# Estructuras y Tipos de Datos

Tipo Abstracto de Datos Listas

## Tipo de Dato Abstracto Lista

Las listas son una estructura de datos fundamental, empleadas para almacenar una colección de elementos. Su importancia radical reside en que puede utilizarse para implementar una amplia variedad de otros tipos de datos. Es decir TAD Lista sirve como bases para implementar otros TAD más complicados.

Las listas constituyen una estructura flexible, ya que puede crecer y acortarse según se requiera, los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. También se puede operar sobre listas, por ejemplo concatenarlas, dividirlas en sub-listas o mezclarlas.

## Tipo de Dato Abstracto Lista

Una lista es una secuencia de ceros o más elementos de un determinado tipo y se definir como una n-tupla dinámica (donde dinámica significa que los elementos pueden cambiar con el tiempo) de la siguiente forma:

$$L=(a_1, a_2, \ldots , a_n)$$

Donde $n \geq 0$, n es la longitud de la lista o $|L|$. Si n=0 se dice que la lista es vacía y para $n \geq 1$, $a_1$ es el primer elemento o cabeza, $a_n$ es el último elemento o cola y en general $a_i$ está en la posición i.

## Tipo de Dato Abstracto Lista

No se impone ninguna restricción en los elementos de una lista, si todos los elementos son de un mismo tipo (p.e. enteros) se dice que es homogénea. Sin embargo diferentes tipos (p.e. enteros, flotantes, complejos) pueden ser almacenados en la lista entonces se dice que la lista es heterogénea. En algunas aplicaciones se trabaja con lista de listas:

L=((3),(4,5,6),(12,8,(10,11),2,()))

Una propiedad importante de una lista es que los elementos pueden estar ordenados de forma lineal de acuerdo a sus posiciones en la misma. Se dice que $a_i$ precede a $a_{i+1}$ para i=1, 2, …, n-1 y $a_i$ sucede a $a_{i-1}$ para i=2, 3, …, n.

## Tipo de Dato Abstracto Lista

### Ejemplo

Se pueden utilizar lista para representar polinomios

Dado $p(x) = a_n X^n + a_{n-1} X^{n-1} + \ldots + a_1 x + a_0$

Se puede representar como una lista donde cada elemento de la lista almacenen el coeficiente y el exponente de un término en particular, mientras $a_i \neq 0$

Para:

$p(x) = 5X^3 + 10X - 7$

se almacena como

L=((5,3),(10,1),(-7,0))

## Tipo de Dato Abstracto Lista

Sobre una lista se puede definir un conjunto de operaciones. Aquí L es una lista, x es un elemento y p es una posición, y se supone que la lista mantiene su posición actual, que se refiere a cierto elemento, y sobre la que se puede actual.

**Insertar (L,x,p).** Inserta x en la posición p de la Lista L, si tiene éxito retorna ok sino falla. Hace que los elementos $a_p$, $a_{p+1}$, …, $a_n$ pasen a ser $a_{p+1}$, $a_{p+2}$, …, $a_{n+1}$.

**Añadir (L,x).** Añade el elemento x a la Lista L, si tiene éxito retorna ok sino falla.

**Obtener (L,p).** Retorna el elemento que está en la posición p de la lista L. o el valor nulo si la posición no existe.

## Tipo de Dato Abstracto Lista

### Operaciones con Listas

**Eliminar (L,p).** Eliminar el elemento en la posición p de la lista L, si tiene éxito retorna ok sino falla. Hace que los elementos $a_{p+1}$, …, $a_n$ pasen a ser $a_p$, $a_{p+1}$, …, $a_{n-1}$.

**Longitud (L).** Retorna el número de elementos en L es decir |L|.

**Localiza (x,L).** Retorna la posición de x en la lista L, si estas más de una vez retorna la aparición de la primera ocurrencia y si x no se encuentra en la lista retorna NULL.

**Inicio (L).** Sitúa la posición actual en el primer elemento o en la cabeza y retorna el primer elemento, si la lista es vacía retorna NULL.

## Tipo de Dato Abstracto Lista

### Operaciones con Listas

**Siguiente (L).** Incrementa la posición actual y retorna el valor de la posición.

**Anterior (L).** Decrementa la posición actual y retorna el valor de la posición.

**Actual (L).** Retorna el valor de la posición actual.

**Anular (L).** Convierte a L en una lista vacía.

**Listar (L).** Imprime los elementos de L en orden.

Se debe tener en cuenta que esta son un pequeño grupo de operaciones de las que se pueden definir.

## Tipo de Dato Abstracto Lista

### Implementación

Ahora cuando los elementos que forman una estructura de datos están almacenados unos detrás del otros en posiciones consecutiva de la memoria se dice que estructura de datos tiene una la **disposición secuencial**. Una disposición secuencial hace que los elementos de la estructura de datos se accedan en tiempo constante, básicamente calculando el desplazamiento desde el inicio de la estructura, siendo independiente del tamaño de la estructura. Un arreglo es un ejemplo de una estructura de datos con disposición secuencial y debido a que se tarda la misma cantidad de tiempo en acceder a cualquier elemento se le denomina **estructura de datos de acceso directo**.

## Tipo de Dato Abstracto Lista

### Implementación

El TAD Lista se puede implementar utilizando arreglos lo suficientemente grandes como para almacenar todos los elementos de la lista. Donde cada elemento $a_i$ de la lista se almacena en la posición i-1 del arreglo, y los elementos sucesivos de la lista se almacenan en posiciones consecutivas del arreglo.

Las ventajas es su simplicidad, así como la eficiencia con que las operaciones del TAD pueden ser implementadas. La principal desventaja resulta que al definir un tamaño máximo al arreglo se establecer un límite a priori sobre el número de elementos que pueden ser almacenados en la lista.

