APUNTADORES

Un apuntador 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 = 57;
```

Dirección		11010	11011	11100	11101		
...	...	57	...	...	...	...	

La dirección de la variable x es 11010
El contenido de la variable x es 57

Al trabajar con apuntadors se emplean dos operadores específicos:

- ✓ Operador de dirección: &
  Representa la dirección de memoria de la variable que le sigue:
  &x representa la dirección de x.
- ✓ Operador de contenido o indirección: *
  El operador * aplicado al nombre de un apuntador indica el valor de la variable apuntada:

```
float x = 33.15, *p;
p = &x; //inicialización del apuntador
```

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

```
int *p;
```

Es una variable apuntador que apunta a variable que contiene un dato de tipo entero llamada p.

```
char *c:
```

Es un apuntador a variable de tipo carácter.

Es decir: hay tantos tipos de apuntadors como tipos de datos, aunque también pueden declararse apuntadors a estructuras más complejas (funciones, registros, ficheros...) e incluso apuntadors vacíos (void) y apuntadors nulos (NULL).

Declaración:

```
<tipo de dato> *<nombre_apuntador>;
```

Utilidad de los apuntadores:

En lenguaje C/C++, los apuntadores cumplen los siguientes objetivos:

a) Paso de parámetros por referencia en C

En C no existen explícitamente los llamados "parámetros por referencia", hay que usar apuntadores para simular este pasaje de argumentos.

Por ejemplo, si tenemos

```
void suma (int a, int b, int c)
{
   c = a+b;
}
main ()
{
  int x,y,z;
  x = 4;
  y= 2;
  suma (x,y,z);
  printf ("z = ", z);
}
```

El valor que se imprime sería indefinido, ya que el parámetro c es local y no es "asignado" a z al concluir la invocación a "suma". Para ello se realizarían los siguientes cambios

```
void suma (int a, int b, int *c)  //c guarda una dirección, la de z
{
  *c = a+b;   //pone en la posición de memoria indicada por c el valor de a + b
}             //la posición indicada por c corresponde a la variable z,
              //por lo tanto, la actualiza
main ()
{
  int x,y,z;
  x = 4;
  y= 2;
  suma (x,y,&z);      // se pasa la dirección de z a c
  printf ("z = ", z);
}
```

En C++, si existen parámetros "explícitos" por referencia, por lo que bastaría con modificar la declaración de "suma":

