Unidad 2: Estructuras de Datos

Tema III. Estructuras de datos Compuestas.

Pilas, Colas y Listas implementadas con punteros

Algoritmos y Estructuras de Datos II (Plan 2009)

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

Punteros

- Las estructuras de datos estudiadas hasta ahora se almacenan estáticamente en la memoria física del computador.

- Esta rigidez en las estructuras de datos estáticas hace que no pueden crecer o menguar durante la ejecución de un programa.

- Por otra parte, la representación de ciertas construcciones (como las listas) usando las estructuras conocidas (concretamente los arrays) tiene que hacerse situando elementos consecutivos en componentes contiguas, de manera que las operaciones de inserción de un elemento nuevo o desaparición de uno ya existente requieren el desplazamiento de todos los posteriores para cubrir el vacío producido, o para abrir espacio para el nuevo.

MEMORIA DINÁMICA

- La memoria dinámica es un espacio de almacenamiento que se solicita en tiempo de ejecución. De esa manera, a medida que el proceso va necesitando espacio para más líneas, va solicitando más memoria al sistema operativo para guardarlas. El medio para manejar la memoria que otorga el sistema operativo, es el **puntero**, puesto que no podemos saber en tiempo de compilación dónde nos dará huecos el sistema operativo (en la memoria de nuestro PC).

- Un dato importante es que como tal este tipo de datos se crean y se destruyen mientras se ejecuta el programa y por lo tanto la estructura de datos se va dimensionando de forma precisa a los requerimientos del programa, evitándonos así perder datos o desperdiciar memoria si hubiéramos tratado de definirla cantidad de memoria a utilizar en el momento de compilar el programa.

MEMORIA DINÁMICA

VENTAJAS:

- Es posible disponer de un espacio de memoria arbitrario que dependa de información dinámica (disponible sólo en ejecución): Toda esa memoria que maneja es implementada por el programador cuando fuese necesario.
- Se puede ir incrementando durante la ejecución del programa. Esto permite, por ejemplo, trabajar con arreglos dinámicos.
- Es memoria que se reserva en tiempo de ejecución. Su tamaño puede variar durante la ejecución del programa y puede ser liberado mediante la función free.

DESVENTAJAS:

- Es difícil de implementar en el desarrollo de un programa o aplicación.
- Es difícil implementar estructuras de datos complejas como son los tipos recursivos (árboles, grafos, etc.).Por ello necesitamos una forma para solicitar y liberar memoria para nuevas variables que puedan ser necesarias durante la ejecución de nuestros programas: Heap.
- La memoria dinámica puede afectar el rendimiento. Puesto que con la memoria estática el tamaño de las variables se conoce en tiempo de compilación, esta información está incluida en el código objeto generado. Cuando se reserva memoria de manera dinámica, se tienen que llevar a cabo varias tareas, como buscar un bloque de memoria libre y almacenar la posición y tamaño de la memoria asignada, de manera que pueda ser liberada más adelante. Todo esto representa una carga adicional, aunque esto depende de la implementación y hay técnicas para reducir su impacto.

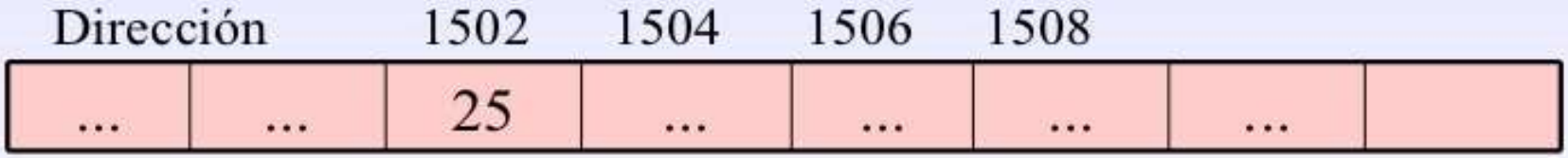
PUNTEROS

¿Qué es un PUNTERO?:

Un puntero es un objeto que apunta a otro objeto. Es decir, una variable cuyo valor es la dirección de memoria de otra variable.

No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.

```
int x = 25;
```



La dirección de la variable x (&x) es 1502

El contenido de la variable x es 25

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

PUNTEROS

Las direcciones de memoria dependen de la arquitectura del ordenador y de la gestión que el sistema operativo haga de ella.

En lenguaje ensamblador se debe indicar numéricamente la posición física de memoria en que queremos almacenar un dato. De ahí que este lenguaje dependa tanto de la máquina en la que se aplique.

En C no debemos, ni podemos, indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable. Lo que nos interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

PUNTEROS

Una variable puntero se declara como todas las variables. Debe ser del mismo tipo que la variable apuntada. Su identificador va precedido de un asterisco (*):

```
int *punt;
```

Es una variable puntero que apunta a variable que contiene un dato de tipo entero llamada punt.

```
char *car:
```

Es un puntero a variable de tipo carácter.

```
long float *num;

float *mat[5];   // . . .
```

FUND. PROG.

Un puntero tiene su **propia dirección** de memoria:

&punt

&car 4

PUNTEROS

```
int *punt = NULL, var = 14;

punt = &var;
printf("%#X,    %#X", punt, &var)  //la misma salida: dirección

printf("\n%d,  %d", *punt, var);   //salida:  14,  14
```

Hay que tener cuidado con las direcciones apuntadas:

```
printf("%d,  %d", *(punt+1), var+1);
```

*(punt + 1) repesenta el valor contenida en la dirección de memoria aumentada en una posición (int=2bytes), que será un valor no deseado. Sin embargo var+1 representa el valor 15.

punt + 1 representa lo mismo que &var + 1 (avance en la dirección de memoria de var).

