Estructuras lineales en memoria dinámica

Programación 2

Profesorado de Informática
CeRP del Suroeste, Uruguay

5 de mayo de 2016

Estructuras de datos dinámicas

- Hasta el momento se repasaron algunas estructuras de datos estáticas vistas en 1o año.

Estructuras de datos dinámicas

- Hasta el momento se repasaron algunas estructuras de datos estáticas vistas en 1o año.
- Recordemos que las estructuras estáticas son aquellas en las que la cantidad de memoria necesaria se determina en forma previa a ejecutarse el programa.

Estructuras de datos dinámicas

- Hasta el momento se repasaron algunas estructuras de datos estáticas vistas en 1o año.
- Recordemos que las estructuras estáticas son aquellas en las que la cantidad de memoria necesaria se determina en forma previa a ejecutarse el programa.
- Repasaremos ahora algunas de las estructuras dinámicas vistas en 1o año. Se trata de aquellas en las que la cantidad de memoria necesaria se determina durante la ejecución del programa.

Estructuras de datos dinámicas

- Hasta el momento se repasaron algunas estructuras de datos estáticas vistas en 1o año.
- Recordemos que las estructuras estáticas son aquellas en las que la cantidad de memoria necesaria se determina en forma previa a ejecutarse el programa.
- Repasaremos ahora algunas de las estructuras dinámicas vistas en 1o año. Se trata de aquellas en las que la cantidad de memoria necesaria se determina durante la ejecución del programa.
- Existen múltiples estructuras dinámicas posibles, de momento repasaremos solamente las siguientes: arreglos dinámicos, listas y árboles binarios.

Estructuras de datos dinámicas :: Listas

- Surgen como una alternativa a los arreglos, cuando la cantidad de elementos que se desea almacenar cambia constantemente en tiempo de ejecución. En una lista, los elementos se van insertando de a uno, y por cada nueva inserción se solicita la memoria necesaria en el mismo momento de realizarla.

Estructuras de datos dinámicas :: Listas

- Surgen como una alternativa a los arreglos, cuando la cantidad de elementos que se desea almacenar cambia constantemente en tiempo de ejecución. En una lista, los elementos se van insertando de a uno, y por cada nueva inserción se solicita la memoria necesaria en el mismo momento de realizarla.

- Nótese la diferencia con los arreglos dinámicos, en los cuales la totalidad de celdas necesarias era pedida de una sola vez.

Nodos

- En una lista, cada elemento se almacena en una estructura especial denominada nodo. Un nodo está compuesto por el valor que se desea almacenar (la información) y un puntero que referencia al siguiente nodo de la lista. Luego la lista será una concatenación de nodos.

Nodos

- La definición de un nodo (en lenguaje C) en el cual la información que se desea almacenar es de tipo int es la siguiente:

```
typedef struct nodo_l {
        int info;
        nodo_l *sig;
} nodo;
```

Nodos

- La definición de un nodo (en lenguaje C) en el cual la información que se desea almacenar es de tipo int es la siguiente:

```
typedef struct nodo_l {
        int info;
        nodo_l *sig;
} nodo;
```

- Cabe preguntarse ahora, ¿qué pasa con el campo sig del último nodo de la lista?, es decir, es un puntero, a dónde debería apuntar para que se sepa que ese nodo es el último?.

Nodos

- La definición de un nodo (en lenguaje C) en el cual la información que se desea almacenar es de tipo int es la siguiente:

```
typedef struct nodo_l {
        int info;
        nodo_l *sig;
} nodo;
```

- Cabe preguntarse ahora, ¿qué pasa con el campo sig del último nodo de la lista?, es decir, es un puntero, a dónde debería apuntar para que se sepa que ese nodo es el último?.
- Otro problema que se plantea es, ¿qué sucede cuando una lista es vacía?. Ya que si se ve a la lista como al primero de sus nodos, en el caso de que no se tenga ninguno, no sería posible representar una lista vacía.

Lista simple

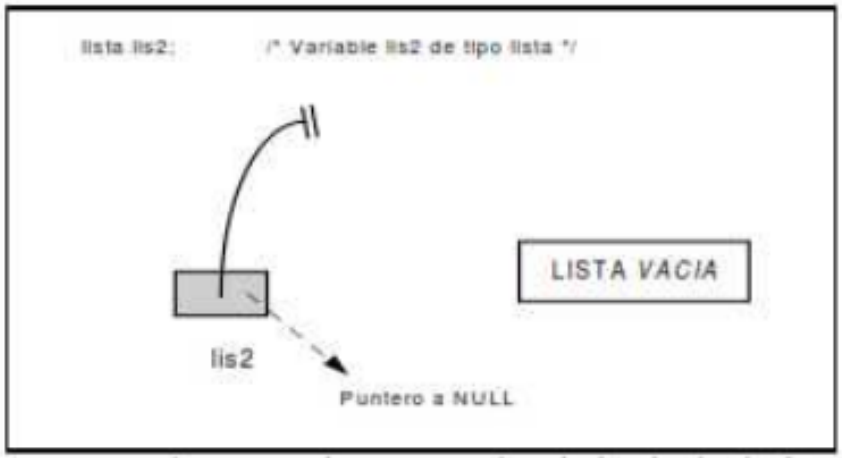
- La solución tiene que ver con lo expuesto anteriormente para el último nodo. De hecho si consideramos la sub-lista conformada por lo que se encuentra después del último nodo, esa sub- lista es una lista vacía, y la manera de representarla es mediante un puntero a NULL. Por lo tanto, consideraremos a una lista como un puntero que apunta a NULL si la lista es vacía, o apunta al primer nodo si no lo es. En conclusión, una lista es un puntero a nodo.

