6. Listas Generalizadas

Una lista enlazada es una estructura de datos en la cual los objetos están organizados linealmente. Listas enlazadas proveen una forma simple de estructuar conjuntos dinámicos de datos. Una lista puede tener diferentes formas, puede ser enlazadas simple, doblemente enlazada, ordenada o circular.

Una lista generalizada, A, es una secuencia finita de $n \geq 0$ elementos, $\alpha_1, \ldots, \alpha_n$, donde α_i es un elemento átomo o una lista. Si es una lista, se llama sublista de A. Si $n \geq 1$, entonces α_1 se dice la cabeza de A y el resto la cola.

Ejemplos: A = (a,(b,c)) B = (A,A,()) C = (a,C)

Consecuencias: Listas generalizadas pueden ser compartidas o pueden ser recursivas.

structure *LIST(ITEM)*

declare

$CREATE() \rightarrow list$
$ADD\_ITEM(list, item) \rightarrow list$
$ADD\_LIST(list, list) \rightarrow list$
$DELETE_ITEM(list, item) \rightarrow list$
$DELETE_LIST(list, list) \rightarrow list$
$HEAD\_ITEM(list) \rightarrow item$
$HEAD\_LIST(list) \rightarrow list$
$IS\_EMPTY(list) \rightarrow boolean$
$EQUAL(list, list) \rightarrow boolean$

for all $l, l_1, l_2, l_3 \in LIST, i, i_1 \in ITEM$ **let**

$IS\_EMPTY(CREATE()) ::= true$
$IS\_EMPTY(ADD\_ITEM(l, i)) ::= false$
$IS\_EMPTY(ADD\_LIST(l, l_1)) ::= false$
$EQUAL(CREATE(), l) ::= IS\_EMPTY(l)$
$EQUAL(l, CREATE()) ::= IS\_EMPTY(l)$
$EQUAL(ADD\_ITEM(l, i), ADD\_ITEM(l_1, i_1) ::=$ if $i = i_1$ then $EQUAL(l, l_1)$ else $false$
$EQUAL(ADD\_ITEM(l, i), ADD\_LIST(l_1, l_2) ::= false$
$EQUAL(ADD\_LIST(l, l_1), ADD\_ITEM(l_2, i)) ::= false$
$EQUAL(ADD\_LIST(l, l_1), ADD\_LIST(l_2, l_3)) ::= EQUAL(l, l_2) \wedge EQUAL(l_1, l_3)$
$HEAD\_ITEM(CREATE()) ::= error$
$HEAD\_ITEM(ADD\_ITEM(l, i)) ::=$ if $IS\_EMPTY(l)$ then i else $HEAD\_ITEM(l)$
$HEAD\_ITEM(ADD\_LIST(l, l_1)) ::=$ if $IS\_EMPTY(l)$ then $error$ else $HEAD\_ITEM(l)$
$HEAD\_LIST(CREATE()) ::= error$
$HEAD\_LIST(ADD\_ITEM(l, i)) ::=$ if $IS\_EMPTY(l)$ then $error$ else $HEAD\_LIST(l)$
$HEAD\_LIST(ADD\_LIST(l, l_1)) ::=$ if $IS\_EMPTY(l)$ then l_1 else $HEAD\_LIST(l)$
$DELETE\_ITEM(CREATE(), i) ::= CREATE()$
$DELETE\_ITEM(ADD\_ITEM(l, i), i_1) ::=$ if $i = i_1$ then $DELETE\_ITEM(l, i_1)$
else $ADD\_ITEM(DELETE(l, i_1), i)$
$DELETE\_ITEM(ADD\_LIST(l, l_1), i) ::=$
$ADD\_LIST(DELETE\_ITEM(l, i), DELETE\_ITEM(l_1, i))$
$DELETE\_LIST(CREATE(), l) ::= CREATE()$
$DELETE\_LIST(ADD\_ITEM(l, i), l_1) ::= ADD\_ITEM(DELETE\_LIST(l, l_1), i)$
$DELETE\_LIST(ADD\_LIST(l, l_1), l_2) ::=$ if $EQUAL(l_1, l_2)$ then $DELETE\_LIST(l, l_2)$
else $ADD\_LIST(DELETE\_LIST(l, l_2), DELETE\_LIST(l_1, l_2))$

Supongamos una representación por punteros de modo que se tiene un nodo con tres campos:

- *tag* (0: atomo, 1: sublista)
- data (Si $tag = 0$, data es un dato que se almacena, and si $tag = 1$, data es un puntero a la sublista).
- puntero al siguiente elemento de la lista.