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

PUNTEROS

Al trabajar con punteros se emplean dos operadores específicos:

- Operador de dirección: **&** Representa la dirección de memoria de la variable que le sigue:

&fnum representa la dirección de fnum.

- Operador de contenido o indirección: *

El operador * aplicado al nombre de un puntero indica el valor de la variable apuntada:

```
float  altura = 26.92,  *apunta;

apunta = &altura; //inicialización del puntero
```


Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

Un puntero siempre está asociado a objetos de un tipo → sólo puede apuntar a objetos (variables o vectores) de ese tipo.

```
int *ip;    /* Sólo puede apuntar a variables enteras */
char *c;    /* Sólo puede apuntar a variables carácter */
double *dp, /* dp sólo puede apuntar a variables reales */
     atof (char *); /* atof es una función que devuelve un real
                       dada una cadena que se le pasa como
                       puntero a carácter*/
```

Es necesario utilizar paréntesis cuando aparecen en la misma expresión que otros operadores unarios como ++ o --, ya que en ese caso se evaluarían de izquierda a derecha.

```
++*ip;
(*ip)++;
```

Dado que los punteros son variables, también pueden usarse como asignaciones entre direcciones. Así:

```
int *ip, *iq;
iq = ip;   /* Indica que iq apunta a donde apunte el puntero ip. */
```

En C, por defecto, todos los parámetros a las funciones se pasan por valor (la función recibe una copia del parámetro, que no se ve modificado por los cambios que la copia sufra dentro de la función).
Ejemplo: Intercambio de dos valores

{VERSION ERRONEA}

```
intercambia (int a, int b) {
int tmp;

tmp = a;
a = b;
b = tmp
}
```

{VERSION CORRECTA}

```
intercambia (int *a, int *b) {
int tmp;

tmp = *a;
*a = *b;
*b = tmp
}
```

Para que un parámetro de una función pueda ser modificado, ha de pasarse por referencia, y en C eso sólo es posible pasando la dirección de la variable en lugar de la propia variable.
Si se pasa la dirección de una variable, la función puede modificar el contenido de esa posición (no así la propia dirección, que es una copia).

Punteros y vectores

En C los punteros y los vectores están fuertemente relacionados, hasta el punto de que el nombre de un vector es en sí mismo un puntero a la primera (0-ésima) posición del vector. Todas las operaciones que utilizan vectores e índices pueden realizarse mediante punteros.

int v[10];

1	3	5	7	9	11	13	15	17	19
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]

v: designa 10 posiciones consecutivas de memoria donde se pueden almacenar enteros.

int *ip;

Designa un puntero a entero, por lo tanto puede acceder a una posición de memoria. Sin embargo, como se ha dicho antes v también puede considerarse como un puntero a entero, de tal forma que las siguientes expresiones son equivalentes:

```
ip = &v[0]
x = *ip;
*(v + 1)
v + I
```

```
ip = v
x = v[0];
v[1]
&v[i]
```

Aritmética de punteros

El compilador C es capaz de "adivinar" cuál es el tamaño de una variable de tipo puntero y realiza los incrementos/decrementos de los punteros de la forma adecuada. Así:

```
int v[10], *p;
p = v;

/* p          Apunta a la posición inicial del vector*/
/* p + 0 Apunta a la posición inicial del vector*/
/* p + 1 Apunta a la segunda posición del vector*/
/* p + i  Apunta a la posición i+1 del vector*/

p = &v[9];  /* p apunta ahora a la última posición (décima) del vector */

/* p - 1  Apunta a la novena posición del vector*/
/* p - i  Se refiere a la posición 9 - i en v*/
```

Aritmética de punteros

Ejemplo de recorrido de un vector utilizando índices y punteros.

```
/* prog3.c*/
main(){
int v[10];
int i, *p;

for (i=0; i < 10; i++) v[i] = i;

for (i=0; i < 10; i++) printf ("\n%d", v[i]);

p = v;
for (i=0; i < 10; i++) printf ("\n%d", *p++);
/* Tras cada p++ el puntero señala a la siguiente posición en v */
}
```


Mostrar programa en “C” …

Asignación dinámica de memoria

- `void *calloc(size_t nobj, size_t size)`

calloc obtiene (reserva) espacio en memoria para alojar un vector (una colección) de nobj objetos, cada uno de ellos de tamaño size. Si no hay memoria disponible se devuelve NULL. El espacio reservado se inicializa a bytes de ceros.

Obsérvese que calloc devuelve un (void *) y que para asignar la memoria que devuelve a un tipo Tipo_t hay que utilizar un operador de ahormado: (Tipo_T *)

Ejemplo:

```
char * c;
c = (char *) calloc (40, sizeof(char));
```

Asignación dinámica de memoria

- `void *malloc(size_t size)`

malloc funciona de forma similar a calloc salvo que: a) no inicializa el espacio y b) es necesario saber el tamaño exacto de las posiciones de memoria solicitadas.

El ejemplo anterior se puede reescribir:

```
char * c;
c = (char *) malloc (40*sizeof(char));
```

Asignación dinámica de memoria

- `void *realloc(void *p, size_t size)`

realloc cambia el tamaño del objeto al que apunta p y lo hace de tamaño size. El contenido de la memoria no cambiará en las posiciones ya ocupadas. Si el nuevo tamaño es mayor que el antiguo, no se inicializan a ningún valor las nuevas posiciones. En el caso en que no hubiese suficiente memoria para "realojar" al nuevo puntero, se devuelve NULL y p no varía.

