Unidad 3 Estructuras Lineales

3.1 Pilas
3.2 Colas
3.3 Listas Encadenadas

3.1 Pilas

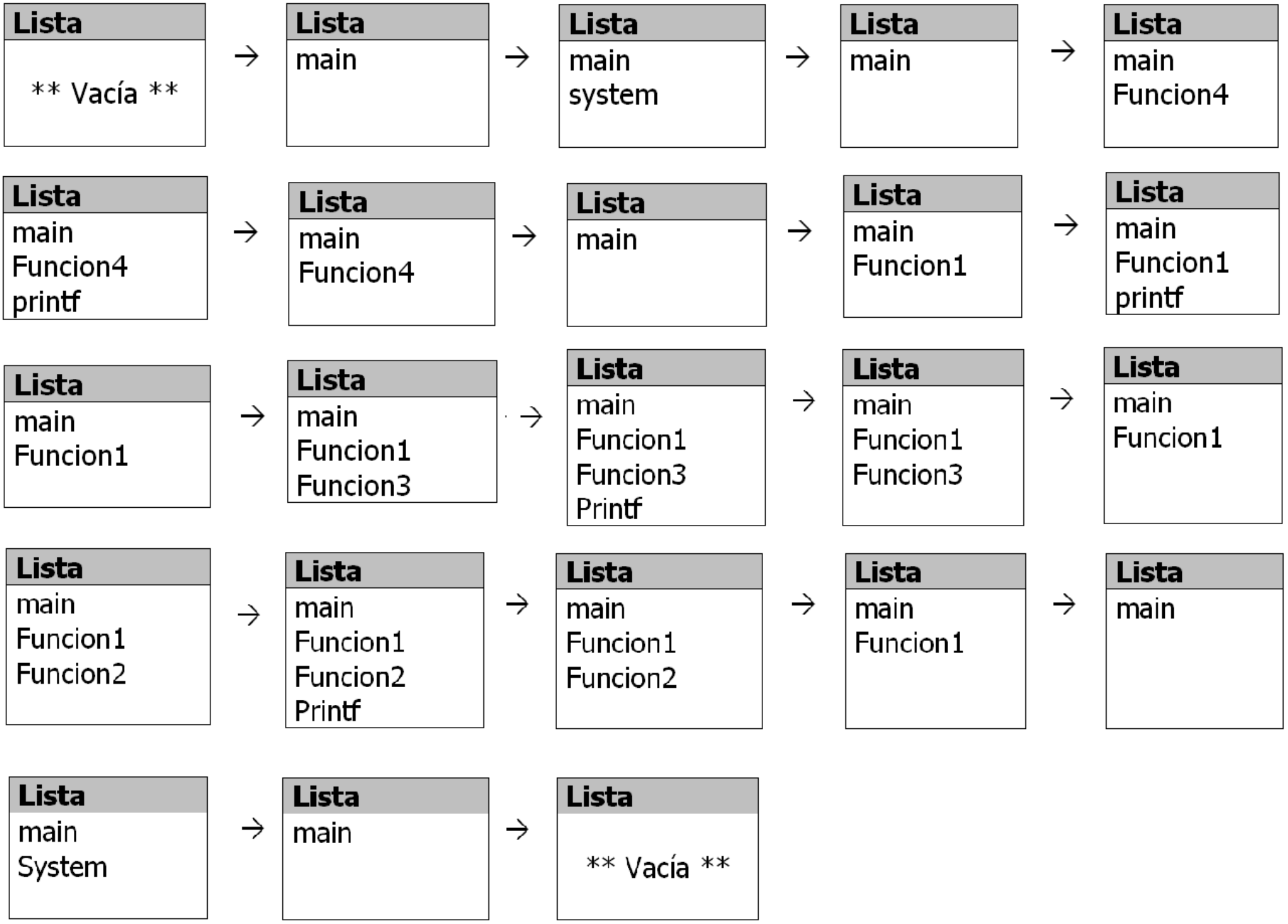
```
#include <iostream>
using namespace std;
void Funcion1(); void Funcion2(); void Funcion3(); void Funcion4();
int main() {
     system("cls");
     Funcion4();
     Funcion1();
     system("pause");
}
void Funcion4() {
       printf("A"); };
void Funcion2() {
       printf("B"); };
void Funcion3() {
       printf("C"); };
void Funcion1() {
       printf("D");
       Funcion3();
       Funcion2(); };
```

¿En que orden se mostrarán en la pantalla las letras A, B, C, y D?

3.1 Pilas

La respuesta es ADCB porque el Sistema Operativo controla la ejecución de los programas respetando el orden siguiente:

1. El programa *main* se ejecuta en primer lugar.
2. Luego se ejecuta la primera instrucción dentro de *main*.
3. Al terminar la ejecución de una instrucción, se continúa con la siguiente.
4. Una instrucción puede ser el nombre de un subprograma.
5. Al iniciar la ejecución de una instrucción se anota su nombre en una lista, si la lista contiene otras instrucciones, se conservan y el nuevo nombre se añade al final de la lista.
7. Mientras una instrucción se encuentra en ejecución, su nombre se mantiene anotado en la lista.
8. Cuando una instrucción termina, su nombre se remueve de la lista.
9. Si una instrucción se borra de la lista, la instrucción que queda al final es la que continuará su ejecución.


Lista
** Vacía **
Lista
main
Lista
main
system
Lista
main
Lista
main
Funcion4
Lista
main
Funcion4
printf
Lista
main
Funcion4
Lista
main
Lista
main
Funcion1
Lista
main
Funcion1
printf
Lista
main
Funcion1
Lista
main
Funcion1
Funcion3
Lista
main
Funcion1
Funcion3
Printf
Lista
main
Funcion1
Funcion3
Lista
main
Funcion1
Lista
main
Funcion1
Funcion2
Lista
main
Funcion1
Funcion2
Printf
Lista
main
Funcion1
Funcion2
Lista
main
Funcion1
Lista
main
Lista
main
System
Lista
main
Lista
** Vacía **

3.1 Pilas

- La lista que provee el S.O. durante la ejecución de un programa es una **PILA**.
- El ejemplo de las páginas previas es una herramienta didáctica, en realidad algo de lo que se guarda en la pila para controlar el orden de ejecución, son direcciones de memoria de retorno.
- La aplicación de pilas que acabamos de conocer es una de las más importantes.
- Las pilas son estructuras **UEPS** o **LIFO** (*Ultimas Entradas Primeras Salidas* o en inglés *Last Input, First Output*).

3.1 Pilas

- Las pilas se crearon como una necesidad.
- Para entender su funcionamiento podríamos usar el concepto de ***Vector Dinámico*** que estudiamos previamente:
- Cada nuevo elemento de la lista se debe **añadir al final** del vector.
- El único dato de la lista, que se puede eliminar es el de la última posición.
- Un poco más adelante, enseguida crearemos una clase llamada Pila, ya que la clase Vector Dinámico tiene más operaciones que no son requeridas en una Pila.

Antes de escribir la clase pila, escriba un programa, que compruebe si cierta frase es un palíndroma.

El programa debe solicitar que el usuario escriba una frase y responder si se trata de un palíndroma o no (los espacios no deben tomarse en cuenta).

PALINDROMAS:

Anilina

Dábale arroz a la zorra el abad

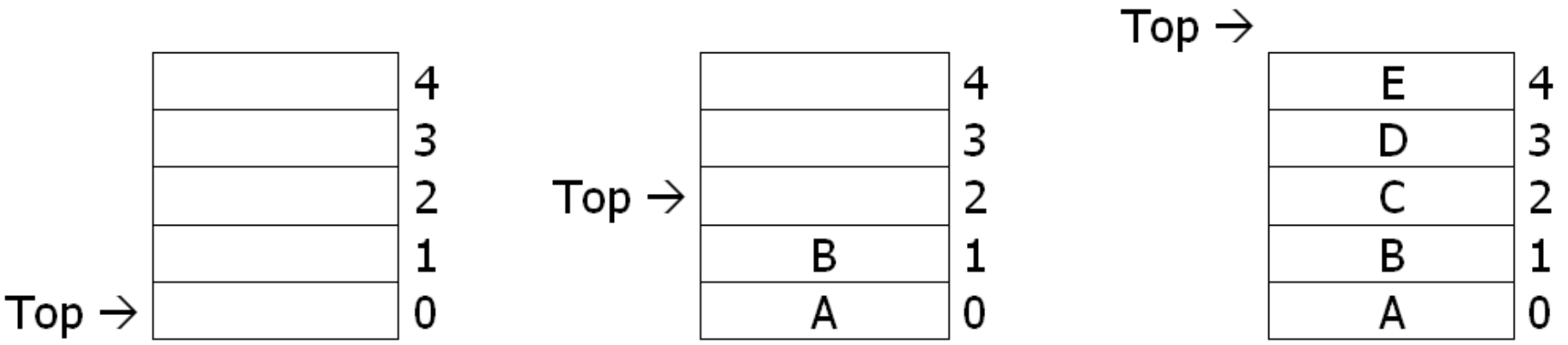
nion anomemata me monan oin *("Lavad vuestros pecados, no solo vuestra cara" en griego)*

NO SON PALINDROMAS:

Estructuras de Datos

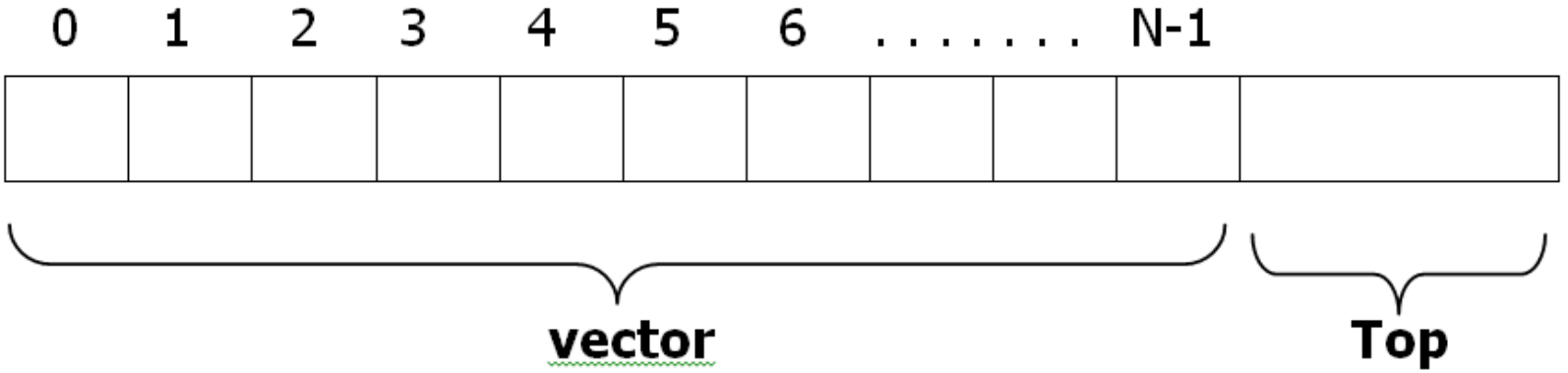
Dábale el abad arroz a la zorra

Diferentes estados de una pila en un vector
(el tamaño físico del vector es 5).



Pila Vacía **Pila Típica** **Pila Llena**

Clase Pila



3.1 Pilas

Métodos	Seudocódigo	Observaciones
Constructor	top ← 0	
push	vector[top] ← dato top++	Guardar un dato en la pila
pop	top - - resulta vector[top]	Elimina un dato De la pila
vacia	si top == 0 resulta true	
llena	si top == max resulta true	

3.1 Pilas

```
// Pila.h

#define TamVector 1000
class Pila{
      private:
        int top;
        char vector[TamVector];
      public:
        Pila()              {top=0;};
        void push(char x) { vector[top]=x; top++; };
        char pop()          { top--; return vector[top]; };
        bool vacia()        { return (top==0); };
        bool llena()        { return (top==TamVector); };
};
```

3.1 Pilas

Escriba un programa, **usando una pila**, que compruebe si cierta frase es un palíndroma.

PALINDROMAS:

Anilina

Dábale arroz a la zorra el abad

nion anomemata me monan oin *("Lavad vuestros pecados, no solo vuestra cara" en griego)*

NO SON PALINDROMAS:

Estructuras de Datos

Dábale el abad arroz a la zorra

3.1 Pilas

// VerificaPalindroma.cpp

```
#include <iostream>
using namespace std;
#include "Pila.h"
int main()
{
    char frase[200];
    cout << "Escribe una frase: ";gets(frase);
```

3.1 Pilas

```
Pila x1,x2,x3; // se requieren 3 pilas
for(int i=0;frase[i]!=0;i++)
    if (frase[i]!=32) // Ignora los espacios.
                      // Falta ignorar la H e uniformar
                      // acentos, mayúsculas, minúsculas
        {
        x1.push(frase[i]);
        x2.push(frase[i]);
        };
while (!x1.vacia()) // invertir contenido de la pila 1
        x3.push(x1.pop());
bool EsPalindroma=true;
while (!x2.vacia()) // Comparar pilas 2 y 3
        if (x3.pop()!=x2.pop())
           EsPalindroma=false;
```

3.1 Pilas

```
    if (EsPalindroma)
        cout << "ES PALINDROMA\n";
    else
        cout << "**NO** ES PALINDROMA\n";
    system("pause");
    return 0;
}
```

Una pequeño recordatorio de programación:

Elimine el constructor de pilas.h y pruebe el valor de top en cada una de las pilas al inicio de la aplicación (intente con las siguientes instrucciones):

cout<< x1.top << " " << x2.top << " " << x3.top << endl;
system("pause");

3.1 Pilas

Ejercicio 2.

Escribir un programa que verifique que una expresión aritmética se encuentre agrupada correctamente. Los símbolos de agrupación a considerar son paréntesis, corchetes y llaves. No se requiere revisar la validez de la expresión desde otro punto de vista, solo de la agrupación.

Expresiones Válidas

```
(a+b)*(c-d)
(a+b)*x[1]+y[0,0]
xzy-(*a){ [ ] }
((a+b)/(c-d)+x)/{((x+1)/2)/(z-3)}
a*(b[R]+c{3})
{ }
( )
[ ]
```

Expresiones Inválidas

```
(a+b[i]
)x+y[3,0](
a+b[3)+c{1}
(a[1)+1]*2
)
(
```

3.1 Pilas

// VerificaParentesis.cpp

```
#include <iostream>
using namespace std;
#include "pila.h"
#define PARIZQ '(' // PARentesis
#define PARDER ')'
#define CORIZQ '[' // CORchetes
#define CORDER ']'
#define LLAIZQ '{' // LLAves
#define LLADER '}'
int main()
{
    char expresion[100];
    cout << "Expresión a revisar: ";gets(expresion);
```

3.1 Pilas

```
Pila x; bool EsCorrecta=true;
for(int i=0;expresion[i]!=0;i++)
   if (expresion[i]==PARIZQ)
      x.push(PARDER);
   else if (expresion[i]==CORIZQ)
      x.push(CORDER);
   else if (expresion[i]==LLAIZQ)
      x.push(LLADER);
   else if ((expresion[i]==PARDER)or(expresion[i]==CORDER)
            or(expresion[i]==LLADER))
      if (x.vacia())
         EsCorrecta=false;
      else if (expresion[i]!=x.pop())
         EsCorrecta=false;
if (!x.vacia())
     EsCorrecta=false;
```

3.1 Pilas

```
    if (EsCorrecta)
          cout << "EXPRESIÓN VALIDA\n";
    else
          cout << "EXPRESIÓN INVÁLIDA\n";
    return 0;
}
```

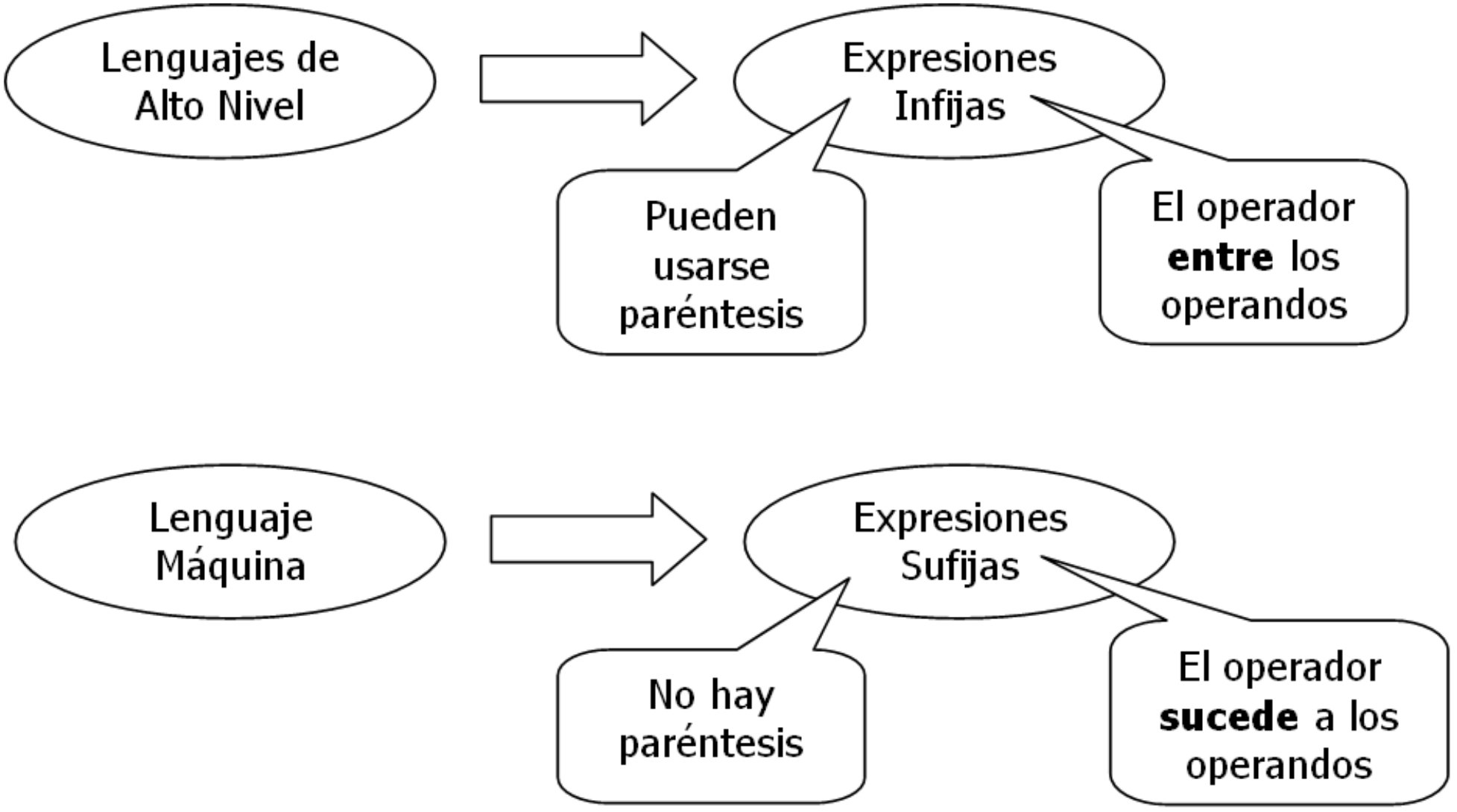
3.1 Pilas

Ejercicio:

Modifique el programa anterior para que se pueda incluir como pareja de símbolos de agrupación a los caracteres de “*Menor que*” **<** y “*Mayor que*” **>**

3.1 Pilas

Conversión de Expresiones Aritméticas a la Forma Sufija



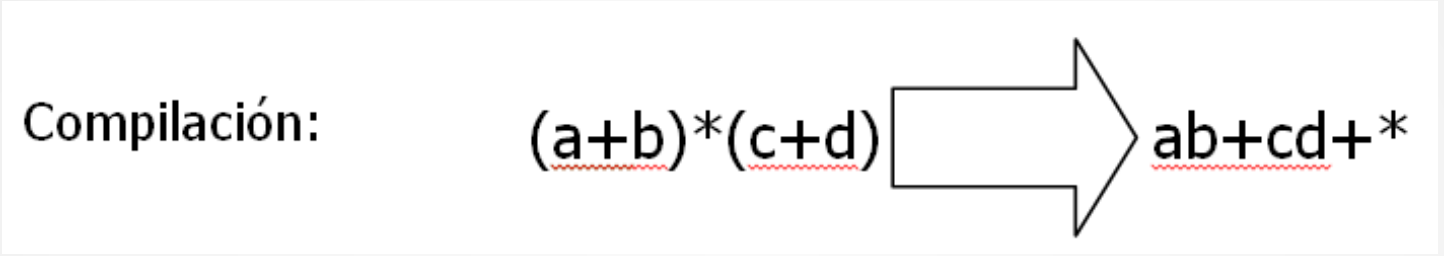
3.1 Pilas

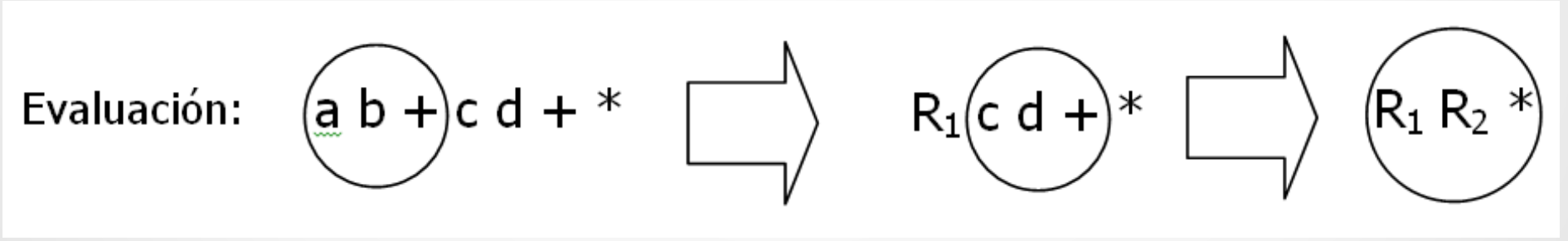
Forma infija	Forma sufija equivalente
a+b	a b +
5+1	5 1 +
a+b*c	a b c * +
5+1*3	5 1 3 * +
(a+b)*c	a b + c *
(5+1)*3	5 1 + 3 *
(a+b)^2*(c-d)	a b + 2 ^ c d - *
(5+1)^2*(3-4)	5 1 + 2 ^ 3 4 - *

3.1 Pilas

Jerarquía de operadores (de mayor a menor):

^ potencia * / multiplicación y división + - suma y resta





Las expresiones en la forma sufija se evalúan de izquierda a derecha.
El primer operador que se encuentre, se aplica a los 2 operandos previos.
El resultado obtenido se convierte en un operando y se inicia de nuevo desde la izquierda la búsqueda de un operador.

