LISTAS

2002

GRUPO # 22

Alumnos:

Aguilar Elba
Barrios Miguel
Camacho Yaquelin
Ponce Rodríguez Jhonny

ESTRUCTURAS DE DATOS

TEMA 5

Estructura de datos Lista

ÍNDICE

BIBLIOGRAFÍA

- (Joyanes y Zahonero, 1998), Cap. 5 y 6.
- (Joyanes y Zahonero, 1999), Cap. 4.
- (Dale y Lilly, 1989), Cap. 6 y 7.
- (Horowitz y Sahni, 1994).

OBJETIVOS

- Conocer el concepto, funcionamiento y utilidad del tipo Lista.
- Conocer el TAD Lista y sus operaciones asociadas.
- Saber implementar una lista mediante variables estáticas.
- Saber implementar una lista mediante variables dinámicas. Conocer distintas implementaciones dinámicas del tipo, utilizando simple y doble enlace.
- Conocer el TAD Lista ordenada y su implementación mediante variables dinámicas.
- Conocer el concepto y utilidad de las multilistas.

5.1. Definición

Al igual que ocurría en el caso de la pila y la cola, estamos bastante familiarizados con el concepto de lista. Si pensamos por ejemplo en una lista de la compra, o en una guía telefónica, estamos manejando listas. En el primero de los casos, los elementos no tienen porque estar ordenados por su valor, mientras en el segundo se ordenan según un índice alfabético. No obstante, en ambos casos estamos hablando de listas de elementos.

En el caso de las estructuras de datos, definiremos una lista del siguiente modo:

Definición

Una lista es un conjunto ordenado de elementos homogéneos en la que no hay restricciones de acceso, la introducción y borrado de elementos puede realizarse en cualquier posición de la misma.

Junto con la pila y la cola, la lista forma parte de lo que se denominan estructuras lineales. Intuitivamente son estructuras en los que los distintos elementos se sitúan en línea, de ahí su nombre. Dicho de otro modo, cada elemento de una estructura lineal, salvo el primero y el último, tan sólo tiene un anterior y un siguiente. La diferencia entre estos tres tipos de estructuras lineales radica en las restricciones impuestas en el acceso a los elementos.

- En una pila los elementos tan sólo pueden añadirse y borrarse por uno de sus extremos.
- En una cola, los elementos se añaden por un extremo y se borran por el otro.
- En una lista, los elementos pueden añadirse y borrarse en cualquier posición, no sólo en los extremos.

De un modo más formal, una lista es un triplete (P,R,v) en donde:

- P: es un conjunto de posiciones, en particular un conjunto de posiciones que contienen datos en la lista.
- R: es una relación de orden sobre el conjunto P. Los elementos se sitúan en línea (orden), no se ordenan sus valores.
- v: es una función de evaluación, que para cada posición P define el valor del dato almacenada en la misma.

5.2. El TAD Lista

En este apartado vamos a ver una posible definición del TAD Lista. En este caso estaremos hablando de una lista general en la que sus elementos no tienen porque estar ordenados por su valor. Por otro lado, es necesario decir que existen muchas formas de definir una lista, tal y como puede verse en la literatura al respecto. Asimismo, y como veremos en apartados posteriores, existen numerosas estructuras relacionadas directamente con la lista y distintas variantes de su implementación que ofrecen diferentes propiedades de acceso.

TAD: Lista

Operaciones:

`CrearLista: → Lista`
Crea una lista vacía.

`ListaVacia: Lista → Logico`
Dada una lista nos dice si está vacía.

`Primero: Lista → Posicion`
Devuelve la posición del primer elemento de la lista.

`Ultimo: Lista → Posicion`
Devuelve la posición del último elemento de la lista.

`Siguiente: Lista x Posicion → Posicion`
Dada una posición p en la lista, devuelve la posición del elemento situado a continuación.

`Anterior: Lista x Posicion → Posicion`
Dada una posicion p en la lista, devuelve la posición del situado justo antes del mismo.

`Dato: Lista x Posicion → Tipobase`
Dada una posición en la lista, devuelve el dato que contiene.

`Almacenar: Lista x Tipobase → Lista`
Dado un dato e, lo añade a la lista como último elemento.

`InsAntes: Lista x Posicion x Tipobase → Lista`
Dado un dato e y una posición p, añade el dato justo antes de la posición.

`InsDespues: Lista x Posicion x Tipobase → Lista`
Dado un dato e y una posición p, añade el dato justo después de la posición.

`Borrar: Lista x Posicion → Lista`
Dada una posición, elimina de la lista el elemento que la ocupa.

`Buscar: Lista x Tipobase → Posicion`
Dado un dato e, lo busca en la lista y devuelve la posición que ocupa.

`Modificar: Lista x Posicion x Tipobase → Lista`
Dado un dato e y una posición p, sustituye el valor que ocupa esa posición por e.

`Longitud: Lista → Entero`
Dada una lista, devuelve su longitud.

Axiomas: ∀ L ∈ Lista, ∀ p ∈ Posicion, ∀ e, f ∈ TipoBase,

```
1)  Listavacia(CrearLista) = cierto
2)  Listavacia(Almacenar(L, e)) = falso
3)  Longitud(CrearLista) = 0
4)  Longitud(Almacenar(L, e)) = Longitud(L) + 1
5)  InsDespues(CrearLista, p, e) = Almacenar(CrearLista, e)
6)  InsDespues(Almacenar(L, e), p, f) =
    Si p = Ultimo(Almacenar(L, e))
    entonces Almacenar(Almacenar(L, e), f)
    sino Almacenar(InsDespues(L, p, f), e)
7)  Dato(CrearLista, p) = error
```

```
8)   Dato(Almacenar(L, f), p) =
     Si p = Ultimo(Almacenar(L, f)) entonces f
     sino Dato(L, p)
9)   Borrar(CrearLista, p) = CrearLista
10)  Borrar(Almacenar(L, e), p) =
     Si p = Ultimo(Almacenar(L, e)) entonces L
     sino Almacenar(Borrar(L, p), e)
10)  Modificar(Almacenar(L, e), p, f) =
     Si p = Ultimo(Almacenar(L, e)) entonces Almacenar(L, f)
     sino Almacenar(Modificar(L, p, f), e)
```

5.3. Implementaciones del tipo Lista

Dentro de este apartado vamos a ver con mayor o menor detalle tres implementaciones del TAD Lista. En primer lugar veremos como puede implementarse de forma eficiente mediante una estructura estática de tipo vector. A continuación veremos dos implementaciones posibles mediante variables dinámicas que difieren en la eficiencia con que se desarrollan las distintas operaciones de acceso.

5.3.1. Implementación estática mediante un vector

En una primera aproximación, podríamos pensar en identificar la lista con un vector de elementos del tipo base. Dado que no existe ninguna posición específica en la cual se almacenen o borren los elementos, no sería necesario guardar ningún valor adicional, tal y como ocurre en implementación estática de la pila o la cola. Los elementos podrían añadirse o borrarse de cualquier posición del vector.

Sin embargo, inmediatamente podemos observar problemas con esta implementación. En primer lugar, para añadir un elemento en la lista, sería necesario desplazar todos los elementos situados a continuación del mismo en el vector. En segundo lugar, conforme fuésemos borrando elementos de la lista, iríamos generando huecos en el vector. Si quisiéramos hacer un uso óptimo del espacio disponible, sería necesario desplazar los elementos del vector para rellenar los huecos generados. Así pues, las operaciones de modificación de la pila resultan muy costosas utilizando un vector de esta forma. Además, nos encontraríamos con el problema derivado de manejar una estructura dinámica, como es la lista, mediante una estructura estática, como es el vector. Podría darse el caso de que la lista se llenara, al disponer de un espacio limitado por el tamaño del vector para almacenar sus elementos.