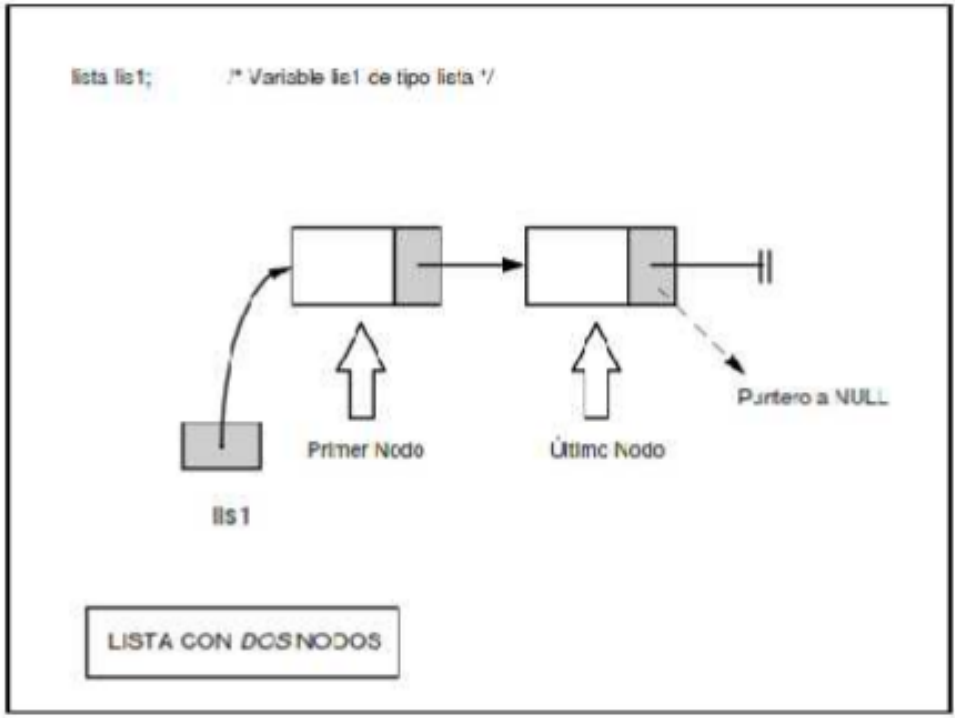
```
typedef struct nodo_l {
        int info;
        nodo_l *sig;
} nodo;

typedef nodo *lista;
```

Lista simple

- Gráficamente una lista vacía se puede representar de la siguiente forma:





Inserción de un Nodo

- Ahora veremos una operación que, dada una lista y un nuevo valor a almacenar en ella, inserta el valor al principio de la lista. Nótese que en una lista se puede insertar también al final, o incluso en algún lugar entre medio, dependiendo del criterio que se desee tener en relación al orden de los valores en la lista. Dado que la inserción al comienzo de la lista es el mecanismo más sencillo, lo proponemos ahora por tratarse de un primer ejemplo.

```
void InsFront(lista &l, T e){
        lista aux = new nodo; // pido memoria para el nuevo nodo
        aux->info=e; // almaceno el valor en el campo información
        aux->sig=l; // hago apuntar el siguiente a lo que apunta l.
        l = aux; // por último, hago apuntar l al nuevo nodo
}
```

Pausa en al lectura...

Iteraciones y recursiones

- Una vez que se cuenta con una lista armada, es posible realizar operaciones que realicen recorridas sobre la lista. Veremos que existen dos maneras posibles de realizar dichas recorridas, en forma **iterativa** y en forma **recursiva**.

Iteraciones y recursiones

- **Ejemplo:** La siguiente función calcula en forma iterativa la cantidad de elementos de una lista:

```
int SumaIterativa (lista l){
        int cont = 0;
        while (l != NULL){
                cont = cont + 1;
                l = l->sig;
        }
        return cont;
}
```

Iteraciones y recursiones

- **Ejemplo:** La siguiente función calcula en forma iterativa la cantidad de elementos de una lista:

```
int SumaIterativa (lista l){
        int cont = 0;
        while (l != NULL){
                cont = cont + 1;
                l = l->sig;
        }
        return cont;
}
```

- Ahora presentamos una implementación recursiva para la función anterior:

```
int SumaRecursiva (lista l){
        if (l == NULL)
                return 0;
        else
                return 1 + SumaRecursiva(l->sig);
}
```