Algoritmo para la conversión de una expresión infija a la forma sufija

a) Se usarán dos pilas, llamadas pila de **operadores** y pila ***polish***.
b) La expresión infija se encontrará en un **string**.
c) Si la expresión contiene paréntesis, debe estar correctamente agrupada.
d) Se repetirá el siguiente procedimiento para cada elemento de la expresión infija:
   - Se toma un elemento de la expresión infija.
   - Si es un operando[1] se guarda en la pila ***polish***.
   - Si es el primer operador que llega[2] o es un paréntesis izquierdo, se guarda en la pila de **operadores**.
   - Si lo que llega es un paréntesis derecho, se transfieren **uno a uno** los operadores de la pila de **operadores** hacia la pila ***polish*** hasta que se encuentre un paréntesis izquierdo[3], y se desechan ambos paréntesis.
   - Si no se trata de ninguno de los dos casos anteriores, se comparará la jerarquía del operador que está en la cima de la pila de **operadores** con la del operador que quiere entrar:
     - Si la jerarquía del operador que llega es mayor que la jerarquía del operador que está en la cima, se almacena el operador que llega, en la pila de **operadores**.
     - En caso contrario, se transfiere el operador que está en la cima de la pila de **operadores** hacia la pila ***polish*** y de nuevo se hace la comparación entre el operador que llega y el que ahora queda en la cima de la pila de **operadores**.
e) Una vez que ya no haya elementos en la expresión infija, se transfieren uno a uno los operadores de la pila de operadores hacia la pila *polish*.

Referencias:

1. Un operando es una variable o una constante.
2. Se asumirá que es el primer operador que llega verificando si la pila de **operadores** está vacía, o si en el tope de ella hay un paréntesis izquierdo.
3. El último que entró, es decir, el que corresponde con el paréntesis derecho que ha llegado.

3.1 Pilas

```
#include <iostream>
using namespace std;
#include "Pila.h"
#define PARIZQ '('
#define PARDER ')'
bool EsOperando(char);
bool EsOperador(char);
int Jerarquia(char);
int main()
{
   char Infija[100];
   cout << "Expresión INFIJA: ";gets(Infija);
      Pila Polish,Operadores;
      for(int i=0;Infija[i]!=0;i++){
          if (EsOperando(Infija[i]))
                    Polish.push(Infija[i]);
       else if (Infija[i]==PARIZQ
          Operadores.push(Infija[i]);
```

3.1 Pilas

```
else if ((EsOperador(Infija[i])) and
(Operadores.vacia()or(Operadores.ver()==PARIZQ)))
      Operadores.push(Infija[i]);
else if (Infija[i]==PARDER){
      char temp;
      temp = Operadores.pop();
      while (temp!=PARIZQ){
            Polish.push(temp);
            temp = Operadores.pop();
            };
      }
else if (Jerarquia(Infija[i])>Jerarquia(Operadores.ver()))
      Operadores.push(Infija[i]);
else {
      Polish.push(Operadores.pop());
      i--;
      };
}
```

3.1 Pilas

```
    while (!Operadores.vacia())
          {
                Polish.push(Operadores.pop());
          }
    while (!Polish.vacia())
          {
                Operadores.push(Polish.pop());
          };
    while (!Operadores.vacia())
          {
                cout << Operadores.pop();
          };
  cout << endl;
  return 0;
}
```

3.1 Pilas

```
bool EsOperador(char valor){
      if ((valor=='*') or (valor=='/') or
       (valor=='+') or (valor=='-') or (valor=='^'))
            return true;
      else
            return false;
};

bool EsOperando(char valor) {
      if ((valor>=65)and(valor<=90))
            return (true);
      else if ((valor>=97)and(valor<=122))
            return (true);
      else if ((valor>=48)and(valor<=57))
            return (true);
      else
            return false;
};
```

3.1 Pilas

```
int Jerarquia(char operador)
{
      if (operador=='^') return 5;
      else if (operador=='*') return 3;
      else if (operador=='/') return 3;
      else if (operador=='+') return 0;
      else if (operador=='-') return 0;
};
```

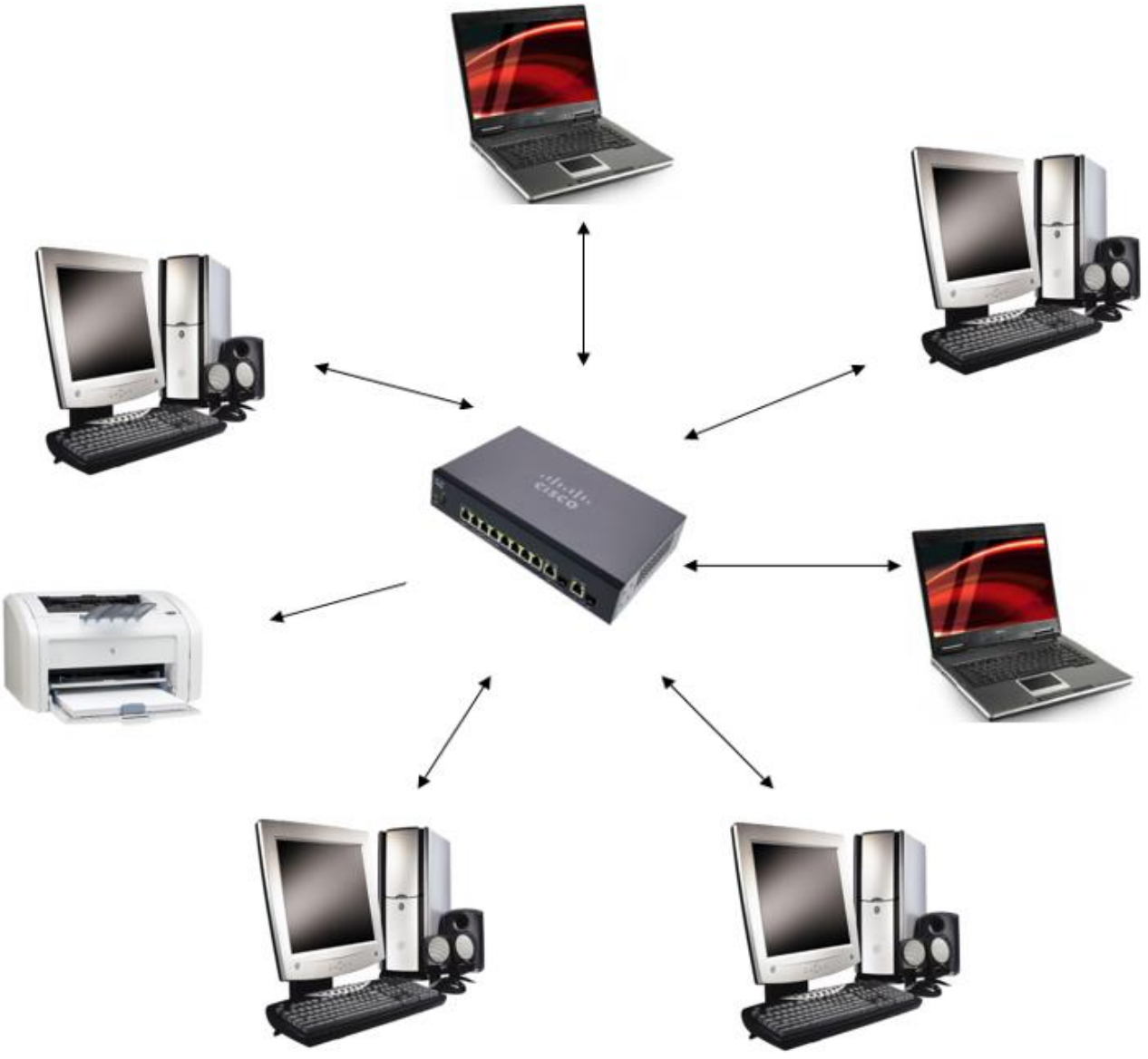
```
// El método siguiente debe añadirse a la
// Clase Pila
    char ver()
          {return vector[top-1];};
```

3.2 Colas

Una de las aplicaciones:

Impresión en red

1) Un usuario de la red desea imprimir un documento.
2) La impresora está disponible.
3) El documento es de tamaño importante.
4) Casi simultáneamente otros tres usuarios desean imprimir pero la impresora está ocupada.
5) ¿A cual de los tres se le asignará la impresora una vez que termine la impresión del documento grande?



3.2 Colas

Otra aplicación:

Un sistema operativo multitarea debe distribuir el tiempo de operación del CPU para atender las tareas que corren **"simultáneamente"** durante la operación regular del equipo, por ejemplo:

- Antivirus
- Reproductor de Música
- Microsoft Word®

3.2 Colas

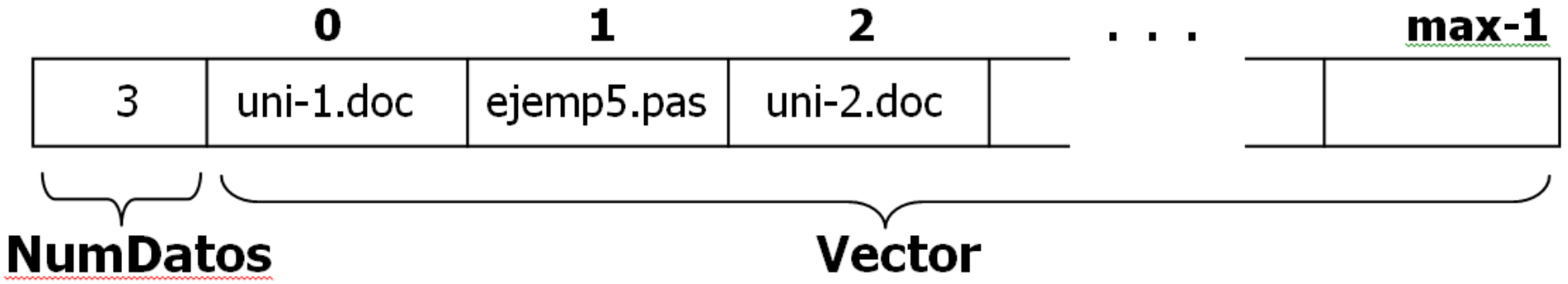
Las colas son estructuras de datos **PEPS** (**P**rimeras **E**ntradas, **P**rimeras **S**alidas) o **FIFO** (**F**irst **I**nput, **F**irst **O**utput).

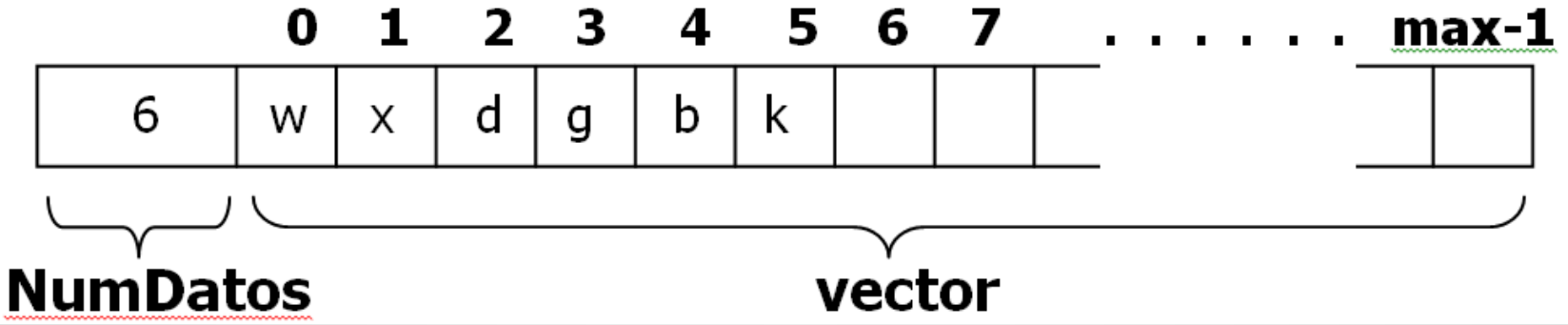
Cola con corrimiento

- Las colas en los sistemas son como las de un Banco o del cine.
- Solo que no atienden personas, sino procesos o tareas.
- Una **cola con corrimiento** guarda los datos de los procesos en el orden que serán atendidos cuando sea posible.
- Una cola siempre inicia vacía y su tamaño cambia continuamente.
- Se usará el mismo concepto del vector dinámico.
  - Cada dato que se va a colocar en espera se añadirá al final del vector.
  - Las tareas que ya fueron atendidas (por ejemplo la impresión cuya atención se inició) deben ser eliminados de la cola, por lo que el dato de la **posición 0** del vector es el que se elimina.

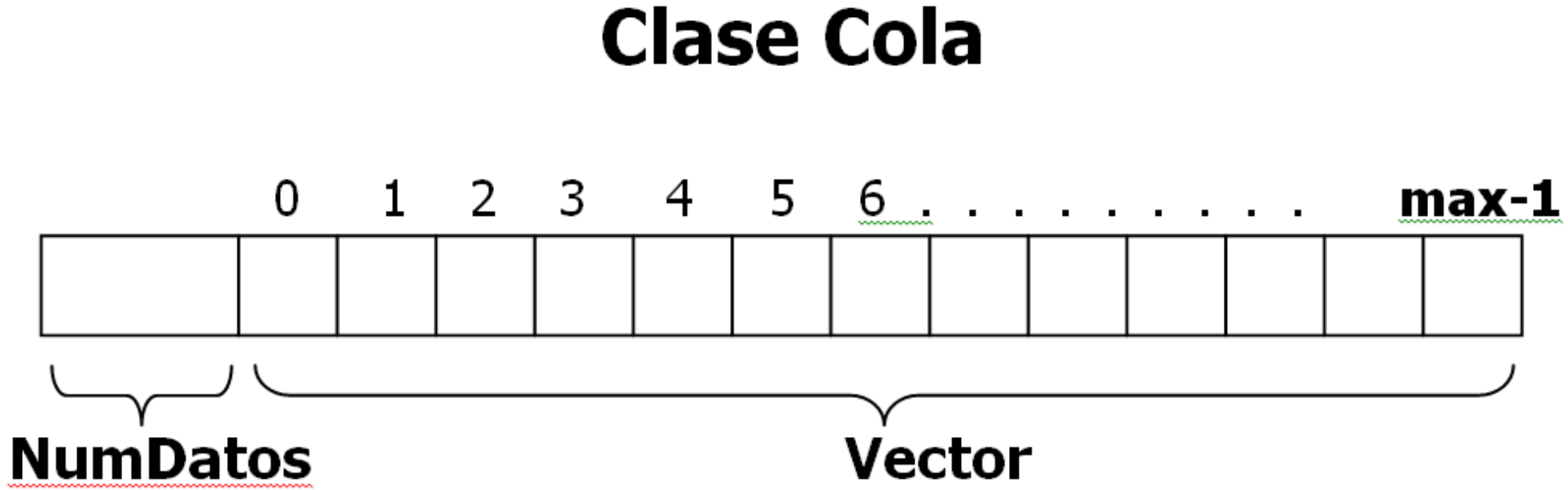
3.2 Colas

Los siguientes son dos arreglos dinámicos en cierto momento (el primero es una cola de impresión y el segundo una aplicación cualquiera no especificada).





3.2 Colas



3.2 Colas

Método	Objetivo y Observaciones	Seudocódigo
Constructor	✓ Se asegura que la cola esté vacía. ✓ Equivale a **ModifTam(0)**	numdatos ← 0
Entra	✓ Añade un nuevo dato a la cola. ✓ Equivale a **Añadir(dato)**	vector[numdatos] ← dato numdatos++
Sale	✓ Recupera el dato al que toca turno y lo retira de la cola. ✓ Equivale a **Eliminar(0)**	Resulta vector[0] Para i ←1 hasta numdatos-1 ejecutar vector[i – 1] ← vector[i] numdatos--
Vacia	✓ Indica si la cola no tiene ningún dato en espera.	Si numdatos=0 Resulta TRUE
Llena	✓ Indica si no hay capacidad en la cola para un dato más.	si numdatos=max Resulta TRUE

```
#define MaxCola 1000
class Cola {
      private:
        int numdatos;
        char vector[MaxCola];
      public:
        Cola();
        void entra(char);
        char sale();
        bool vacia();
        bool llena();
    };

Cola::Cola() {
   numdatos=0;
};

void Cola::entra(char x) {
    vector[numdatos]=x;
    numdatos++;
}
```

3.2 Colas

```
char Cola::sale() {
    char temp=vector[0];
    for(int i=1;i<numdatos;i++)
       vector[i-1]=vector[i];
    numdatos--;
    return temp;
}

bool Cola::vacia() {
     return (numdatos==0);
};

bool Cola::llena() {
     return (numdatos==MaxCola);
};
```

3.2 Colas

Ejercicio

Escribir un programa que realice lo siguiente:

- Tomar una frase del teclado.
- Guardar cada uno de los caracteres en una cola llamada x1.
- Repetir cierto número de veces las dos secuencias siguientes:
  - Hasta que **x1** quede vacía, ejecutar:
    - Tomar de la cola **x1** un valor e imprimirlo.
    - Hacer un retraso de 1 seg. y guardar el dato en otra cola llamada **x2**.
  - Hasta que **x2** quede vacía, ejecutar:
    - Tomar un valor de la cola **x2** e imprimirlo.
    - Hacer un retraso de 1 segundo y guardar el dato en **x1**.