Así pues, optaremos por una solución alternativa para implementar una lista de forma más eficiente mediante un vector. Para no tener que desplazar elementos al añadir o borrar, lo que haremos es no obligar a que elementos sucesivos de la lista se almacenen en posiciones consecutivas del vector. Para ello será necesario que cada elemento guarde, además de la información contenida en la lista, información sobre cual es la posición del siguiente elemento en la misma. De este modo, cada elemento del vector tendrá dos componentes. Una posible implementación de esta estructura en Pascal es la siguiente:

```
CONST
  MAX = ... {Tamaño del vector y máximo tamaño posible de la lista}
TYPE
  Posicion = 0 .. MAX;
  Elemento = RECORD
               info: <TipoBase>;
               sig: Posicion
             END;
  TipoLista= RECORD
               primero, vacios: Posicion;
               mem = ARRAY [1..MAX] OF Elemento
             END;
VAR
  v:TipoLista;
```

Como vemos, cada componente de la lista es un registro con dos campos: un campo `info` que contiene el elemento de la lista y un campo `sig` que contiene el índice en el vector del siguiente elemento. A su vez, la lista es un registro con tres componentes

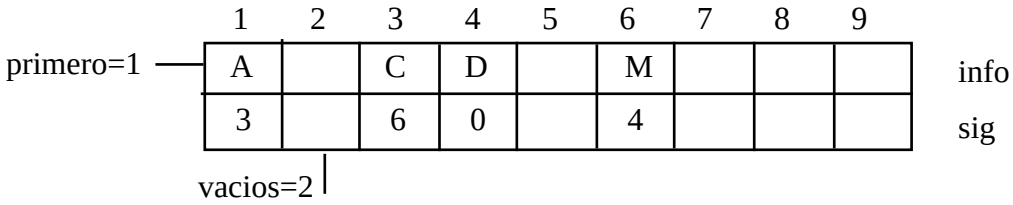
mem: contiene los elementos de la lista

primero: contiene el índice del primer elemento de la lista en el vector

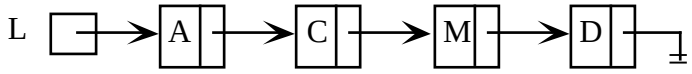
vacios: contiene el índice del primer elemento vacío del vector

Con esta implementación, los enlaces entre elementos de la lista no son punteros, sino que son simulados mediante enteros en el rango 0..MAX que contienen índices de componentes del vector.

Veamos gráficamente un ejemplo de esta implementación:

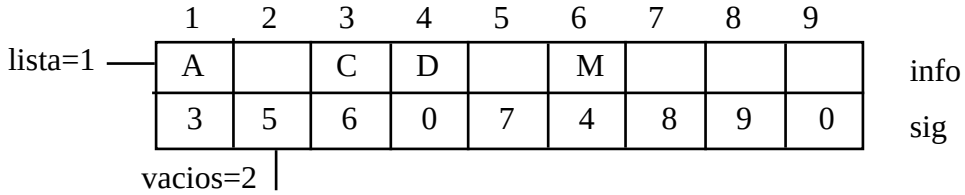


En este caso la lista contiene 4 elementos {A, C, M, D}. Dado que el campo `primero` vale 1, el primer elemento de la lista será el que ocupe esta posición en el vector: A. Dado que el campo `sig` de este elemento vale 3, el segundo elemento de la lista será el que ocupe esta posición en el vector: C. Se sigue este razonamiento hasta llegar al elemento D cuyo campo `sig` vale 0. Tomaremos como criterio general el que el valor 0 en este campo indique el final de la lista. Así pues, la lista implementada mediante la estructura anterior es de la siguiente forma:

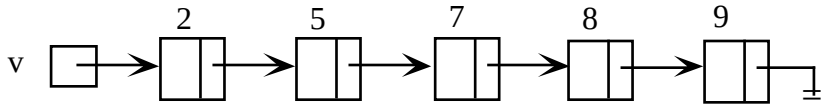


Démonos cuenta que con esta implementación de la lista estamos "simulando" la memoria dinámica mediante un vector. Los elementos de la lista se almacenan en un vector, y los punteros al siguiente elemento son simulados mediante enteros que indican (apuntan) cual es la posición del mismo en el vector.

Los distintos elementos de la lista no tienen porque encontrarse consecutivos en el vector y además, durante el manejo de la estructura, se habrán ido produciendo huecos en el mismo. A la hora de insertar nuevos elementos será necesario conocer la posición de los distintos huecos. Para ello lo que haremos es enlazar las componentes del vector que no contengan elementos de la lista para formar otra lista de elementos vacíos. De este modo, el vector contendrá realmente dos listas: una de elementos y otra de huecos. Veámoslo gráficamente:



La lista de elementos del ejemplo empieza en la posición 1 del vector, mientras la lista de huecos empieza en la posición 2. La lista de elementos será la representada anteriormente, mientras la de huecos enlazará las siguientes componentes del vector:



Para poder manejar esta implementación de la lista será necesario implementar dos operaciones adicionales a las incluidas en el TAD. Estas operaciones simularán las funciones `New` y `Dispose` proporcionadas por el Pascal para la gestión de punteros:

La operación `Nuevo(L)` devolverá, si existe, una componente del vector que esté vacía para poder almacenar el siguiente elemento de la lista `L`. Para ello consultará la lista de huecos, devolverá su primera componente y la eliminará de esta lista.

```
FUNCTION Nuevo(VAR L: TipoLista):Posicion;
BEGIN
  IF L.vacios=0 THEN { no quedan componentes en el vector }
    { mensaje de error de lista llena }
  ELSE BEGIN
    Nuevo := L.vacios;
    L.vacios := L.mem[L.vacios].sig
  END
END;
```

La operación `Liberar(L, e)` añadirá al principio de la lista de huecos la componente `e` del vector. Se supone que esta componente acaba de ser borrada de la lista de elementos.

```
PROCEDURE Liberar(VAR L: TipoLista; e: Posicion);
BEGIN
  L.mem[e].sig := L.vacios;
  L.vacios := e
END;
```

Se deja como ejercicio la implementación de las operaciones que definen el TAD Lista utilizando la estructura descrita en este apartado.

5.3.2. Implementación dinámica de una lista con enlace simple

En este aparado veremos cómo implementar una lista mediante variables dinámicas y punteros. En una primera aproximación consideraremos que cada nodo de la lista tiene dos componentes: una de ellas contiene la información del elemento de la lista, mientras la otra contiene un puntero al siguiente elemento de la misma. Al utilizar un solo enlace por cada nodo de la lista denominaremos a esta implementación de enlace simple.

La implementación directa de esta idea nos lleva a la siguiente estructura de datos en Pascal

```
TYPE
  Posicion = ^Elemento;
  Elemento = RECORD
               info: <TipoBase>
               sig: Posicion
             END;
  TipoLista = Posicion;
```

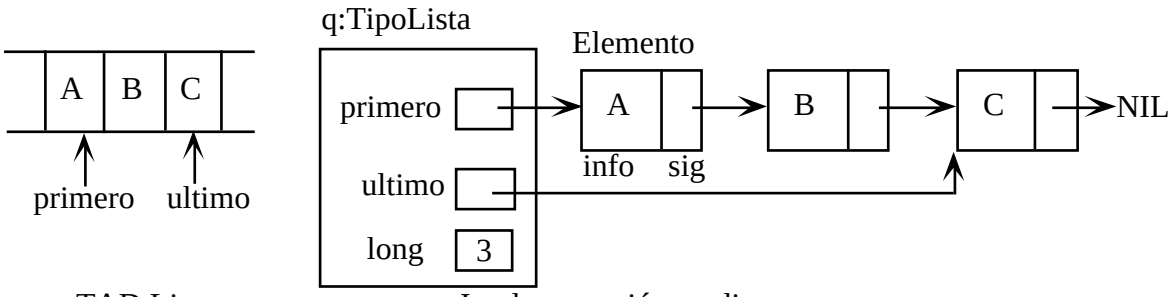
En este caso el tipo TipoLista es un puntero al primer elemento de la lista. Podríamos realizar una implementación utilizando esta estructura, pero para simplificar las distintas operaciones del TAD, haremos que la lista sea un registro con tres componentes:

- Una componente que apunte al primer elemento de la lista
- Una componente que apunte al último elemento de la lista
- Una componente que nos dé la longitud de la lista.

