un posible TAD de las listas cuenta con las siguientes operaciones:
  - crear la lista vacía,
  - generar una lista unitaria formada por un elemento dado,
  - añadir un elemento por la izquierda,
  - añadir un elemento por la derecha,
  - consultar el elemento más a la izquierda,
  - consultar el elemento más a la derecha,
  - eliminar el elemento más a la izquierda,
  - eliminar el elemento más a la derecha,
  - determinar si una lista es vacía,
  - concatenar dos listas, y
  - calcular la longitud de una lista.

Especificación algebraica

especificación *LISTAS*[*ELEM*]
usa *BOOLEANOS, NATURALES*
tipos *lista*
operaciones

[ ]	:		$\rightarrow$ *lista*	{ constructora }
_ : _	:	*elemento lista*	$\rightarrow$ *lista*	{ constructora }
[_]	:	*elemento*	$\rightarrow$ *lista*	
_ # _	:	*lista elemento*	$\rightarrow$ *lista*	
izquierdo	:	*lista*	$\rightarrow_p$ *elemento*	
elim-izq	:	*lista*	$\rightarrow_p$ *lista*	
derecho	:	*lista*	$\rightarrow_p$ *elemento*	
elim-der	:	*lista*	$\rightarrow_p$ *lista*	
es-lista-vacía?	:	*lista*	$\rightarrow$ *bool*	
_ ++ _	:	*lista lista*	$\rightarrow$ *lista*	
longitud	:	*lista*	$\rightarrow$ *nat*	

Especificación algebraica

variables

$e, f \ : \ elemento$

$x, y, z \ : \ lista$

ecuaciones

$[e] = e : [\ ]$

$x \# e = x ++ [e]$

$\text{izquierdo}([\ ]) = \text{error}$

$\text{izquierdo}(e : x) = e$

$\text{elim-izq}([\ ]) = \text{error}$

$\text{elim-izq}(e : x) = x$

$\text{derecho}([\ ]) = \text{error}$

$\text{derecho}(e : [\ ]) = e$

$\text{derecho}(e : f : x) = \text{derecho}(f : x)$

Especificación algebraica

$$\begin{aligned}
\text{elim-der}([\ ]) &= \text{error} \\
\text{elim-der}(e : [\ ]) &= [\ ] \\
\text{elim-der}(e : f : x) &= e : \text{elim-der}(f : x)
\end{aligned}$$

$$\begin{aligned}
\text{es-lista-vacía?}([\ ]) &= \text{cierto} \\
\text{es-lista-vacía?}(e : x) &= \text{falso}
\end{aligned}$$

$$\begin{aligned}
[\ ] \mathbin{++} y &= y \\
(e : x) \mathbin{++} y &= e : (x \mathbin{++} y)
\end{aligned}$$

$$\begin{aligned}
\text{longitud}([\ ]) &= 0 \\
\text{longitud}(e : x) &= 1 + \text{longitud}(x)
\end{aligned}$$

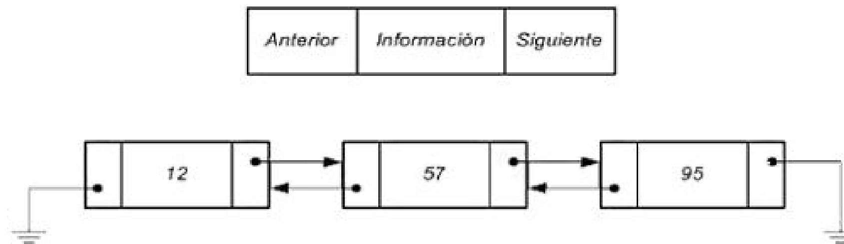
fespecificación

- También podemos considerar como constructoras la operación que crea la lista vacía y la que añade un elemento por la derecha (situación simétrica a la anterior); o bien elegir la lista vacía, la operación de construcción de la lista unitaria y la operación de concatenación.

Implementación dinámica

- Los elementos de una lista se representan mediante una **sucesión de nodos doblemente enlazados**: cada nodo, además del valor del elemento concreto almacenado, contiene enlaces al nodo siguiente y al anterior en la sucesión.