3.2 Colas

```
#include <iostream>
using namespace std;
#include "windows.h"
#include "cola.h"
int main()
{
    char frase[50];
    cout << "Escribe una frase: ";gets(frase);
    int veces;
    cout << "Número de Veces: ";cin>>veces;
    Cola x1,x2;
    for(int i=0;frase[i]!=0;i++)
        x1.entra(frase[i]);
```

3.2 Colas

```
    char letra;
    for(int i=0;i<veces;i++) {
        while (!x1.vacia()) {
              Sleep(200);
              letra=x1.sale();
              cout<<letra;
              x2.entra(letra);
        };
        cout << endl;
        while (!x2.vacia()) {
              letra=x2.sale();
              x1.entra(letra);
        };
    };
    return 0;
}
```

3.2 Colas

¿Tienen alguna desventaja las colas con corrimiento?

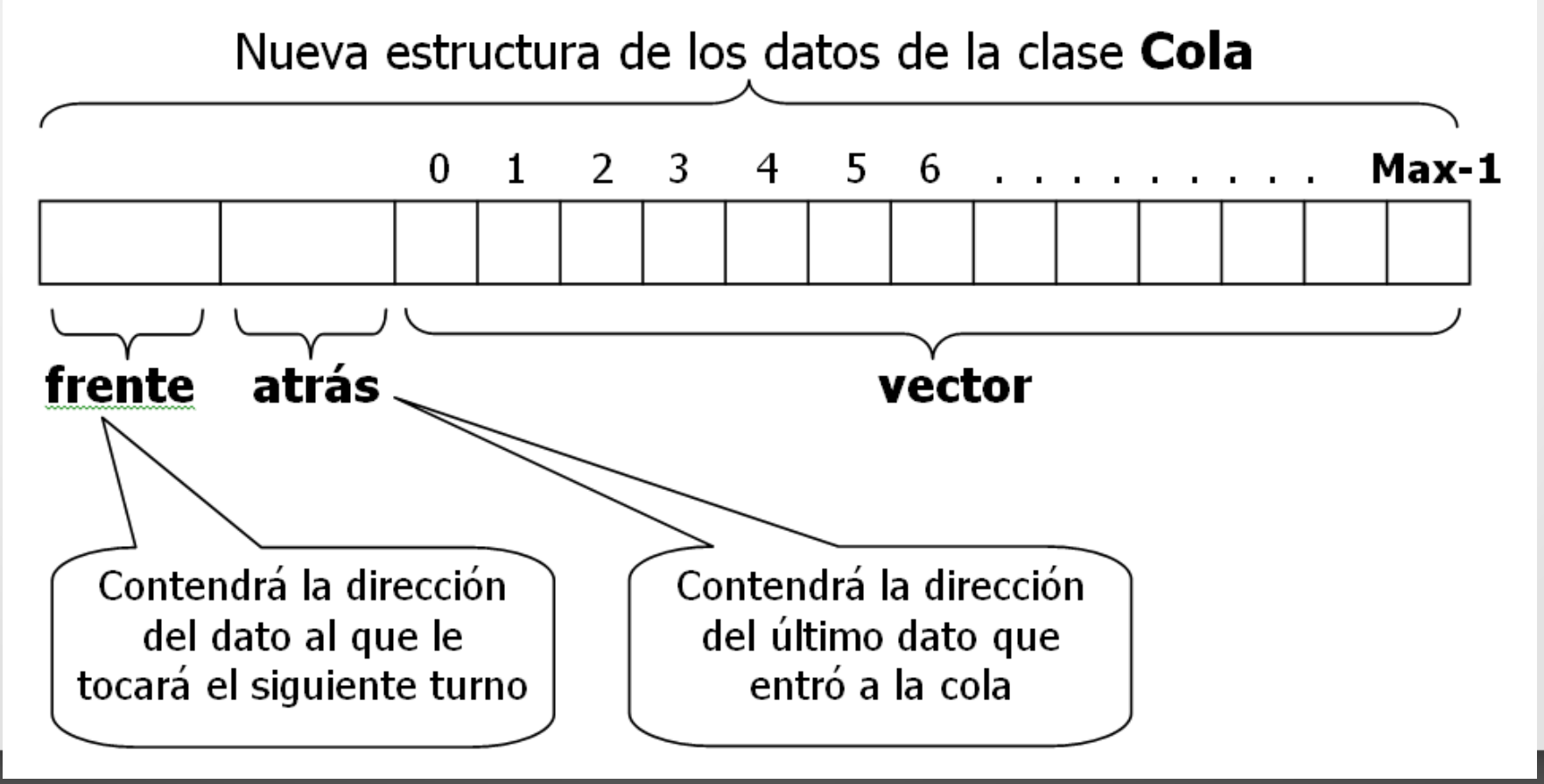
Si

el corrimiento del resto de los datos cuando sale uno de ellos (es un trabajo innecesario como lo veremos enseguida)

3.2 Colas. **Cola Simple**

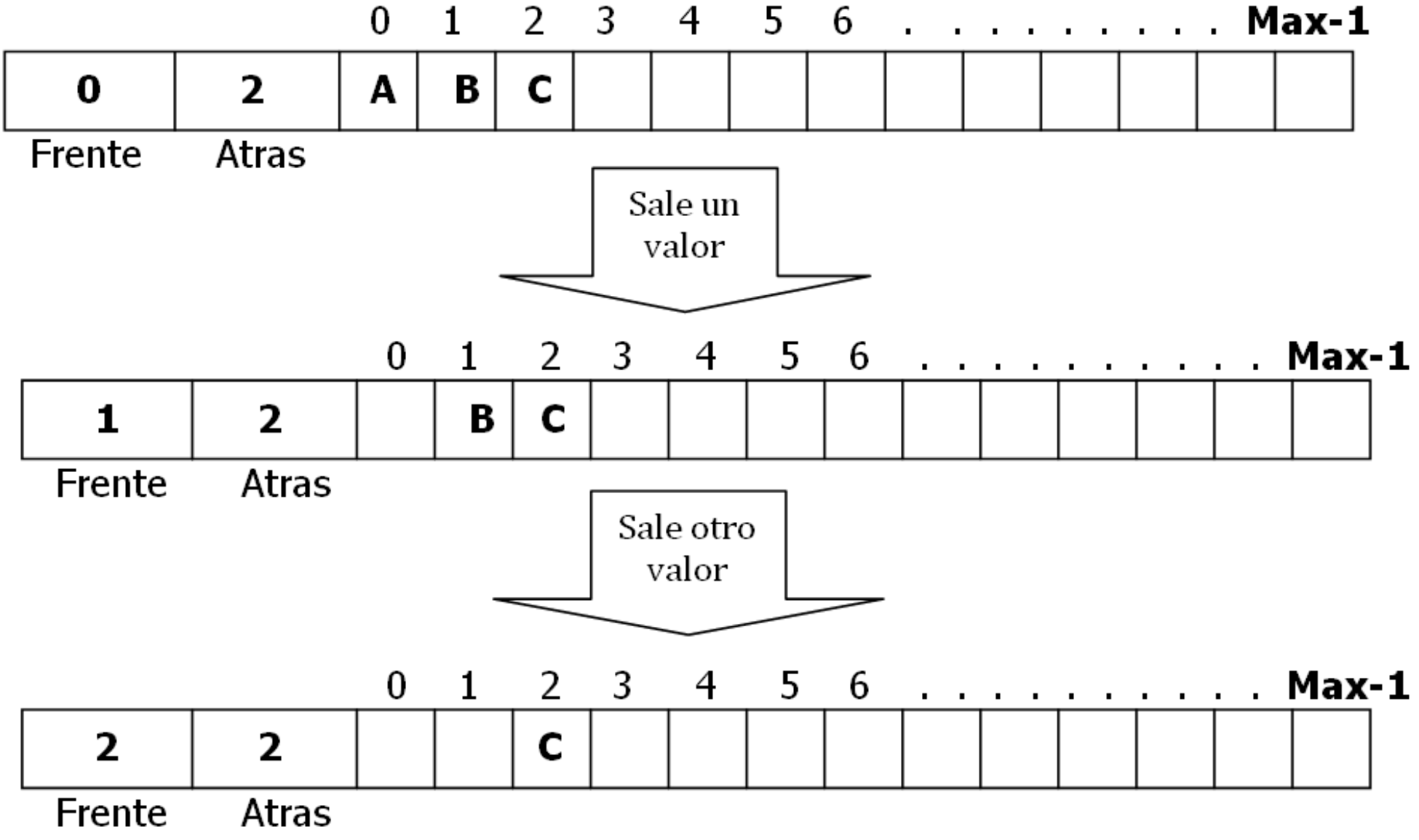
Hay una solución llamada ***Cola Simple*** que elimina el corrimiento.

Cambio de paradigma para la clase *Cola* : Se incorporan dos apuntadores: ***frente*** y ***atrás***, que reemplazan a la variable **NumDatos**

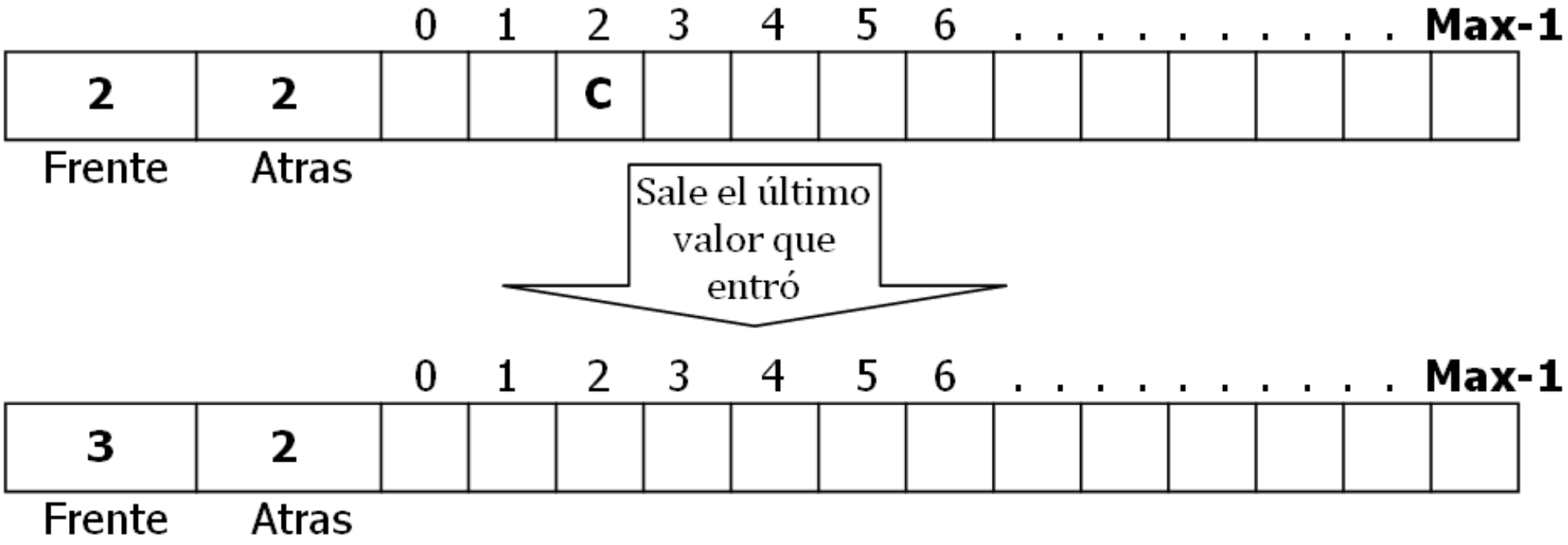


3.2 Colas

Estado de la *Cola* en cierto momento (entraron 3 valores y no ha salido ninguno)

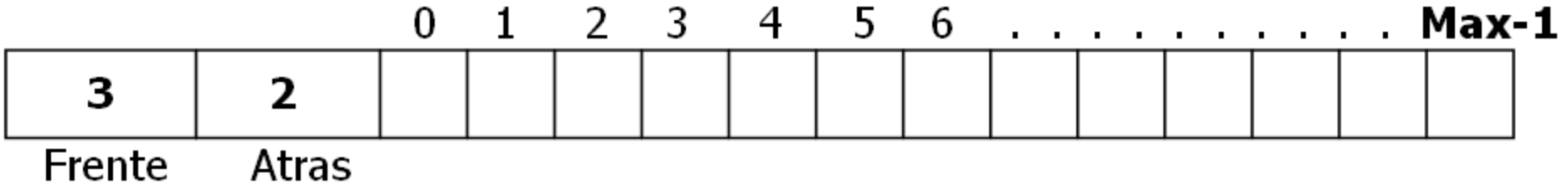


3.2 Colas



Atras se mueve cada vez que entra un nuevo valor,
Frente se mueve cada vez que sale un valor

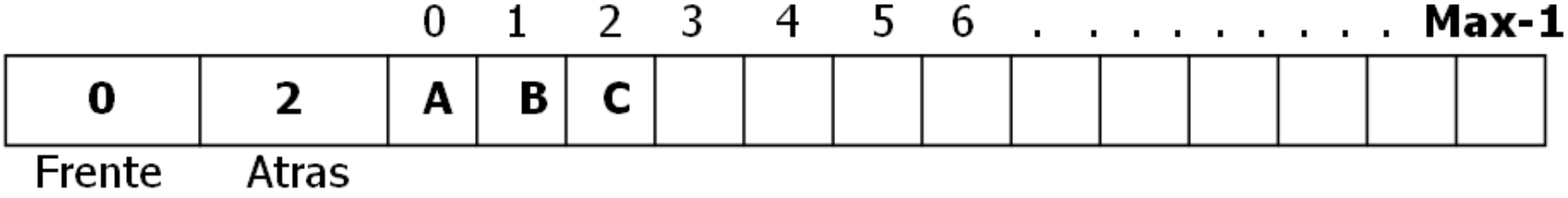
3.2 Colas



Se debe observar que en este estado, **la Cola se encuentra vacía** ya que el único valor que había, se retiró.

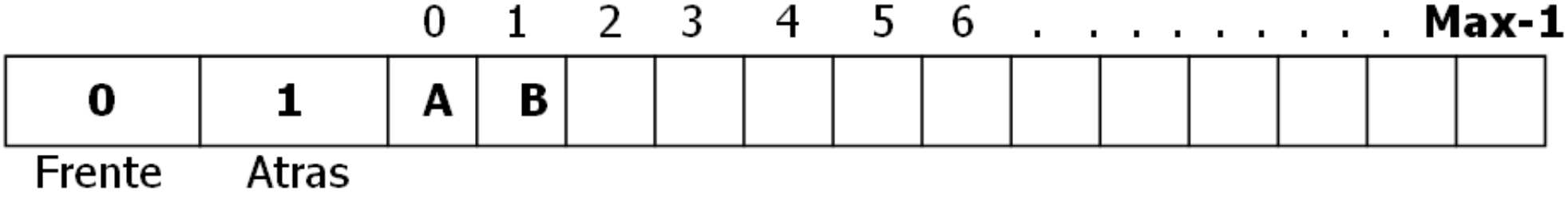
3.2 Colas

Ahora, se va a empezar con el estado del que se partió antes:

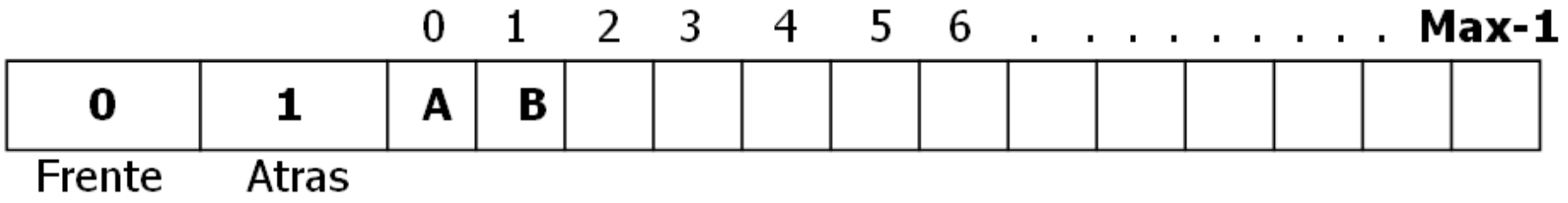


¿Cómo se llegó a este estado?

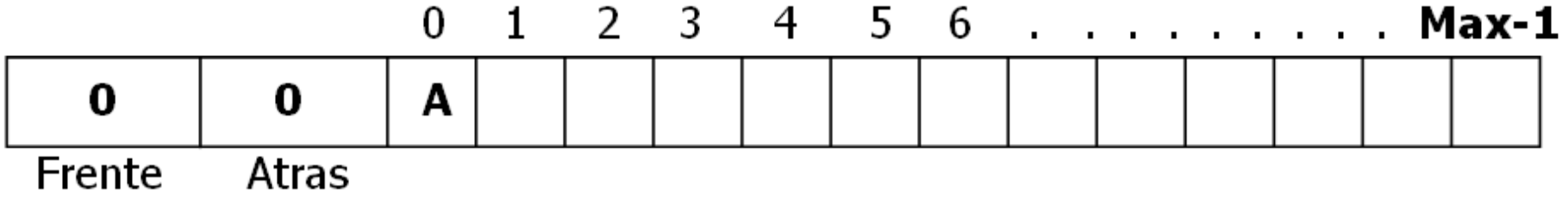
Como el último valor que entró es una **C**, significa que el estado anterior, forzosamente tuvo que ser el que se muestra enseguida:



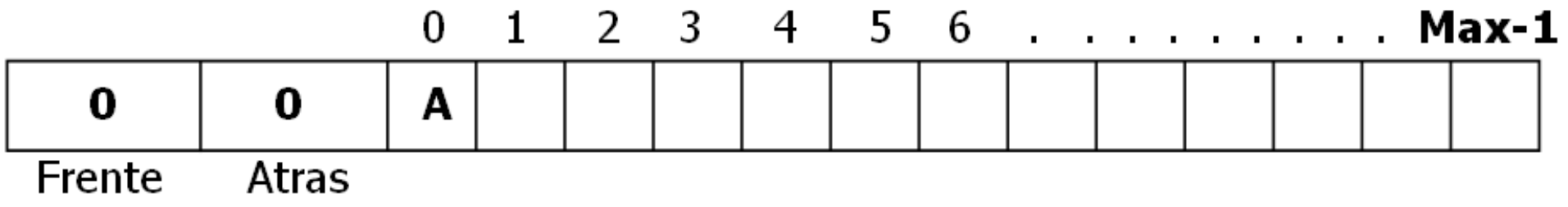
3.2 Colas



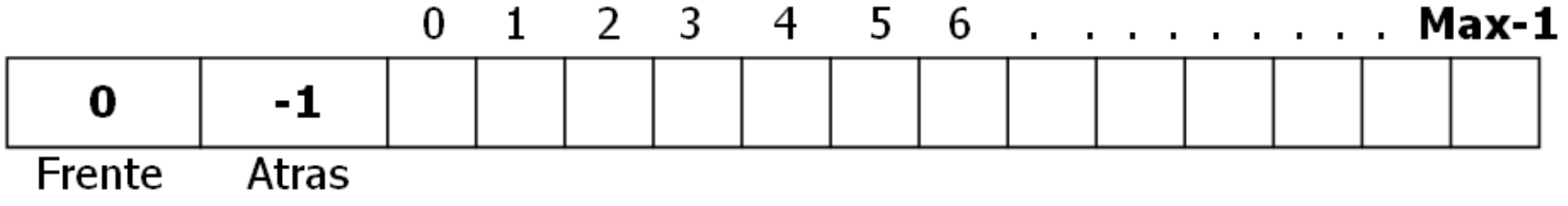
Y el estado anterior tuvo que ser éste:



3.2 Colas



Como la **A** es el único valor que hay en la Cola, significa que, considerando que el apuntador ***Atras*** se incrementa en 1 cada vez que entra un valor, el estado anterior tuvo que ser este:



por lo tanto éste tiene que ser el estado inicial de una ***Cola Simple***

3.2 Colas

Métodos de la clase **Cola**

Métodos	**Seudocódigo**
Constructor	Atras ← **−1** Frente ← **0**
Entra	Mover ***Atras*** a la siguiente dirección (sumar 1 a ***Atras***) vector[Atras] ← dato
Sale	Valor Resultante: vector[Frente] Mover ***Frente*** a la siguiente dirección (sumar 1 a ***Frente***)
Vacia	Si Frente > Atras Resulta TRUE
Llena	Si Atras == Max-1 Resulta TRUE

3.2 Colas

Ejercicio

Escriba la clase **Cola** en C++ bajo el criterio de **cola simple**. **Pruébela** con la aplicación **marquesina.cpp**, solo asegúrese de que el número de veces de impresión multiplicado por la longitud de la frase exceda el valor de MaxCola para que pueda observar el problema que presenta la Cola Simple.