- `void free(void *p)`

free() libera el espacio de memoria al que apunta p. Si p es NULL no hace nada. Además p tiene que haber sido "alojado" previamente mediante malloc(), calloc() o realloc().

Asignación dinámica de memoria

El puntero que se pasa como argumento ha de ser NULL o bien un puntero devuelto por malloc(), calloc() o realloc().

```
#define N 10
#include <stdio.h>

main(){
char c, *cambiante;
int i;

i=0;
cambiante = NULL;

printf("\nIntroduce una frase. Terminada en [ENTER]\n");
while ((c=getchar()) != '\n') {
  if (i % N == 0){
   printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
   cambiante=(char *)realloc((char *)cambiante,(i+N)*sizeof(char));
   if (cambiante == NULL) exit(-1);
  }
  /* Ya existe suficiente memoria para el siguiente carácter*/
  cambiante[i++] = c;
}
 /* Antes de poner el terminador nulo hay que asegurarse de que haya
 suficiente memoria */
 if ((i % N == 0) && (i != 0)){
       printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
       cambiante=realloc((char *) cambiante, (i+N)*sizeof(char));
         if (cambiante == NULL) exit(-1);
   }
 cambiante[i]=0;

 printf ("\nHe leido %s", cambiante);
 }
```

LISTAS … recordemos

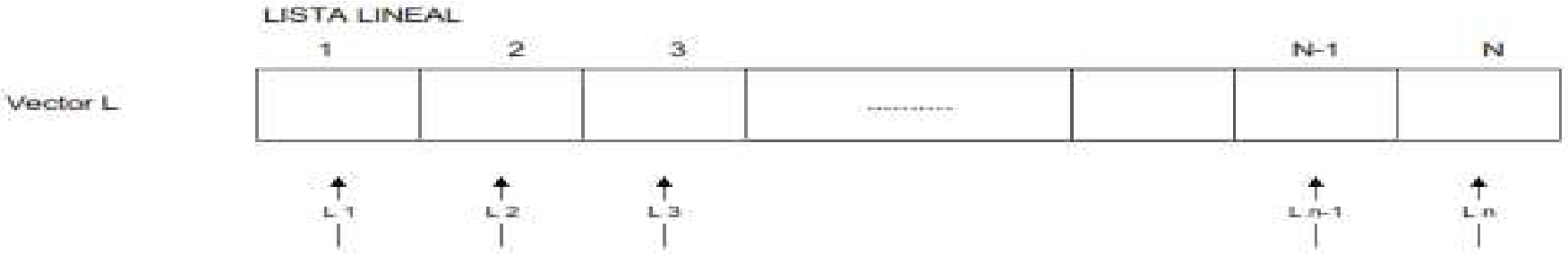
Una **lista** es un conjunto ordenado de elementos de un tipo, la misma puede variar el número de elementos.

***Lista* L: L1, L2, L3, L4,....., LN** *donde LN es un elemento de la lista.*

Cada elemento de la lista – a excepción del primero – tiene un único predecesor y a su vez cada elemento – a excepción del último – tiene un único sucesor.

Los elementos de una lista lineal normalmente se almacenan uno detrás de otro en posiciones consecutivas de memoria, por ejemplo las entradas en una guía telefónica. Cuando una lista se almacena en la memoria principal de una computadora lo está haciendo en posiciones sucesivas de memoria; cuando se almacena en cinta magnética, los elementos sucesivos se presentan en sucesión en la cinta.
Esta asignación de memoria toma el nombre de *almacenamiento secuencial*, en apartados posteriores veremos el tipo de *almacenamiento encadenado o enlazado*.

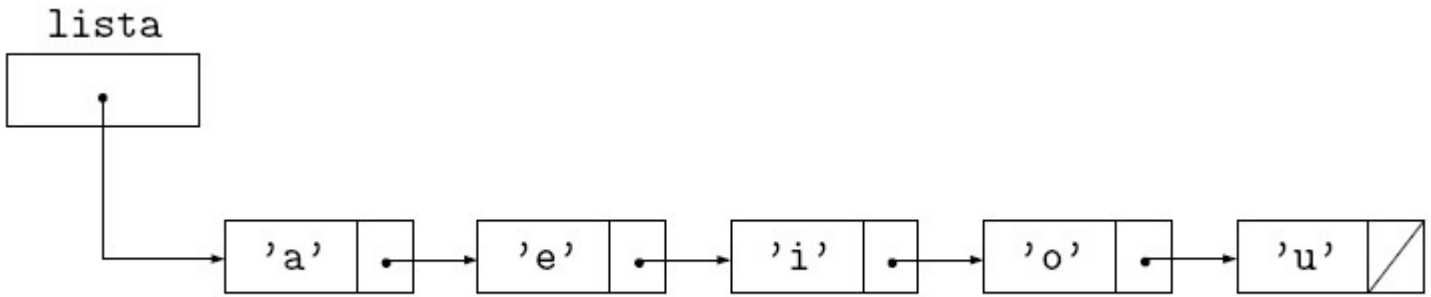
Uno de los medios mas clásicos de realizar una lista lineal es mediante un vector, donde los elementos se sitúan en posiciones físicamente contiguas. Su representación



Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

Estructuras lineales: las listas enlazadas

- Las representaciones para listas (con punteros) nos permitirán insertar y borrar elementos más fácil y eficientemente que en una implementación estática (para listas acotadas) usando el tipo de datos array.



Cómo funciona una lista

- Para crear una lista enlazada recordamos los conocimientos sobre estructuras y asignación dinámica de memoria.

  Crear una sencilla agenda que contiene el nombre y el número de teléfono.