```
void suma (int a, int., int &c)
```

b) Acceso a memoria física

Útil para manejo de drivers, programación de sistemas operativos, etc.

c) Implementación de estructuras dinámicas de información

Una de las aplicaciones más importantes de la memoria dinámica y los apuntadores son las estructuras dinámicas de datos. Los arreglos están compuestos por un determinado

número de elementos, número que se decide en la fase de diseño y que no puede variar durante la ejecución del programa.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales de información. Adicionalmente, que sean flexibles en cuanto al orden de los elementos que contienen y sus relaciones con los demás elementos. Para ello, podemos definir, con el uso de apuntadores, "listas enlazadas"

LISTAS ENLAZADAS

Una lista es una estructura de datos secuencial, donde se almacenan cero o más elementos de un tipo determinado. Representa una estructura dinámica de información, ya que nos permite "crecer" durante la ejecución del programa, creando a petición, nuevas estructuras de datos o elementos a los que llamaremos nodos los cuales se agrupan o se "encadenan" unos con otros mediante el uso de apuntadores del mismo tipo nodo.

Una estructura básica de un nodo para crear listas de datos en C sería:

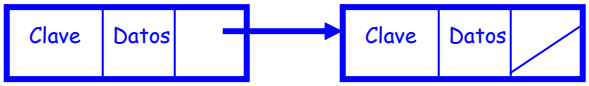
```
struct nodo
{
  int clave;
  struct nodo *proximo;
};
```

El campo "próximo" puede apuntar a un objeto del tipo nodo. De este modo, cada nodo puede usarse como un elemento para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos.

Para acceder a un nodo de la estructura sólo necesitaremos un apuntador a un nodo.

De ahora en adelante, representaremos "NULL" con una raya diagonal. Así:

Gráficamente se suele representar así:



Características:

- El largo de la lista es por lo general, variable.
- Cada elemento de la lista es un "nodo" y puede estar compuesto de varios campos (registro)
- Cada nodo tiene la misma estructura, es decir, es del mismo tipo.
- Es muy importante siempre guardar la posición de memoria donde se encuentra el "primero" de la lista para poder hacer el recorrido.

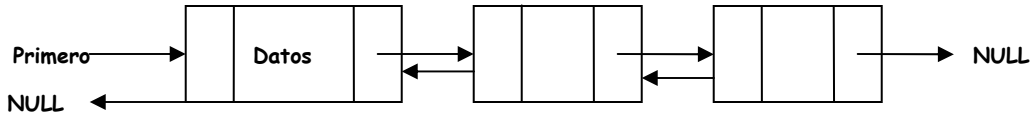
Clasificación

- Estructuras secuenciales
  - Listas
  - Pilas
  - Colas
- Estructuras jerárquicas
  - Árboles
  - Grafos

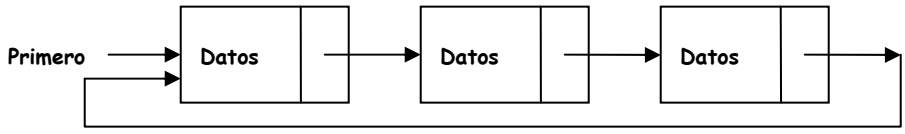
Tipos de Listas:

1) Listas simplemente enlazadas

2) Listas doblemente enlazadas



3) Listas circulares



Operaciones básicas con listas:

Se pueden realizar diversas operaciones con listas, las más básicas son:

- ✓ Añadir o insertar elementos.
- ✓ Buscar elementos.
- ✓ Borrar elementos.
- ✓ Moverse a través de una lista, anterior, siguiente, primero.

Cada una de estas operaciones tendrá varios casos especiales, por ejemplo, no será lo mismo insertar un nodo en una lista vacía, o al principio de una lista no vacía, o al final, o en una posición intermedia.