3.2 Colas

```
#define MaxCola 100

class Cola {
      private:
        int frente,atras;
        char vector[MaxCola];
      public:
        Cola();
        void entra(char);
        char sale();
        bool vacia();
        bool llena();
    };

Cola::Cola() {
       frente=0;
       atras=-1;
};
```

3.2 Colas

```
void Cola::entra(char x) {
    atras++;
    vector[atras]=x;
}

char Cola::sale() {
    char temp=vector[frente];
    frente++;
    return temp;
}

bool Cola::vacia() {
     return (frente>atras);
};

bool Cola::llena() {
     return (atras==MaxCola-1);
};
```

3.2 Colas

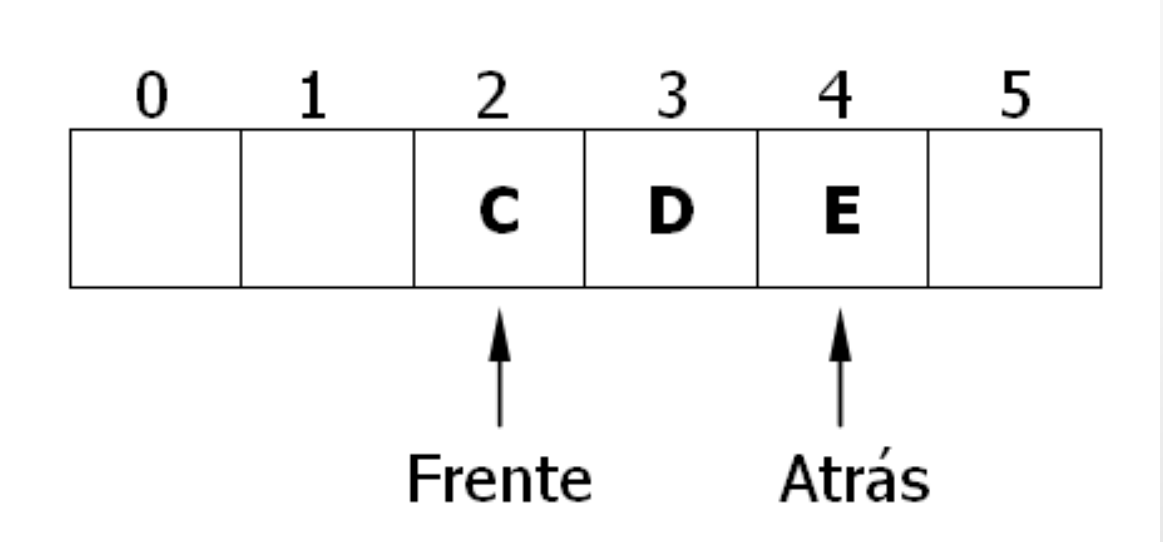
Aunque los datos no se mueven al salir uno de ellos y con ello se evita un trabajo innecesario, cuando se llega al final del vector, si no se prevé la **verificación de que esté llena la Cola**, se invadirán direcciones de memoria asignadas a **otras variables** causando resultados imprevisibles

3.2 Colas

Cola Circular

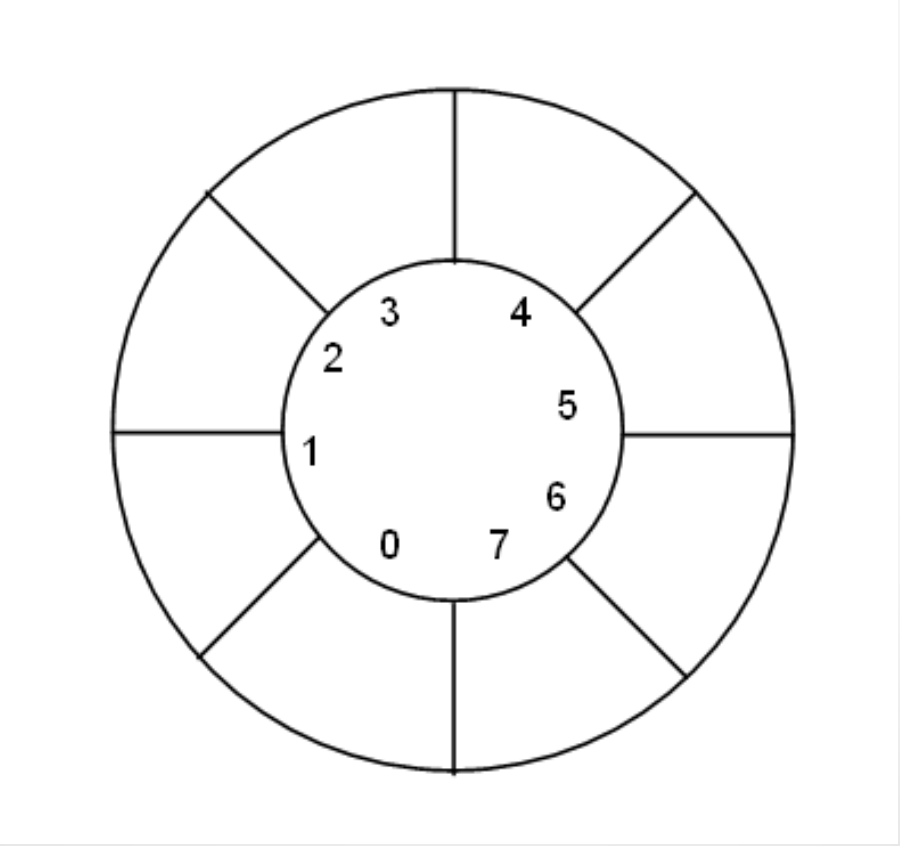
En una Cola Simple, los lugares libres no se vuelven a ocupar.

Se pueden proponer muchas soluciones a este problema, por ejemplo: mover en cierto momento los apuntadores a la posición inicial, sin embargo surgen muchas consideraciones técnicas.



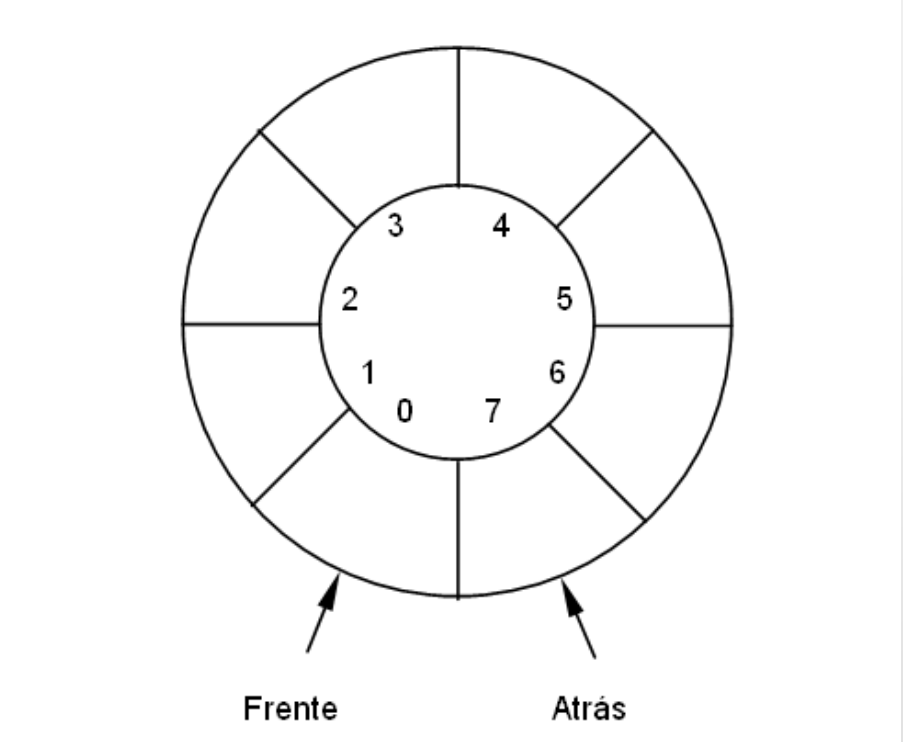
3.2 Colas

La propuesta ideal bajo memoria estática es usar un concepto llamado *vector circular*, lo que implica establecer una **secuencia lógica** entre la última dirección del vector y la primera



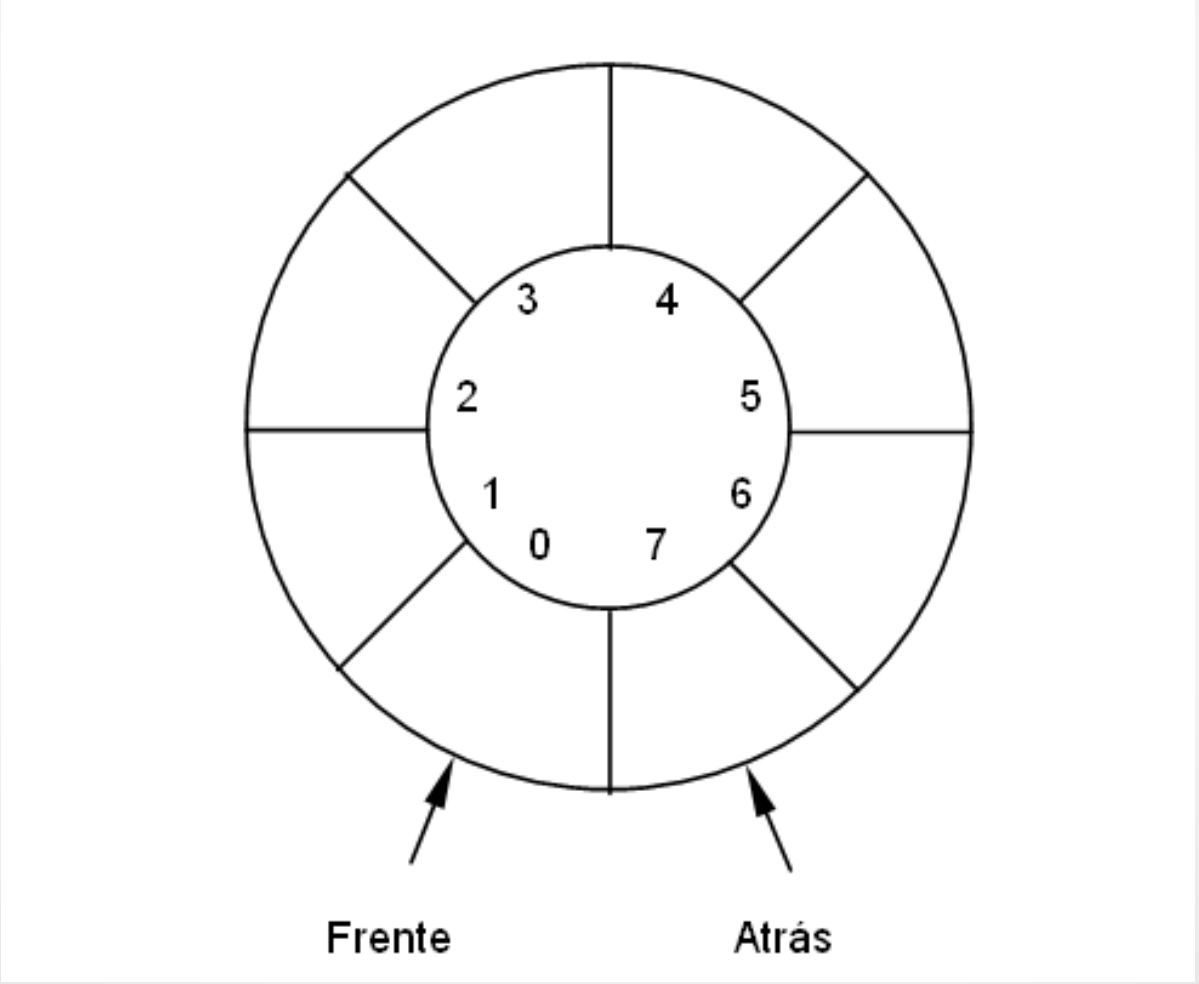
3.2 Colas

Haremos una simulación del comportamiento de una cola como esta usando los mismos algoritmos que se tienen para la cola simple.
En una cola simple, los valores iniciales de los apuntadores son **atrás = -1** y **frente** = **0**
Como ahora se usará un vector circular, suena razonable que **atrás** = **Max-1** y **frente** = **0**



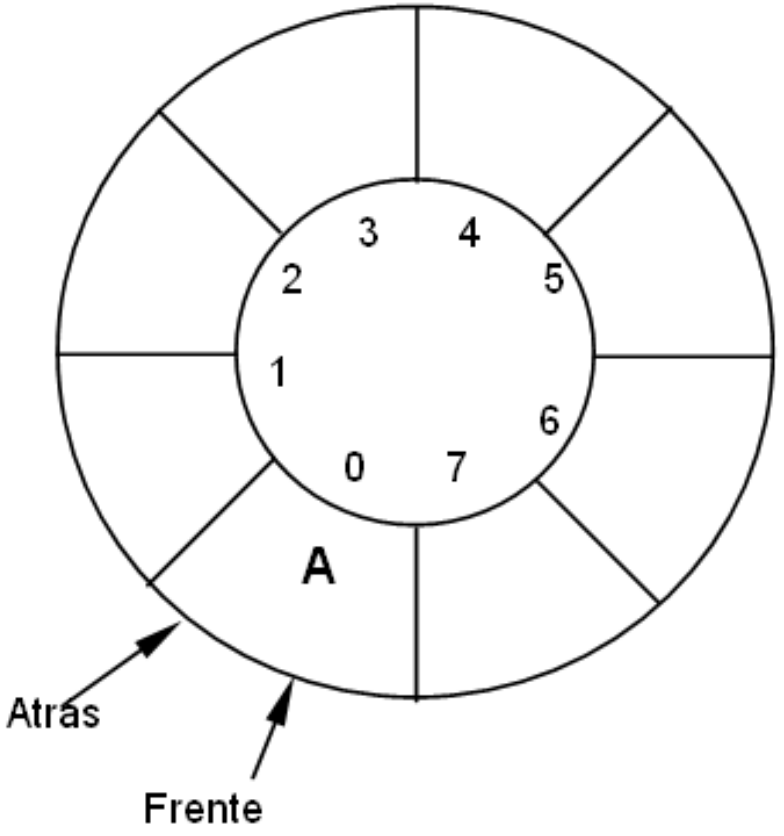
3.2 Colas

Estado inicial
Cola Vacía

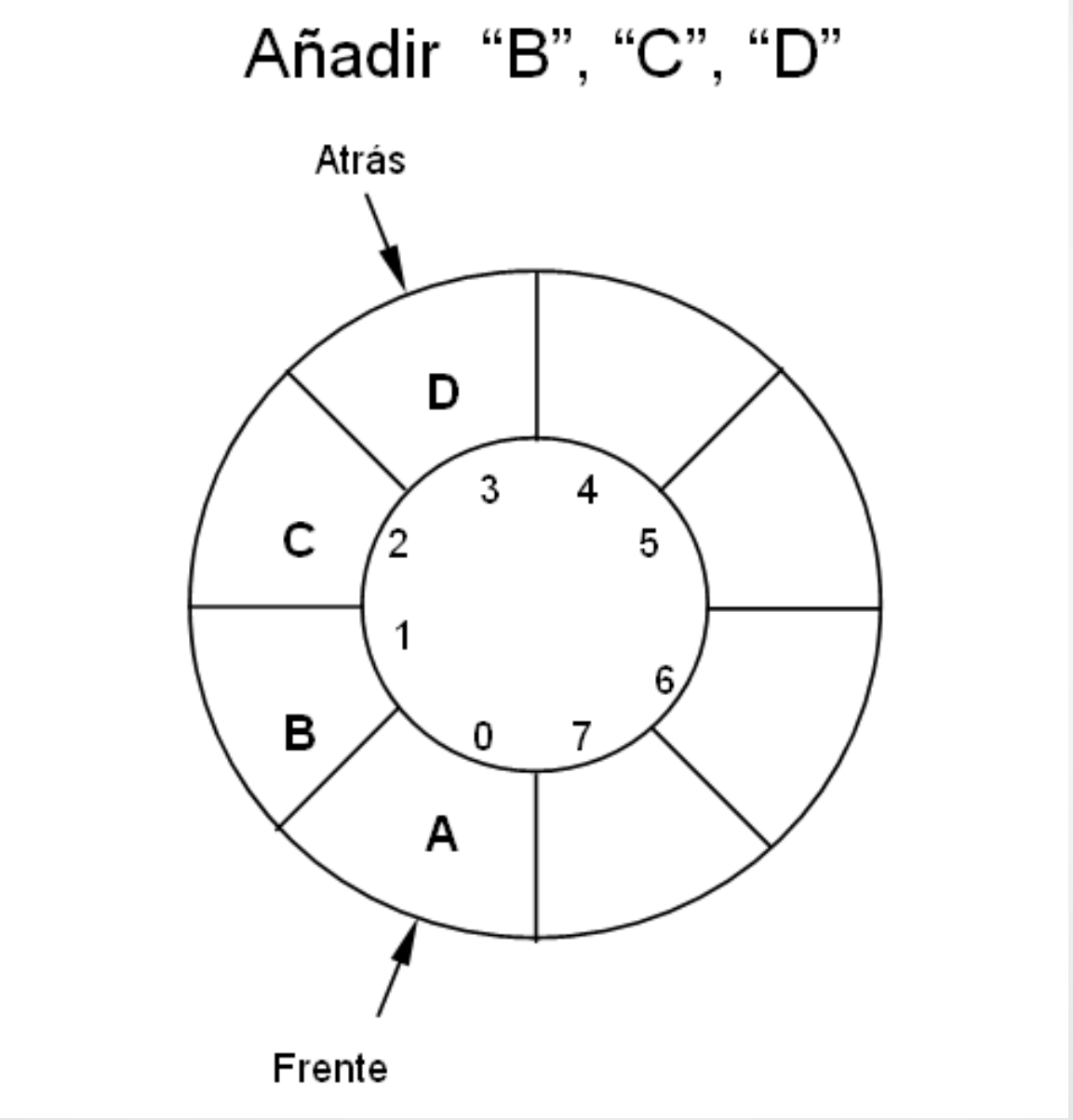


3.2 Colas

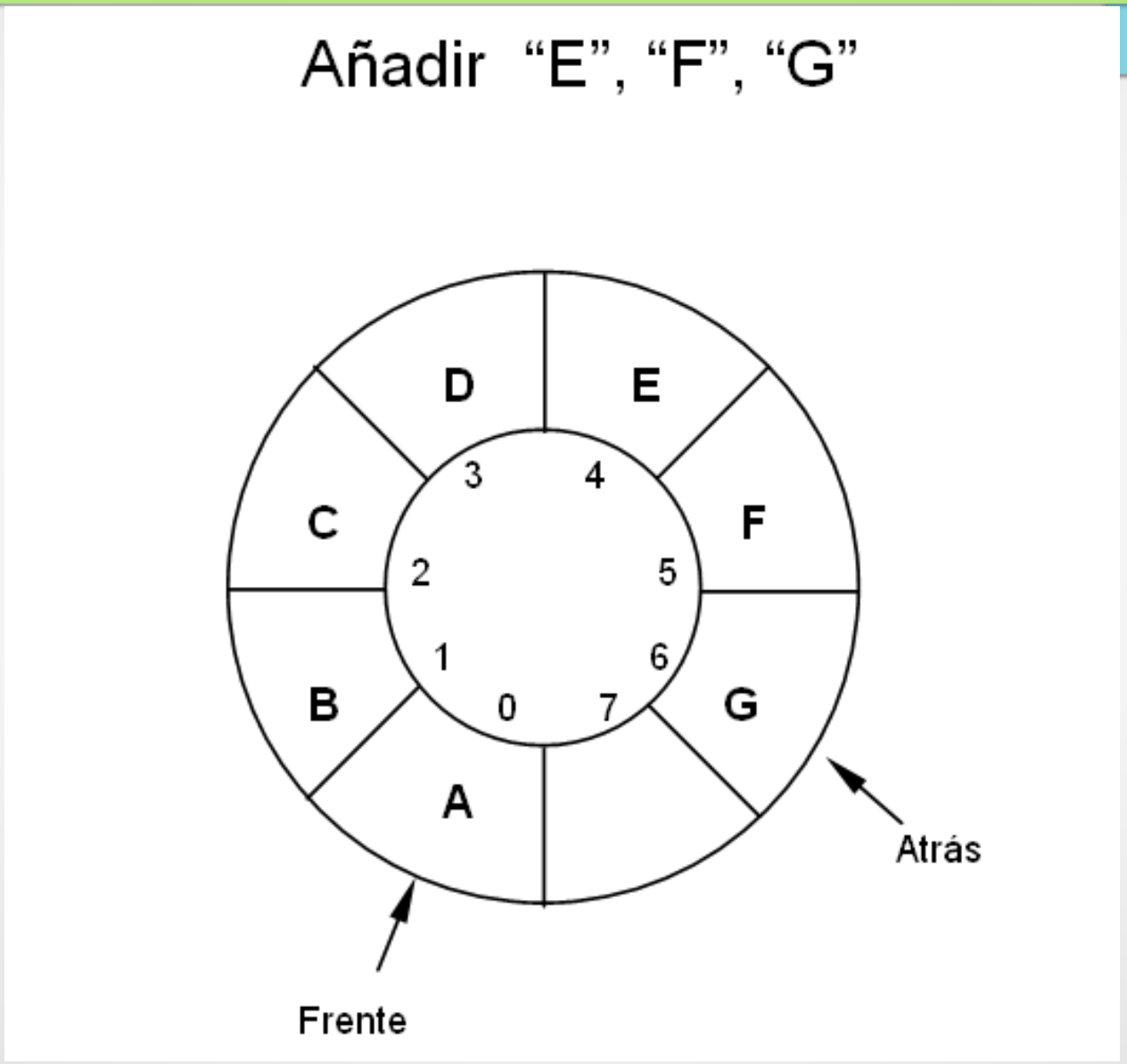
Entra un valor “A”