- La ventaja de tener estos dobles enlaces es que así se pueden eliminar elementos por ambos extremos de la lista con un **coste en tiempo constante**.



- Una lista será un registro con un campo que guarda la longitud de la lista y enlaces a los nodos más a la izquierda y más a la derecha de la sucesión.

Implementación dinámica

```
tipos
    enlace-lista     =  puntero a nodo-lista
    nodo-lista       =  reg
                           valor : elemento
                           sig, ant : enlace-lista
                        freg
    lista            =  reg
                           longitud : nat
                           izquierdo, derecho : enlace-lista
                        freg
ftipos
```

Implementación dinámica

- La lista vacía no tiene elementos, por lo que la longitud es cero, y como no hay elementos, los punteros *izquierdo* y *derecho* son nulos.

fun lista-vacía() **dev** l : lista $\{O(1)\}$

l.longitud := 0

l.izquierdo := nil

l.derecho := nil

ffun

- Una lista es vacía cuando los punteros (uno o ambos, pues ambas condiciones son equivalentes) son nulos pues eso significa que no hay nodos en la estructura.

fun es-lista-vacía?(l : lista) **dev** r : bool $\{O(1)\}$

r := (*l.izquierdo* = nil)

ffun

Implementación dinámica

- Añadir un elemento por la izquierda consiste en reservar memoria para un nuevo nodo, almacenar en él el nuevo elemento, y enlazarlo con la estructura, distinguiendo el caso en el que la lista inicial sea vacía. En cualquier caso la longitud de la lista aumenta en una unidad.

proc añadir-izq(**e** *e* : elemento, *l* : lista) $\{O(1)\}$
var *p* : enlace-lista
  reservar(*p*)
  p↑.*valor* := *e* ; *p*↑.*ant* := nil
  si *l*.*izquierdo* = nil **entonces** { *l* es vacía }
    p↑.*sig* := nil ; *l*.*derecho* := *p*
  si no
    p↑.*sig* := *l*.*izquierdo* ; (*l*.*izquierdo*)↑.*ant* := *p*
  fsi
  l.*izquierdo* := *p*
  l.*longitud* := *l*.*longitud* + 1
fproc

Implementación dinámica

- Añadir un elemento por la derecha es la operación simétrica.

proc añadir-der(*l* : lista, **e** *e* : elemento) $\{O(1)\}$

var *p* : enlace-lista

reservar(*p*)

p↑.*valor* := *e* ; *p*↑.*sig* := nil

si *l.derecho* = nil **entonces**

p↑.*ant* := nil ; *l.izquierdo* := *p*

si no

p↑.*ant* := *l.derecho* ; (*l.derecho*)↑.*sig* := *p*

fsi

l.derecho := *p*

l.longitud := *l.longitud* + 1

fproc

Implementación dinámica

- Construir una lista unitaria a partir de un elemento dado consiste en crear una nueva lista con un único nodo que es apuntado tanto por *izquierdo* como por *derecho*. La longitud es 1.

fun unitaria(e : elemento) **dev** l : lista $\{O(1)\}$
var p : enlace-lista
    reservar(p)
    $p\uparrow.valor := e$
    $p\uparrow.ant :=$ nil ; $p\uparrow.sig :=$ nil
    $l.izquierdo := p$; $l.derecho := p$
    $l.longitud := 1$
ffun

Implementación dinámica

- El elemento más a la izquierda de una lista no vacía se obtiene accediendo al campo *valor* del nodo apuntado por *izquierdo*. Si la lista es vacía se produce un mensaje de error.
- La operación derecho es simétrica.

fun izquierdo(*l* : lista) **dev** *e* : elemento $\{O(1)\}$
    si *l.izquierdo* = nil **entonces** error(Lista vacía)
    si no *e* := (*l.izquierdo*)↑.*valor*
    fsi
ffun

fun derecho(*l* : lista) **dev** *e* : elemento $\{O(1)\}$
    si *l.derecho* = nil **entonces** error(Lista vacía)
    si no *e* := (*l.derecho*)↑.*valor*
    fsi
ffun