Con esta representación, las lista dadas como ejemplo quedan representadas como:

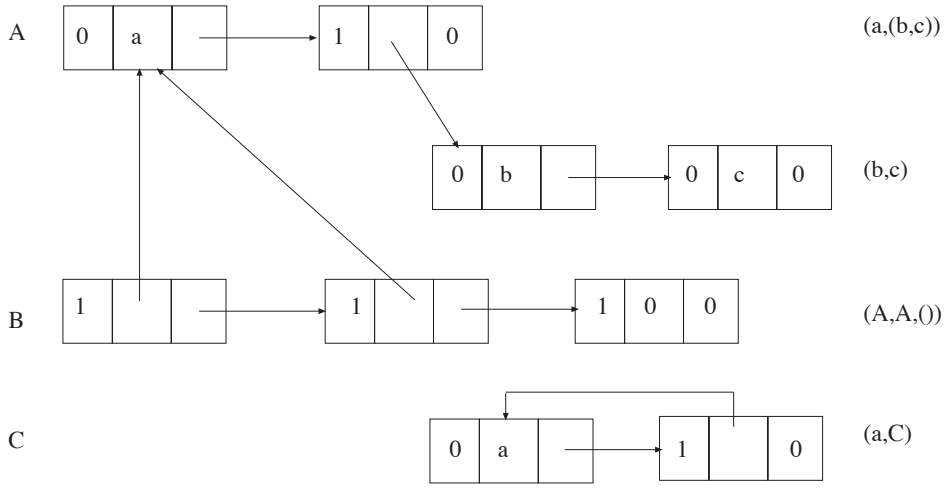

Figura 8: Ejemplo de listas generalizadas

Un ejemplo de aplicación de lista generalizadas en la representación de un polinomio. Ejemplo:

$$\begin{aligned} P(x,y,z) &= x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz \\ & \quad z^2(x^{10}y^3 + 2x^8y^3 + 3x^8y^2) + z(x^4y^4 + 6x^3y^4 + 2y) \\ & \quad z^2(y^3(x^{10} + 2x^8) + y^2(3x^8)) + z(y^4(x^4 + 6x^3) + y(2x^0)) \end{aligned}$$

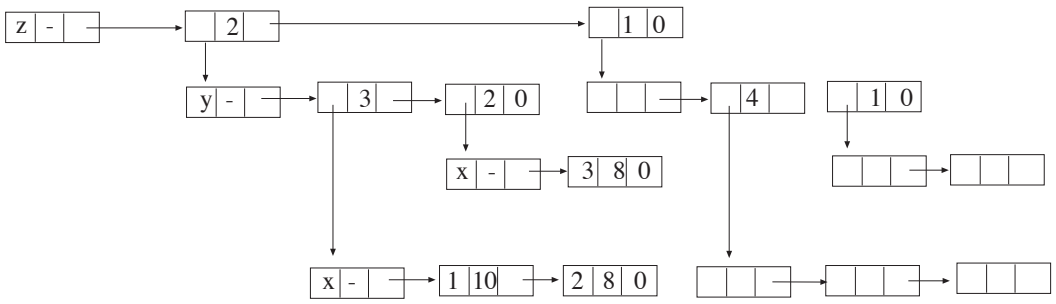

Figura 9: Ejemplo de un polinomio como lista

¿ Cuál es la representación de los nodos de esta lista?

El hecho que listas puedan ser compartidas puede resultar en un mejor manejo de espacio. Para hacer esto, extenderemos la definición de una lista para permitir el manejo de nombres de sublistas. Por ejemplo, en la lista A=(a,(b,c)), la sublista (b,c) puede ser llamada Z de manera que A = (a,Z(b,c)), o en forma más consistente A(a,Z(b,c)).

Listas que son compartidas por otras, tal como A en los ejemplos, crea problemas cuando uno desea agregar o borrar un nodo que está al frente. Si el primer nodo de A es borrado, es necesario mover los punteros de la lista B al segundo elemento de A. El problema también aparece cuando se agrega un nodo al comienzo. Aún más, no siempre se conocen todas las listas que comparten una definición. Una solución fácil es agregar un nodo cabecera. El *tag* de la cabecera no importa y es inicializado con 2. Adicionalmente al uso de la cabecera para indicar el comienzo de una lista, esta cabecera puede contener un contador indicando el número de listas que hacen referencia a ella. Así cuando uno desea eliminar una lista, esta lista sólo puede ser eliminada si es que no hay listas que hagan referencia a ella.