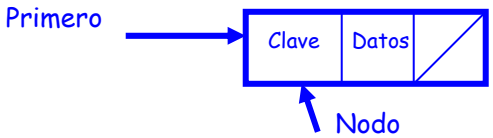
A continuación, veremos algunos casos con listas simplemente enlazadas.

INSERCIÓN

Insertar un elemento en una lista vacía:

El proceso es muy simple, bastará con que:

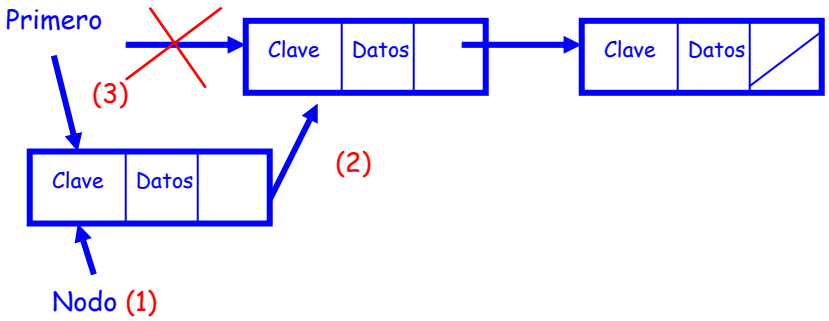
1. Crear el nodo
2. nodo->siguiente apunte a NULL.
3. Primero apunte a nodo.



Insertar un elemento en la primera posición de una lista:

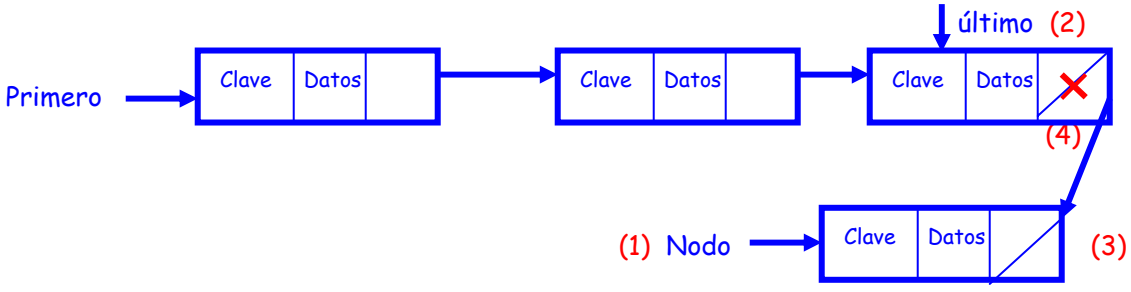
El proceso es:

1. Crear el nodo
2. nodo->siguiente lo apuntamos a Primero
3. Primero apunte a nodo.



Insertar un elemento en la última posición de una lista:

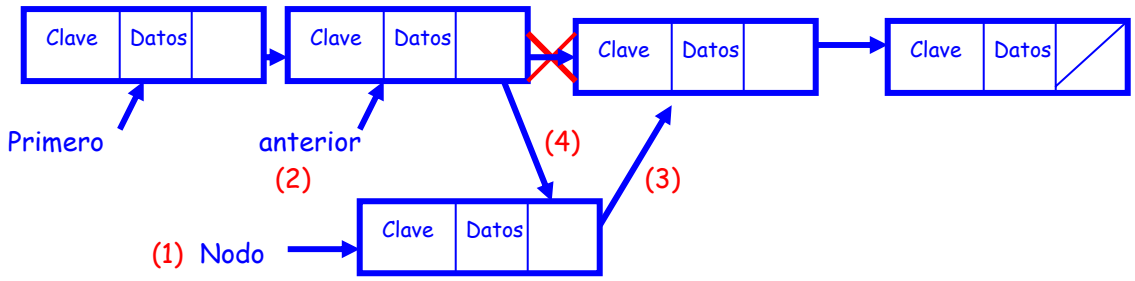
1. Crear el nodo
2. Necesitamos un apuntador que señale al último elemento de la lista. La manera de conseguirlo es empezar por el primero y avanzar hasta que el nodo que tenga como siguiente el valor NULL. (ultimo)
3. Hacer que nodo->siguiente sea NULL.
4. Hacer que ultimo->siguiente sea nodo.



Insertar un elemento a continuación de un nodo cualquiera de una lista:

El proceso a seguir será:

1. Crear el nodo
2. Recorremos la lista desde Primero hasta que estemos en el nodo anterior al nodo donde queremos insertar (apuntado por anterior)
3. nodo->siguiente debe apuntar a anterior->siguiente.
4. anterior->siguiente debe apuntar a nodo.



ELIMINACIÓN

Podemos encontrarnos con varios casos, según la posición del nodo a eliminar.

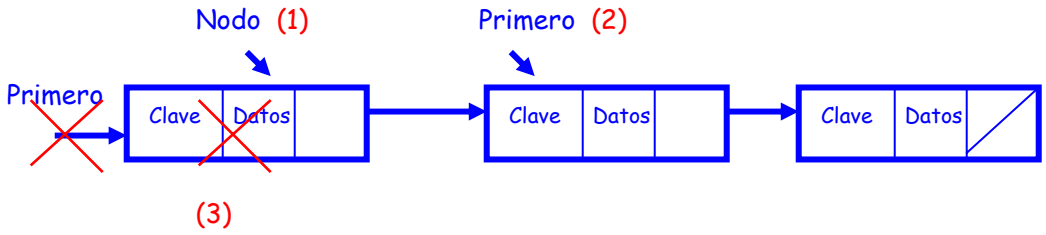
Eliminar el primer nodo:

1. Apuntamos "nodo" al primer elemento de la lista, es decir a Primero.
2. Asignamos a Primero la dirección del segundo nodo de la lista: Primero->siguiente.
3. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

Debemos guardar el apuntador al primer nodo antes de actualizar "Primero" porque, si no no podríamos liberar la memoria que ocupa. Si liberamos la memoria antes de actualizar Primero, perderemos el apuntador al segundo nodo.

Si la lista sólo tiene un nodo, el proceso es también válido, ya que el valor de Primero->siguiente es NULL, y después de eliminar el primer nodo la lista quedará vacía, y el valor de Priemro será NULL.

De hecho, el proceso que se suele usar para borrar listas completas es eliminar el primer nodo hasta que la lista esté vacía.

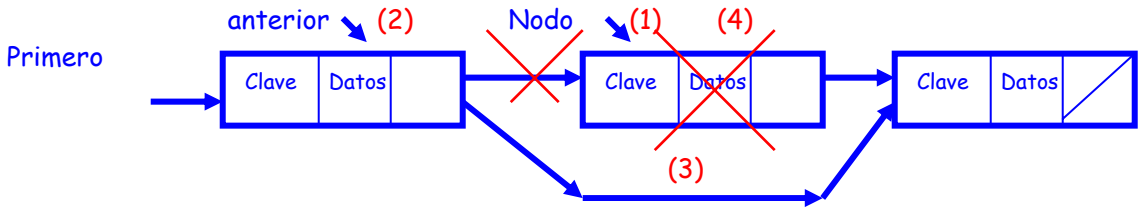


Eliminar un nodo cualquiera:

El proceso es:

1. Buscamos a el nodo que queremos eliminar.
2. Asignamos "anterior" al nodo anterior a "nodo"
3. anterior->siguiente = nodo->siguiente.
4. Eliminamos la memoria asociada al nodo que queremos eliminar.

Si el nodo a eliminar es el último, es procedimiento es igualmente válido, ya que anterior pasará a ser el último, y anterior->siguiente valdrá NULL.



Ejemplo de Codificación de listas en C