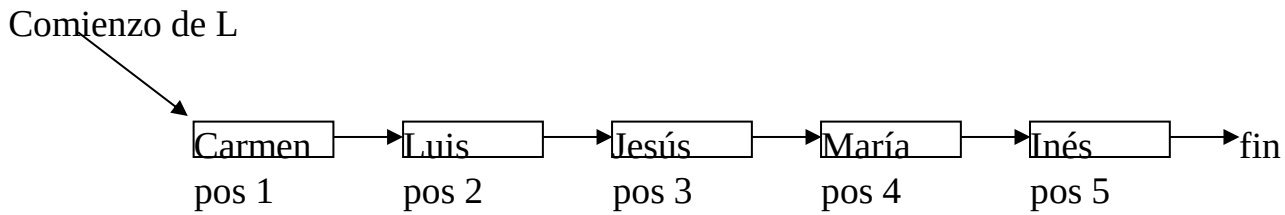
## Tipo de Dato Abstracto Lista

### Implementación

Escoger un tamaño muy grande del arreglo, nos puede conducir a ineficiencias, en términos de memoria desperdiciada, si el tamaño promedio de las lista resultar ser en tiempo de ejecución bastante pequeñas. Una solución puede ser asignando el tamaño de forma dinámica y/o realizando redimensionamientos, el cual consiste en utilizar arreglos obtenidos de forma dinámica y dependiendo de la cantidad de elementos (relación de carga), se obtiene un nuevo arreglo se copia en el los elementos del viejo, que luego se libera. De esta forma la estructura puede crecer y decrecer.

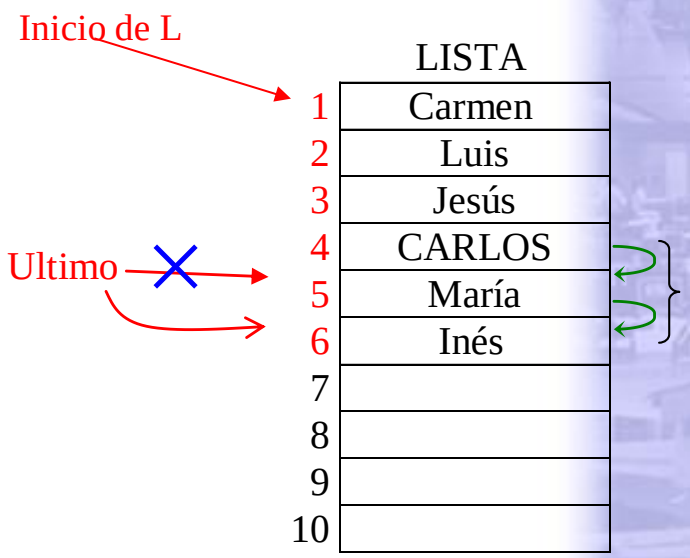
# Ejemplo

*Lista de estudiantes*




*Representación en Memoria*


*Insertar un elemento en la posición p de la lista*
*(Insertar CARLOS en la posición 4)*



# Tipo de Dato Abstracto Lista

## Implementación con arreglos

```
#include <stdio.h>
#define MAX_ELEMENTOS 15

class lista {
private:
    int longitud;
    int actual;
    int elementos[MAX_ELEMENTOS];
public:
    lista();
    int insertarElemento(int element, int pos);
    int obtenerElemento(int pos);
    int eliminarElemento(int pos);
    int longitudLista();
    int primerElemento();
    int siguienteElemento();
    int actualElemento();
    void anularLista();
    void listarElementos();
};
```

# Tipo de Dato Abstracto Lista

## Implementación con arreglos

```
lista::lista(){
   anularLista();
}

int lista::insertarElemento(int element, int pos=-1){
     register int i;

     if (pos == -1)
          pos = (longitud==0?1:longitud+1);

     if(longitud==MAX_ELEMENTOS || pos<1 || pos>longitud+1)
          return 0;
     for(i=longitud;i>=pos;i--)
          elementos[i]=elementos[i-1];
     elementos[pos-1]=element;
     longitud++;
     return 1;
};
```

# Tipo de Dato Abstracto Lista

## Implementación con arreglos

```
int lista::obtenerElemento(int pos){
    if(pos<1 || pos>longitud)
        return -1;
    return elementos[pos-1];
};

int lista::eliminarElemento(int pos){
    register int i;

    if (pos < 1 || pos> longitud)
        return -1;
    longitud--;
    for(i=pos-1;i<longitud;i++)
        elementos[i]=elementos[i+1];
    return 1;
};

int lista::longitudLista(){
    return longitud;
};
```

# Tipo de Dato Abstracto Lista

## Implementación con arreglos

```
int lista::primerElemento(){
    if(longitud == 0)
        return -1;
    actual=0;
    return elementos[0];
};

int lista::siguienteElemento(){
    if (actual+1 >= longitud)
        return -1;
    actual++;
    return elementos[actual];
};

int lista::actualElemento(){
    if(longitud==0)
        return -1;
    return actual+1;
}
```

## Tipo de Dato Abstracto Lista

### Implementación con arreglos

```
void lista::anularLista(){
    longitud = 0;
    actual   = -1;
}

void lista::listarElementos(){
    register int i;

    for(i=0;i<longitud;i++)
        printf("%d ",elementos[i]);
}
```