De este modo, la estructura de datos utilizada quedará así:

```
TYPE
  Posicion = ^Elemento;
  Elemento = RECORD
               info: <TipoBase>
               sig: Posicion
             END;
  TipoLista = RECORD
                longitud: INTEGER;
                primero, ultimo: Posicion
              END;
```

Gráficamente



TAD Lista Implementación mediante punteros

Esta estructura es muy similar a la que hemos utilizado para implementar una cola mediante memoria dinámica. Sin embargo, la diferencia estriba en las operaciones da acceso, dado que éstas pueden añadir y borrar elementos en cualquier posición de la estructura.

Veamos como se implementan las distintas operaciones que definen el TAD Lista:

```
PROCEDURE CrearLista (VAR L: TipoLista);
BEGIN
  L.longitud := 0;
  L.primero := NIL;
  L.ultimo := NIL
END;
```

Al crear una lista, su longitud será 0 y no estarán definidos ni su primer ni su último elemento.

```
FUNCTION ListaVacia(L: TipoLista):BOOLEAN;
BEGIN
  ListaVacia := (L.longitud=0)
END;
```

Se considerará que la lista está vacía cuando su longitud sea 0.

```
FUNCTION Primero (L:TipoLista):Posicion;
BEGIN
     Primero := L.primero
END;
```

El primer elemento de la lista es el apuntado por el campo primero.

```
FUNCTION Ultimo (L: TipoLista): Posicion;
BEGIN
  Ultimo := L.ultimo
END;
```

El último elemento de la lista es el apuntado por el campo ultimo. Si no hubiésemos incluido este campo en la estructura de datos `TipoLista`, sería necesario recorrer toda la lista desde su primer elemento para devolver la posición del último. En las dos funciones anteriores no hemos considerado el caso en el que la lista estuviese vacía de un modo especial. En este caso, por el modo en que el resto de subprogramas tratan los distintos campos de la estructura, la posición devuelta será NIL.

```
FUNCTION Siguiente ( L: TipoLista; p: Posicion): Posicion;
BEGIN
  Siguiente := p^.sig
END;
```

En una implementación completamente libre de errores deberíamos haber contemplado el caso en el que la posición `p` pasada como argumento no correspondiese a un elemento de la lista. En ese caso deberíamos haber devuelto algún tipo de error. Sin embargo, para no complicar el código, vamos a suponer que al usar esta función, siempre se pasa una posición que apunta a uno de los elementos de la lista. Este elemento puede haberse encontrado por ejemplo usando la función `Buscar` que veremos más adelante.

```
FUNCTION Longitud ( L : TipoLista ): INTEGER;
BEGIN
  Longitud := L.longitud
END;
```

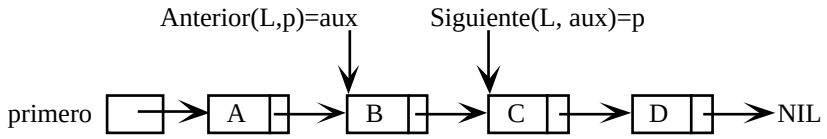
Al disponer de un campo longitud en la estructura que representa la lista, es muy sencillo implementar esta operación. Si no hubiésemos incluido este campo, habría sido necesario recorrer toda la lista y contar sus componentes.

```
FUNCTION Anterior (L: TipoLista; p:Posicion): Posicion;
VAR
  aux: Posicion;
BEGIN
  IF p = Primero(L) THEN
    aux := NIL
  ELSE BEGIN
    aux := Primero (L);
    WHILE (Siguiente(L,aux) <> p) DO
      aux := Siguiente(L,aux)
  END;
  Anterior := aux
END;
```

En la función `Anterior` de nuevo hemos supuesto que la posición pasada como argumento corresponde con un elemento de la lista. Al implementarla hemos de contemplar dos casos:

En el primer caso, cuando la lista la posición pasada como argumento corresponde al primer elemento de la lista, devolveremos como anterior al mismo un valor indefinido: NIL;

En el segundo caso, cuando la posición indicada no es la primera, hemos de recorrer la lista hasta encontrar el elemento apuntado. Para ello utilizaremos una variable auxiliar que irá apuntando a los sucesivos elementos recorridos. Comenzaremos apuntando al primer elemento de la lista y recorreremos los sucesivos elementos con ayuda de la función Siguiente. Cuando el siguiente elemento al apuntado por `aux` corresponda a la posición `p`, `aux` será el anterior que buscamos.




```
PROCEDURE Dato (L: TipoLista; p: Posicion; VAR d: TipoBase);
BEGIN
  d := p^.info
END;
```

Para acceder al dato contenido en el nodo en la posición `p`, basta con devolver el campo `info` de ese nodo. Si el tipo base lo permite, la operación anterior puede implementarse como una función del siguiente modo:

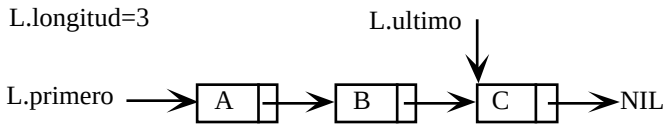
```
FUNCTION Dato (L: TipoLista; p: Posicion): TipoBase;
BEGIN
  Dato := p^.info
END;

FUNCTION Buscar (L: TipoLista; e: TipoBase):Posicion;
VAR
  aux: Posicion;
  d:TipoBase;
BEGIN
  aux := Primero(L);
  Dato(L, aux, d);
  WHILE ( d <> e) AND (aux <> NIL) DO BEGIN
    aux := Siguiente (L, aux);
    IF (aux <> NIL) THEN Dato (L, aux, d)
  END;
  Buscar := aux
END;
```

La operación `Buscar` devolverá la posición del dato `e` en la lista `L` si se encuentra, y NIL en caso contrario. Para buscar el dato, se recorre la lista desde su primer elemento dado por `Primero(L)`, mediante un puntero auxiliar. Por cada elemento recorrido se extrae el dato que contiene mediante la función `Dato(L,aux,d)`. Si el dato coincide con el

buscado, se devuelve su posición y si no coincide se pasa al siguiente mediante la operación `Siguiente(L,aux)`. El recorrido finaliza cuando hemos encontrado el elemento (`d=e`) o cuando hemos recorrido todos los elementos (`aux=NIL`).