Implementación dinámica

- Para eliminar el izquierdo se libera el primer nodo de la estructura, dejándola bien enlazada. Las operaciones derecho y elim-der son simétricas a las dos anteriores.

proc elim-izq(*l* : lista) $\{O(1)\}$
var *p* : enlace-lista
  si *l.izquierdo* = nil **entonces** error(Lista vacía)
  si no
    p := *l.izquierdo* ; *l.izquierdo* := *p*↑.*sig*
    si *l.izquierdo* = nil **entonces** *l.derecho* := nil
    si no (*l.izquierdo*)↑.*ant* := nil
    fsi
    l.longitud := *l.longitud* − 1
    liberar(*p*)
  fsi
fproc

Implementación dinámica

- La operación elim-der es simétrica.

proc elim-der(l : lista) $\{O(1)\}$
var p : enlace-lista
  si $l.derecho$ = nil **entonces** error(Lista vacía)
  si no
    $p := l.derecho$; $l.derecho := p\uparrow.ant$
    si $l.derecho$ = nil **entonces** $l.izquierdo$:= nil
    si no $(l.derecho)\uparrow.sig$:= nil
    fsi
    $l.longitud := l.longitud - 1$
    liberar(p)
  fsi
fproc

Implementación dinámica

- La lista resultado de concatenar las dos listas que recibe como argumento, comienza siendo una copia de la primera lista. A continuación se recorre la sucesión de nodos de la segunda lista, añadiendo cada elemento por la derecha a la lista resultado. El coste en tiempo de concatenar es lineal respecto a la suma de las longitudes de las listas concatenadas.

```
fun concatenar(x, y : lista) dev z : lista        {O(longitud(x)+longitud(y))}
var p : enlace-lista
    z := lista-vacía() ; p := x.izquierdo
    mientras p ≠ nil hacer
        añadir-der(z, p↑.valor) ; p := p↑.sig
    fmientras
    p := y.izquierdo
    mientras p ≠ nil hacer
        añadir-der(z, p↑.valor) ; p := p↑.sig
    fmientras
ffun
```

Implementación dinámica

- Si se implementara la operación concatenar mediante un procedimiento, se podría conseguir un coste constante modificando el primer argumento para que también fuera el resultado, para lo cual bastaría encadenar el nodo *derecho* de x con el *izquierdo* de y, y sumar las dos longitudes.

- La longitud de una lista se obtiene consultando el campo correspondiente, con un coste en tiempo constante.

fun longitud(l : lista) **dev** n : nat $\{O(1)\}$

$n := l.longitud$

ffun

Extensión de operaciones

- Extender la especificación para considerar las siguientes operaciones:
  - determinar si un elemento aparece en una lista,
  - localizar la posición de un elemento (devolviendo 0 si el elemento no está),
  - calcular el número de repeticiones de un elemento,
  - eliminar todas las apariciones de un elemento,
  - determinar si dos listas son iguales,
  - determinar si una lista es capicúa,
  - consultar el elemento en una posición,
  - insertar un elemento en una posición,
  - eliminar el elemento en una posición, y
  - modificar el elemento en una posición.

Extensión de la especificación algebraica

especificación *LISTAS*+[*ELEM*=]

usa *LISTAS*[*ELEM*=], *NATURALES, BOOLEANOS*

operaciones

está?	:	*elemento lista*	$\rightarrow$	*bool*
posición	:	*elemento lista*	$\rightarrow$	*nat*
repeticiones	:	*elemento lista*	$\rightarrow$	*nat*
eliminar	:	*elemento lista*	$\rightarrow$	*lista*
_ == _	:	*lista lista*	$\rightarrow$	*bool*
es-capicúa?	:	*lista*	$\rightarrow$	*bool*
_ [_]	:	*lista nat*	$\rightarrow_p$	*elemento*
insertar	:	*lista nat elemento*	$\rightarrow_p$	*lista*
eliminar	:	*lista nat*	$\rightarrow_p$	*lista*
modificar	:	*lista nat elemento*	$\rightarrow_p$	*lista*