# Tipo de Dato Abstracto Lista

## Implementación con arreglos

Ejemplo de uso del TAD

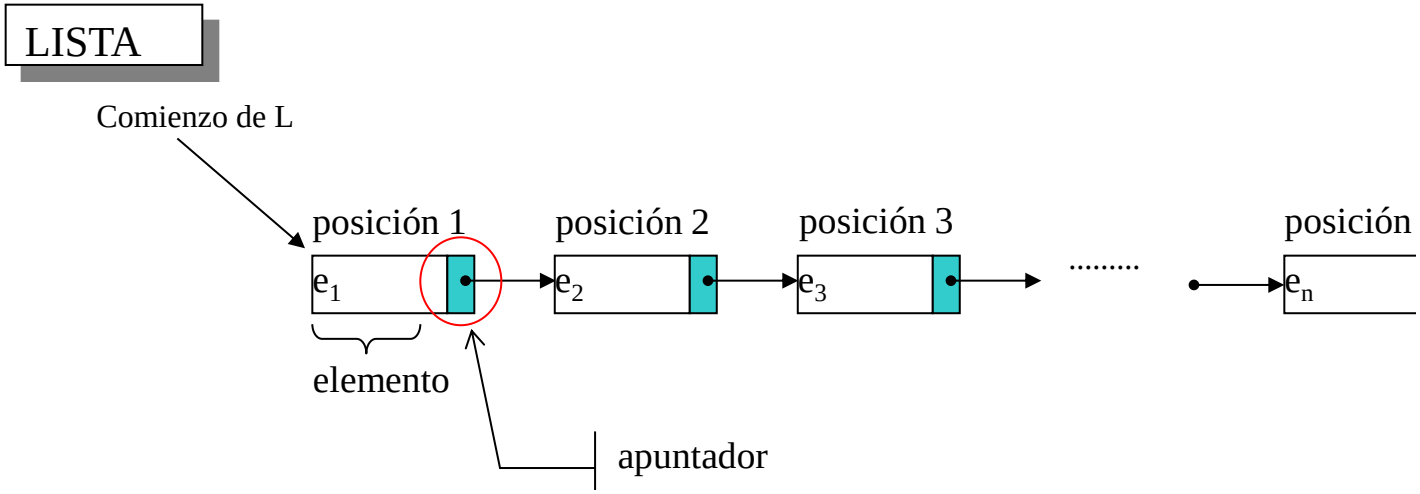
```
Implementacion de Lista Ordenada
0 1 2 3 3 4 5
0 1 2 3 3 4 5 Actual:7
```

```
main() {

    lista L; int e;

    printf("\nImplementacion de Lista Ordenada\n");
    L.insertarElemento(2);
    L.insertarElemento(4);
    L.insertarElemento(3);
    L.insertarElemento(1);
    L.insertarElemento(5);
    L.insertarElemento(3);
    L.insertarElemento(0);
    L.listarElementos();
    printf("\n");
    for(e=L.primerElemento();e!=-1;e=L.siguienteElemento())
        printf("%d ",e);
    printf("\nActual:%d\n",L.actualElemento());

}
```

# Representación con Apuntadores

## Estrategia

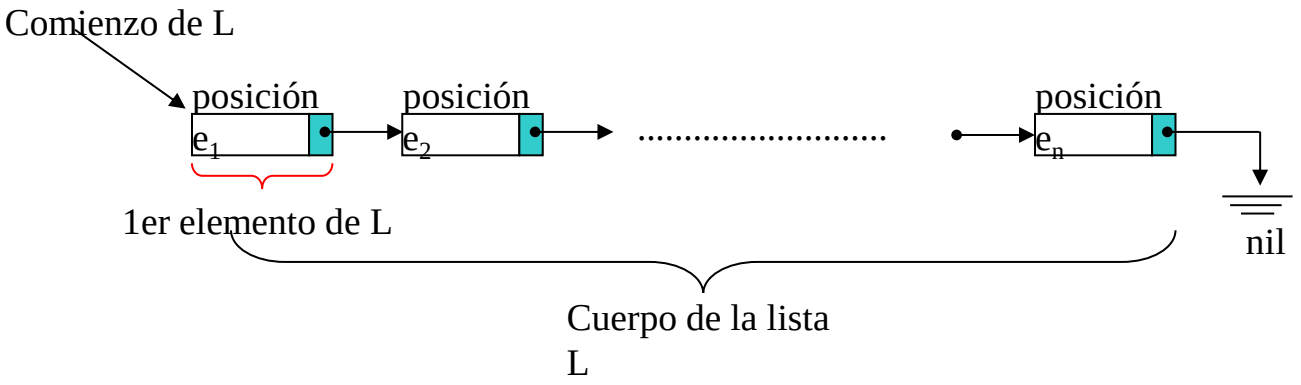
- *Los elementos están enlazados a través de un campo llamado apuntador*
- *Un apuntador es una celda que contiene la dirección en memoria en donde se próximo elemento de la lista*
- *Un elemento de lista consiste en uno o más campos asociados a ese elem apuntador al próximo elemento de la lista*
- *El uso de apuntadores define una estructura dinámica*