A continuación se presenta el dibujo de la representación con cabecera de las lista de ejemplo:

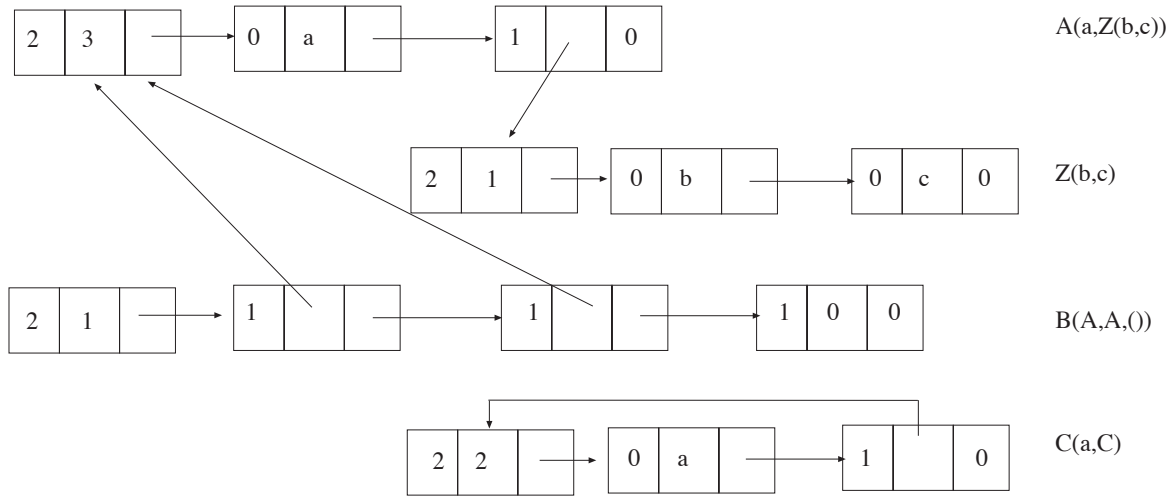

Figura 10: Ejemplo de lista generalizadas con encabezamiento

El uso del contador resuelve el problema de listas compartidas, pero cuando existen listas con ciclos, el contador nunca llega a cero. Por ejemplo:

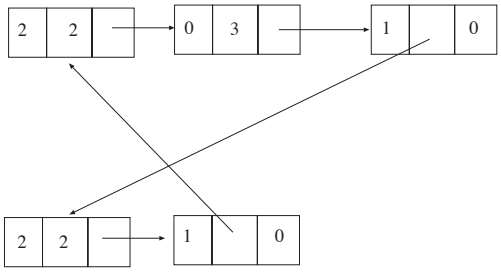

Figura 11: Listas con ciclos

Desafortunadamente, cuando se usan listas recursivas (con ciclos) es posible que se acabe el espacio disponible aunque no todos los nodos estén siendo usados, ya que no se pueden devolver los nodos a medida que se van borrando. Cuando esto ocurre, es posible recolectar el espacio sin uso mediante un proceso conocido como colección de basura o *garbage collection*.

6.1. Aplicación de Listas: Recolección de Basura y Compactación

En un ambiente de computadores multi-procesos, muchos programas residen en memoria al mismo tiempo. Diferentes programas tiene diferentes requerimientos de memoria. Así un programa puede requerrir 60K o 300K de memoria. En cualquier momento que el sistema necesite memoria, necesita localizar memoria continua del tamaño deseado. Cuando la ejecución de un programa es terminada, la memoria que ha sido ocupada debe ser liberada y disponible para otro programa. Más aún, bloques de memoria pueden ser liberados en una secuencia diferente a la que fueron solicitados. Suponga por ejemplo el siguiente caso:

Se tiene una memoria de 100000 palabras y 5 programasa P1, P2, P3, P4, y P5 que requieren bloques de memoria de tamaños 10000, 15000, 6000, 8000 y 20000. La vista de la memoria al localizar el programa P5 sería:

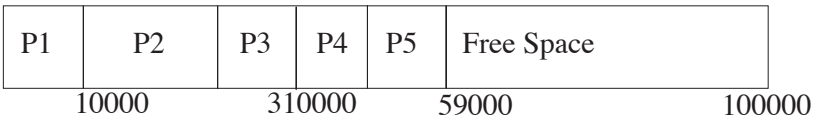

Figura 12: Memoria con 5 procesos en ejecución

Asuma ahora que el programa P4 y P2 completaron su ejecución y que por lo tanto la memoria es liberada:

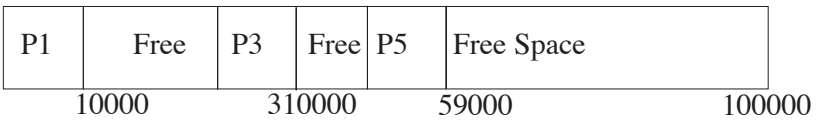

Figura 13: Memoria con 2 procesos terminados

Ahora se tienen 3 bloques de memoria contigua libres y tres ocupados. Con el propósito de seguir permitiendo la localización de memoria, es necesario mantener registro de los bloques que no están en uso. Para eso se usa un lista de bloques de memoria que no están en uso.

