U.D.4

Estructuras de almacenamiento

(6ª parte)

PRO – IES EL MAJUELO

ÍNDICE

ASIGNACIÓN DINÁMICA DE MEMORIA

Problemas de las estructuras estáticas:

- Debe conocerse su tamaño de antemano.
- El tamaño es invariable (puede producirse desperdicio o puede faltar espacio).

Principios de las estructuras dinámicas:

- El espacio se reserva conforme el programa va creando las estructuras y se libera conforme el programa deja de necesitarlas.
- Los límites no son marcados por el programador.

Funciones para la asignación dinámica de memoria

(Se encuentran en la librería stdlib.h)

- `malloc`: Reserva de espacio.
- `calloc`: Reserva e inicialización de espacio.
- `realloc`: Redimensionamiento de espacio reservado.
- `free`: Liberación de espacio.

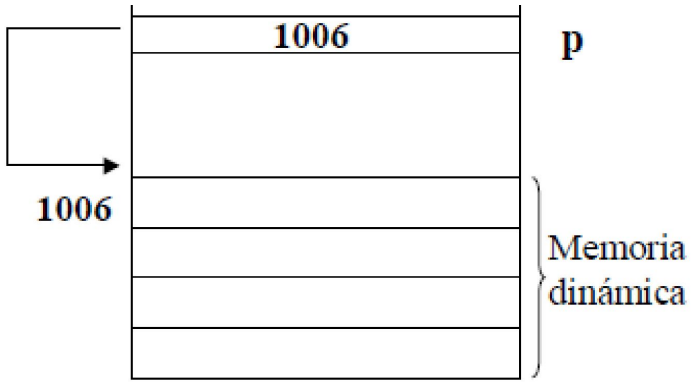
Función `malloc`

```
void * malloc (numb)
```

- Asigna un espacio de memoria de tamaño `numb` bytes y devuelve la dirección de comienzo.
- Si no hay espacio suficiente devuelve el puntero a `NULL`.
- El puntero devuelto es de tipo “no especificado” (`void *`) por lo que habrá que realizar una conversión de tipos explícita.
- La función no inicializa el espacio de memoria.

Ejemplo uso `malloc`

```
int * p;
p = (int *) malloc (4 * sizeof(int));

if (p == NULL)
   printf("No hay memoria");
else
{
   //Tratamiento de esa zona.
   //Liberar.
}
```



Función `free`

```
void free (void *)
```

- Libera el espacio de memoria asignado mediante las funciones `calloc`, `malloc`, `realloc`.
- De esta forma se indica al S.O. que no se volverá a necesitar esta zona de memoria y que queda libre.

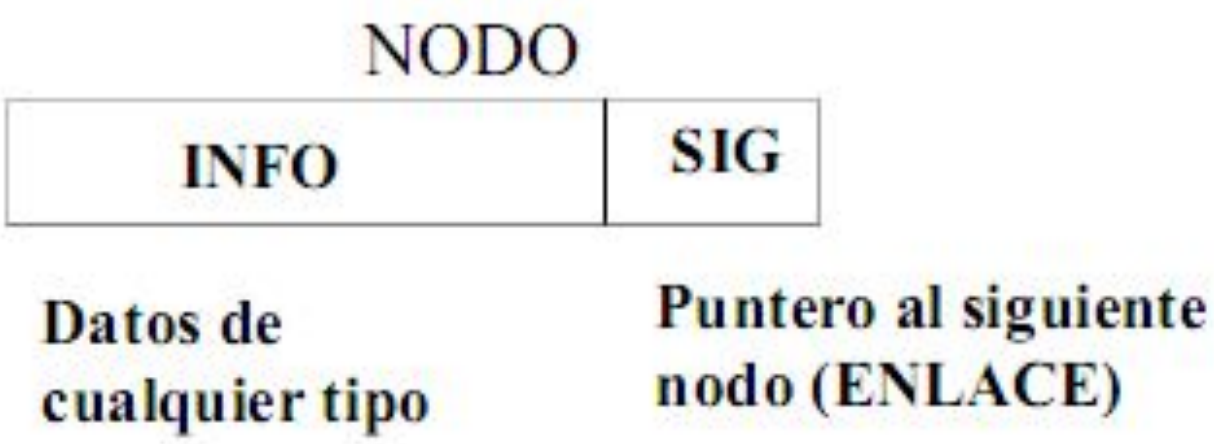
```
int * p;
p = (int *) malloc (4* sizeof(int));
.......
free (p);
```