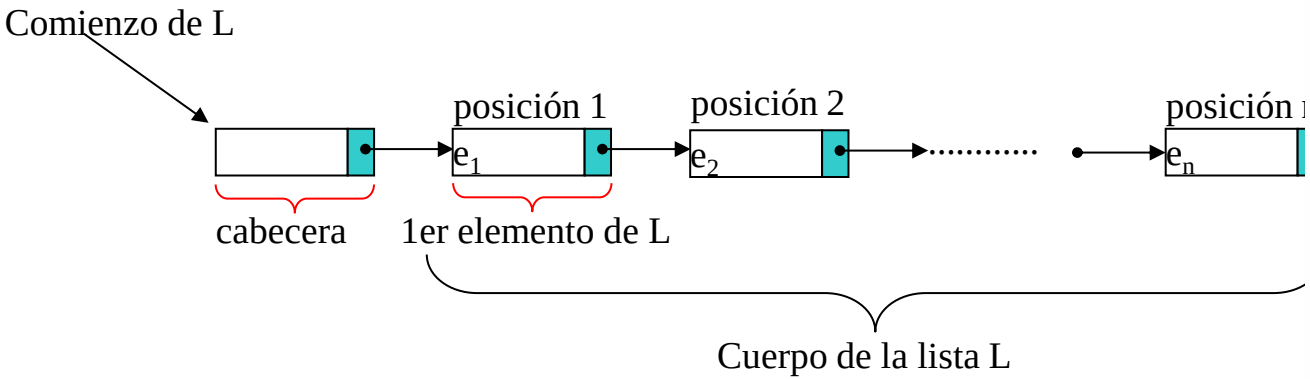
# Representación con Apuntadores

## Formas de Manipular la Lista

*Listas sin cabecera*




*Listas con cabecera*

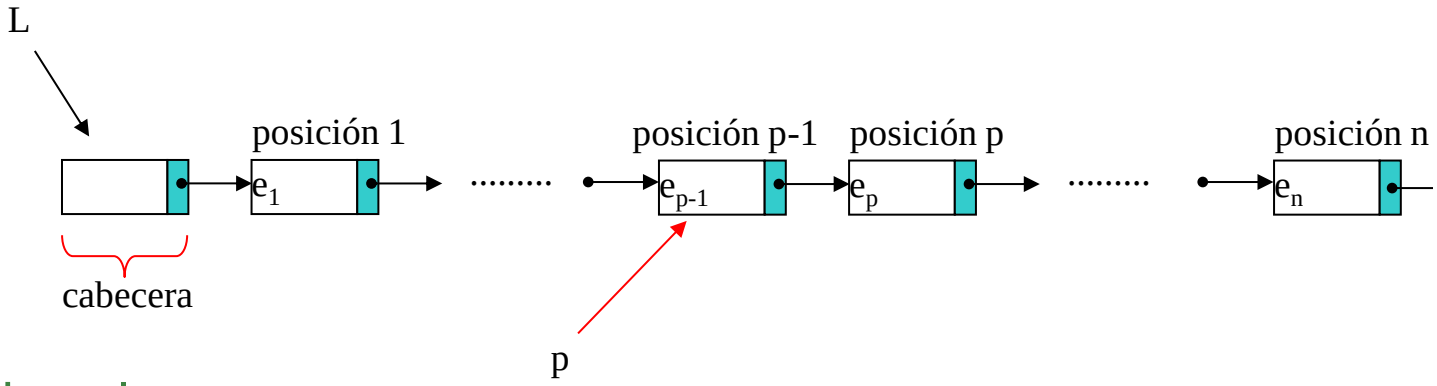


# Representación con Apuntadores

## Formas de Manipular la Lista

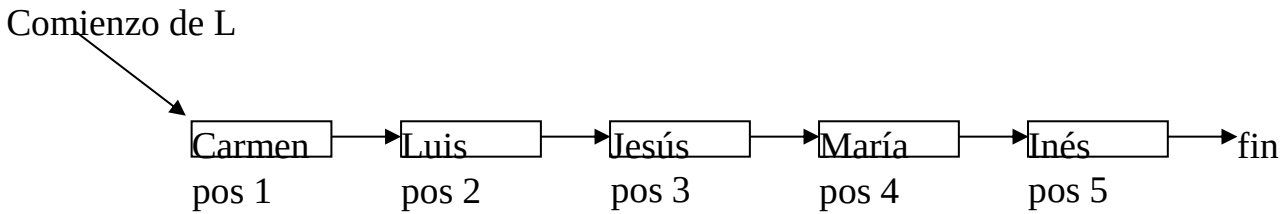
*Listas con cabecera, Cont…*

- *La posición p es un apuntador a la casilla que contiene el elemento* $e_{p-1}$
- *La posición 1 es un apuntador a la cabecera de la lista*
- *La posición FIN(L) es un apuntador al último elemento de la lista*