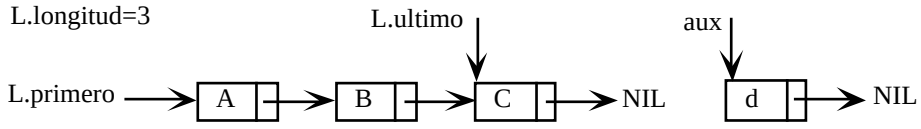
```
PROCEDURE Almacenar (VAR L: TipoLista; d: TipoBase );
VAR
  aux: Posicion;
BEGIN
  new(aux);
  aux^.info := d;
  aux^.sig := NIL;
  IF ListaVacia(L) THEN
    L.primero := aux
  ELSE
    L.ultimo^.sig := aux;
  L.ultimo := aux;
  L.longitud := L.longitud+1
END;
```

Supongamos que partimos de una lista como la siguiente:

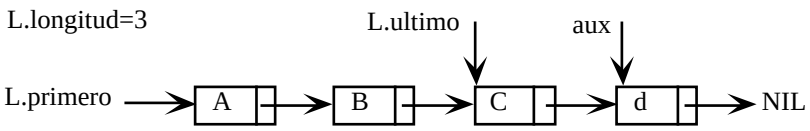


Para almacenar un elemento al final debemos recorrer los siguientes pasos:

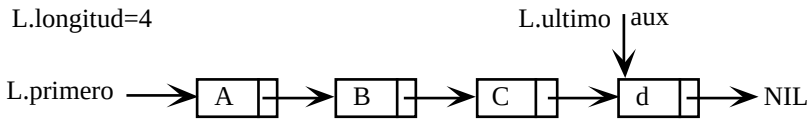
- En primer lugar creamos un nuevo nodo mediante la operación `new(aux)`.
- A continuación rellenamos los dos campos del nodo. El campo `info` con el valor `d` pasado como parámetro. Dado que el nuevo elemento va a ser el último de la lista, el campo `sig` se rellena con NIL.



- Si la lista estaba vacía, el campo `L.primero` debe apuntar al nuevo nodo añadido,
- en caso contrario, debemos enlazar el último nodo de la lista con el recién creado: `L.ultimo^.sig := aux`



- Al almacenar el elemento al final, el campo `L.ultimo` pasará a apuntar al nuevo nodo y el campo `L.longitud` se incrementará en 1.



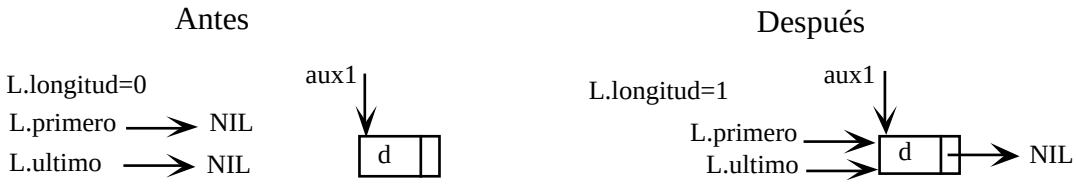

```
PROCEDURE InsAntes (VAR L: TipoLista; p: Posicion; d: TipoBase );
VAR
  aux1,aux2: Posicion;
BEGIN
  new(aux1);
  aux1^.info := d;
  IF ListaVacia(L) THEN BEGIN
       L.primero := aux1;
       L.ultimo := aux1;
       aux1^.sig := NIL
     END
  ELSE
    IF p = Primero(L) THEN BEGIN
         aux1^.sig := Primero(L);
         L.primero := aux1
       END
    ELSE BEGIN { cualquier posición }
       aux2 := Anterior (L, p);
       aux2^.sig := aux1;
       aux1^.sig := p
    END;
  L.longitud := L.longitud+1
END;
```

A la hora de insertar un elemento antes de una posición dada, debemos considerar tres casos posibles:

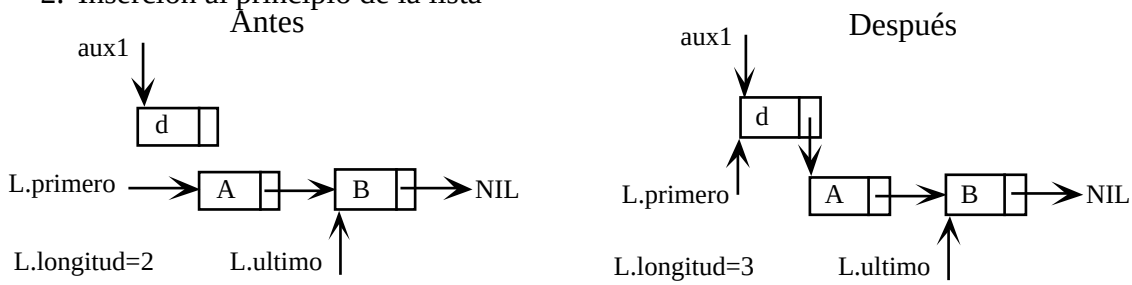
1. Que la lista este vacía
2. Que la inserción se realice al principio de la lista
3. Que la inserción se realice en cualquier otra posición de la lista

En los tres casos debemos comenzar por crear el nuevo nodo y almacenar en su campo `info` la información pertinente, y debemos finalizar incrementando la longitud de la lista en 1.

1. Inserción en una lista vacía

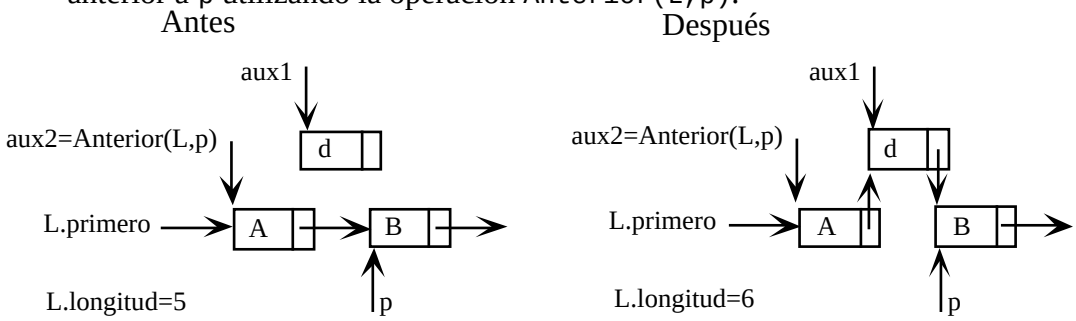


2. Inserción al principio de la lista



2. Inserción a partir del primer elemento

En este caso debemos insertar el nuevo nodo entre el apuntado por `p` y el anterior. Esto es, debemos hacer que el anterior apunte al nuevo nodo y que el nuevo nodo apunte a `p`. Para ello, lo primero que haremos es localizar el nodo anterior a `p` utilizando la operación `Anterior(L,p)`.




```
PROCEDURE InsDepues (VAR L: TipoLista; p: Posicion; d: TipoBase );
VAR
  aux1: Posicion;
BEGIN
  new(aux1);
  aux1^.info := d;
  IF ListaVacia(L) THEN BEGIN
       L.primero := aux1;
       L.ultimo := aux1;
       aux1^.sig := NIL
    END
  ELSE
    IF p = Ultimo(L) THEN BEGIN
         p^.sig := aux1;
         L.ultimo := aux1;
         aux1^.sig := NIL
       END
    ELSE BEGIN { cualquier posición }
       aux1^.sig := p^.sig;
       p^.sig := aux1
    END;
  L.longitud := L.longitud+1
END;
```

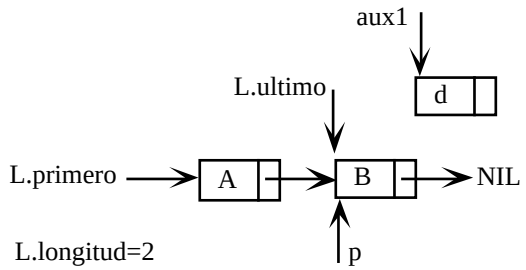
Para insertar después de una posición dada, de nuevo tenemos que contemplar tres casos:

1. Que la lista este vacía
2. Que queramos insertar después del último
3. Que queramos insertar en cualquier otra posición

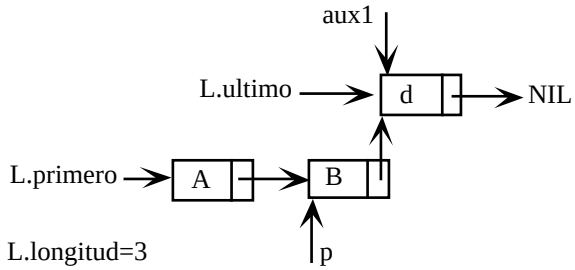
De nuevo debemos comenzar por crear el nuevo nodo y almacenar la información en su campo `info`, y debemos finalizar incrementando en uno la longitud de la lista.