  Una lista enlazada simple necesita una estructura con varios campos, los campos que contienen los datos necesarios (nombre y teléfono) y otro campo que contiene un puntero a la propia estructura. Este puntero se usa para saber dónde está el siguiente elemento de la lista, para saber la posición en memoria del siguiente elemento.

```
struct _agenda {
        char nombre[20];
        char telefono[12];
        struct _agenda *siguiente;
        };
```

Definición de Tipo de Lista

- Esencialmente, una lista será representada como un puntero que señala al principio (o cabeza) de la lista.

```
typedef struct ElementoLista {
        char *dato;
        struct ElementoLista *siguiente;
} Elemento;
```

Para tener el control de la lista es preferible guardar determinados elementos: el primer elemento, el último elemento, el número de elementos. Para ello será empleado otra estructura (no es obligatorio, pueden ser utilizadas variables).

```
typedef struct ListaIdentificar {
        Elemento *inicio;
        Elemento *fin;
        int tamaño;
}Lista;
```

Operaciones sobre las listas enlazadas

Inicialización

void inicializacion (Lista *lista);

Esta operación debe ser hecha antes de otra operación sobre la lista. Esta comienza el puntero **inicio** y el puntero **fin** con el puntero NULL, y el **tamaño** con el valor 0.

```
void inicializacion (Lista *lista){
        lista->inicio = NULL;
        lista->fin = NULL;
tamaño = 0;
}
```

Inserción de un elemento en la lista

A continuación el algoritmo de inserción y el registro de los elementos, pasos:

a.1.- declaración del elemento que se va a insertar;
a.2.- asignación de la memoria para el nuevo elemento;
a.3.- llena el contenido del campo de datos;
a.4.- actualización de los punteros hacia el primer y último elemento si es necesario.

Caso particular: en una lista con un único elemento, el primero es al mismo tiempo el último. Actualizar el tamaño de la lista

Casos para insertar elementos a una lista enlazada:

- en una lista vacía,
- al inicio de la lista,
- al final de la lista
- en otra parte de la lista.

Inserción en una lista vacía

```
int ins_en_lista_vacia (Lista *lista, char *dato);
```

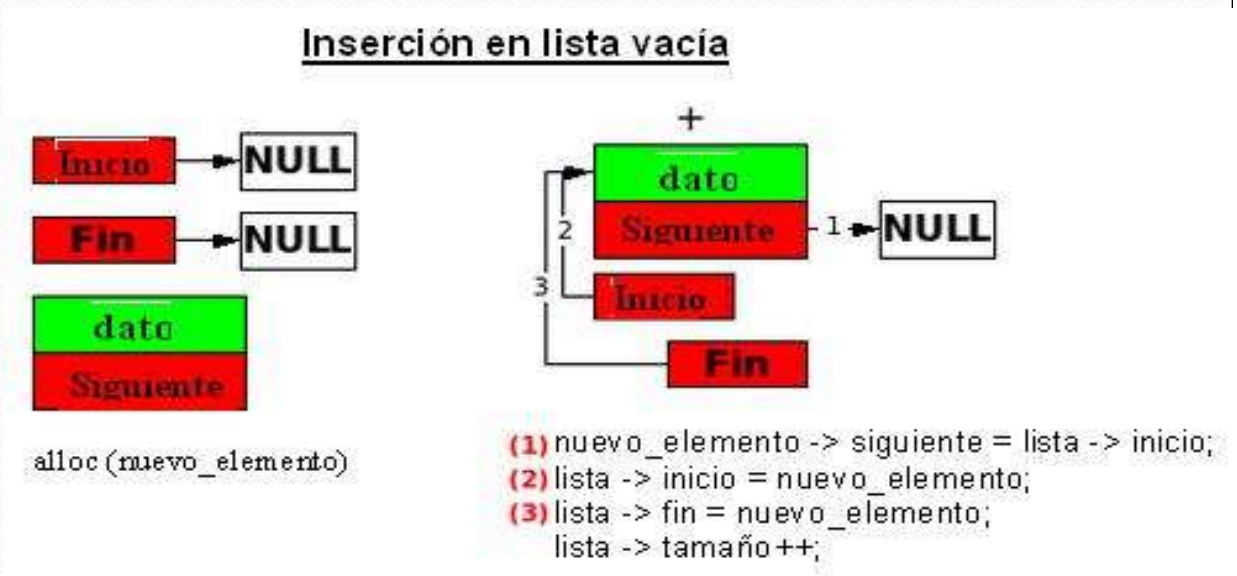
La función retorna 1 en caso de error, si no devuelve 0.

Las etapas son asignar memoria para el nuevo elemento, completa el campo de datos de ese nuevo elemento, el puntero **siguiente** de este nuevo elemento apuntará hacia NULL (ya que la inserción es realizada en una lista vacía, se utiliza la dirección del puntero **inicio** que vale NULL), los punteros **inicio** y **fin** apuntaran hacia el nuevo elemento y el tamaño es actualizado

Inserción en una lista vacía

```
int ins_en_lista_vacia (Lista * lista, char *dato){
Element *nuevo_elemento;
if ((nuevo_elemento = (Element *) malloc (sizeof (Element))) == NULL)
return -1;
if ((nuevo _elemento->dato = (char *) malloc (50 * sizeof (char)))
== NULL)
return -1;
strcpy (nuevo_elemento->dato, dato);

nuevo_elemento->siguiente = NULL;
lista->inicio = nuevo_elemento;
lista->fin = nuevo_elemento;
lista->tamaño++;
return 0;
}
```



Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

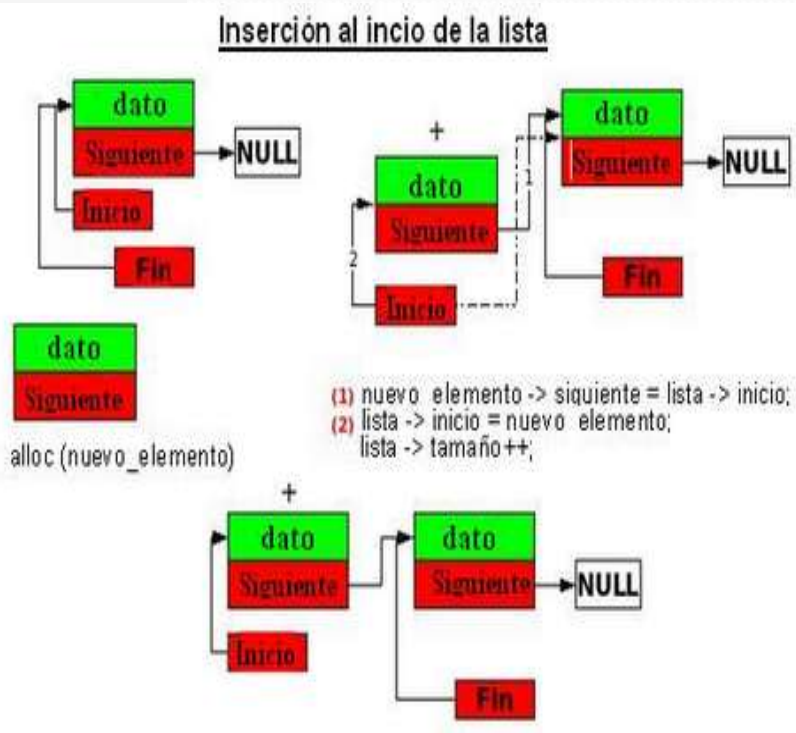
Inserción al inicio de la lista

int ins_inicio_lista (Lista *lista,char *dato);

Etapas: asignar memoria al nuevo elemento, rellenar el campo de datos de este nuevo elemento, el puntero **siguiente** del nuevo elemento apunta hacia el primer elemento, el puntero **inicio** apunta al nuevo elemento, el puntero **fin** no cambia, el tamaño es incrementado.

```
/* inserción al inicio de la lista */
int ins_inicio_lista (Lista * lista, char
*dato){
  Element *nuevo_elemento;
  if ((nuevo_elemento = (Element *) malloc
(sizeof (Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc
(50 * sizeof (char)))
      == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  nuevo_elemento->siguiente = lista->ir
  lista->inicio = nuevo_elemento;
  lista->tamaño++;
  return 0;
}
```



Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

Inserción al final de la lista

int ins_fin_lista (Lista *lista, Element *actual, char *dato);

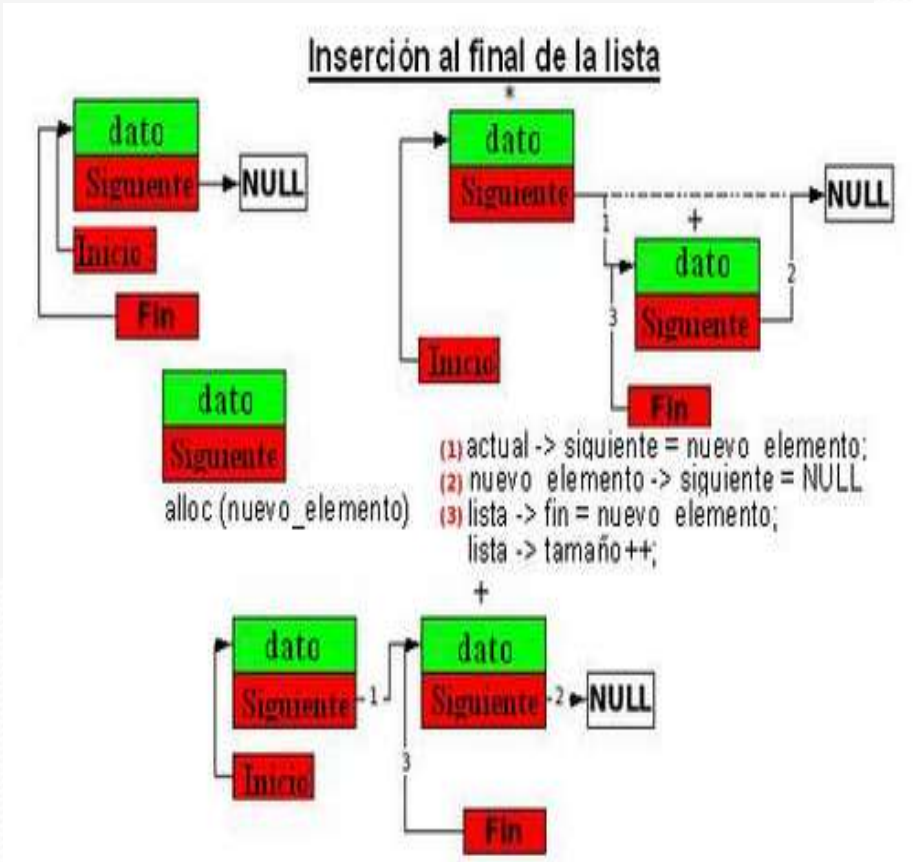
Etapas: proporcionar memoria al nuevo elemento, rellenar el campo de datos del nuevo elemento, el puntero **siguiente** del ultimo elemento apunta hacia el nuevo elemento, el puntero **fin** apunta al nuevo elemento, el puntero **inicio** no varía, el tamaño es incrementado:

```
/*inserción al final de la lista */
int ins_fin_lista (Lista * lista, Element * actual, char *dato){
  Element *nuevo_elemento;
  if ((nuevo_elemento = (Element *) malloc (sizeof (Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 * sizeof (char)))
      == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  actual->siguiente = nuevo_elemento;
  nuevo_elemento->siguiente = NULL;

  lista->fin = nuevo_elemento;

  lista->tamaño++;
  return 0;
}
```



Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

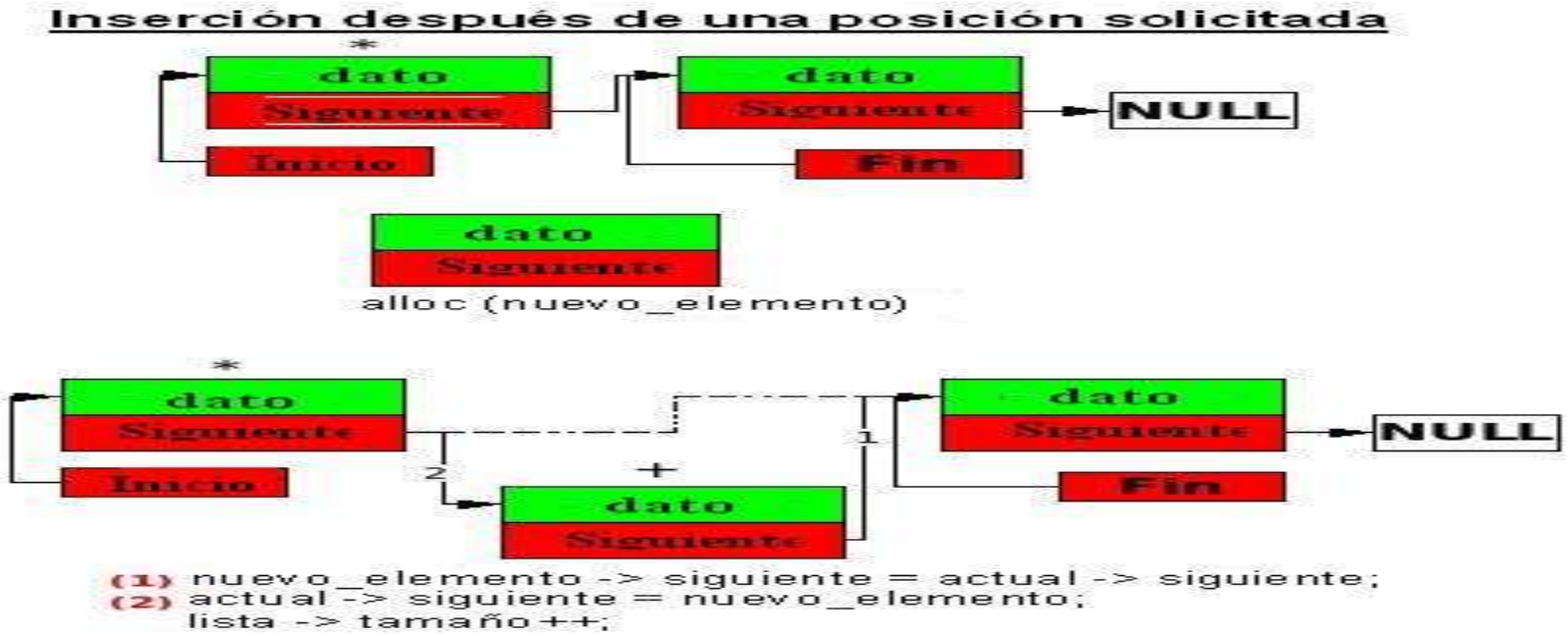
Inserción en otra parte de la lista

int ins_lista (Lista *lista, char *dato,int pos);

La función arroja **-1** en caso de error, si no da **0**.

La inserción se efectuará después de haber pasado a la función una posición como argumento. Si la posición indicada no tiene que ser el último elemento. En ese caso se debe utilizar la función de inserción al final de la lista.

Etapas: asignación de una cantiad de memoria al nuevo elemento, rellenar el campo de datos del nuevo elemento, escoger una posición en la lista (la inserción se hará luego de haber elegido la posición), el puntero **siguiente** del nuevo elemento apunta hacia la dirección a la que apunta el puntero **siguiente** del elemento actual, el puntero **siguiente** del elemento actual apunta al nuevo elemento, los punteros **inicio** y **fin** no cambian, el tamaño se incrementa en una unidad:



```
/* inserción en la posición solicitada */
int ins_lista (Lista * lista, char *dato, int pos){
  if (lista->tamaño < 2)
    return -1;
  if (pos < 1 || pos >= lista->tamaño)
    return -1;

  Element *actual;
  Element *nuevo_elemento;
  int i;

  if ((nuevo_elemento = (Element *) malloc (sizeof (Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 * sizeof (char)))
      == NULL)
    return -1;

  actual = lista->inicio;
  for (i = 1; i < pos; ++i)
    actual = actual->siguiente;
  if (actual->siguiente == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  nuevo_elemento->siguiente = actual->siguiente;
  actual->siguiente = nuevo_elemento;
  lista->tamaño++;
  return 0;
```

Eliminación de un elemento de la lista

A continuación un algoritmo para eliminar un elemento de la lista: uso de un puntero temporal para almacenar la dirección de los elementos a borrar, el elemento a eliminar se encuentra después del elemento actual, apuntar el puntero **siguiente** del elemento actual en dirección del puntero **siguiente** del elemento a eliminar, liberar la memoria ocupada por el elemento borrado, actualizar el tamaño de la lista.