## Ejemplo

*Lista de estudiantes*

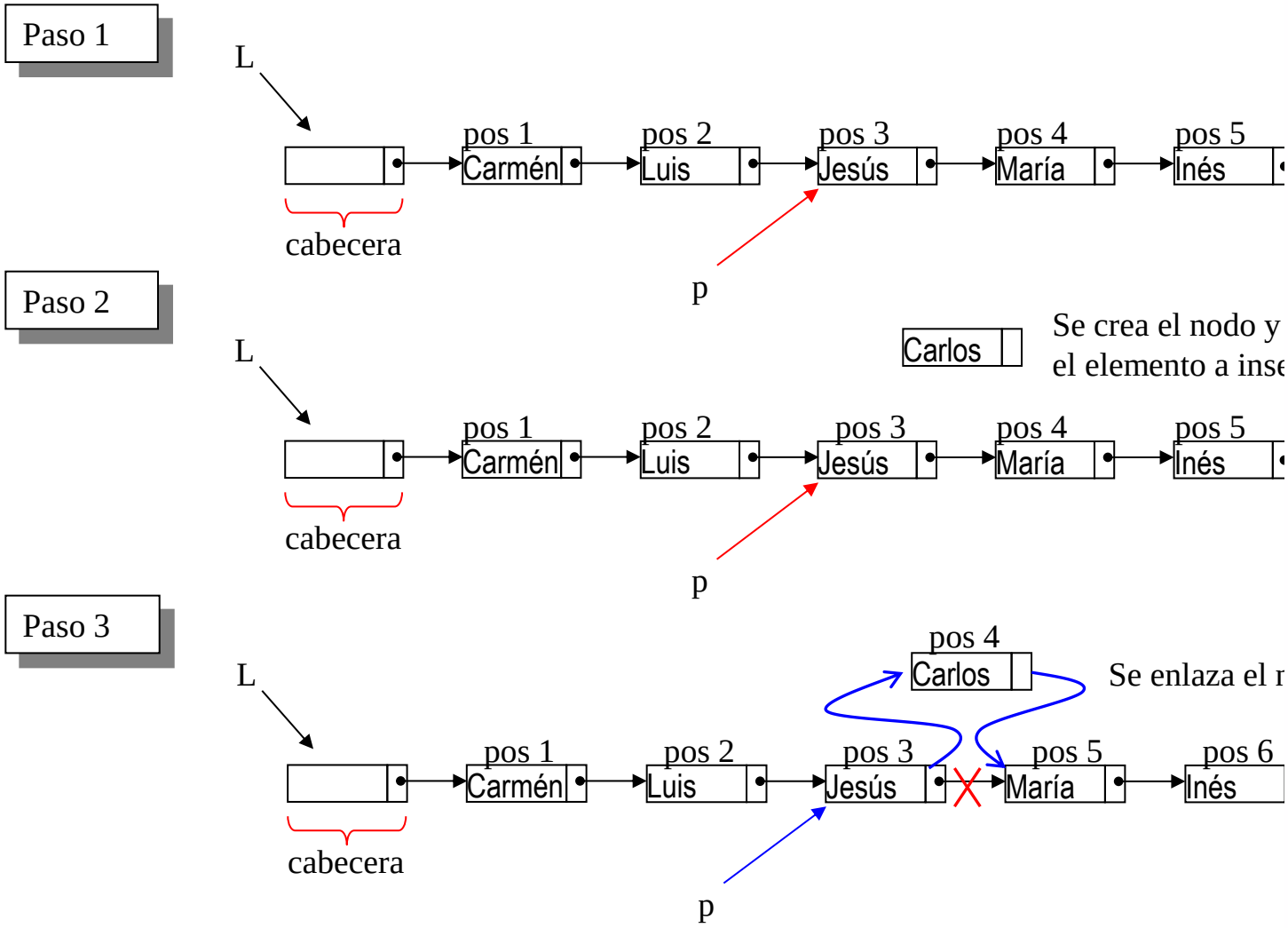



*Queremos agregar Carlos en la posición 4 de la lista*

# Representación con Apuntadores

## Ejemplo

*Lista de estudiantes, Cont…*



# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
struct Nodo {
    int elemento;
    Nodo* sig;

    Nodo(int data, Nodo* nextPtr = NULL): elemento(data),sig(nextPtr){}
};

class lista {
private:
    Nodo *cab, *cola, *actual;
    int longitud;
public:
    lista();
    lista(lista& otraLista);
    ~lista();

    bool insertarCabeza(int element);
    bool insertarCola(int element);
    bool sacarInicio();
    bool sacarFinal();
    bool eliminar(int pos);
    bool eliminarLista();
    Nodo* inicio();
    Nodo* fin();
    Nodo* siguienteNodo();
    Nodo* actualNodo() const;

    int longitudLista();
    void listarElementos();
};
```

...

# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
lista::lista(){
  cab = cola = actual = NULL;

  longitud = 0;
};

lista::lista(lista& otraLista){
    cab = cola = actual = NULL;
    longitud = 0;

    for (Nodo* iptr=otraLista.inicio(); iptr != NULL; iptr = otraLista.siguienteNodo()) {
        insertarCola(iptr->elemento);
    }
}

lista::~lista(){

    while (cab != NULL) {
      actual = cab;
      cab = cab->sig;
      delete actual;
    }
    cola = NULL;
    longitud = 0;
}
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
bool lista::insertarCabeza(int element){
    Nodo *np;

    np=new Nodo(element,NULL);
    if(cab==NULL)
        cab= cola = np;
    else{
        np->sig=cab;
        cab=np;
    }

    longitud++;
    return true;
};

bool lista::insertarCola(int element){
    Nodo *np;

    np=new Nodo(element,NULL);
    if(cab==NULL)
        cab= cola = np;
    else {
        cola->sig = np;
        cola=np;
    }

    longitud++;
    return true;
};
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
bool lista::sacarInicio(){
    Nodo* ptr;

    if (cab != NULL) {
       ptr = cab;

        if (cab == cola) // Un solo elemento en la cola
            cab = cola = actual = NULL;
        else {
            cab = cab->sig;
            actual = cab;
        }
        delete ptr;
        longitud--;
        return true;
    }
    return false;
}
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
bool lista::eliminar(int pos){
    Nodo *ptr = cab, *ptrPrevio = cab;
    int i= 1;

    if ( pos < 1 || pos > longitud)
        return false;

    while (i < pos && ptr != NULL ) {
        ptr = ptr->sig;
        ptrPrevio = (i==1?ptrPrevio:ptrPrevio->sig);
        i++;
    }
    if (ptr != NULL){
        if (ptr == cab && ptr == cola) { // Un solo elemento en la cola
           cab = cola = actual = NULL;
        }
        else if (ptr == cab) {  // Se elimina el primero
            cab = cab->sig;
        }
        else if (ptr == cola){ // Se eleimina el ultimo
            cola = ptrPrevio;
            ptrPrevio->sig = NULL;
        } else // Se elimina un elemento intermedio
            ptrPrevio->sig = ptr->sig;
        delete ptr;
        longitud--;
        return true;
    }
    return false;
}
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Simplemente Enlazadas

```
main() {

    lista L; int e;

    cout <<("\nImplementacion de Listas Enlazadas\n");
    L.insertarCabeza(2);
    L.insertarCabeza(4);
    L.insertarCabeza(3);
    L.insertarCabeza(1);
    L.insertarCabeza(5);
    L.insertarCabeza(3);
    L.insertarCabeza(0);
    L.listarElementos();

    lista L2(L);

    cout << endl;
    cout << "Eliminar posicion 3" << endl;
    L.eliminar(3);
    L.listarElementos();

    cout << endl;
    cout << "Eliminar posicion 10" << endl;
    L.eliminar(10);
    L.listarElementos();