1. Si la lista está vacía se opera igual que en `InsAntes`.
2. Si queremos insertar después del último elemento

Antes

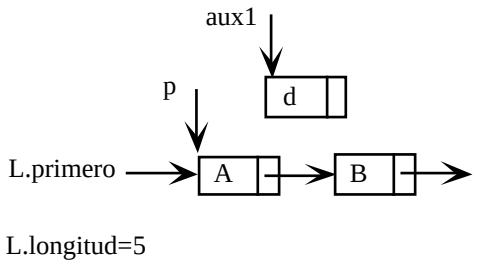


Después

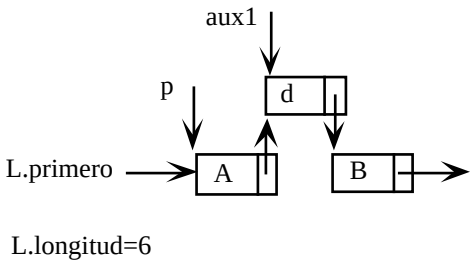


3. Si queremos insertar en cualquier otra posición

Antes



Después



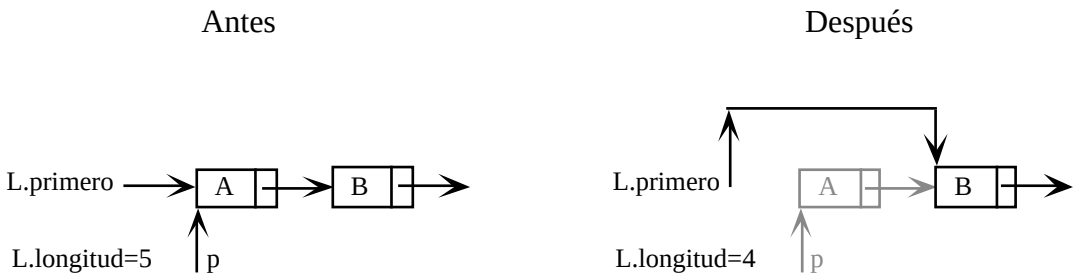

```
PROCEDURE Modificar (VAR L: TipoLista; p: Posicion; d: TipoBase);
BEGIN
  p^.info := d
END;
```

Modificar un elemento de la lista cuya posición pasamos como parámetro consiste simplemente en asignarle el nuevo valor a su campo `info`.

```
PROCEDURE Borrar ( VAR L: TipoLista; p: Posicion );
VAR
  aux: Posicion;
BEGIN
  IF p = Primero(L) THEN BEGIN
       L.primero := p^.sig;
       IF Primero(L) = NIL THEN
         L.ultimo := NIL
    END
  ELSE BEGIN
       aux := Anterior (L,p);
       aux^.sig := Siguiente (L,p);
       IF p = Ultimo(L) THEN
         L.ultimo := aux
  END;
  dispose(p);
  L.longitud := L.longitud - 1
END;
```

A la hora de borrar un elemento de la lista debemos contemplar dos casos básicos: que el elemento a borrar sea el primero o que no. En ambos casos debemos acabar por borrar el nodo mediante un dispose y disminuir la longitud de la lista en uno.

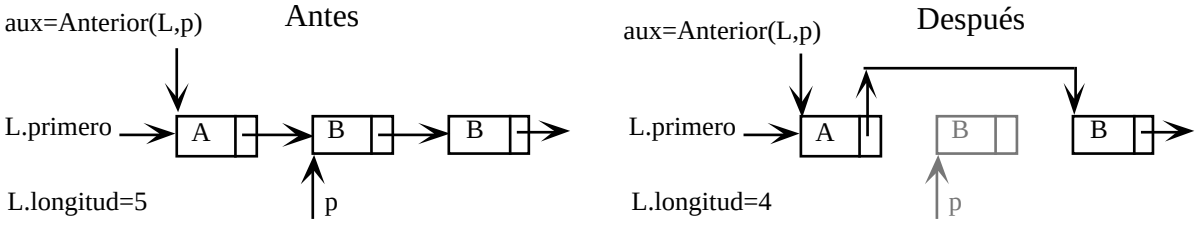
1. Si el elemento a borrar es el primero



   Si además el elemento borrado es el único de la lista (`Primero(L)=NIL`), debemos hacer que `L.ultimo` apunte a NIL

2. Si el elemento a borrar no es el primero

   En este caso debemos enlazar el nodo anterior a `p` con el siguiente a `p`. Para ello buscamos el nodo anterior mediante la operación `Anterior(L,p)`.

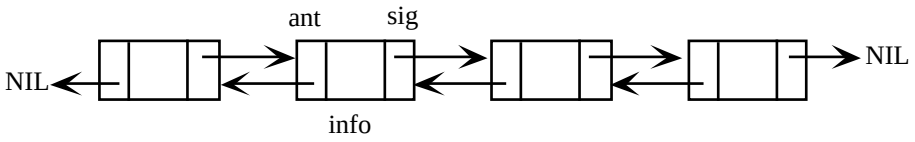


Además, si el nodo a borrar es el último de la lista, debemos hacer que el campo `L.ultimo` apunte al anterior (aux).

5.3.3. Implementación dinámica como lista doblemente enlazada

Uno de los problemas que plantea la implementación dinámica mediante simple enlace es el coste de insertar y borrar nuevos elementos en la lista. En varios casos es necesario recorrerla desde el principio para poder acceder al elemento anterior al dado como parámetro. Además, tan sólo es posible recorrer una lista así enlazada en una sola dirección.

Para solucionar ambos problemas se puede utilizar una lista doblemente enlazada. En este caso, cada nodo de la lista apuntará tanto al anterior como al siguiente.



Para implementar en Pascal este tipo de listas, utilizaremos las siguientes definiciones:

```
TYPE
  Posicion = ^Elemento;
  Elemento = RECORD
               info: <TipoBase>;
               ant, sig: Posicion
             END;
  TipoLista = RECORD
                longitud: INTEGER;
                primero, ultimo : Posicion
              END;
```

Con esta definición de una lista doblemente enlazada, es necesario modificar algunas operaciones del TAD Lista para poder adaptarlas al nuevo tipo de nodo utilizado.