```
// Lista en C
// (C) Abril 2001, Salvador Pozo
// Este ejemplo fue bajado de Internet


#include <stdlib.h>
#include <stdio.h>

typedef struct _nodo {
  int valor;
  struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;

/* Funciones con listas: */
void Insertar(Lista *l, int v);
void Borrar(Lista *l, int v);

int ListaVacia(Lista l);

void BorrarLista(Lista *);
void MostrarLista(Lista l);

int main()
{
  Lista lista = NULL;
  pNodo p;

  Insertar(&lista, 20);
  Insertar(&lista, 10);
  Insertar(&lista, 40);
  Insertar(&lista, 30);

  MostrarLista(lista);

  Borrar(&lista, 10);
  Borrar(&lista, 15);
  Borrar(&lista, 45);
  Borrar(&lista, 30);
  Borrar(&lista, 40);

  MostrarLista(lista);
```

```
  BorrarLista(&lista);

  system("PAUSE");
  return 0;
}

void Insertar(Lista *lista, int v)
{
  pNodo nuevo, anterior;

  /* Crear un nodo nuevo */
  nuevo = (pNodo)malloc(sizeof(tipoNodo));
  nuevo->valor = v;

  /* Si la lista está vacía */
  if(ListaVacia(*lista) || (*lista)->valor > v) {
    /* Añadimos la lista a continuación del nuevo nodo */
    nuevo->siguiente = *lista;
    /* Ahora, el comienzo de nuestra lista es en nuevo nodo */
    *lista = nuevo;
  }
  else {
    /* Buscar el nodo de valor menor a v */
    anterior = *lista;
    /* Avanzamos hasta el último elemento o hasta que el siguiente tenga
      un valor mayor que v */
    while(anterior->siguiente && anterior->siguiente->valor <= v)
      anterior = anterior->siguiente;
    /* Insertamos el nuevo nodo después del nodo anterior */
    nuevo->siguiente = anterior->siguiente;
    anterior->siguiente = nuevo;
  }
}

void Borrar(Lista *lista, int v)
{
  pNodo anterior, nodo;

  nodo = *lista;
  anterior = NULL;
  while(nodo && nodo->valor < v) {
    anterior = nodo;
    nodo = nodo->siguiente;
  }
  if(!nodo || nodo->valor != v) return;
  else { /* Borrar el nodo */
```