Para eliminar un elemento de la lista hay varios casos:

eliminación al inicio de la lista

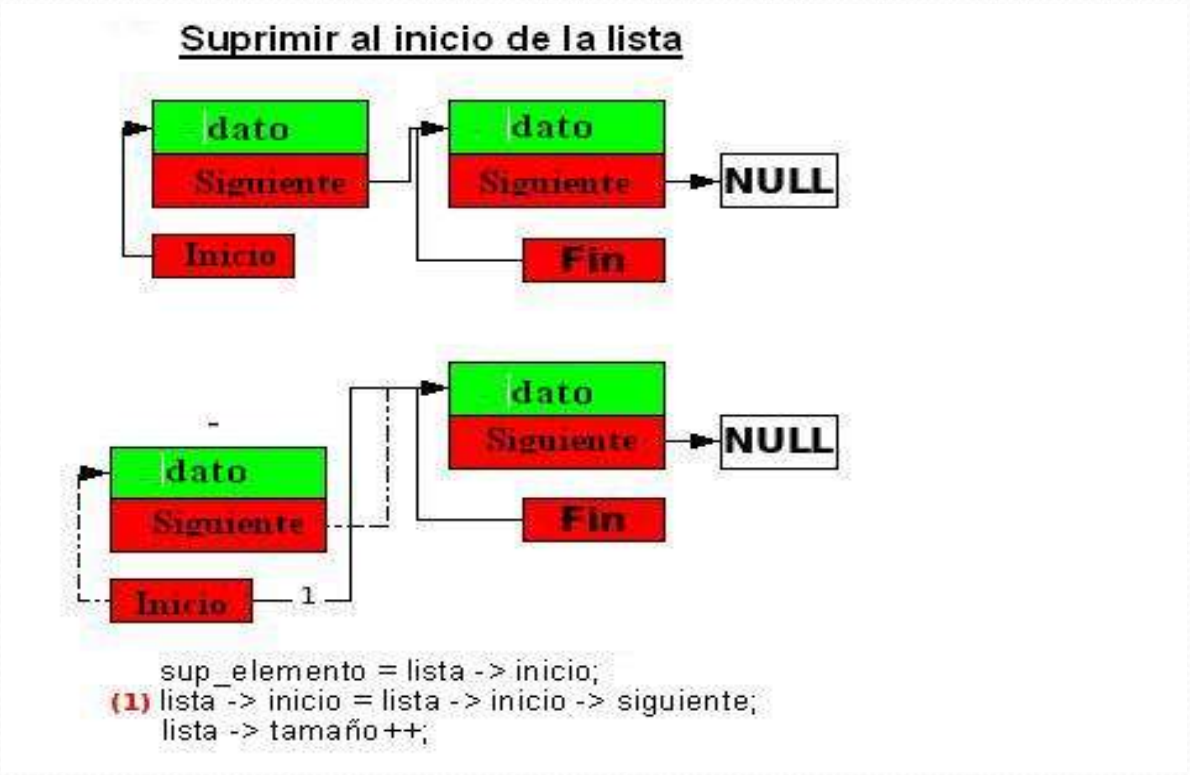
eliminación en otra parte de la lista.

eliminación al final de la lista (penúltimo elemento)

Eliminación al inicio de la lista

int sup_inicio (Lista *lista);

La función devuelve **-1** en caso de equivocación, de lo contrario da **0**. Etapas: el puntero **sup_elem** contendrá la dirección del 1er elemento, el puntero **inicio** apuntara hacia el segundo elemento, el tamaño de la lista disminuirá un elemento:



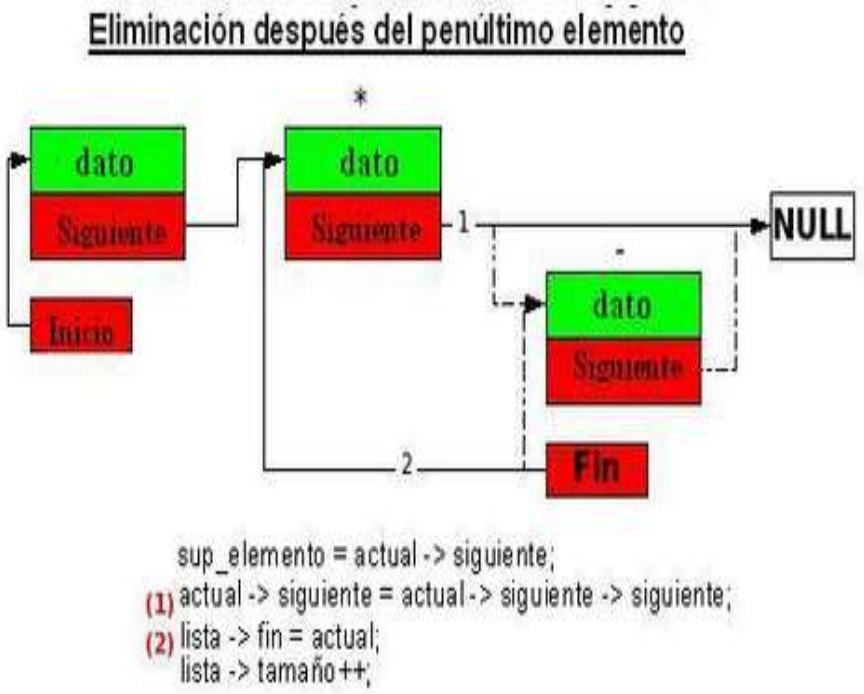
```
/* eliminación al inicio de la lista */
int sup_inicio (Lista * lista){
  if (lista->tamaño == 0)
    return -1;
  Element *sup_elemento;
  sup_element = lista->inicio;
  lista->inicio = lista->inicio->siguiente;
  if (lista->tamaño == 1)
    lista->fin = NULL;
  free (sup_elemento->dato);
  free (sup_elemento);
  lista->tamaño--;
  return 0;
}
```