El proceso de recolección de basura es llevado a cabo en dos pasos: (1) marcación de los bloques que están en uso y (2) compactación de los bloques libres a memoria disponible. Este segundo paso es trivial si los bloques son de igual tamaño llegando a ser de $O(n)$, siendo n el número de nodos. Cuando existe un tamaño variable por nodo, es deseable compactar la memoria de manera que todos los nodos libres queden en memoria contigua, lo que se llama compactación de memoria. La compactación de memoria reduce el tiempo de recuperación.

6.2. Representation de listas

Considere el caso simple de listas con elementos de un sólo tipo:

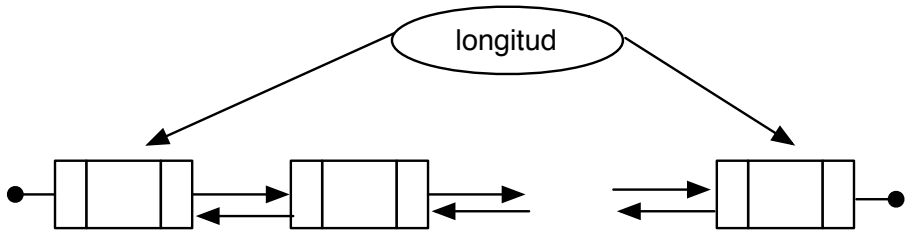

Figura 14: Lista doblemente encadenada

La representación de esta lista es:

Tipos
nodo_lista = **reg**
valor : *elemento*
sig, *ant* : **puntero a** *nodo_lista*
freg

lista = **reg**
longitud : *nat*
izquierda, *derecha* : **puntero a** *nodo_lista*
freg

Algunas algoritmos son los siguientes. En todos los casos la complejidad es de orden lineal $O(n)$ con n siendo el numero de elementos en la lista.

Procedure *concatenar*(*lista x*; *lista y*; `var` *lista z*)
$z \leftarrow lista\_vacia()$
$p \leftarrow x.izquierda$
while $p \neq nil$ **do**
$agregar\_der(z, p \rightarrow valor)$
$p \leftarrow p \rightarrow sig$
end while
$p \leftarrow y.izquierda$
while $p \neq nil$ **do**
$agregar\_der(z, p \rightarrow valor)$
$p \leftarrow p \rightarrow sig$
end while
end

Procedure *inversa*(*lista l*)
    $p \leftarrow l \rightarrow derecha$
    $r \leftarrow nil$
    while $p \neq nil$ **do**
        $q \leftarrow p \rightarrow sig$
        $p \rightarrow sig \leftarrow r$
        $p \rightarrow ant \leftarrow q$
        $r \leftarrow p$
        $p \leftarrow q$
    end while
    $l \leftarrow r$
    end

Func *buscar*(*lista l*, *elem e*)
    $p \leftarrow l \rightarrow derecha$
    while $p \neq nil$ and $p \rightarrow valor \neq e$ **do**
        $p \leftarrow p \rightarrow sig$
    end while
    if $p \neq nil$ **then return** false
    else return true
    end

Ahora considere listas doblemente encadenadas y, además, circulares:

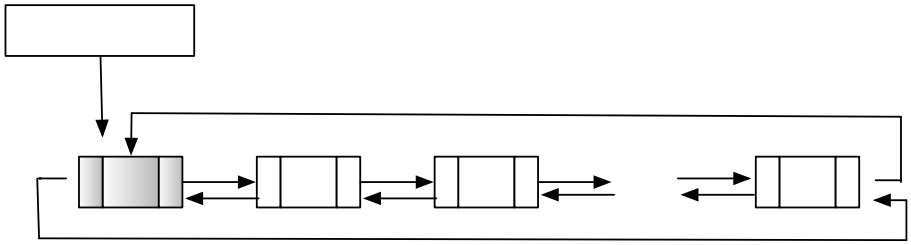

Figura 15: Lista doblemente encadenada y circular

La representación de esta lista es por punteros es:

Tipos
nodo_lista = **reg**
    valor : *elemento*
    sig, *ant* : **puntero a** *nodo_lista*
    freg

lista = **reg**
    longitud : *nat*
    sentinela : **puntero a** *nodo_lista*
    freg

En esta representación un nodo es usado como cabecera o sentinela de la lista de nodos. Los algoritmos soy muy parecidos a los anteriores, teniendo en cuando que los punteros siguientes y anteriores nunca son nulos. La condición de parada es que el puntero sea igual a la cabecera o

sentinela.