```
    if(!anterior) /* Primer elemento */
      *lista = nodo->siguiente;
    else  /* un elemento cualquiera */
      anterior->siguiente = nodo->siguiente;
    free(nodo);
  }
}

int ListaVacia(Lista lista)
{
  return (lista == NULL);
}

void BorrarLista(Lista *lista)
{
  pNodo nodo;

  while(*lista) {
    nodo = *lista;
    *lista = nodo->siguiente;
    free(nodo);
  }
}

void MostrarLista(Lista lista)
{
  pNodo nodo = lista;

  if(ListaVacia(lista)) printf("Lista vacía\n");
  else {
    while(nodo) {
      printf("%d -> ", nodo->valor);
      nodo = nodo->siguiente;
   }
   printf("\n");
  }
}
```

Ejemplo de Codificación en listas en C++

```
// Lista en C++
// (C) Abril 2001, Salvador Pozo
// Este ejemplo fue bajado de Internet

#include <iostream>
using namespace std;

class nodo {
  public:
   nodo(int v, nodo *sig = NULL)
   {
     valor = v;
     siguiente = sig;
   }

  private:
   int valor;
   nodo *siguiente;

  friend class lista;
};

typedef nodo *pnodo;

class lista {
  public:
   lista() { primero = actual = NULL; }
   ~lista();

   void Insertar(int v);
   void Borrar(int v);
   bool ListaVacia() { return primero == NULL; }
   void Mostrar();
   void Siguiente();
   void Primero();
   void Ultimo();
   bool Actual() { return actual != NULL; }
   int ValorActual() { return actual->valor; }

  private:
   pnodo primero;
   pnodo actual;
};

lista::~lista()
{
```

```
  pnodo aux;

  while(primero) {
    aux = primero;
    primero = primero->siguiente;
    delete aux;
  }
  actual = NULL;
}

void lista::Insertar(int v)
{
  pnodo anterior;

  // Si la lista está vacía
  if(ListaVacia() || primero->valor > v) {
    // Asignamos a lista un nievo nodo de valor v y
    // cuyo siguiente elemento es la lista actual
    primero = new nodo(v, primero);
  }
  else {
    // Buscar el nodo de valor menor a v
    anterior = primero;
    // Avanzamos hasta el último elemento o hasta que el siguiente tenga
    // un valor mayor que v
    while(anterior->siguiente && anterior->siguiente->valor <= v)
      anterior = anterior->siguiente;
    // Creamos un nuevo nodo después del nodo anterior, y cuyo siguiente
    // es el siguiente del anterior
    anterior->siguiente = new nodo(v, anterior->siguiente);
  }
}

void lista::Borrar(int v)
{
  pnodo anterior, nodo;

  nodo = primero;
  anterior = NULL;
  while(nodo && nodo->valor < v) {
    anterior = nodo;
    nodo = nodo->siguiente;
  }
  if(!nodo || nodo->valor != v) return;
  else { // Borrar el nodo
    if(!anterior) // Primer elemento
```

```
      primero = nodo->siguiente;
    else  // un elemento cualquiera
      anterior->siguiente = nodo->siguiente;
    delete nodo;
  }
}

void lista::Mostrar()
{
  nodo *aux;

  aux = primero;
  while(aux) {
    cout << aux->valor << "-> ";
    aux = aux->siguiente;
  }
  cout << endl;
}

void lista::Siguiente()
{
  if(actual) actual = actual->siguiente;
}

void lista::Primero()
{
  actual = primero;
}

void lista::Ultimo()
{
  actual = primero;
  if(!ListaVacia())
    while(actual->siguiente) Siguiente();
}

int main()
{
  lista Lista;

  Lista.Insertar(20);
  Lista.Insertar(10);
  Lista.Insertar(40);
  Lista.Insertar(30);

  Lista.Mostrar();
```

```
  cout << "Lista de elementos:" << endl;
  Lista.Primero();
  while(Lista.Actual()) {
    cout << Lista.ValorActual() << endl;
    Lista.Siguiente();
  }
  Lista.Primero();
  cout << "Primero: " << Lista.ValorActual() << endl;

  Lista.Ultimo();
  cout << "Ultimo: " << Lista.ValorActual() << endl;

  Lista.Borrar(10);
  Lista.Borrar(15);
  Lista.Borrar(45);
  Lista.Borrar(30);
  Lista.Borrar(40);

  Lista.Mostrar();

  cin.get();
  return 0;
}
```