Estructuras dinámicas más utilizadas

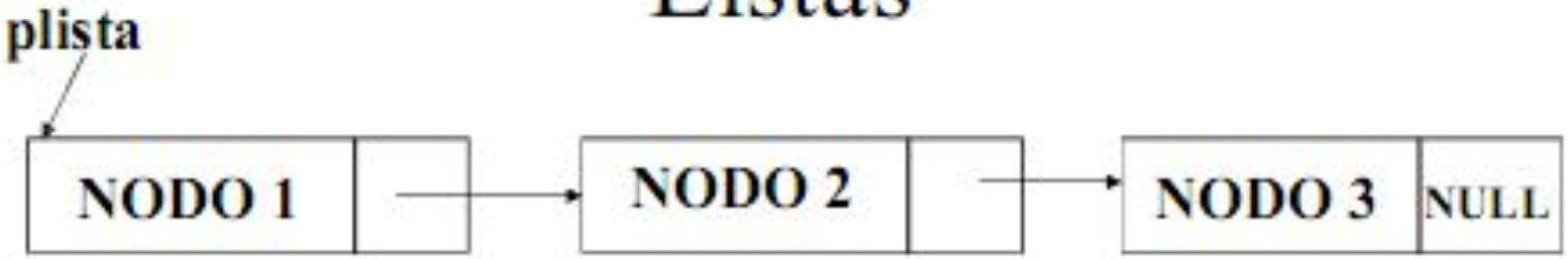
- Listas:
  - Enlace simple.
  - Enlace doble.
- Cola.
- Pila.
- Árbol.
  - Árbol Binario.
  - Árbol B.
- Grafo.