    cout << endl;
    cout << "Eliminar posicion 1" << endl;
    L.eliminar(1);
    L.listarElementos();
```

# Representación Listas Doblemente Enlazadas

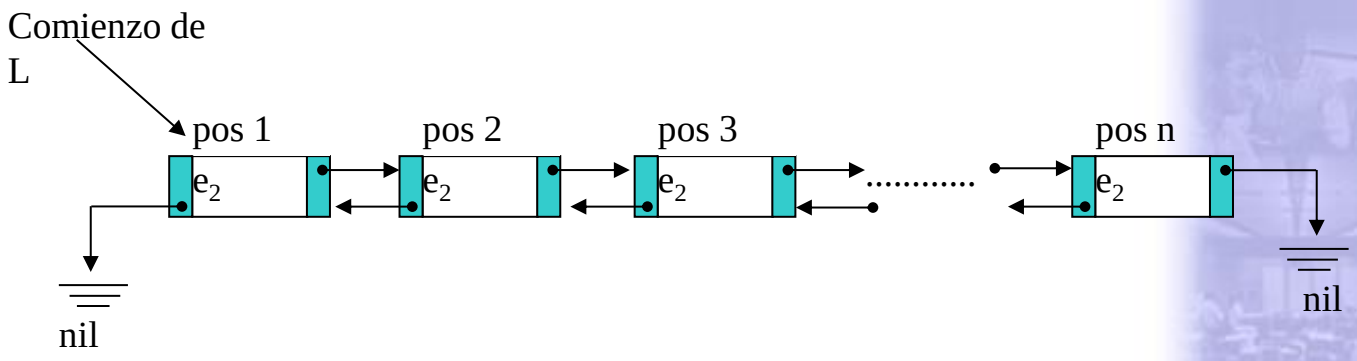
## Estrategia

*Desplazarnos hacia delante y hacia atrás* → *Ir desde la pos p → hacia la p+1 y desde la pos p → hacia la p-1.*

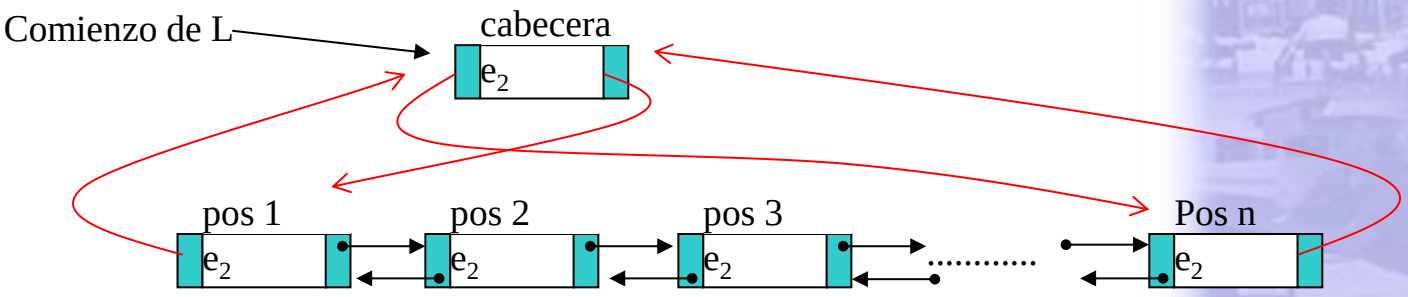
*¿Como?* *Cada nodo de la lista tiene un apuntador al siguiente elemento de la lista y un apuntador al anterior.*

*Dos maneras de formarlas:* →
- *Usando la lista sin cabecera*
- *Usando la lista con cabecera:*

*Sin Cabecera*




*Con Cabecera*



# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

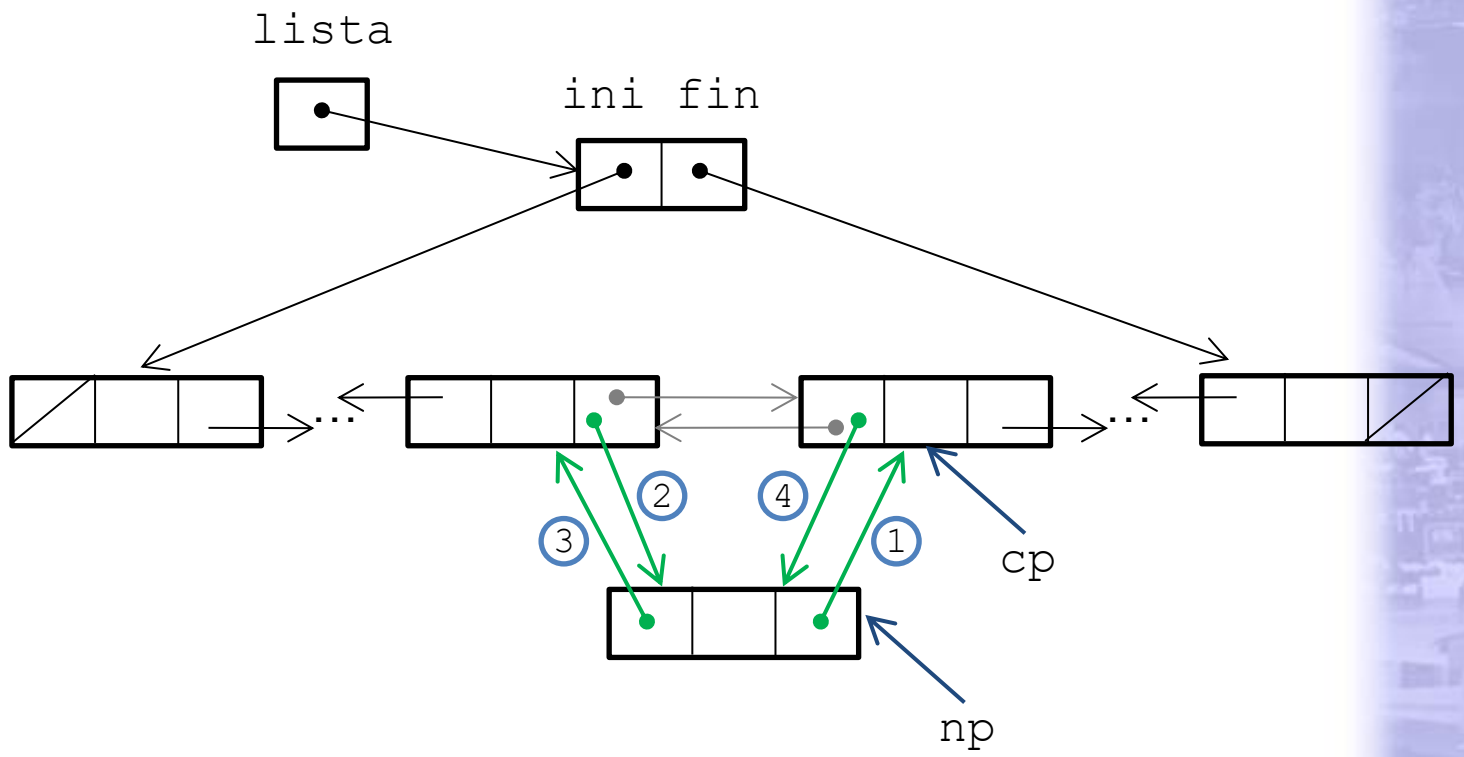
```
struct NodoD {
    int elemento;
    NodoD *sig, *ant;