variables

e, f : *elemento* ; x, y, z : *lista* ; i : *nat*

Extensión de la especificación algebraica

ecuaciones

está?(e, [ ])	=	falso
está?(f, e : x)	=	f == e $\vee$ está?(f, x)
posición(e, x)	=	0 $\Leftarrow$ $\neg$está?(e, x)
posición(e, e : x)	=	1
posición(e, f : x)	=	1 + posición(e, x) $\Leftarrow$ e $\neq f$ $\wedge$ está?(e, x)
repeticiones(e, [ ])	=	0
repeticiones(e, e : x)	=	1 + repeticiones(e, x)
repeticiones(e, f : x)	=	repeticiones(e, x) $\Leftarrow$ e $\neq f$

Extensión de la especificación algebraica

$$
\begin{aligned}
\text{eliminar}(e, [\ ]) &= [\ ] \\
\text{eliminar}(e, e : x) &= \text{eliminar}(e, x) \\
\text{eliminar}(e, f : x) &= f : \text{eliminar}(e, x) \Leftarrow \text{e} \neq f
\end{aligned}
$$

$$
\begin{aligned}
[\ ] == [\ ] &= \text{cierto} \\
[\ ] == e : x &= \text{falso} \\
e : x == [\ ] &= \text{falso} \\
e : x == f : y &= e == f \ \land \ x == y
\end{aligned}
$$

$$
\begin{aligned}
\text{es-capicúa?}([\ ]) &= \text{cierto} \\
\text{es-capicúa?}(e : [\ ]) &= \text{cierto} \\
\text{es-capicúa?}(e : x) &= e == \text{derecho}(x) \land \text{es-capicúa?}(\text{elim-der}(x)) \Leftarrow \neg\text{es-lista-vacía?}(x)
\end{aligned}
$$

Extensión de la especificación algebraica

$$x[i] = \text{error} \Leftarrow i == 0 \lor i > \text{longitud}(x)$$
$$(e : x)[1] = e$$
$$(e : x)[i] = x[i - 1] \Leftarrow 1 < i \land i \leq \text{longitud}(e : x)$$

$$\text{insertar}(x, i, e) = \text{error} \Leftarrow i == 0 \lor i > \text{longitud}(x) + 1$$
$$\text{insertar}(x, 1, e) = e : x$$
$$\text{insertar}(e : x, i, f) = \text{e} : \text{insertar}(x, i - 1, f) \Leftarrow 1 < i \land i \leq \text{longitud}(e : x) + 1$$

$$\text{eliminar}(x, i) = \text{error} \Leftarrow i == 0 \lor i > \text{longitud}(x)$$
$$\text{eliminar}(e : x, 1) = x$$
$$\text{eliminar}(e : x, i) = e : \text{eliminar}(x, i - 1) \Leftarrow 1 < i \land i \leq \text{longitud}(e : x)$$

$$\text{modificar}(x, i, f) = \text{insertar}(\text{eliminar}(x, i), i, f)$$

fespecificación

Extensión de la implementación dinámica

- La declaración de tipos es la misma.

- Para averiguar si un elemento está en una lista se recorre la estructura de nodos enlazados hasta que se encuentre el elemento o se acabe la estructura. En cada nodo del recorrido se consulta si el elemento en el campo *valor* es igual al elemento *e* dado.

fun está?(e : elemento, l : lista) **dev** b : bool $\{ O(\text{longitud}(l)) \}$
var p : enlace-lista
    b := falso
    p := $l.izquierdo$
    mientras $p \neq$ nil $\land \neg b$ **hacer**
        b := $(e = p\uparrow.valor)$
        p := $p\uparrow.sig$
    fmientras
ffun

Extensión de la implementación dinámica

- La función para localizar un elemento es muy similar. Al recorrer la lista vamos guardando en una variable auxiliar i la posición del elemento en el que nos encontramos. Si el valor apuntado es igual al que buscamos, asignamos el valor de i al parámetro de salida n.