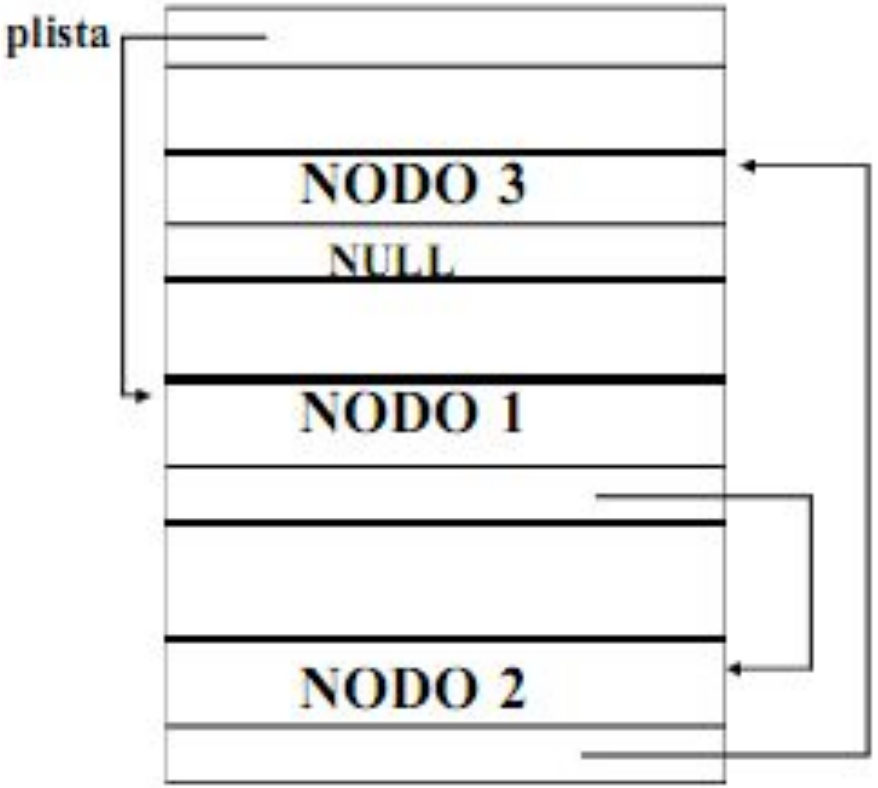
TIPO LISTA

- Una LISTA es una sucesión de elementos del mismo tipo, que tienen un orden pero cuyo número es indeterminado pudiendo variar durante la ejecución del programa.
- A los elementos de una lista se le llama nodos.
- Los nodos no se almacenan en posiciones consecutivas de memoria, por lo que se necesita la existencia de un enlace entre los nodos.



Listas





- El último elemento de la lista debe contener en el enlace un valor NULL.
- Para poder acceder a todo la lista se necesita un puntero al primer nodo de la lista (plista)

Creación de una lista

Definir un registro con los campos de información:

```
typedef struct
{
   char nombre[30];
   int edad;
}Persona;
```

Definir el registro Nodo:

```
typedef struct nodo
{
   Persona datospers;
   struct nodo * sig;
}Nodo;
```

Definir la lista:

```
void main()
{
   Nodo * lista = NULL; // Inicialmente estará vacía.
   . . . . . .
}
```

OPERACIONES BÁSICAS

a. Inserción de un elemento:
   - Al principio de la estructura.
   - Detrás de otro elemento.

b. Recorrido de la estructura.

c. Borrado de un elemento.

d. Búsqueda de un elemento.

e. Liberación de la memoria reservada para la estructura.

A continuación se describirán ejemplos de estas operaciones basados en la estructura de "lista simplemente enlazada".

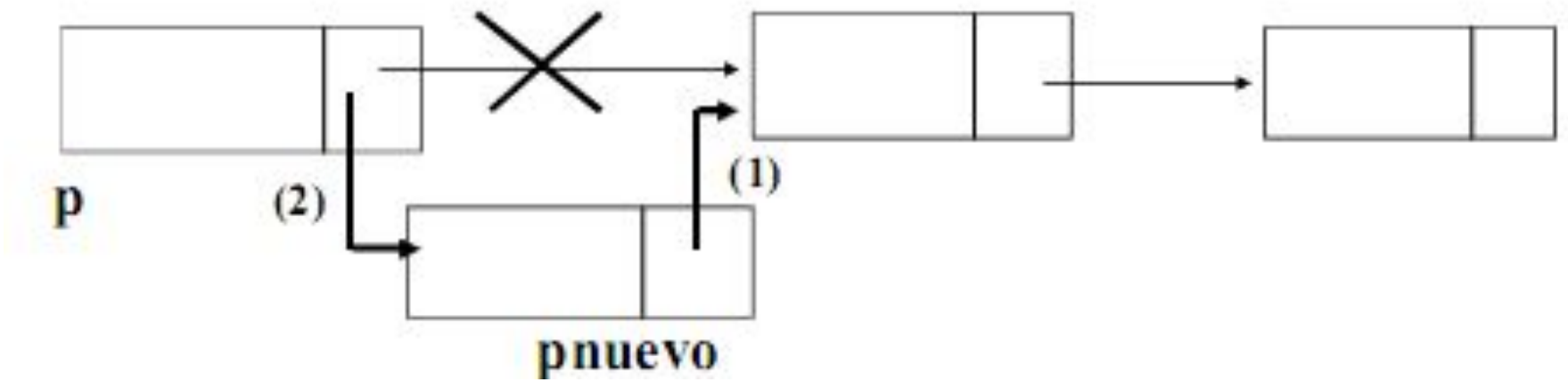
Inserción al principio

1. Reservar memoria para el nuevo nodo y darle valores.
2. Reasignar los punteros para que el nuevo nodo sea el primero de la lista.

```
Nodo * pnuevo;

// Reservar y asignar:
pnuevo = (Nodo *) malloc (sizeof(Nodo));
strcpy ((pnuevo -> datospers).nombre, “Pepe”);
pnuevo -> datospers.edad = 18;

// Reasignar punteros:
pnuevo -> sig = lista;
lista = pnuevo;
```

