

1.2.4 Listas enlazadas

Las listas enlazadas son tipos de datos dinámicos que se construyen con *nodos*. Un nodo es un registro con al menos, dos campos, uno de ellos contiene las componentes y se le denomina **data** (estructura de registro), el otro es un valor que señala al siguiente nodo y se le denomina **enlace** (next).

Como definición más formal se tiene que: El TDA lista enlazada es una colección de nodos ordenada según su posición, tal que cada uno de ellos es accedido a través del campo *enlace* del nodo anterior.

Las listas enlazadas se implementan aprovechando todas las ventajas de la asignación dinámica de memoria. **Ésta es la forma habitual y más eficaz de realizar su implementación.** Sin embargo, ésta no es la única implementación posible. También pueden ser implementadas estáticamente definiendo el tipo arreglo como nodos y utilizar los índices en el campo *enlace* para determinar el siguiente nodo.

Aunque pueda parecer más eficiente implementar una lista enlazada mediante arreglos, realmente no lo es. En el caso de un arreglo, siempre se deberá mantener una lista paralela de nodos vacíos de manera que cuando se necesite alojar un nuevo nodo, se tome de esta última lista un nodo disponible y cuando se quiera liberar un nodo habrá de ser devuelto a la lista.

Es importante observar la distinción existente entre un tipo de datos abstracto y su implementación. Ambos conceptos pueden ser considerados de forma estática o dinámica. Una variable de tipo array es una estructura típicamente estática, pero puede ser almacenada en memoria de forma estática o dinámica (mediante punteros). Lo mismo sucede con los conceptos **pila** o **lista**, que por naturaleza son estructuras puramente dinámicas pero que pueden implementarse con asignación de memoria estática (dentro de un array).

Una definición de un tipo nodo puede ser la siguiente:

```
TYPE Ptr_nodo = POINTER TO Nodo;
      Nodo = RECORD
          datos: Tipo_datos
          enlace: Ptr_nodo;
      END
```

Para acceder a los elementos de una lista se necesita un primer elemento o **puntero externo**. Este puntero apuntará al primer elemento de la lista. Para ello será necesario declarar una variable a tal fin:

```
VAR lista: Ptr_nodo;
```

Acciones a realizar en una lista enlazada:

Inserción por la cabeza

1. Crear un nuevo nodo.
2. Almacenar el dato en el campo correspondiente (datos).
3. Como va a ser el primer nodo, su enlace deberá apuntar al que hasta ahora era el primer nodo (apuntado por el enlace externo).
4. El enlace externo deberá apuntar al nuevo nodo (que ahora es el primero).

Inserción por el final

Para insertar por el final, se debe llegar hasta el último nodo de la lista. Para ello, y partiendo del enlace externo, se saltará de nodo a nodo (a través del campo enlace), hasta alcanzar un nodo cuyo enlace sea nulo (NIL). En ese momento, el campo enlace se cambiará por el enlace hacia el nuevo nodo (recordar que al crear un nuevo nodo, por defecto, su enlace debe estar siempre a NIL).

Suprimir por la cabeza

1. Almacenar la dirección del primer elemento apuntado por el enlace externo en una variable puntero temporal.
2. Se asigna al enlace externo el elemento que figura en el enlace del primer elemento.
3. Se elimina el nodo apuntado por el puntero auxiliar.

Suprimir por el final

En este caso se procederá del mismo modo que en la inserción por el final pero con una variable puntero auxiliar se irá guardando la dirección del nodo anterior. Al alcanzar el nodo con enlace=NIL, se borrará y al nodo anterior apuntado por el puntero auxiliar, se modificará su enlace a NIL.

Inserción según criterio de orden

Para todos los casos que serán expuestos a continuación, previamente será necesario crear un nuevo nodo **Nuevo_nodo** que contenga el elemento de datos y que su **enlace** al siguiente nodo sea NIL.

Para ello, será necesario, en primer lugar, recorrer la lista desde el enlace externo hasta localizar el nodo con un valor entero menor que 8 que además sea el predecesor de un nodo con un valor igual o superior a 8 (en ningún momento se ha hablado de valores únicos).

En general, se considerarán tres posibles situaciones: que la posición a insertar sea la primera de la lista, es decir en valor del primer nodo es mayor que el valor del nodo que deseamos insertar, que la posición a insertar sea la última de la lista es decir, que el último valor de la lista sea mayor que el valor que se desea insertar, o bien, como en este caso, que el valor a insertar se encuentre entre dos nodos adyacentes.

En el primer caso, se utilizará el procedimiento **insertar por la cabeza**. En el segundo caso, se empleará el procedimiento **insertar por el final**. Estos dos procedimientos ya han sido descritos anteriormente. Debe prestarse especial atención al caso de que la lista puede estar vacía (lista=NIL), analizando esta condición por separado.

En el caso de tener que realizar una inserción entre dos nodos serán necesarios dos punteros que almacenen la dirección del nodo anterior y del nodo actual respectivamente. Situando el valor del puntero **anterior**=al inicio de la lista y el puntero **actual**=al enlace del nodo actual, se recorrerá iterativamente la lista hasta detectar que el valor del dato del nodo que se está analizando es mayor o igual que el valor del nuevo nodo a insertar. Llegado este caso, el puntero **anterior** contendrá la dirección del nodo con elemento de datos menor que el valor a insertar y el puntero **actual** contendrá la dirección del nodo cuyo elemento de datos es mayor o igual al elemento que se pretende insertar.

Con esta información, el enlace de **anterior**.**enlace** deberá apuntar al nuevo nodo creado y el enlace del nuevo nodo creado deberá apuntar a la dirección del nodo apuntada por **actual**

(Nuevo_nodo^.enlace=actual).