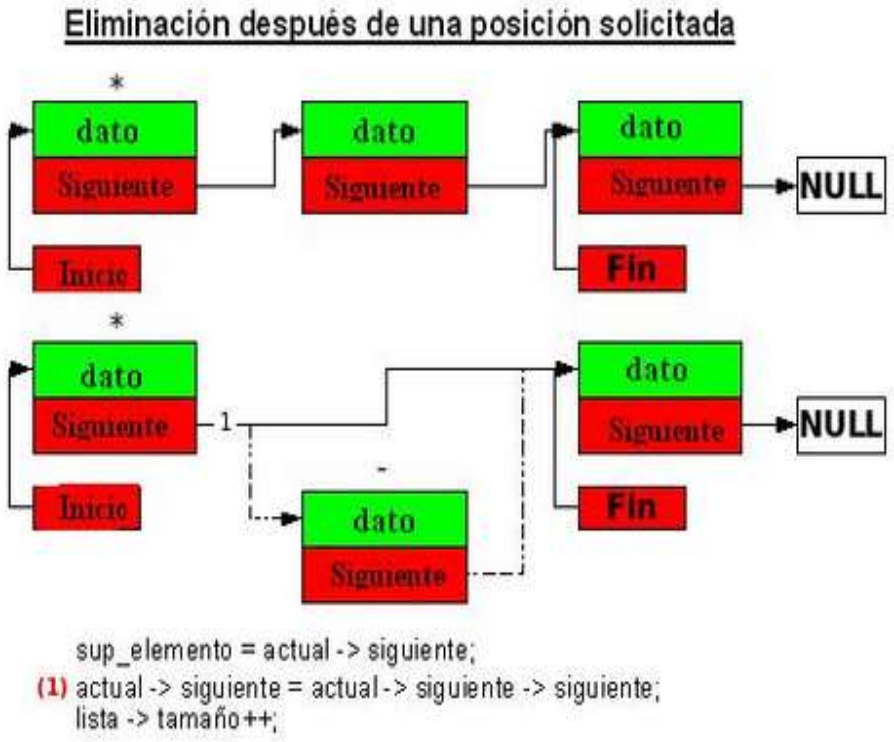
Eliminación en otra parte de la lista

int sup_en_lista (Lista *lista, int pos);

La función da **-1** en caso de error, si no devuelve **0**.

Etapas: el puntero **sup_elem** contendrá la dirección hacia la que apunta el puntero **siguiente** del elemento **actual**, el puntero **siguiente** del elemento actual apuntara hacia el elemento al que apunta el puntero **siguiente** del elemento que sigue al elemento **actual** en la lista. Si el elemento**actual** es el penúltimo elemento, el puntero **fin** debe ser actualizado. El tamaño de la lista será disminuido en un elemento:





```
/* eliminar un elemento después de la posición solicitada */
int sup_en_lista (Lista * lista, int pos){
  if (lista->tamaño <= 1 || pos < 1 || pos >= lista->tamaño)
    return -1;
  int i;
  Element *actual;
  Element *sup_elemento;
  actual = lista->inicio;

  for (i = 1; i < pos; ++i)
    actual = actual->siguiente;

  sup_elemento = actual->siguiente;
  actual->siguiente = actual->siguiente->siguiente;
  if(actual->siguiente == NULL)
          lista->fin = actual;
  free (sup_elemento->dato);
  free (sup_elemento);
  lista->tamaño--;
  return 0;
}
```

Visualización de la lista

Para mostrar la lista entera hay que posicionarse al inicio de la lista (el puntero **inicio** lo permitirá). Luego usando el puntero **siguiente** de cada elemento la lista es recorrida del primero al ultimo elemento.
La condición para detener es proporcionada por el puntero **siguiente** del ultimo elemento que vale NULL.

```
/* visualización de la lista */
void visualización (Lista * lista){
  Element *actual;
  actual = lista->inicio;
  while (actual != NULL){
      printf ("%p - %s\n", actual, actual->dato);
      actual = actual->siguiente;
  }
}
```

Bibliografía

- Mark Allen Weiss - Estructuras de Datos y Algoritmos - Florida International University - Año: 1995 - Editorial: Addison-Wesley Iberoamericana .

- Joyanes Aguilar, Luis - Programación en Pascal - 4ª Edición - Año: 2006 - Editorial: McGraw-Hill/Interamericana de España, S.A.U.

- Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Andeyro Quesada, Carlos Rossi Jiménez - Algoritmos y Programación en Pascal.

- Joyanes Aguilar, Luis - Fundamentos de la Programación. Algoritmos, Estructuras de Datos y Objetos - 3ª Edición - Editorial: McGraw-Hill

Link de videos para tratamiento de listas enlazadas

- https://www.youtube.com/watch?v=2YPMA1p5KoM
- https://www.youtube.com/watch?v=fcpZ_77Ncm8
- https://www.youtube.com/watch?v=7yOnE3yVjMY
- https://www.youtube.com/watch?v=9-18Qp50QDg
- https://www.youtube.com/watch?v=qI_09fwv3b4
- https://www.youtube.com/watch?v=ljYbVM6j11s
- https://www.youtube.com/watch?v=WxoGvBzWuGs