En este apartado queda como ejercicio la justificación de la implementación de las operaciones mostradas.

```
FUNCTION Anterior (L : TipoLista; p: Posicion ):Posicion;
BEGIN
  Anterior := p^.ant
END;
```

```
PROCEDURE InsAntes( VAR L: TipoLista; p: Posicion; d: TipoBase );
VAR
  aux : Posicion;
BEGIN
  new (aux);
  aux^.info := d;
  IF ListaVacia(L) THEN BEGIN
       L.primero := aux;
       L.ultimo := aux;
       aux^.sig := NIL;
       aux^.ant := NIL
    END
  ELSE
    IF p = Primero(L) THEN BEGIN
         aux^.sig := Primero(L);
         L.primero^.ant := aux;
         aux^.ant := NIL;
         L.primero := aux
       END
    ELSE BEGIN { cualquier posición }
      aux^.ant := Anterior(L,p);
      aux^.sig := p;
      p^.ant^.sig := aux;
      p^.ant := aux
    END;
  L.longitud := L.longitud+1
END;

PROCEDURE InsDespues( VAR L : TipoLista; p: Posicion; d: TipoBase );
VAR
  aux: Posicion;
BEGIN
  new(aux);
  aux^.info := d;
  IF ListaVacia(L) THEN BEGIN
       aux^.sig := NIL;
       aux^.ant := NIL;
       L.primero := aux;
       L.ultimo := aux
    END
  ELSE
    IF p = Ultimo(L) THEN BEGIN
         aux^.ant := Ultimo(L);
         L.ultimo^.sig := aux;
```

```
          aux^.sig := NIL;
          L.ultimo := aux
        END
      ELSE BEGIN { cualquier posición }
        aux^.ant := p;
        aux^.sig := Siguiente (L,p);
        p^.sig^.ant := aux;
        p^.sig := aux
      END;
  L.longitud := L.longitud + 1
END;

PROCEDURE Borrar ( VAR L: TipoLista; p: Posicion );
VAR
  aux1, aux2 : Posicion;
BEGIN
  IF p = Primero(L) THEN BEGIN
       L.primero := Siguiente(L,p);
       IF Primero(L) = NIL THEN
         L.ultimo := NIL
       ELSE
         L.primero^.ant := NIL
    END
  ELSE
    IF p = Ultimo(L) THEN BEGIN
         aux1 := Anterior (L,p);
         L.ultimo := aux1;
         aux1^.sig := NIL
       END
    ELSE BEGIN { Cualquier posición }
      aux1 := Anterior (L,p);
      aux2 := Siguiente (L,p);
      aux1^.sig :=aux2;
      aux2^.ant :=aux1
    END;
  dispose(p);
  L.longitud := L.longitud - 1
END;
```

5.4. TAD's relacionados con la Lista

En función del problema que queramos resolver y de las características específicas que queramos conseguir en una lista, existen toda una serie de variantes de este TAD. En este apartado vamos a ver algunas de ellas, pero puede consultarse la literatura al respecto para encontrar otras muchas.

5.4.1. TAD Lista Ordenada

Supongamos que queremos manejar una lista en la que sus distintos elementos estén ordenados en función de su valor. En este caso podemos definir una variante del TAD Lista que denominaremos Lista Ordenada. Una definición de este nuevo tipo de datos puede ser la siguiente:

Definición

Una lista ordenada es una estructura lineal y homogénea en la que se pueden añadir o eliminar elementos en cualquier posición, pero de manera que los elementos siempre permanezcan ordenados según algún criterio de ordenación.

En el caso de las listas ordenadas el TAD asociado tendrá una estructura similar a la vista en el apartado 5.1.2. Sin embargo, en este caso a la hora de insertar elementos no necesitaremos indicar ninguna posición, dado que ésta quedará definida por su valor. El nuevo elemento se insertará de modo que se mantenga el orden en la lista.

TAD: ListaOrd

Operaciones:

`CreaListOrd: → ListaOrd`

Crea una lista ordenada vacía.

`ListOrdVacia: ListaOrd → Logico`

Dada una lista ordenada, nos dice si está vacía.

`Primero: ListaOrd → Posicion`

Dada una lista ordenada, nos devuelve la posición de su primer elemento.

`Ultimo: ListaOrd → Posicion`

Dada una lista ordenada, nos devuelve la posición de su último elemento.

`Siguiente: ListaOrd x Posicion → Posicion`

Dada una lista ordenada y una posición en la misma, nos devuelve la posición del siguiente elemento.

`Anterior: ListaOrd x Posicion → Posicion`

Dada una lista ordenada y una posición en la misma, nos devuelve la posición del elemento anterior.

`Almacenar: ListaOrd x Tipobase → ListaOrd`

Dada una lista ordenada y un dato del tipo base, inserta el dato en la última posición de la lista.

`InsOrd: ListaOrd x TipoBase → ListaOrd`

Dada una lista ordenada y un elemento d, inserta el elemento en la lista en su posición correspondiente.

`Borrar: ListaOrd x Posicion → ListaOrd`

Dada una lista ordenada y una posición p, borra de la lista el elemento indicado por la posición.

`Dato: ListaOrd x Posicion → TipoBase`

Dada una lista ordenada y una posición en la misma, nos devuelve el valor del elemento situado en la misma.

`Buscar: ListaOrd x TipoBase → Posicion`

Dada una lista ordenada y un valor del tipo base, busca dicho valor en la lista y nos devuelve su posición.

`Longitud: ListaOrd → Entero`

Dada una lista ordenada, nos devuelve su longitud.

Axiomas: ∀ L ∈ ListaOrd, ∀ p ∈ Posicion, ∀ e, f ∈ TipoBase,

```
1)  ListaOrdVacia(CrearListOrd) = cierto
2)  ListaOrdVacia(Almacenar(L,e)) = falso
3)  Longitud(CrearListOrd) = 0
4)  Longitud(Almacenar(L,e)) = Longitud(L) + 1
5)  InsOrd(CrearListOrd, e) = Almacenar(CrearListOrd, e)
6)  InsOrd(Almacenar(L, e), f) =
    Si e <= f entonces Almacenar(Almacenar(L, e), f)
    sino Almacenar(InsOrd(L, f), e)
7)  Dato(CrearListOrd, p) = error
8)  Dato(Almacenar(L, e), p) =
    Si p = Ultimo(Almacenar(L, e)) entonces e
    sino Dato(L, p)
9)  Borrar(CrearListOrd, p) = error
10) Borrar(Almacenar(L, e), p) =
    Si p = Ultimo(Almacenar(L, e)) entonces L
    sino Borrar(L, p)
```

A la hora de implementar una lista ordenada podemos elegir las mismas opciones que hemos descrito en el caso de una lista general. Así, podemos utilizar una implementación estática mediante un vector o podemos utilizar una implementación dinámica mediante el uso de punteros. También en este último caso podemos implementar una lista ordenada simplemente enlazada o doblemente enlazada.

Dado que ya hemos entrado en detalle en la descripción de las tres implementaciones en el caso de una lista general, no lo volvemos a hacer con una lista ordenada. Sin embargo, sí es interesante describir cuál sería la implementación de la nueva operación añadida al TAD: `InsOrd`. Para ello utilizaremos una implementación dinámica simplemente enlazada. La estructura de datos definida en Pascal es la misma que en el caso de la lista general. Lo que cambia es el método de inserción utilizado.

Cuando implementamos esta operación hemos de distinguir dos casos fundamentales: que la lista este vacía o que no lo esté. En ambos casos hemos de comenzar por crear el nuevo nodo y almacenar en él la información, y finalizar incrementando la longitud de la lista.

Si la lista está vacía integrar el nuevo nodo es trivial. Si la lista no está vacía, debemos buscar la posición que le corresponde al dato y insertarlo en la misma. Para buscar la posición en que debemos insertar el nuevo dato, se recorre la lista desde su primer elemento hasta encontrar el primero que sea mayor que el mismo. El nuevo nodo se insertará justo antes de este elemento.