fun posición(e : elemento, l : lista) **dev** n : nat $\{O(\text{longitud}(l))\}$
var p :enlace-lista
  $n := 0$; $i := 0$; $p := l.izquierdo$
  mientras $p \neq \text{nil} \wedge n = 0$ **hacer**
    si $e = p\uparrow.valor$ **entonces**
      $n := i$
    fsi
    $p := p\uparrow.sig$
    $i := i + 1$
  fmientras
ffun

Extensión de la implementación dinámica

- Para contar el número de apariciones de un elemento hay que recorrer toda la lista.

fun repeticiones(e : elemento, l : lista) **dev** n : nat $\{O(\text{longitud}(l))\}$
var p : enlace-lista
  $n := 0$
  $p := l.izquierdo$
  mientras $p \neq$ nil **hacer**
    si $e = p\uparrow.valor$ **entonces**
      $n := n + 1$
    fsi
    $p := p\uparrow.sig$
  fmientras
ffun

Extensión de la implementación dinámica

- Para eliminar todas las apariciones de un elemento se recorre la estructura de nodos enlazados. Cuando se encuentra un nodo con un elemento igual al que hay que eliminar, hay que modificar todos los enlaces para evitar dicho nodo que se libera, tratando apropiadamente los casos en que éste es el primero o el último. La longitud va decreciendo a medida que se eliminan nodos.

proc eliminar(e : elemento, l : lista) $\{O(\text{longitud}(l))\}$
var p, q : enlace-lista
  $p := l.izquierdo$
  mientras $p \neq \text{nil}$ **hacer**
    si $p\uparrow.valor = e$ **entonces**
      si $p = l.izquierdo$ **entonces**
        $l.izquierdo := p\uparrow.sig$
        $p\uparrow.sig\uparrow.ant := \text{nil}$
      si no
        $p\uparrow.ant\uparrow.sig := p\uparrow.sig$
      fsi

Extensión de la implementación dinámica

```
        si p = l.derecho entonces
            l.derecho := p↑.ant
            p↑.ant↑.sig := nil
        si no
            p↑.sig↑.ant := p↑.ant
        fsi
        q := p
        p := p↑.sig
        liberar(q)
        l.longitud := l.longitud – 1
    si no
        p := p↑.sig
    fsi
  fmientras
fproc
```

Extensión de la implementación dinámica

- Para determinar si dos listas son iguales, primero se comprueba si tienen la misma longitud. Si es así, se utilizan dos punteros auxiliares p y q para recorrer cada una de las listas, comprobando si los elementos apuntados por ambos son iguales.

```
fun igual(x, y : lista) dev b : bool                {O(longitud(x))}
var p, q : enlace-lista
    si  x.longitud ≠ y.longitud  entonces
       b := falso
    si no
       b := cierto
       p := x.izquierdo  ;  q := y.izquierdo
       mientras p ≠ nil  ∧  b  hacer
          b := (p↑.valor = q↑.valor)
          p := p↑.sig  ;  q := q↑.sig
       fmientras
    fsi
ffun
```

Extensión de la implementación dinámica

- Para comprobar si una lista es capicúa utilizamos dos enlaces: p recorrerá los elementos de la lista de izquierda a derecha, y q recorrerá los elementos de derecha a izquierda. Ya que no se puede comprobar fácilmente cuando se cruzan los punteros en su recorrido, utilizamos una variable m para contar cuantos elementos hay desde p hasta q (ambos incluidos).

```
fun es-capicúa?(l : lista) dev b : bool        {O(longitud(l))}
var p, q : enlace-lista
    b := cierto
    p := l.izquierdo  ;  q := l.derecho
    m := l.longitud
    mientras m > 1 ∧ b hacer
        b := (p↑.valor = q↑.valor)
        p := p↑.sig  ;  q := q↑.ant
        m := m – 2
    fmientras
ffun
```

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003. Capítulo 5

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. Capítulo 6

(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)