El siguiente algoritmo presenta la solución al problema:

```
PROCEDURE Insertar_orden_delante (VAR lista: Ptr_Nodo; dato : Tipo_datos);
VAR
  Encontrado: BOOLEAN;
  Anterior, Actual, Nuevo_nodo: Ptr_Nodo;

BEGIN
  IF (lista=NIL)OR(dato<lista^.dato) THEN    (*Insercion al principio*)
    Insertar_cabeza(lista, dato)
  ELSE
    ALLOCATE(Nuevo_nodo,SIZE(Nodo));
    Nuevo_nodo^.dato:=dato;
    Nuevo_nodo^.enlace:=NIL;
    (*busca el punto de insercion*)
    Anterior:=lista;
    Actual:=lista^.enlace;
    Encontrado := FALSE;
    WHILE (Actual<>NIL) AND (NOT Encontrado) DO
      IF dato>Actual^.dato THEN
        Anterior:=Actual;
        Actual:=Actual^.enlace;
      ELSE
        encontrado := TRUE;
      END;
    END;
    (*insertar el nuevo nodo*)
    Nuevo_nodo^.enlace:=Actual;
    Anterior^.enlace:=Nuevo_nodo;
  END
END Insertar_orden_delante;
```

Imprimir una lista enlazada

Después de haber visto los métodos utilizados para mantener una lista enlazada, resulta trivial pensar en un procedimiento que sirva para imprimir su información.

Este procedimiento debería partir del enlace externo y mediante un puntero auxiliar (Aux) se recuperaría cada nodo mediante una iteración cuya función de cota sería que el enlace del nodo actual fuera NIL.

Para cada nodo se imprimiría el valor de su campo de datos, por ejemplo `WriteInt(Aux^.dato, 7);`

```
PROCEDURE Imprimir_lista (lista: Ptr_Nodo);
VAR
  Aux: Ptr_Nodo;

BEGIN
  Aux:=lista;
  WHILE Aux<>NIL DO
    WriteInt(Aux^.dato, 7);
    Aux := Aux^.enlace;
  END;
END Imprimir_lista;
```

1.2.5 TDA Lista implementado con Listas enlazadas

Los apartados anteriores han servido para especificar algunas particularidades sobre el manejo de listas enlazadas, es evidente que cuando el TDA lista se implementa con listas enlazadas, sus operadores asociados se implementan análogamente. En este caso, los operadores se generalizaran, pensando en una lista enlazada general, es decir, sin ningún criterio de orden. Es por ello, que en los procedimientos presentados en las páginas 35-39 del libro base de teoría, aparecen parámetros como p (*posición dentro de la lista*). Toda vez que no hay criterio de orden, las operaciones: *Insertar*, *Recuperar* y *Suprimir* presentan en su entrada el parámetro p . Así, la Inserción, recibe como parámetro un valor de p que indica la posición donde se deberá insertar el nuevo elemento. La explicación del libro, además, indica que el algoritmo insertará el nuevo elemento en la posición p y desplazará todos los elementos siguientes a p a la posición siguiente (es una forma de implementar el algoritmo, no haría falta desplazar ningún nodo, simplemente actuar como se vio en las inserciones entre dos nodos).

El resto de operadores son: *Localizar* (localiza la posición p en que se encuentra un elemento dado x), *Recuperar* (recupera el nodo x que se encuentra en la posición p), *Suprimir* (Suprime el nodo que se encuentra en la posición p), *Suprimir_dato* (elimina de la lista L todas los nodos que contengan un elemento de valor x), *Anula* (Vacía la lista), *Primero* (obtiene el primer nodo de la lista), *Último* (obtiene el último nodo de la lista).

1.3 Algoritmos

Un algoritmo es un método o proceso seguido para resolver un problema. En este contexto, un programa es la implementación de un algoritmo para su ejecución en un computador. Las propiedades básicas que debe satisfacer todo algoritmo son las siguientes:

- En primer lugar el algoritmo debe ser correcto, es decir, debe solucionar el problema en todos los casos posibles.
- Además, debe estar compuesto por una serie dada de pasos y para cada uno de ellos no debe existir ambigüedad en la definición de cual es el siguiente.
- El número de pasos deber ser finito y el algoritmo debe terminar.

En relación a la última propiedad, existen algoritmos que terminan y otros que no, por ejemplo Euclides para calcular MCD, terminará en un número de pasos finitos, sin embargo, el algoritmo para la expansión de la raíz cuadrada de un número natural en fracciones decimales no garantiza que finalice en un número finito de pasos, ya que el resultado puede ser irracional. Sin embargo, en estos casos se entiende que el algoritmo termina introduciendo el concepto de precisión requerida o solución suficientemente terminada.

En general existen diferentes algoritmos para resolver un mismo problema. La elección de uno u otro dependerá de factores tales como el número de datos a manejar. De ahí la importancia del análisis del costo computacional, que indicará cuando un algoritmo es mejor o peor que otro, permitiendo la elección del más adecuado al tipo de problema que trata de resolver.

Los dos factores fundamentales a considerar en el costo de un algoritmo son el tiempo de ejecución y el espacio de almacenamiento requerido para implementarlo.

El coste de los algoritmos se calcula en función de las operaciones características más frecuentes de los mismos (como pueden ser comparaciones y movimientos en el caso de algoritmos de clasificación).

El segundo aspecto básico del análisis del algoritmo está relacionado con el espacio requerido para almacenar los datos. Esta cuestión es fundamental. Dados dos algoritmos con similar coste computacional, se elegirá aquel que requiera un menor almacenamiento.