3.2 Colas

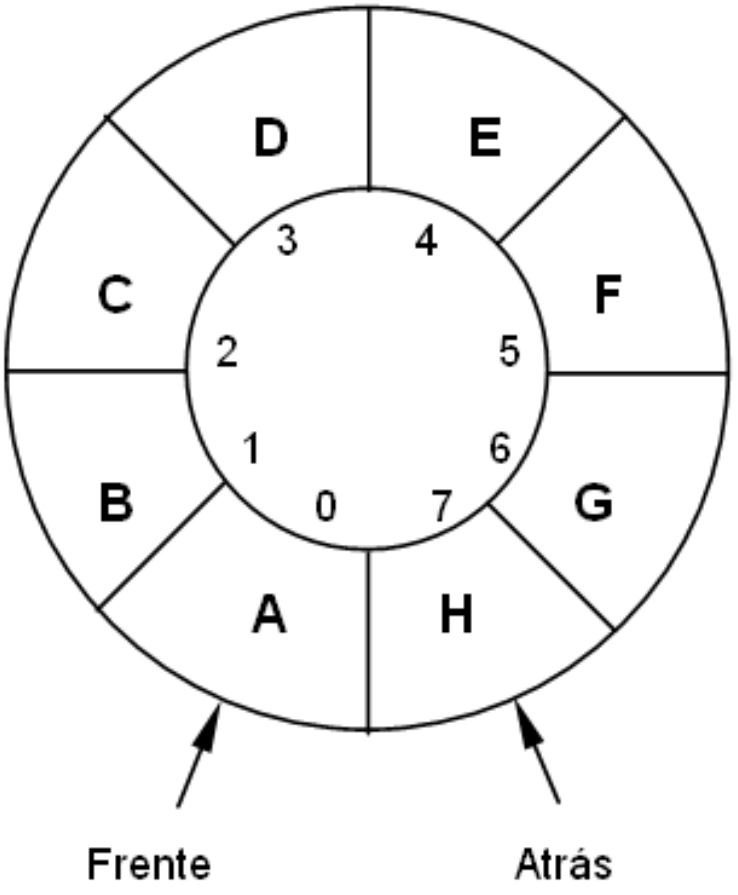


3.2 Colas



3.2 Colas

Añadir "H"

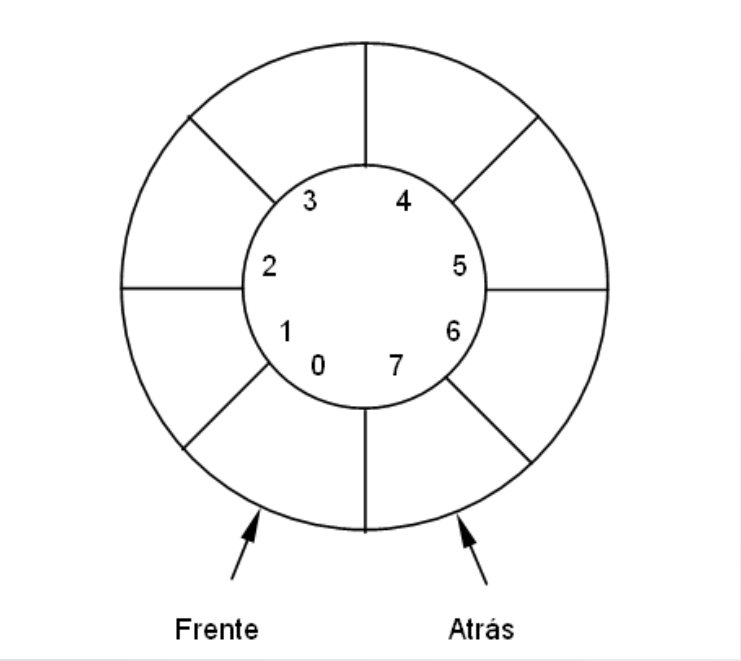


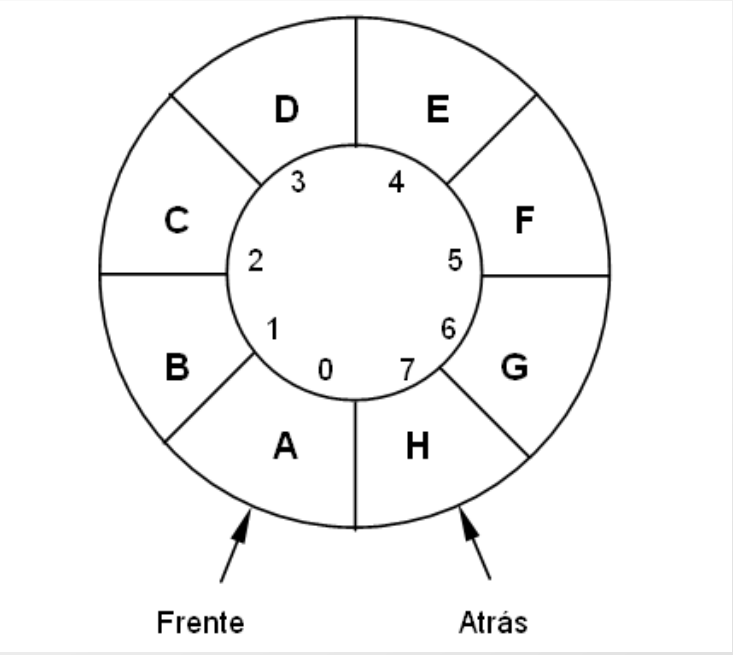
3.2 Colas

¿Puede entrar un dato más?

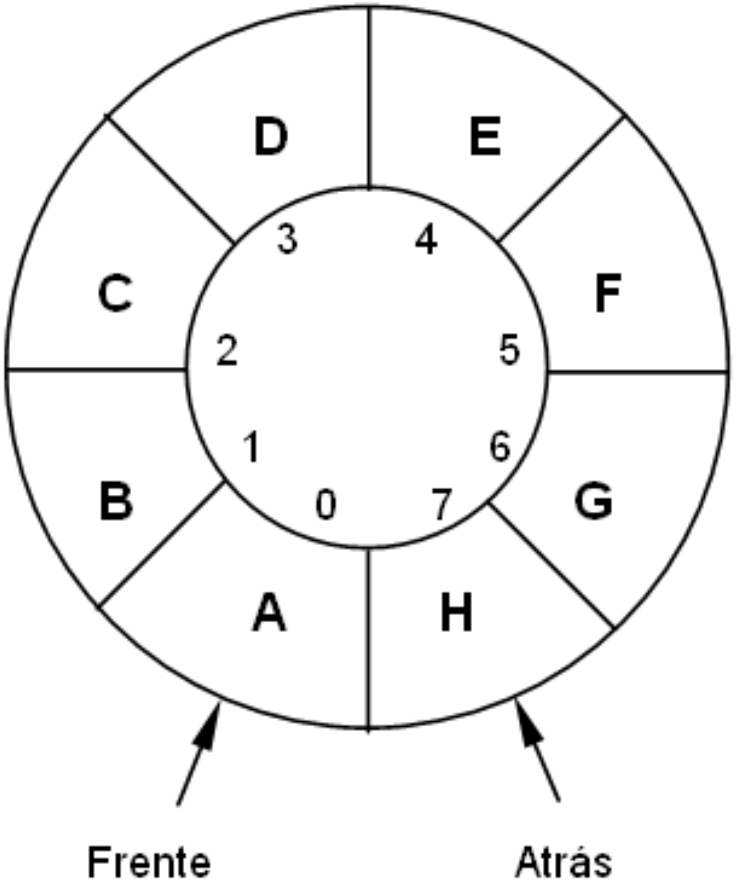
¿En que posiciones se encuentran *Frente* y *Atrás*?

¿En que posiciones estaban *Frente* y *Atrás* cuando la cola estaba vacía?

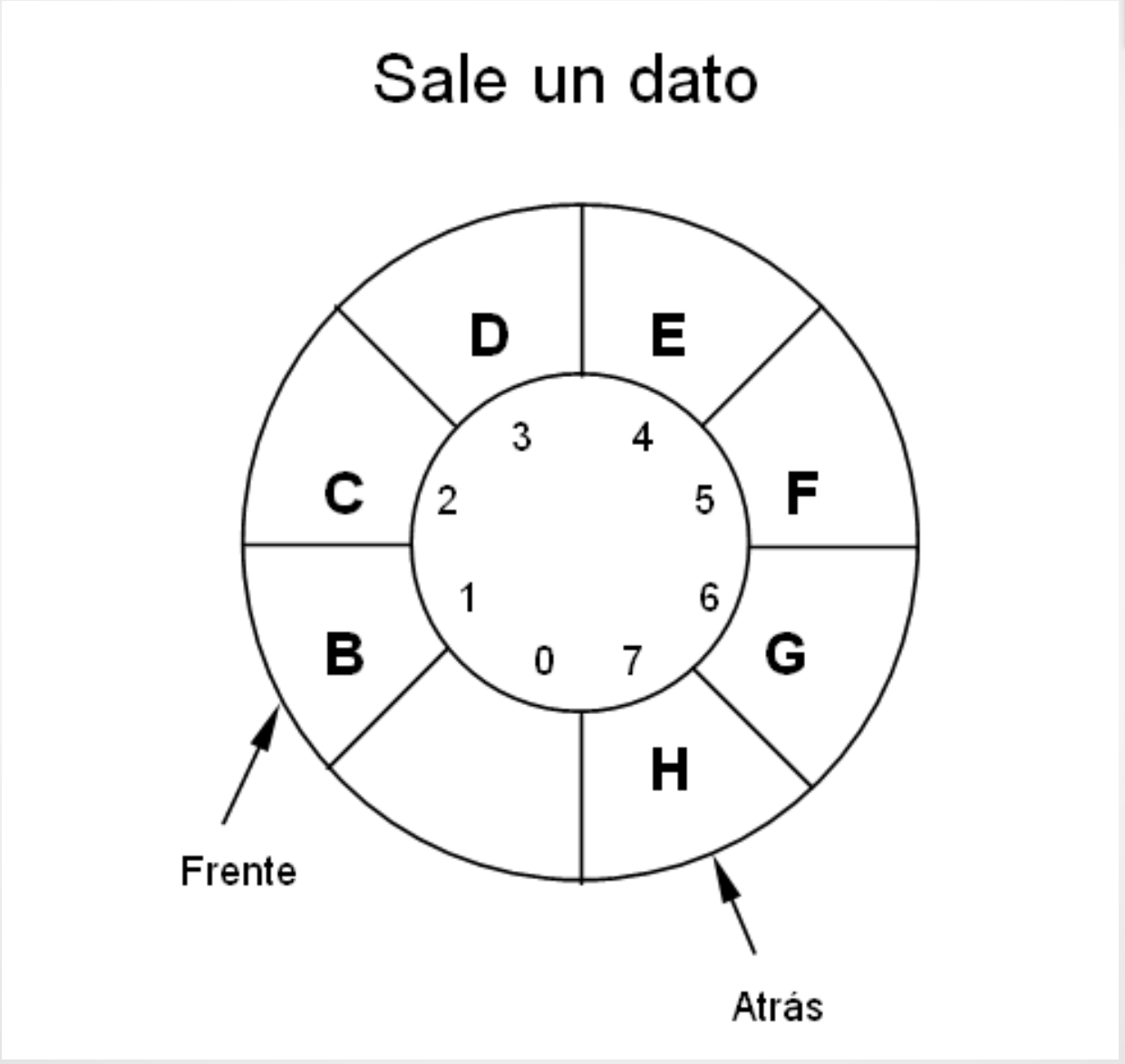




3.2 Colas

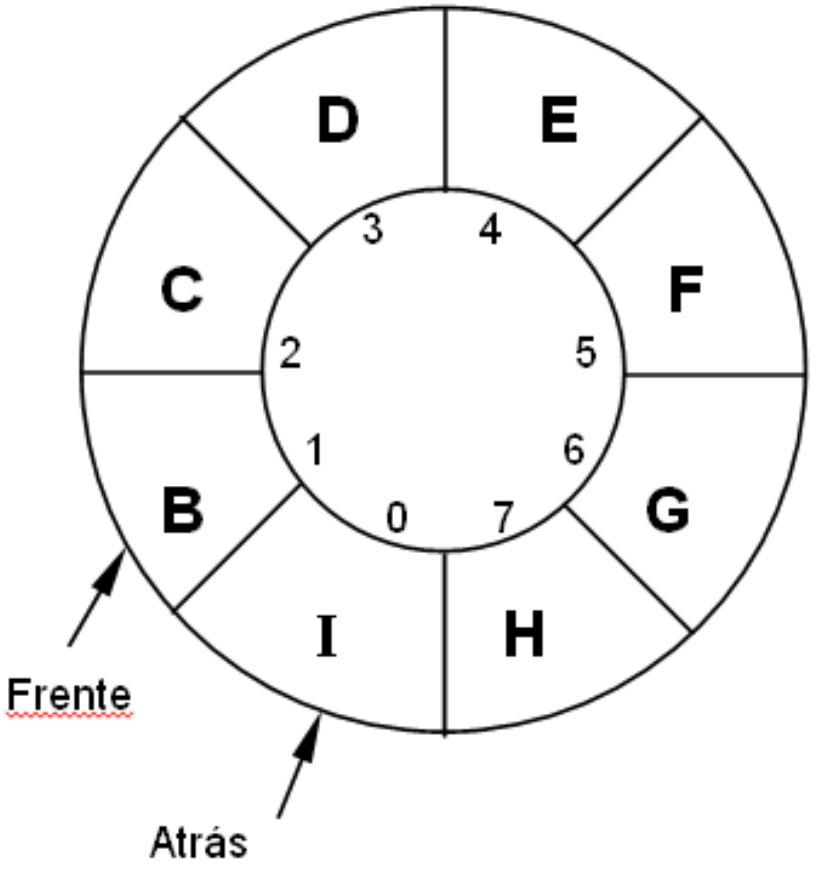


3.2 Colas



3.2 Colas

Se añade "I"

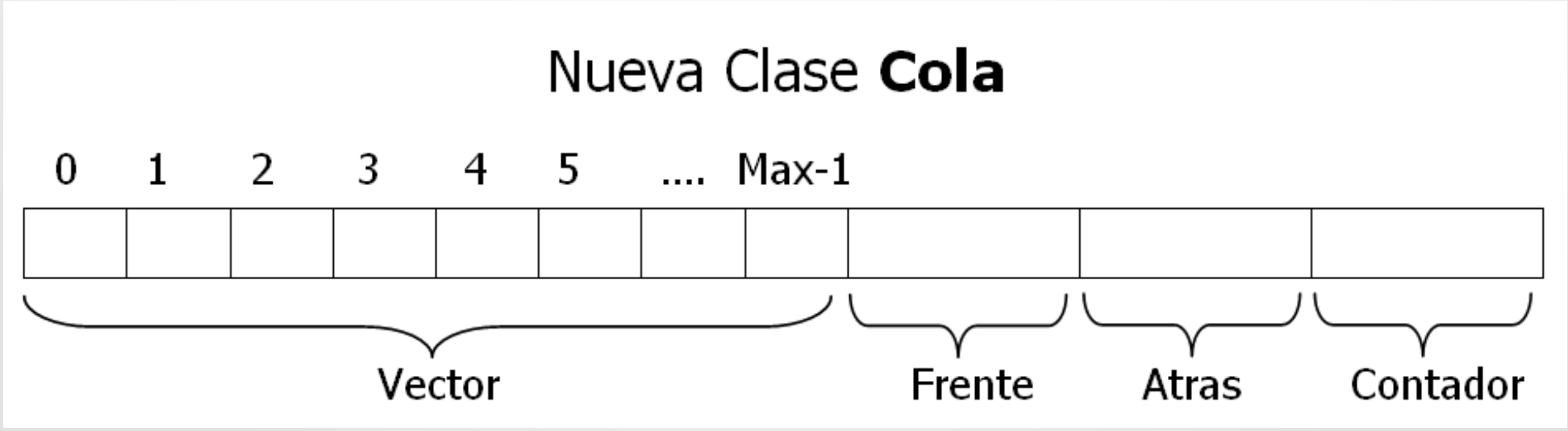


3.2 Colas

Retiremos 8 datos uno a uno, ¿en que posiciones quedarán *Frente* y *Atrás*?

La secuencia se controla bien con los apuntadores ….