Dentro de este caso en el que la lista no está vacía, hemos de distinguir dos situaciones especiales a la hora de actualizar los campos de la misma. La primera situación se da cuando el nodo debe insertarse en la primera posición de la lista. En este caso

debemos actualizar el campo `L.primero`. La segunda situación especial se da cuando acabamos el recorrido de la lista sin encontrar ningún elemento mayor, es decir, cuando debemos insertar el nuevo nodo en la última posición de la lista. En este caso debemos actualizar el campo `L.ultimo`. El siguiente subprograma contempla todos los casos citados.

```
PROCEDURE InsOrd (VAR L: TipoLista; d: TipoBase );
VAR
  aux1,aux2,ant : Posicion;
BEGIN
  new(aux1); aux1^.info := d;
  IF ListaOrdVacia(L) THEN BEGIN
       L.primero := aux1;
       L.ultimo := aux1;
       aux1^.sig := NIL
    END
  ELSE
    IF d <= Dato(L, Primero(L)) THEN BEGIN { se inserta el primero}
         aux1^.sig := Primero (L);
         L.primero:= aux1
      END
    ELSE BEGIN
      aux2 := Primero(L);
      WHILE (Dato(L,aux2) < d) AND (aux2 <> Ultimo(L)) DO BEGIN
        ant := aux2;
        aux2 := Siguiente (L,aux2)
      END;
      IF (Dato (L,aux2) >= d) THEN BEGIN
           aux1^.sig := aux2;
           ant^.sig := aux1
        END
      ELSE BEGIN { Se inserta despues del ultimo }
        aux1^.sig := NIL;
        L.ultimo^.sig := aux1;
        L.ultimo := aux1
      END
    END;
    L.longitud := L.longitud + 1
END;
```

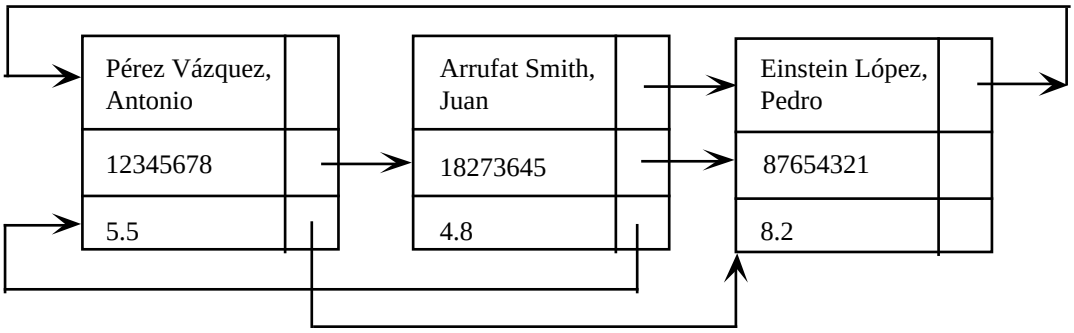
5.4.2. TAD Multilista

En el apartado anterior hemos descrito el TAD Lista ordenada en el que los distintos elementos se ordenan en función del valor de los mismos. Sin embargo, supongamos que los elementos de la lista no son simples, sino que se trata de datos compuestos tales como registros o vectores.

Imaginemos por ejemplo que estamos tratando con una lista de alumnos en la que dos de sus campos son los apellidos y el DNI del alumno. Puede interesarnos en ocasiones recorrer los distintos alumnos por orden de apellidos y en otras ocasiones recorrerlos por orden de DNI. Si implementamos esta lista utilizando el TAD Lista Ordenada nos vemos forzados a elegir uno de los campos para enlazar los distintos elementos en orden. En este caso extraer los elementos ordenados según otro campo sería mucho más costoso.

Para solucionar este problema se utilizan las multilistas. Se trata de una estructura en la que los distintos elementos se enlazan siguiendo el orden marcado por varios componentes de los datos. Estrictamente hablando una multilista no es una estructura lineal, puesto que cada elemento tiene más de un anterior y más de un siguiente. Podríamos decir que una multilista integra varias listas ordenadas, cada una de ellas en función de una componente o campo distinto de los datos almacenado.

Veamos un ejemplo:



La estructura de datos que podemos utilizar en Pascal para implementar una multilista como la anterior es la siguiente

```
TYPE
  Alumno = RECORD
             nombre, DNI: String;
             nota: REAL
           END;
  Posicion = ^ElemMultList;
  ElemMultList = RECORD
                   info: Alumno;
                   signom, sigdni, signot: Posicion
                 END;


  Multilista = RECORD
                 primernom, primerdni, primernot: Posicion
               END;
```

Al implementar la Multilista hemos elegido una estructura dinámica de enlace simple para representar cada una de las listas que contiene. Podríamos haber elegido una estructura estática basada en vectores y registros o haber utilizado enlaces dobles para

alguna de las listas o para todas ellas. También debemos darnos cuenta de que el tipo Multilista tan sólo contiene un campo asociado a cada lista: el que apunta a su primer elemento. Otra opción sería utilizar para cada una de ellas los tres campos que hemos utilizado con el TAD Lista.

Aunque no vamos a describir en detalle el TAD Multilista ni su posible implementación, sí es interesante resaltar algunos aspectos del mismo:

- Las operaciones involucradas serían similares a las del TAD Lista Ordenada.
- Sería necesario incluir una operación Siguiente y una Anterior por cada uno de los campos ordenados.
- Al insertar un nuevo elemento se creará un único nuevo nodo. El nuevo nodo se integrará en cada una de las listas siguiendo el orden adecuado en cada caso.
- Al borrar un elemento deberemos actualizar los enlaces asociados a todas las listas, pero el nodo se eliminará una sola vez.

5.5. Ejercicios

1. Se tiene implementada una lista de empleados de una empresa ordenada alfabéticamente por sus apellidos. Se desea reducir el tiempo de búsqueda de los datos de un empleado y para ello, el programador decide ampliar la estructura de datos añadiendo un vector cuyo índice son las iniciales de los apellidos, y cuyos elementos son los punteros al primer empleado cuyo apellido empiece por dicha inicial. Se pide:

   a) La definición de tipos necesaria.

   b) Determinar qué operaciones de lista se pueden modificar para mejorar su eficiencia con esta nueva estructura y cuáles no.

   c) Implementar las nuevas operaciones de lista (sólo las mejorables).

2. Se dispone de una lista de simple enlace en la que la información almacenada es de tipo STRING. La lista está ordenada por orden alfabético.

   Se pretende utilizar la lista para gestionar el orden en el que se ha de realizar una oposición. El mecanismo habitual es seguir el orden alfabético, pero no se comienza por la letra A, sino por una letra extraída al azar: si, por ejemplo, sale la L, el ejercicio lo comienza el primer candidato cuyo apellido empiece por L, siguiéndose, a partir de ahí, el orden alfabético. Al acabar, se continúa con el primero hasta que todos los candidatos se hayan examinado.

   Sabiendo que nos darán como información de entrada la letra que salga por sorteo, se pide:

   a) Implementar un algoritmo que a partir de la lista original, ordenada, y de la letra, devuelva **otra** lista con la reordenación indicada.

   b) Implementar un algoritmo que a partir de la lista original, ordenada, y de la letra, devuelva **la misma** lista pero con la reordenación indicada.

3. Dada una lista doblemente enlazada, que almacena información de tipo base TB, definir la operación **BORRA_2**, tal que dada la posición de un elemento borre el anterior y el siguiente (si un elemento dado no tuviera anterior y/o siguiente, que borre lo que pueda).

4. Dada una lista de enlace simple, que almacena información de tipo base TB, definir dos implementaciones de la operación **INVERTIR**, tal que a partir de la lista original

   a) se obtenga **otra lista** en la que los elementos de la original estén en orden inverso.

   b) se invierta, **sobre la misma lista**, el orden de los elementos.

5. Dada una lista, desordenada, que almacena información de tipo entero, escribir un algoritmo que obtenga dos listas ordenadas: en una aparecerán los elementos pares de la lista original y en la otra los impares.