    NodoD(int data, NodoD* nextPtr = NULL, NodoD* antPtr = NULL): elemento(data),sig(nextPtr), ant(antPtr){}
};
class listaD {
private:
    NodoD *cab, *cola, *actual;
    int longitud;
public:
    listaD();
    listaD(listaD& otralistaD);
    ~listaD();
    bool insertarCabeza(int element);
    bool insertarCola(int element);
    bool sacarInicio();
    bool sacarFinal();
    bool eliminar(int pos);
    bool eliminarlista();
    NodoD* inicio();
    NodoD* fin();
    NodoD* siguienteNodo();
    NodoD* anteriorNodo();
    NodoD* actualNodo() const;
    int longitudlista();
    void listarElementos();
    void listarElementosI();
};
```

NodoD* actual

## Tipo de Dato Abstracto Lista

### Implementación con Lista Doblemente Enlazadas

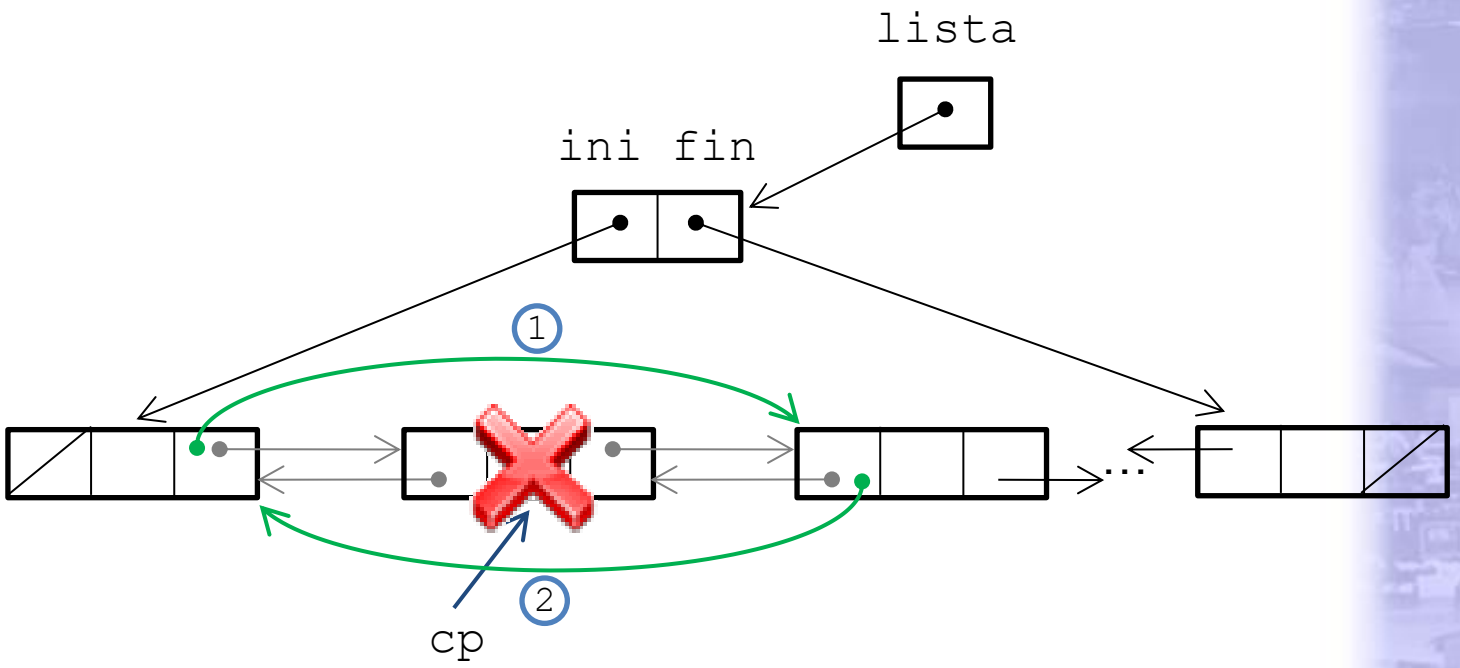
**insertar**



# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

### `eliminar`



# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

```
bool listaD::insertarCabeza(int element){
    NodoD *np;

    np=new NodoD(element,NULL,NULL);
    if(cab == NULL)
        cab = cola = np;
    else{
        np->sig=cab;
        cab->ant = np;
        cab=np;
    }
    longitud++;
    return true;
};

bool listaD::insertarCola(int element){
    NodoD *np;

    np=new NodoD(element,NULL);
    if(cab==NULL)
        cab= cola = np;
    else {
        cola->sig = np;
        np->ant = cola;
        cola=np;
    }
    longitud++;
    return true;
};
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

```
bool listaD::sacarInicio(){
    NodoD* ptr;

    if (cab != NULL) {
       ptr = cab;

        if (cab == cola) // Un solo elemento en la cola
            cab = cola = actual = NULL;
        else {
            cab->sig->ant = NULL;
            cab = cab->sig;
            actual = cab;
        }
        delete ptr;
        longitud--;
        return true;
    }
    return false;
}
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