Solo resta determinar los estados Vacia y Llena



Método	**Seudocódigo**
constructor	Atrás ← Max-1 Frente ← 0 Contador ← 0
entra	Mover atrás a la siguiente dirección *(significa sumar 1, excepto cuando atrás=Max-1, en tal caso Atrás←0)* vector[atrás] ← dato contador++
sale	Resulta vector[frente] Mover frente a la siguiente dirección *(significa sumar 1, excepto cuando frente=Max-1, en tal caso frente←0)* Contador--
vacia	Si Contador = 0 Resulta TRUE
llena	Si Contador=Max Resulta TRUE

3.2 Colas

// Cola Circular

```
#define Max 100
class Cola {
      private:
        int frente,atras,contador;
        char vector[Max];
      public:
        Cola();
        void entra(char);
        char sale();
        bool vacia();
        bool llena();
    };

Cola::Cola() {
   frente=0;atras=Max-1;contador=0;
};
```

3.2 Colas

```
void Cola::entra(char x) {
    if (atras==Max-1)
       atras=0;
    else
       atras++;
    contador++;
    vector[atras]=x;
}

char Cola::sale() {
    char temp=vector[frente];
    if (frente==Max-1)
       frente=0;
    else
       frente++;
    contador--;
    return temp;
}
```

3.2 Colas

```
bool Cola::vacia() {
     return (contador==0);
};

bool Cola::llena() {
     return (contador==Max);
};
```

3.3 Manejo de Memoria Dinámica.

El concepto de apuntador

```
#include <iostream>
using namespace std;
int main()
{
    int Dias[]={31,28,31,30,31,30,31,31,30,31,30,31};
    char NomMeses[][11] =
       {"Enero","Febrero","Marzo","Abril","Mayo","Junio","Julio",
        "Agosto","Septiembre","Octubre","Noviembre","Diciembre"};
    int mes;
    cout << "Escriba un número entre 1 y 12: ";cin >> mes;
    int pos = mes-1;
    cout << "Número de días del mes de " << NomMeses[pos] << ": "
                    << Dias[pos] << endl;
    return 0;
}
```

3.3 Manejo de Memoria Dinámica.

Apuntador es un concepto.
Lo demuestra este ejemplo:
Usamos un par de arreglos para saber el número de días y el nombre equivalentes a cada mes del año.

Cada elemento de los arreglos tiene una dirección única: valores desde 0 a 11.

```
#include <iostream>
using namespace std;
int main()
{
    int Dias[]={31,28,31,30,31,30,31,31,30,31,30,31};
    char NomMeses[][11]={"Enero","Febrero","Marzo","Abril","Mayo",
                         "Junio","Julio","Agosto","Septiembre","Octubre",
                         "Noviembre","Diciembre"};
    int mes;
    cout << "Escriba un número entre 1 y 12: ";cin >> mes;
    int pos = mes-1;
    cout << "Número de días del mes de " << NomMeses[pos] << ": "<< Dias[pos] << endl;
    system("pause
    return
}
```

La variable **"pos"** es un **apuntador** en el amplio sentido del concepto. Aquí se usa para señalar a una posición de los arreglos (conociendo el valor de **"pos"** obtenemos el número de días y el nombre de cierto mes (el que el usuario indicó).

3.3 Manejo de Memoria Dinámica.

Apuntadores a tipos simples

```
#include <iostream>
using namespace std;
int main()
{
    short int i,j;
    short int *p;
    cout << "Direccion asignada a la variable \"p\": " << &p << endl;
    cout << "Direccion asignada a la variable \"j\": " << &j << endl;
    cout << "Direccion asignada a la variable \"i\": " << &i << endl;
    p=&i;
    cout << "     (Instruccion ejecutada: p=&i)" << endl;
    cout << "Valor guardado en  p: " << p << endl;
    i=123;
    cout << "     (Instruccion ejecutada: i=123)" << endl;
    cout << "Valor guardado en  i: " << i << endl;
    cout << "Valor guardado en *p: " << *p << endl;
    return 0;
}
```

3.3 Manejo de Memoria Dinámica.

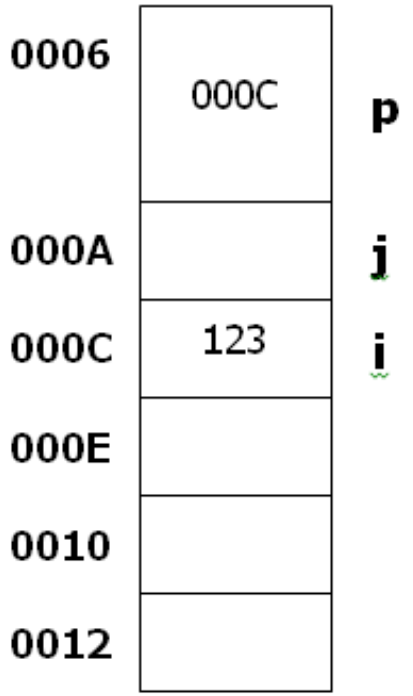
```
#i                    eam>
us                    std;
ir
{
                      ,j;
      rt int *p;
    c   t << "Direccion asignada a la variable \"p\": " << &p
    c   ut << "Direccion asignada a la variable \"j\": " << &j
    co  ut << "Direccion asignada a la variable \"i\": " << &i
    p=&i;
    cout << "                                tada: p=&i)"
    cout << "V                          << p << endl;
    i=123;
    cout << "      (Instruccion ejecutada: i=123)"
    cout << "Valor guardado en  i: " << i << endl;
    cout << "Valor guardado en *p: " << *p << endl;
    system("pause");
    return 0;
}
```

Se asigna a la variable **"p"** la dirección asignada a **"i"**

La variable **"p" NO es de tipo** entero **SINO LA DIRECCIÓN** de un lugar donde se guardan enteros. El asterisco es un operador de **INDIRECCIÓN**. Lo que se indica al compilador es que ***p** es de tipo **int**.

Se guarda en la variable **"i"** un dato cualquiera

NO significa mostrar el contenido de la variable **p** sino el contenido de la **dirección p,**



3.3 Manejo de Memoria Dinámica.

Apuntadores a tipos simples

```
#include <iostream>
using namespace std;
int main()
{
    int *ApuntDato;
    ApuntDato = new int;
    printf("Cada valor se guarda en la direccion %x (reemplaza al anterior)\n",ApuntDato);
    do{
       printf("Escribe un valor entero: ");
              cin >> *ApuntDato;
              *ApuntDato = (*ApuntDato)+1;
              cout << "El dato guardado es: " << *ApuntDato << endl;
       }while (true);
    delete ApuntDato;
    return 0;
}
```

3.3 Manejo de Memoria Dinámica.

Apuntadores a tipos simples

ApuntDato es una variable que guardará la dirección donde se encontrará a un entero.

new int reserva **durante la ejecución**, espacio para un entero.
La dirección correspondiente es asignada a la variable **ApuntDato**.
El Sistema Operativo se asegura que esa dirección no esté ocupada por otro dato.

```
#include <ios     am>
using namespa     std;
int main()
{
    int *ApuntDato;
    ApuntDato = new int;
    printf("Cada valor se guarda en la direccion %x (reemplaza al anterior)\n",ApuntDato);
    do{
       printf("Escribe un valor entero: ");
             cin >> *ApuntDato;
             *ApuntDato = (*ApuntDato)+1;
             cout << "El dato guardado es: " << *ApuntDato << endl;
       }while (true);
    delete ApuntDato;
}
```

Se lee un entero y se guarda en la "**dirección ApuntDato**"

Se incrementa en 1 el valor guardado.
Lo mismo que una variable.

Es indispensable liberar el espacio reservado.
Por cada **new** hay que ejecutar **delete** antes del cierre del ámbito donde se reservó el espacio.

Para que se familiarice con los mensajes de error del compilador, ejecute sin el **operador de indirección** (asterisco).

3.3 Manejo de Memoria Dinámica.

Apuntadores a tipos simples

```
#include <iostream>
using namespace std;
int main()
{
    int *ApuntDato;
    do{
       ApuntDato = new int;
       printf("Escribe un valor entero: ");
       cin >> *ApuntDato;
       *ApuntDato = (*ApuntDato)+1;
       cout << "El dato guardado es: " << *ApuntDato << endl;
       printf("Se encuentra en la direccion %x \n",ApuntDato);
       //delete ApuntDato;
    }while (true);
    return 0;
}
```

3.3 Manejo de Memoria Dinámica.

Apuntadores a tipos simples

```
#include <iostream>
using namespace std;
int main()
{
    int *ApuntDato;
    do{
       ApuntDato = new int;
       printf("Escribe un valor entero: ");
       cin >> *ApuntDato;
       *ApuntDato = (*ApuntDato)+1;
       cout << "El dato guardado es: " << *ApuntDato << endl;
       printf("Se encuentra en la direccion %x \n",ApuntDato);
       //delete ApuntDato;
    }while (true);
    return 0;
}
```

Cada vuelta del ciclo estamos solicitando una nueva dirección, si se ejecuta el **delete** del final del ciclo (quitando las diagonales), es posible que el Sistema Operativo nos de la misma dirección anterior ya que la usamos y la liberamos. Haga varias pruebas para que lo observe.

Si **no** liberamos el espacio, el Sistema Operativo siempre nos dará una nueva dirección y el espacio solicitado se quedará marcado como usado hasta que reinicie el Sistema Operativo.

Apuntadores a tipos simples dentro de estructuras

Operador -> (flecha)

```
#include <iostream>
using namespace std;
struct Persona { char nombre[41]; int edad; };
int main()
{
    Persona *ApuntPadre, *ApuntMadre, *ApuntHijo;
    ApuntPadre = new Persona;
    ApuntMadre = new Persona;
    ApuntHijo  = new Persona;
    cout << "Nombre del Padre  : ";gets(ApuntPadre->nombre);
    cout << "Nombre de la Madre: ";gets(ApuntMadre->nombre);
    cout << "Nombre del Hijo   : ";gets(ApuntHijo->nombre);
    ApuntPadre->edad = 45;
    ApuntMadre->edad = ApuntPadre->edad - 5;
    ApuntHijo->edad = ApuntMadre->edad - 20;
    cout << "Padre: " << ApuntPadre->nombre << " Edad: "
                      << ApuntPadre->edad << endl;
    cout << "Madre: " << ApuntMadre->nombre << " Edad: "
                      << ApuntMadre->edad << endl;
    cout << "Hijo : " << ApuntHijo->nombre << " Edad: "
                      << ApuntHijo->edad << endl;
    delete ApuntPadre; delete ApuntMadre; delete ApuntHijo;
    return 0;
}
```

Apuntadores a tipos simples dentro de estructuras

Operador -> (flecha)

```
#include <iostream>
using namespace std;
struct Persona { char nombre[41]; int edad; };
int main()
{
    Persona *ApuntPadre, *ApuntMadre, *ApuntHijo;
    ApuntPadre = new Persona;
    ApuntMadre = new Persona;
    ApuntHijo  = new Persona;
    cout << "Nombre del Padre  : ";gets(ApuntPadre->nombre);
    cout << "Nombre de la Madre: ";gets(ApuntMadre->nombre);
    cout << "Nombre del Hijo   : ";gets(ApuntHijo->nombre);
    ApuntPadre->edad = 45;
    ApuntMadre->edad = ApuntPadre->edad - 5;
    ApuntHijo->edad = ApuntMadre->edad - 20;
    cout << "Padre: " << ApuntPadre->nombre <<
                         << ApuntPadre->edad <<
    cout << "Madre: " << ApuntMadre->nombre <<
                         << ApuntMadre->edad << endl;
    cout << "Hijo : " << ApuntHijo->nombre << " Edad: "
                         << ApuntHijo->edad << endl;
    delete ApuntPadre; delete ApuntMadre; delete ApuntHijo;
    return 0;
}
```

ApuntPadre, **ApuntMadre**, **ApuntHijo** no son los lugares donde guardar las estructuras, son las variables que contendrán las direcciones de las estructuras.

Equivale a (*ApuntPadre).nombre**.** Sin embargo, **–>** es más simple y es un símbolo fácil de recordar.

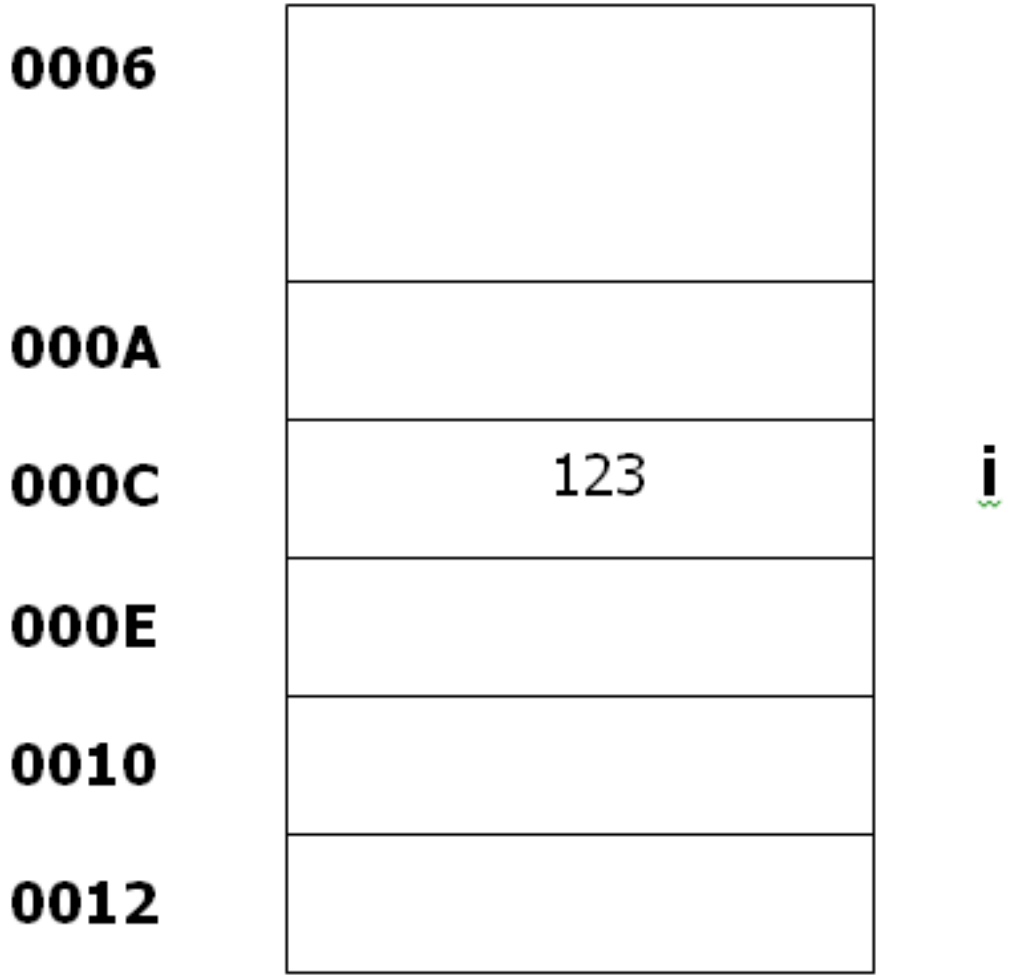
3.3 Listas Encadenadas

Vectores	**Lista Lineal** (Los datos se organizan en una sola ruta)
	Lista Secuencial (los datos adyacentes en forma lógica se encuentran también físicamente juntos)
	Estructura estática (espacio reservado durante la compilación). Los vectores y matrices pueden causar **desperdicio de memoria** si no se usan en forma completa, ya que su espacio se libera hasta que termina el ámbito donde se declaró.
Listas Encadenadas	**Lista Lineal**
	Lista NO SECUENCIAL
	Estructura dinámica (sin necesidad de que se cierre el ámbito donde se reservó cierto espacio, puede liberarse esa memoria ocupada de manera que pueda ser utilizada de nuevo por el mismo u otro proceso, posteriormente).

3.3 Listas Encadenadas

Manejo Estático referencia **directa** a los datos.

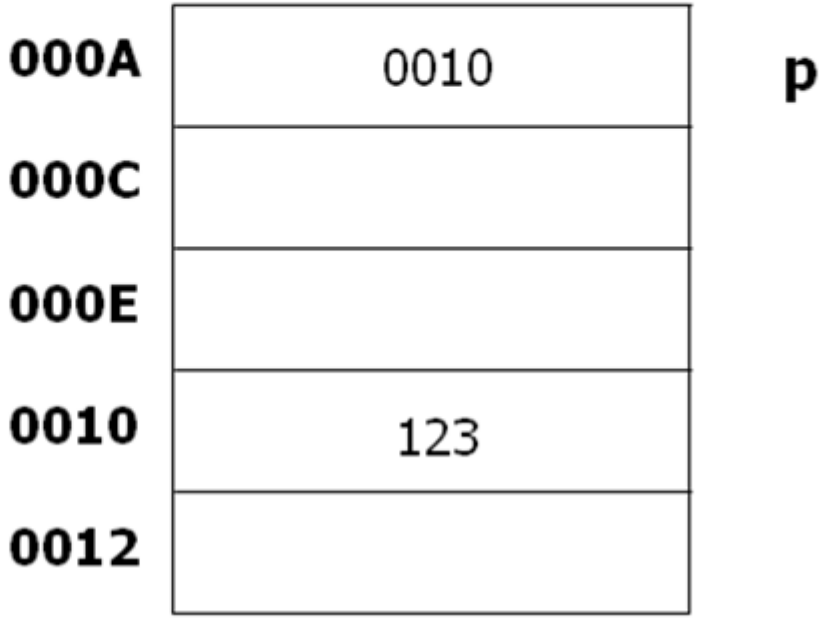
```
short int i;
i=123;
```



3.3 Listas Encadenadas

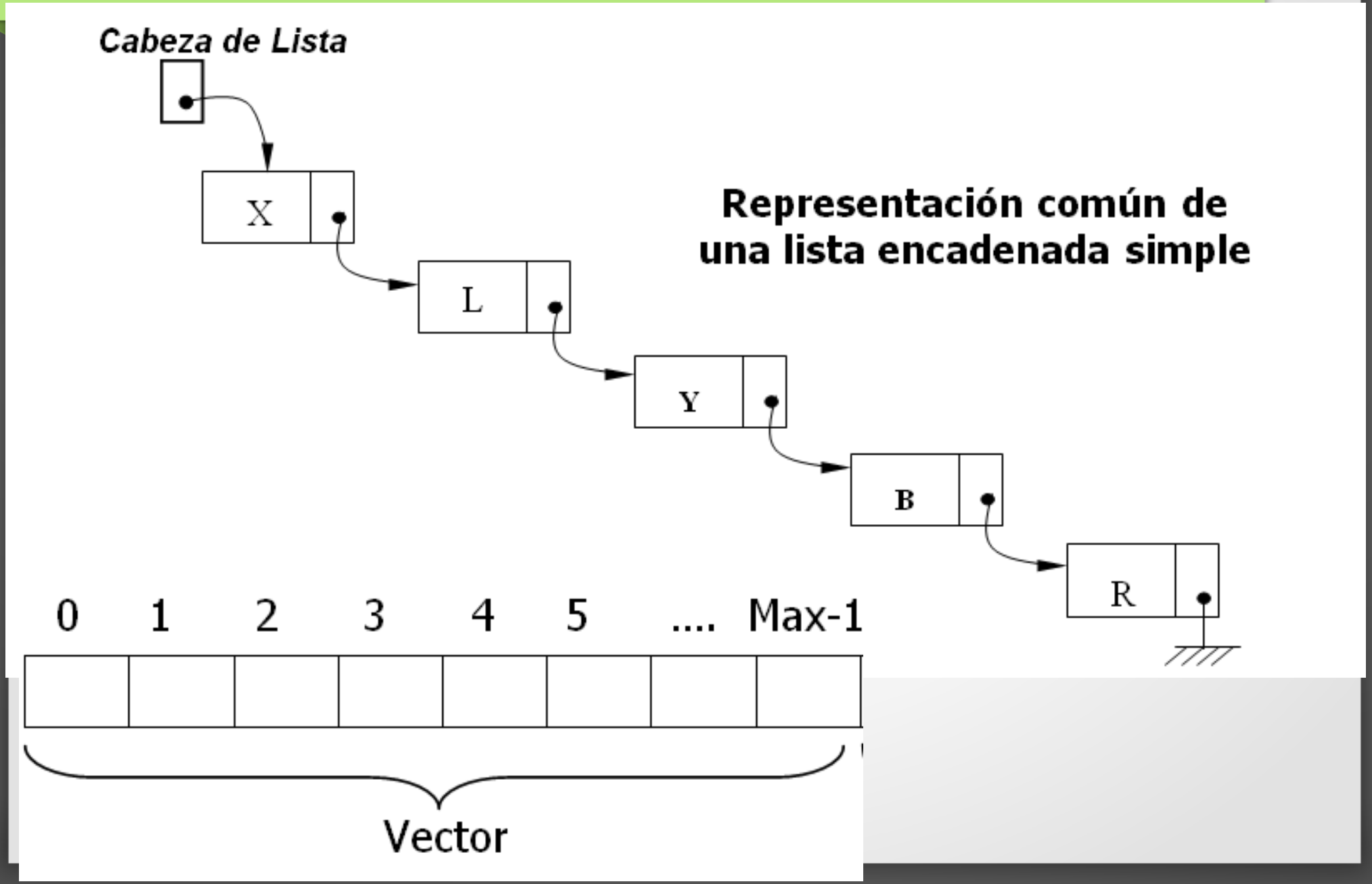
Manejo Dinámico de la memoria referencia **indirecta** a los datos.

```
short int *p;
p = new int;
*p = 123;

.....
.....

delete p;
```



Este concepto permite construir una lista encadenada como la que se muestra en la página siguiente.