Inserción detrás de otro elemento

```
Nodo * pnuevo;

// Reservar y asignar (IDEM ANTERIOR)
// Reasignar punteros (IMPORTANTE EL ORDEN!)
pnuevo -> sig = p -> sig; // (1)
p -> sig = pnuevo; // (2)
```



Recorrido de la lista

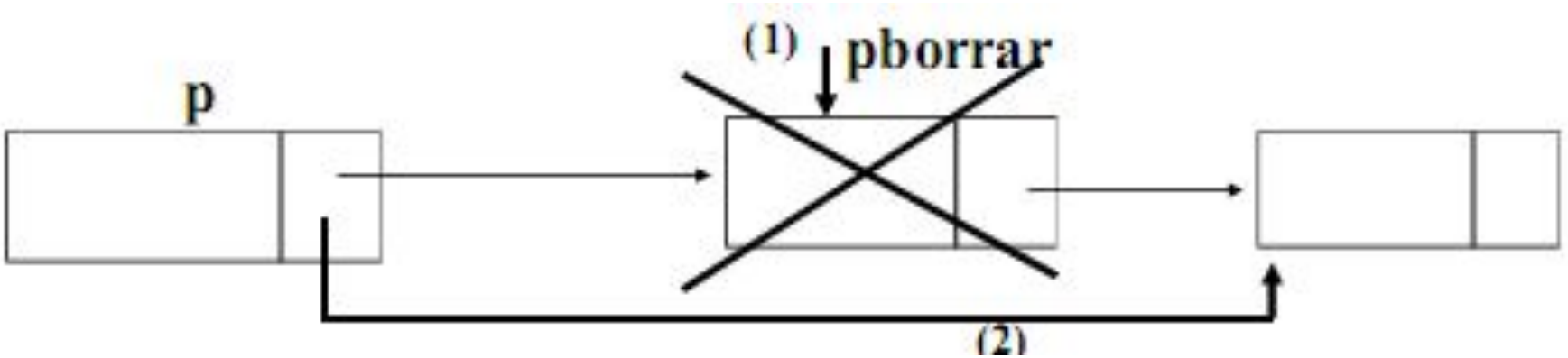
Utilizamos un puntero auxiliar que vaya apuntando a cada elemento de la lista:

```
Nodo * paux;
paux = lista;
while (paux != NULL)
{
   printf (“Nombre %s”, paux -> datospers.nombre);
   printf (“Edad %d”, paux -> datospers.edad);
   paux = paux -> sig;
}
```

Borrado de un elemento

Se borrará el elemento posterior al apuntado por p (nunca podrá eliminarse el primer elemento):

```
Nodo * pborrar;
pborrar = p -> sig; //(1)
p -> sig = pborrar -> sig; //(2)
free (pborrar);
```



Buscar un elemento en la lista

La búsqueda debe realizarse por un campo concreto (por ejemplo, el nombre).

Devuelve NULL si no lo encuentra, y el puntero al nodo si lo encuentra.

```
Nodo * paux = lista;
int encontrado = 0;
while (paux != NULL && encontrado == 0)
{
   if (strcmp(paux -> datospers.nombre,
              nombbuscado) == 0)
       encontrado = 1;
   else
      paux = paux -> sig;
}
return paux;
```

Liberar la lista

Nos hace falta un puntero auxiliar para ir recorriendo la lista y liberando nodo a nodo:

```
Nodo * paux;
paux = lista;
while (paux != NULL)
{
    lista = paux -> sig;
    free (paux);
    paux = lista;
}
```