```
bool listaD::eliminar(int pos){
    NodoD *ptr = cab;
    int i= 1;

    if ( pos < 1 || pos > longitud)
        return false;
    while (i < pos && ptr != NULL ) {
        ptr = ptr->sig;
        i++;
    }
    if (ptr != NULL){
        if (ptr == cab && ptr == cola) { // Un solo elemento en la cola
           eliminarlista();
        }
        else if (ptr == cab) {  // Se elimina el primero
            sacarInicio();
        }
        else if (ptr == cola) // Se elimina el ultimo
            sacarFinal();
        else { // Se elimina un elemento intermedio
            ptr->sig->ant = ptr->ant;
            ptr->ant->sig = ptr->sig;
            delete ptr;
            longitud--;
        }
        return true;
    }
    return false;
}
```

# Tipo de Dato Abstracto Lista

## Implementación con Lista Doblemente Enlazadas

```
main() {

    listaD L; int e;

    cout <<("\nImplementacion de listas Doblemente Enlazadas\n");
    L.insertarCabeza(2);
    L.insertarCabeza(4);
    L.insertarCabeza(3);
    L.insertarCabeza(1);
    L.insertarCabeza(5);
    L.insertarCabeza(3);
    L.insertarCabeza(0);
    L.listarElementos();

    listaD L2(L);

    cout << endl;
    cout << "Eliminar posicion 3" << endl;
    L.eliminar(3);
    L.listarElementos();

    cout << endl;
    cout << "Eliminar posicion 10" << endl;
    L.eliminar(10);
    L.listarElementos();

    cout << endl;
    cout << "Eliminar posicion 1" << endl;
    L.eliminar(1);
```

## Tipo de Dato Abstracto Lista

### Cola de Prioridad

Como veremos posteriormente una cola es lista de elementos en el cual los elementos se insertan por un extremo (el posterior) y se suprimen en el otro (el anterior o frente). Pero en la vida real puede existir excepciones, por ejemplo en las inscripciones de ciertos cursos, donde a veces se le da preferencia a los que tienen los mejores promedios.

Una cola de prioridad (priority queue) es una lista en la que cada elemento tiene asociada una prioridad y la operación de extracción siempre elige el elemento mas antiguo de mayor (menor) prioridad. Puede ser de interés en simulación, gestión de procesos en sistema de operación, algoritmos voraces, etc.

# Tipo de Dato Abstracto Lista

## Cola de Prioridad

```
int InsertaColaPrioridad(COLA_PRIORIDAD cpp, void *elemento,int prioridad){
     nodo_t *np,*ap=NULL,*cp=cpp->ini;

     if((np=(nodo_t *)malloc(sizeof(nodo_t)))==NULL)
          return 0;
     np->elemento = elemento;
     np->prioridad = prioridad;
     while(cp && np->prioridad <= cp->prioridad){
          ap = cp;
          cp=cp->prox;
     }
     np->prox = cp;
     if(ap)
          ap->prox = np;
     else
          cpp->ini = np;
     return 1;
}
```

## Tipo de Dato Abstracto Lista

### Listas Delta

Algunas aplicaciones requieren llevar un control de tiempo asociado a elementos, por ejemplo el tiempo de poner a dormir un proceso o el tiempo máximo de espera por un evento. Cuando la cantidad de elementos crece, se puede llegar a necesitar una manera de mantener un seguimiento eficaz de estos tiempos.

Si los elementos se almacenan simplemente en una lista ordenada, los elementos tendrían que ser visitados en cada actualización del tiempo. Lo que en ciertos casos por cuestión de rendimiento puede ser inaceptable, por ejemplo en la gestión de procesos de un sistema de operación, siendo una lista del delta una opción esta situación.

## Tipo de Dato Abstracto Lista

### Listas Delta

Una lista delta es una estructura de datos se deriva de una cola de prioridad y se puede utilizar para administrar los tiempos como tiempos de espera y retrasos.

Los elementos **están ordenados de modo que su clave es la relativa a la clave de los elementos que le precede**. Por ejemplo, digamos que se tienen los siguientes elementos, a, b, c, d, con retrasos de 10, 15, 12 y 22 unidades de tiempo de espera respectivamente. Se pueden representar en una cola de prioridad (con clave los tiempo de espera) de la siguiente formas: a, c, b, d, con claves de 10, 2, 3 y 7, respectivamente.

# Tipo de Dato Abstracto Lista

## Listas Delta

```
int InsertaListaDelta(LISTADELTA ldp,void *elemento,int espera){
     nododelta_t *np,*ap=NULL,*cp=ldp->ini;
     if((np=(nododelta_t *)malloc(sizeof(nododelta_t)))==NULL)
          return 0;
     np->elemento = elemento;
     np->espera = espera;
     while(cp && np->espera >= cp->espera){
          np->espera -= cp->espera;
          ap = cp;
          cp=cp->prox;
     }
     np->prox=cp;
     if(cp!=NULL)
          cp->espera -= np->espera;
     if(ap)
          ap->prox = np;
     else
          ldp->ini = np;
     return 1;
}
```

# Tipo de Dato Abstracto Lista

## Listas Delta

```
//  Decrementa tiempo en una lista delta,
void DecrementaListaDelta(LISTADELTA ldp,int tiempo){
     nododelta_t *cp=ldp->ini;
     while(cp){
          if(cp->espera >= tiempo){
               cp->espera -= tiempo;
               break;
          }
          else {
               tiempo -= cp->espera;
               cp->espera = 0;
          }
          cp = cp->prox;
     }
}

//  Retorna 1 si el primer elemento de la lista tiene espera==0
int ListoEnListaDelta(LISTADELTA ldp){
     return ldp->ini && !ldp->ini->espera;
}
```


https://www.youtube.com/watch?v=15urP2LmfqY