3.3 Listas Encadenadas



3.3 Listas Encadenadas

Para construir una Lista Encadenada se requiere contar con las siguientes condiciones

1. **Se deben definir elementos independientes que conformarán la lista** (se les llama NODOS)**.**

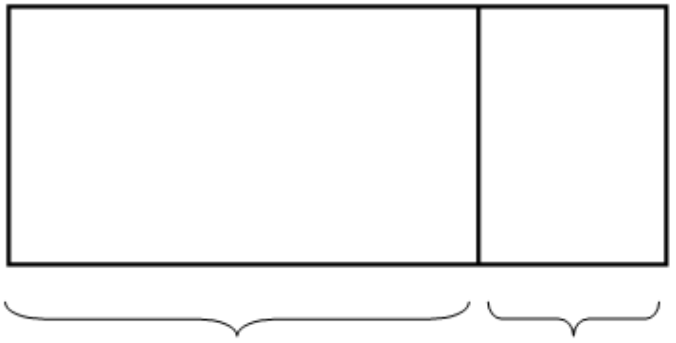
   - **NODO** = lugar en la memoria, reservado durante la ejecución, para guardar un dato.
   - **Apuntador** = Variable que contiene la dirección de un nodo.
   - Los Nodos se deben reservar uno a uno.

   Los elementos de un vector no son independientes ya que no pueden desprenderse físicamente del vector, al ocupar direcciones consecutivas de la memoria.

3.3 Listas Encadenadas

2. Los NODOS deben ser estructuras con 2 datos miembros (para el caso de las listas simples, para las dobles se requieren 3 datos miembros)**.**

- La primera parte es el lugar que corresponde al dato en sí.
- La segunda contendrá la dirección de OTRO NODO (el siguiente en la lista).



3.3 Listas Encadenadas

3. El último nodo de la lista tiene una característica peculiar.

- El último nodo al no tener un nodo después de él, deberá guardar en el campo del *nodo siguiente* un valor que indique el final de la lista.

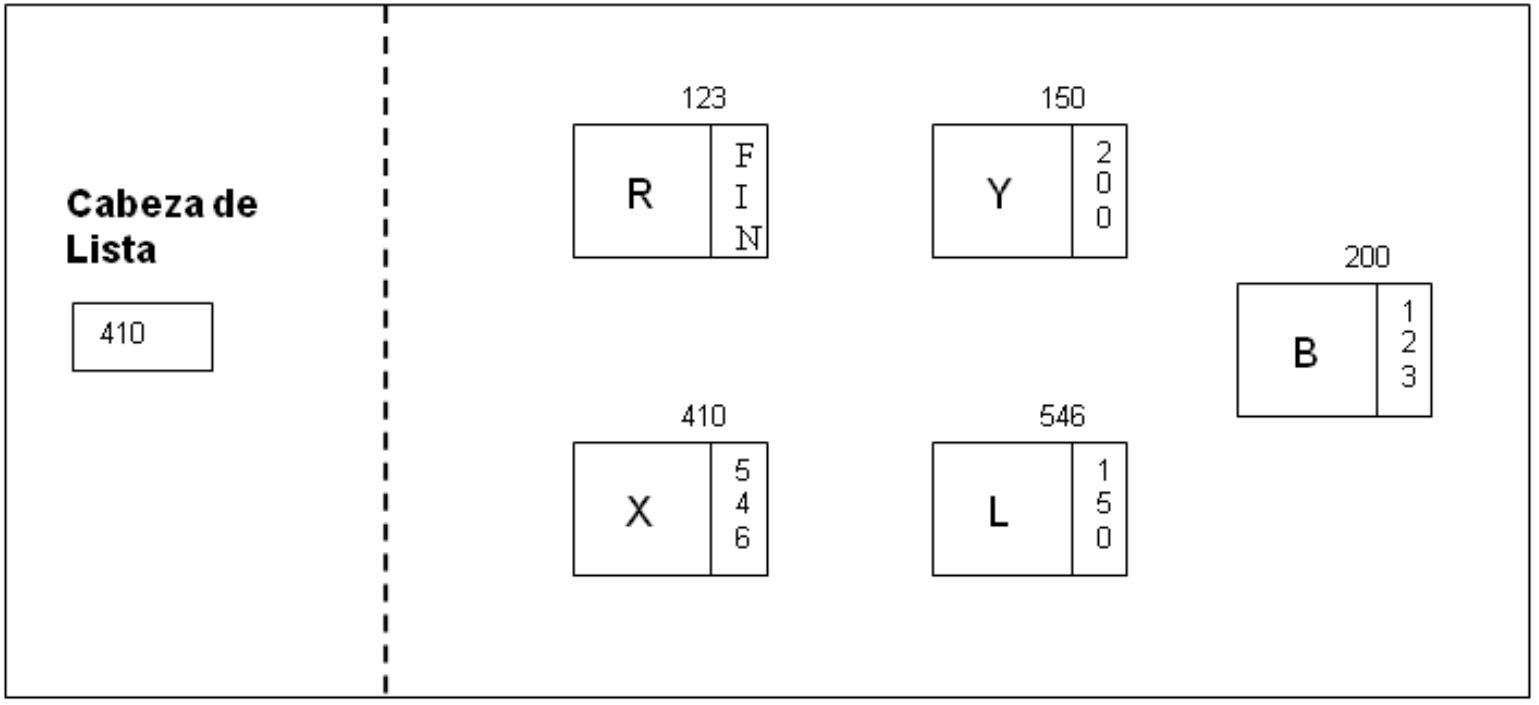
4. El primer nodo de la lista también tiene una peculiaridad.

- La dirección del 2º nodo se conserva en el primer nodo, la del último en el penúltimo, etc.
  - La dirección del primer nodo debe conservarse en una **VARIABLE** tipo apuntador llamada **CABEZA DE LISTA**.

3.3 Listas Encadenadas

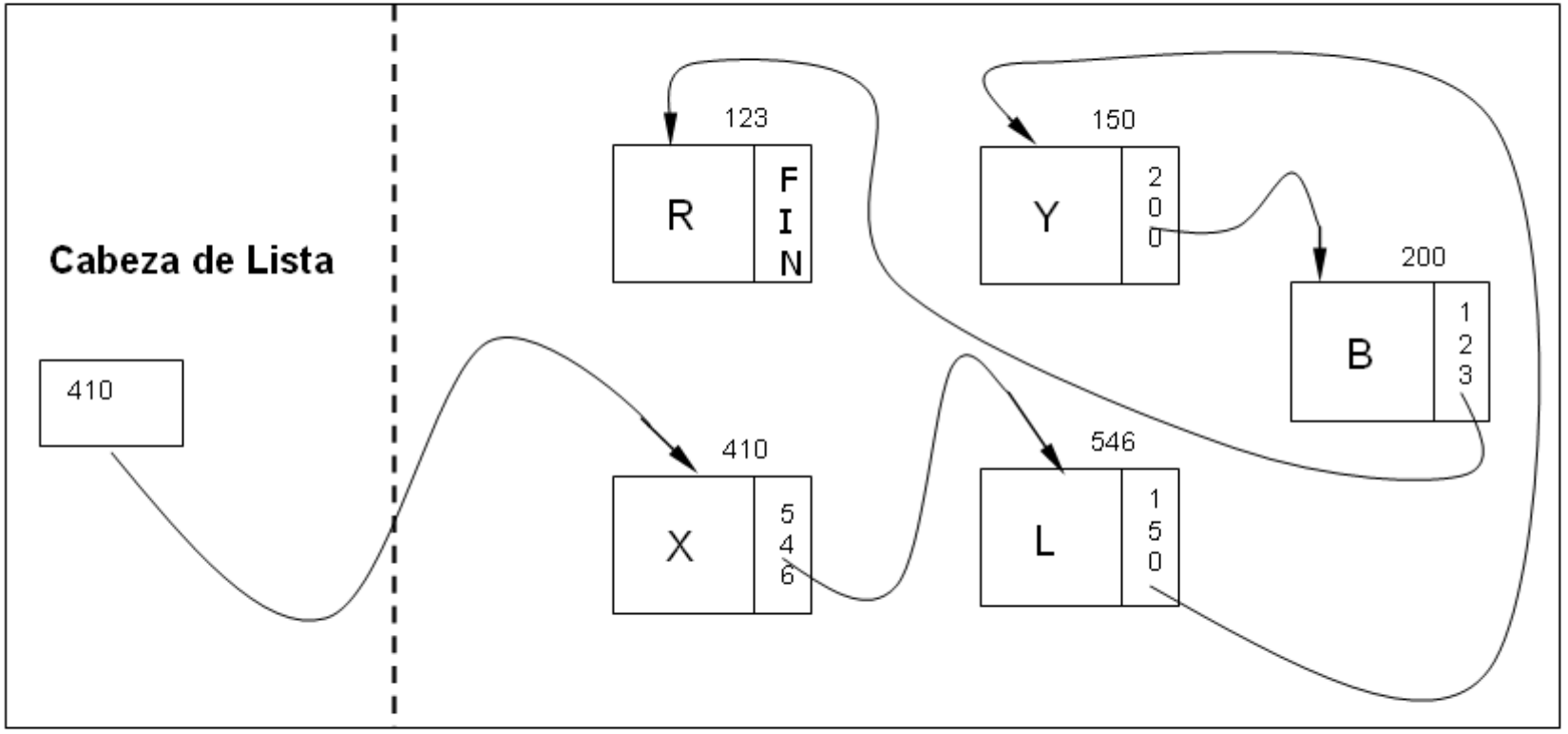
Otra representación de una L.E.

(Los valores numéricos decimales encima de cada nodo representan una hipotética dirección de memoria)



3.3 Listas Encadenadas

Una representación más



3.3 Listas Encadenadas

Algoritmos para la Operación de una Lista Encadenada Simple

Las listas encadenadas siempre inician vacías (**Cabeza** ← **NULL**).

Insertar

- Se **solicita** una **dirección de memoria disponible.**
- Se guarda el nuevo dato en el campo **"info"** del **nuevo nodo**.
- De acuerdo al uso que demos a la lista, se aplicará uno de los 3 siguientes casos.

a) **Si el nuevo nodo debe quedar al inicio de la lista:**
   - Guardar en el campo sucesor del nuevo nodo la dirección que se encuentra en la variable **cabeza de lista**.
   - Guardar en la variable **cabeza de lista** la dirección del nuevo nodo.

b) **Si el nuevo nodo debe ser el último de la lista:**
   - Guardar **NULL** en el campo sucesor del nuevo nodo.
   - Localizar la dirección del último nodo.
   - Guardar en el campo sucesor del último nodo la dirección del nuevo nodo.

c) **Si el nuevo nodo debe quedar entre dos nodos ya existentes:**
   - Asumamos que en **temp1** y **temp2** se encuentran las direcciones de los nodos que serán adyacentes al nuevo nodo.
   - Guardar en el campo sucesor del nuevo nodo el valor de **temp2**.
   - Guardar en el campo sucesor del nodo **temp1** la dirección del nuevo nodo.

3.3 Listas Encadenadas

Eliminar

- Cada aplicación determinará cual nodo se eliminará.
- Puede presentarse uno de los tres casos siguientes.

a) **Si es necesario borrar el primer nodo:**
   - Guardar en **cabeza de lista** la dirección que se encuentra en el campo sucesor del primer nodo.

b) **Si se requiere eliminar el último nodo:**
   - Guardar **NULL** en el campo sucesor del penúltimo nodo.

c) **Para eliminar un nodo que se encuentra entre otros dos:**
   - Siendo **temp1** y **temp2** las direcciones de los nodos adyacentes, hay que guardar en el campo sucesor del nodo **temp1** la dirección **temp2**.

- Una vez eliminado uno nodo (***desprendido de la lista***), hay que liberar ese espacio para que pueda ser aprovechado por el Sistema Operativos.

3.3 Listas Encadenadas

Recorridos

- Algunos algoritmos de los que hemos analizado requieren que se haga un recorrido de la lista.

  Recorrer la lista = visitar todos o algunos de los nodos.

Algoritmo iterativo para imprimir todos los nodos de una lista

1.- Asignar a una variable, podemos llamarla **temp**, la dirección del primer nodo.
2.- Si la variable contiene **FIN**, termina el algoritmo.
3.- Ir al Nodo correspondiente a **temp** e imprimir su información.
4.- Asignar a **temp**, la dirección del nodo siguiente.
5.- Ir al paso 2.

3.3 Listas Encadenadas

Primera Aplicación de Listas Encadenadas

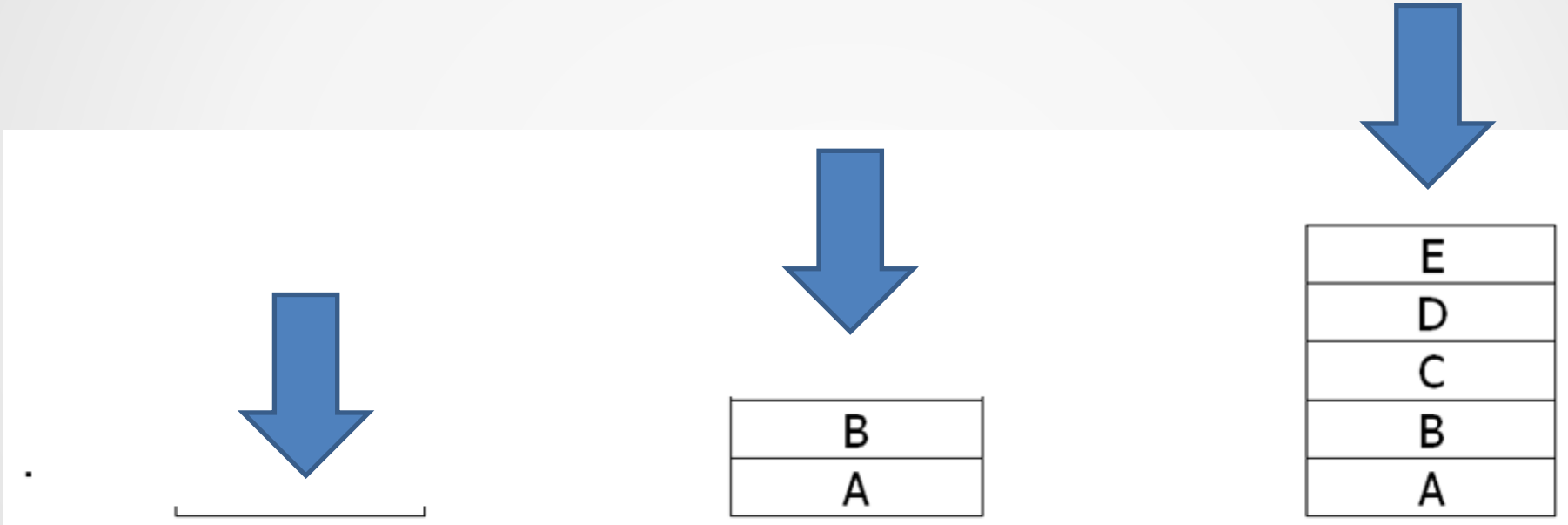
Pilas

Son listas en la que los datos se retiran por el mismo extremo por donde se acumulan.

Se implementa usando una lista encadenada, haciendo que:

- Cada nodo nuevo quede como el primero de la lista (apuntado por la variable *CabezaDeLista*).
- Cuando se deba retirar un dato, se elimine también por *CabezaDeLista*.