6. Una finca tiene **n** pisos y en cada piso hay 5 viviendas identificadas como A, B, C, D y E. Por cada vivienda, un vecino paga una determinada cuota de gastos de escalera. El administrador de la finca sabe el nombre de cada vecino, su cuota y, además, si está alquilado o si es el propietario del piso; en el primer caso, debe saber, además, el nombre del propietario del piso y su número de cuenta corriente, puesto que es el propietario el que corre con los gastos de escalera. En el segundo caso, para poder realizar el cargo sólo se debe saber el número de cuenta del vecino.

   a) Definición completa de la estructura de datos que permita organizar dicha información.

   b) Realizar una operación que permita al administrador sacar un listado en el que, para cada una de las viviendas, se emita un recibo. Un **recibo** indica el **piso**, la **puerta**, el nombre del **vecino**, el nombre del **propietario**, la **cuota** y el **número de cuenta** donde se deben efectuar los cargos del recibo de la escalera.

   c) Al cabo de 15 días desde la emisión del listado y su envío al banco con el que trabaja el administrador, el banco devuelve una **lista** de **recibos** impagados, que el administrador transforma en una **lista** de **morosos** (teniendo en cuenta que puede haber alguien que **posea más de una vivienda** en la finca).

      c.1) Definir las estructuras **lista de recibos** y **lista de morosos**. En la lista de morosos, debe constar el **nombre**, la **cuenta corriente** y el importe total de la **deuda**.

      c.2) Escribir una operación que permita transformar la lista de recibos impagados en la lista de morosos, sin modificar la lista de recibos.

7. En un teatro se identifica cada butaca por su número de fila y su número de butaca dentro de una fila. En total hay N filas y M butacas por fila. Se quiere hacer un programa para el proceso automático de reservas del teatro, de forma que por cada butaca, además de su precio, si está reservada se pueda saber quién la ha reservado.

   a) Definir la estructura de datos **teatro**.

   b) Para realizar una reserva, además del nombre de quien la hace, hay que indicar cuántos asientos se reservan. Las reservas se deben almacenar en una lista, implementada dinámicamente que servirá, además, para cobrar la reserva en el momento del pago.

      b.1) Definir la estructura de datos **reserva**.

      b.2) Definir la operación **Reservar** que, dada la información del nombre y número de asientos, actualice la ocupación del teatro y además añada dicha información a la lista de reservas, incluyendo el precio. La reserva debe asegurar que los asientos sean correlativos. Si no es posible, debe indicarlo.

b.3) Definir la operación **Cancelar** que, dada la información del nombre, permite cancelar una reserva, eliminándola de la lista y liberando, además, los asientos correspondientes.

8. En una determinada Universidad hay **n** campus, cada uno de ellos con **m** aulas informáticas y cada aula con **20** puestos de trabajo. Cada uno de estos puestos de trabajo puede estar ocupado o no con una conexión a la red. Caso de estarlo, el usuario quedará identificado por la siguiente información: su login y la tarea que está realizando. La tarea puede ser **una sola** de estas cuatro: leer las news, hacer un write, acceder al correo electrónico o manejo del Sistema Operativo.

   Se desea escribir un algoritmo que permita a los operadores del sistema saber quién está conectado a la red; para ello, se quiere

   a) La definición de la estructura de datos que soporte la información descrita.

   b) Definir la estructura de datos más adecuada para contener la salida de datos, si lo que se quiere es la relación de los "**logins**" de usuarios conectados. Escribir el algoritmo que permita obtener dicha relación.

   c) Definir la estructura de datos más adecuada para contener la salida de datos, si lo que se quiere es la relación de los "**logins**", **tarea**, **ordenador**, **aula informática** y **campus** de los usuarios conectados. Escribir el algoritmo que permita obtener dicha relación.

   Nota: Un "login" es una identificación única. Se supone que como mucho hay un número máximo de L "logins", L < 256.

9. Un profesor tiene un montón de exámenes y en cada momento sólo puede acceder al examen situado encima del mismo. Por cada uno de ellos guarda el nombre del alumno, su dni y la nota de cada uno de los cinco ejercicios de que consta el examen.

   a) Definir las estructuras de datos más adecuadas para almacenar la información del problema.

   b) El profesor quiere generar el acta en la que los distintos alumnos aparecen ordenados alfabéticamente. Por cada alumno debe guardar entonces su nombre, dni y la nota total obtenida sumando la de los 5 ejercicios.

      b.1) Definir las estructuras de datos más adecuadas para guardar la información.

      b.2) Implementar un procedimiento **publicar** que a partir del montón de exámenes genere el acta. Al finalizar el proceso, el montón de exámenes debe quedar igual.

   c) Implementar un procedimiento denominado **maxnota** que, a partir del acta, escriba el nombre y la nota del alumno que haya obtenido la máxima calificación global.

   d) Suponer que el profesor quiere poder recorrer el acta, tanto por orden alfabético como por orden de la nota global de los distintos alumnos.

      d.1) Modificar la estructura de datos del apartado b) para poder realizar esta operación.

      d.2) Implementar el algoritmo **recorrer** que, en función de un argumento de entrada, permita recorrer el acta por orden de nombre o de nota. Por cada alumno, el algoritmo debe escribir su nombre y la nota total.

10. En una sucursal bancaria se pueden realizar ingresos, reintegros y consultas. La sucursal dispone de dos ventanillas, una de las cuales puede atender cualquier operación (ventanilla

general), mientras que la otra sólo puede realizar ingresos (ventanilla de ingresos). Los clientes, caracterizados por su NIF y la operación que desean realizar, se atienden según el orden de llegada, y pueden ser atendidos indistintamente por una u otra ventanilla en función de que estén libres en ese momento.

Si un cliente es atendido por la ventanilla de ingresos y no desea realizar esa operación, se pasa a una dependencia donde esperan en orden de llegada todos los clientes en la misma situación. Todos los clientes de esta dependencia serán atendidos por la ventanilla general. Además, tienen preferencia sobre los clientes que aún no han visitado ninguna ventanilla.

Cada operación realizada en la sucursal se registra ordenada por NIF del cliente en una estructura denominada **Operaciones** en el momento de efectuarla. Por cada operación se almacena el NIF del cliente y la operación realizada.

a) Definir las estructuras de datos más adecuadas para almacenar la información del problema.

b) Implementar las operaciones de atender un cliente en la ventanilla general y atenderlo en la ventanilla de ingresos.

c) Al finalizar la jornada se reordenan las operaciones efectuadas colocando en primer lugar los ingresos, luego los reintegros y en tercer lugar las consultas, y dentro de cada modalidad manteniendo el orden por NIF del cliente. Se pide realizar un algoritmo que a partir de la estructura **Operaciones** obtenga la estructura **Operaciones por modalidad**.

11 Escribir un algoritmo que permita intercambiar los elementos que ocupan las posiciones `m` y `n` de una lista dinámica de elementos de tipo `<Tipobase>`. Tener en cuenta que no puede modificarse el valor contenido en los nodos de la lista y que `1<=m<=n<=final de la lista`.

12. Queremos almacenar la información sobre 100 CDs de música. Por cada CD guardamos el título, su autor y una lista con las canciones que contiene. Por cada canción guardamos su título y su duración en segundos.

a) (0,5 p.) Definir la estructura de datos más adecuada para almacenar la información.

b) (1 p.) Implementar un procedimiento que, dado un título de CD, nos devuelva su duración total.

c) (0, 75 p.) Implementar un procedimiento que, dado un título de canción, nos devuelva el título y autor del CD en que se encuentra.