Otra representación de lista es usando arreglos. Se pueden usar 3 arreglos para lista doblemente encadenadas, un arreglo de punteros anteriores, un arreglo de puntero al siguiente y un arreglo de valores, tal como la figura que se muestra:

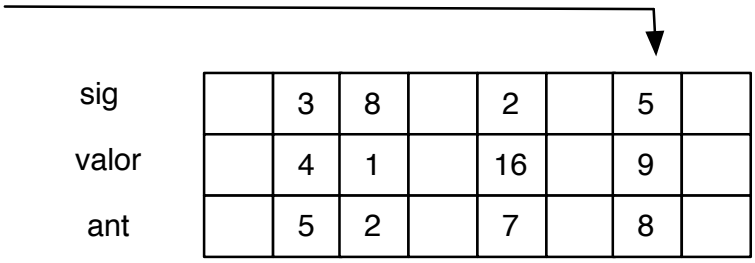

Figura 16: Lista doblemente encadenada y circular con representación por arreglos

Considerando la memoria como un arreglo de una dimensión, otra representación en un sólo arreglo es:

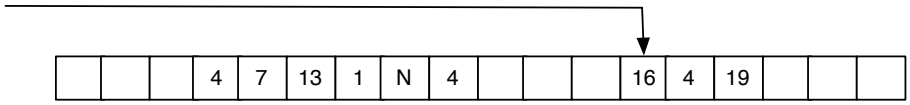

Figura 17: Lista doblemente encadenada y circular con representación por arreglos

En esta representación se maneja un desplazamiento de 0, 1 y 2, para el valor, el próximo y el anterior, respectivamente.

Asumiendo esta represnetación, algunas preguntas interesantes son:

- ?Qué estructuras o variables son necesarias para manejar la lista y usar asignación dinámica de memoria?
- Basada en esta representación, ?qué debería hacerse para manejar la lista en forma compactada, osea, usando los primeros elementos del arreglo?

Para el caso de listas generales:

Tipos

nodo_lista_gen = **reg**
tipo : *bool*
valor : *elemento*
lista : *lista_gen*
sig : **puntero a** *nodo_lista*
freg

lista_gen = **reg**
referencias : *natural*
primero : **puntero a** *nodo_lista_gen*
freg

Procedure *concatenar*(*lista_gen x*; *lista_gen y*; `var` *lista_gen z*)
$v \leftarrow copia\_lista\_gen(x)$
$w \leftarrow copia\_lista\_gen(y)$
if $v = nil$ **then** $z \leftarrow w$
else
[avanzamos hasta el último elemtno de v]
$p \leftarrow v \rightarrow primero$
while $p \rightarrow sig \neq nil$ **do**
$p \leftarrow p \rightarrow sig$
end while
[y lo unimos con w]
$p \rightarrow sig \leftarrow w$
$z \leftarrow v$
end

Procedure *inversa_gen*(*lista_gen l*)
$p \leftarrow l \rightarrow primero$
$r \leftarrow nil$
while $p \neq nil$ **do**
$q \leftarrow p$
$p \leftarrow p \rightarrow sig$
if $\neg(q \rightarrow tipo)$ **then** $inversa\_gen(q \rightarrow lista)$
$q \rightarrow sig \leftarrow r$
$r \leftarrow q$
end while
$l \leftarrow r$
end

```
Func buscar_gen( lista_gen l, elem e)
    p ← l → primero
    while p ≠ nil do
        if p → tipo =' L' then
            if buscar_gen(p → lista, e) then return true
            else p ← p → sig
        else
            if p → valor = e then return true
            else p ← p → sig
    end while
    return false
    end
```

Dos algoritmos de profundidad son:

```
Func depth1( node_list l)
    p ← l
    h1 ← 0
    h2 ← 0
    if p = nil then return h1
    h1 ← 1 + depth1(p → sig)
    if p → tipo =' L' then
        h2 ← 1 + depth1(p → lista → primero)
   return max(h1, h2)
    end

Func depth2( lista_gen l)
    p ← l → primero
    n ← 0
    long ← 0
    while p ≠ nil do
        if p → tipo =' L' then
            n ← max(n, depth2(p → lista))
        long ← long + 1
        p ← p → sig
    end while
    return max(n, long)
    end
```