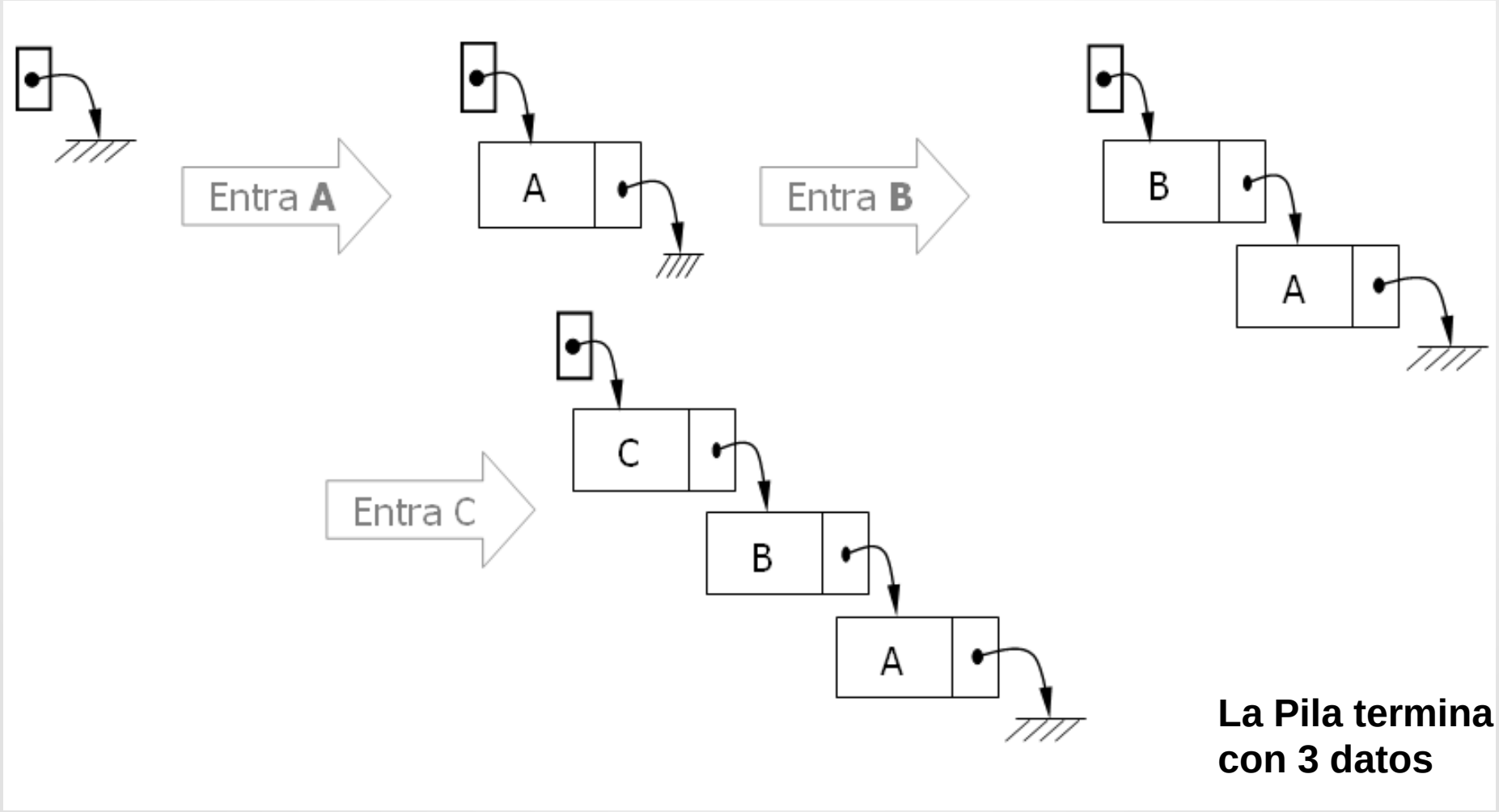
3.1 Pilas



DIVERSOS ESTADOS DE UNA PILA

3.3 Listas Encadenadas

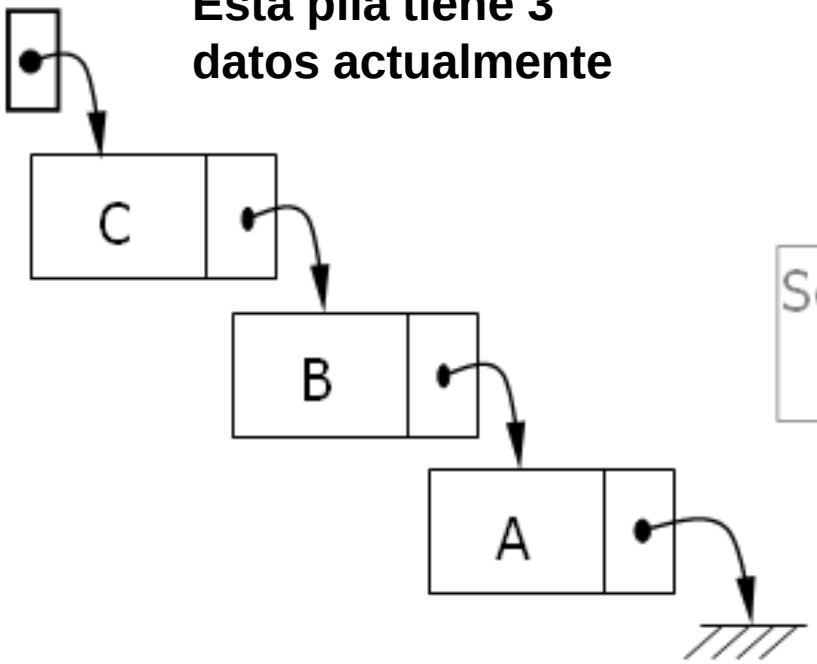
La Pila empieza vacía



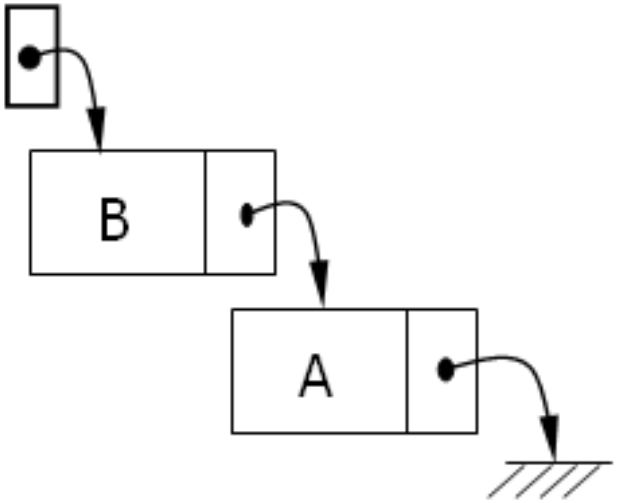
La Pila termina con 3 datos

3.3 Listas Encadenadas

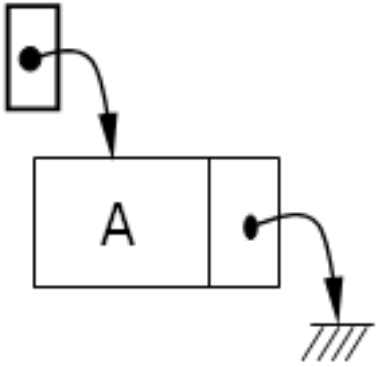
Esta pila tiene 3 datos actualmente



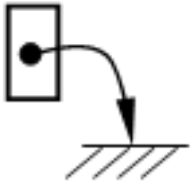
Se retira un dato (C)



Se retira otro dato (B)



Se retirar otro dato (A)



La Pila termina vacía

```
// Ejemplo simple de uso de una lista Encadenada
// Los datos se guardan y retiran del inicio de la lista

#include <iostream>
using namespace std;
struct Nodo {
      char info;
      Nodo *sucesor;
};
int main(){
    Nodo *cabeza;
    cabeza=NULL;
    int opcion;
    do
    {
        system("cls");
        cout << "-----------\n";
        cout << "1. Guardar\n";
        cout << "2. Retirar\n";
        cout << "0. Terminar\n";
        cout << "Opcion: ";cin>>opcion;
        cout << "-----------\n";
```

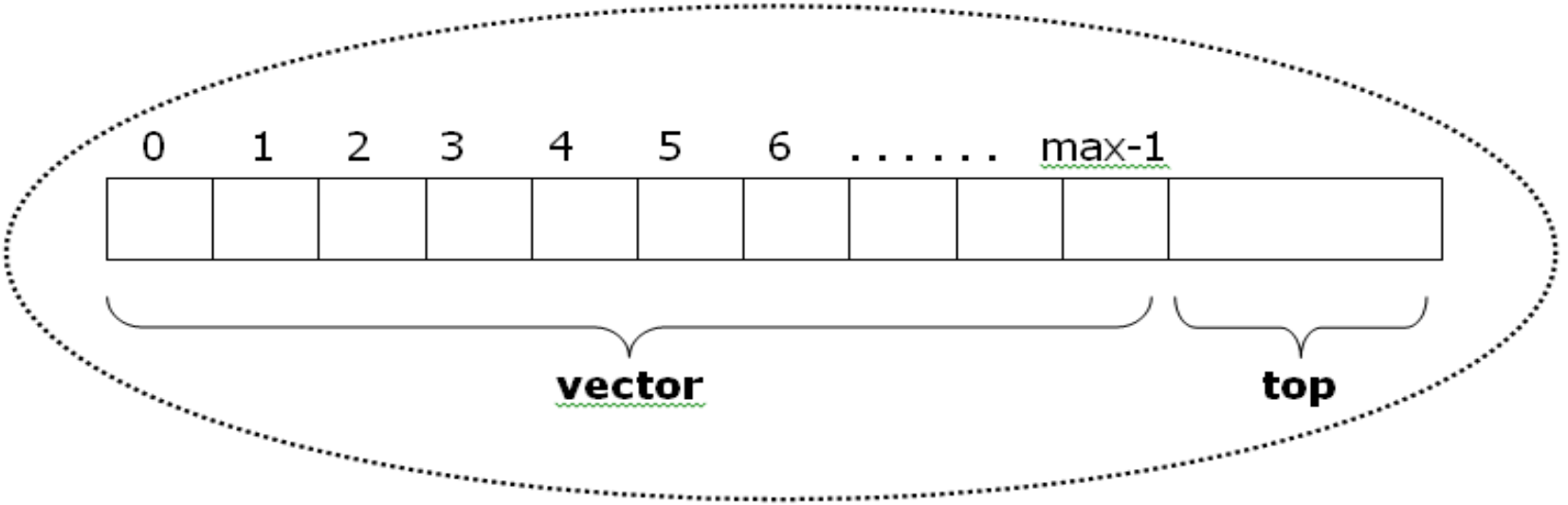
```
    switch(opcion){
            case 1:
                 Nodo *temp;
                 temp=new Nodo;
                 cout << "Caracter a guardar: ";
                 cin >> temp->info;
                 temp->sucesor = cabeza;
                 cabeza = temp;
                 break;
            case 2:
                 if (cabeza==NULL){
                        cout << "no hay datos" << endl;
                        system("pause");}
                 else{
                        Nodo *temp;  temp=cabeza;
                        char result=cabeza->info;
                        cabeza=cabeza->sucesor;
                        delete temp;
                        cout << "dato que salio: "
                         << result << endl;
                        system("pause");}
                 break;
    }
}
while (opcion!=0);
```

3.3 Listas Encadenadas

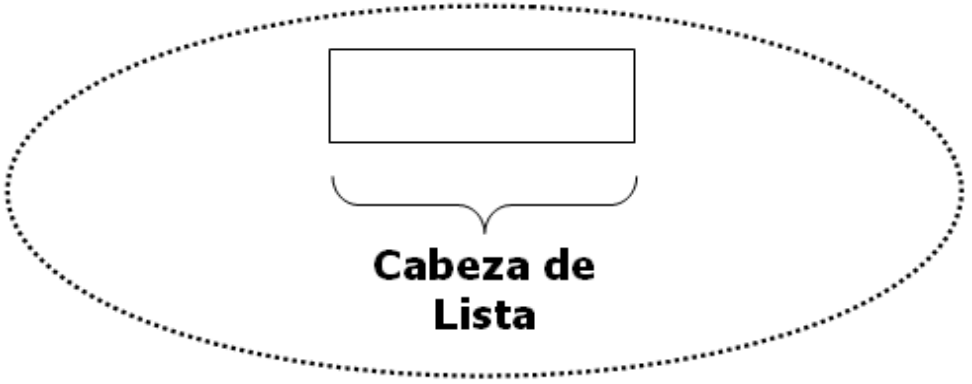
```
    // Se deben eliminar los nodos que no se han liberado

    Nodo *tempnodo;
    while (cabeza!=NULL) {
        tempnodo=cabeza;
        cout << "Borrando el nodo que contiene un caracter "
               << tempnodo->info << endl;
        cabeza = cabeza->sucesor;
        delete tempnodo;
        }
     return 0;
}
```

3.3 Listas Encadenadas (**Clase Pila**)

Clase Pila con un vector



Clase Pila con una Lista Encadenada



3.3 Listas Encadenadas

```
class Pila {
        private:
            struct Nodo
                {
                char info;
                Nodo *sucesor;
                };
            Nodo *cabeza;
        public:
            Pila();
            void push(char x);
            char pop();
            char ver();
            bool vacia();
            bool llena();
            ~Pila();
};
```

3.3 Listas Encadenadas

```
Pila::Pila(){
    cabeza=NULL;
};

void Pila::push(char x) {
    Nodo *temp;
    temp=new Nodo;
    temp->info = x;
    temp->sucesor = cabeza;
    cabeza = temp;
};
```

3.3 Listas Encadenadas

```
char Pila::pop() {
    Nodo *temp=cabeza;
    char result=cabeza->info;
    cabeza=cabeza->sucesor;
    delete temp;
    return result;
    };

bool Pila::vacia() {
    return (cabeza==NULL);
};

bool Pila::llena() {
    return (false);
};
```

3.3 Listas Encadenadas

```
char Pila::ver(){
      return cabeza->info;
};

Pila::~Pila() {
    cout << "Ejecutando el destructor... ";
    system("pause");
    Nodo *temp;
    while (cabeza!=NULL) {
        cout << "Borrando un nodo... ";
        system("pause");
        temp=cabeza;
        cabeza=cabeza->sucesor;
        delete temp;
    };
};
```

Ejercicio

Pruebe todos los programas que hicimos en C++ con pilas, reemplazando solo la clase, por la que recién hemos creado.

Las aplicaciones no deben sufrir ningún cambio (aparte del *#include*), y su funcionamiento debe ser igual, con la ventaja de que internamente el consumo de espacio en la memoria será solo el necesario.

3.3 Listas Encadenadas

Segunda Aplicación de Listas Encadenadas

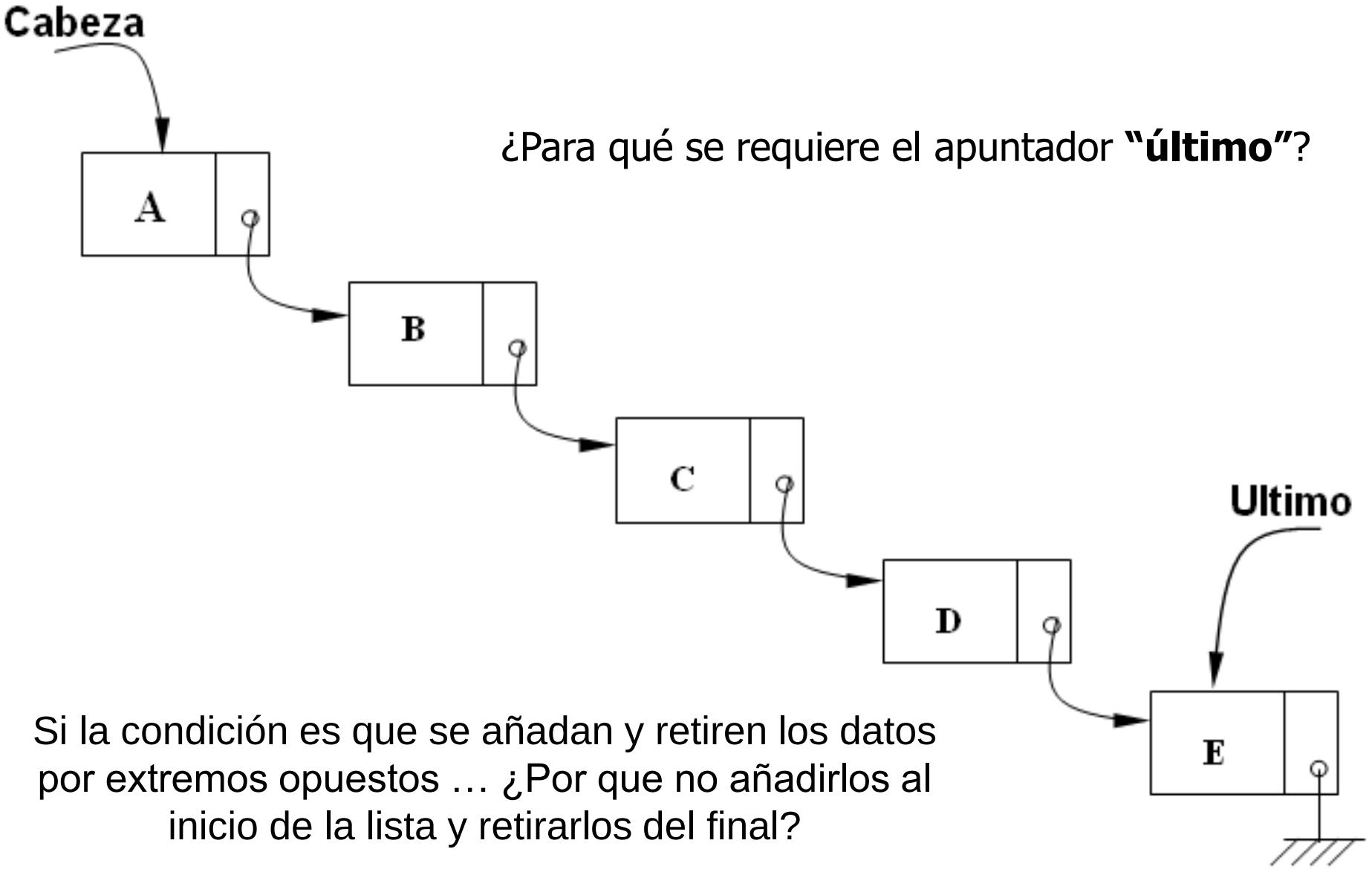
Colas

Cola = Lista cuyos elementos se acumulan y retiran por extremos opuestos.

Al entrar nuevos datos, se guardan al final. Se retiran del inicio.

Se incorpora el uso de un apuntador al final de la Lista (lo llamaremos **ultimo**).

3.3 Listas Encadenadas

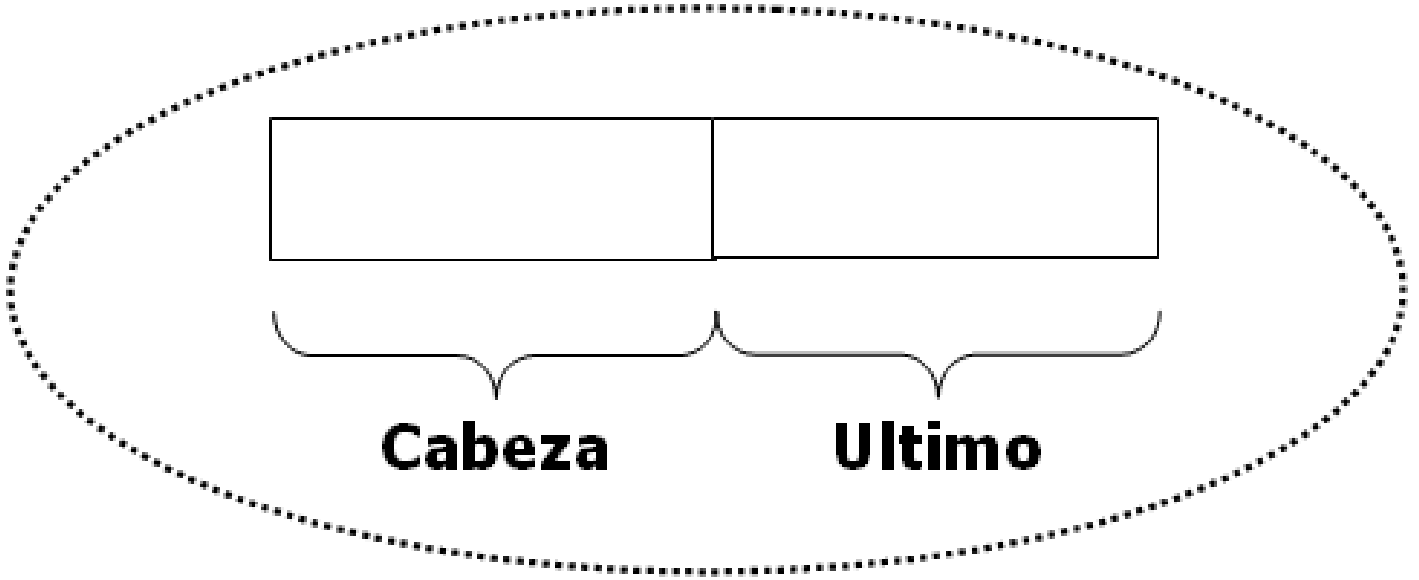


¿Para qué se requiere el apuntador **"último"**?

Si la condición es que se añadan y retiren los datos por extremos opuestos … ¿Por que no añadirlos al inicio de la lista y retirarlos del final?

3.3 Listas Encadenadas

Clase Cola usando una Lista Encadenada



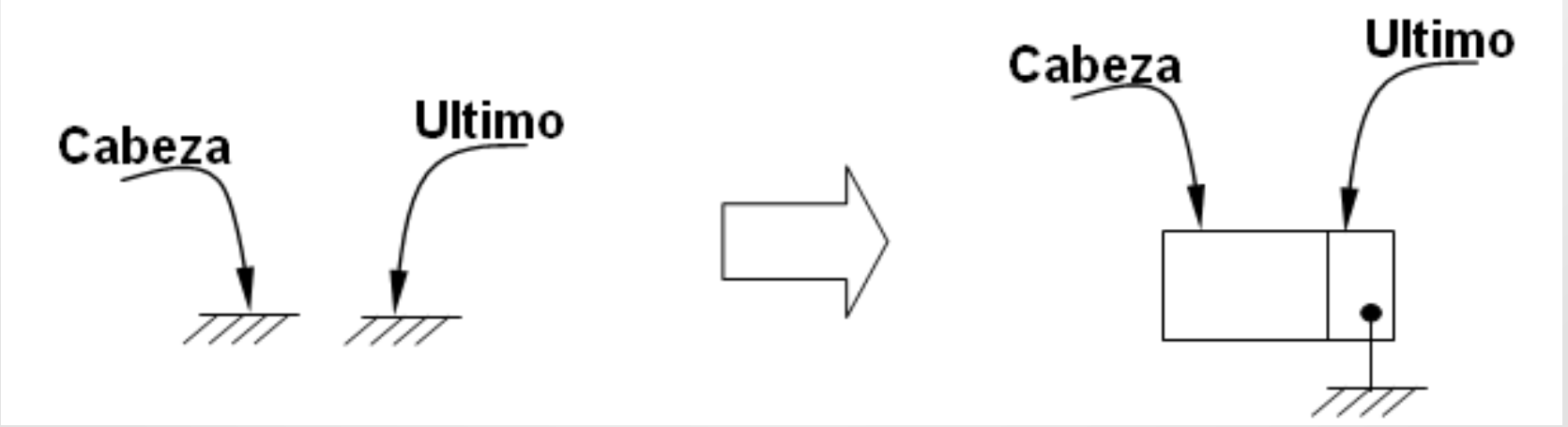
3.3 Listas Encadenadas

Constructor()

cabeza y **ultimo** inician en **NULL**.

entra()

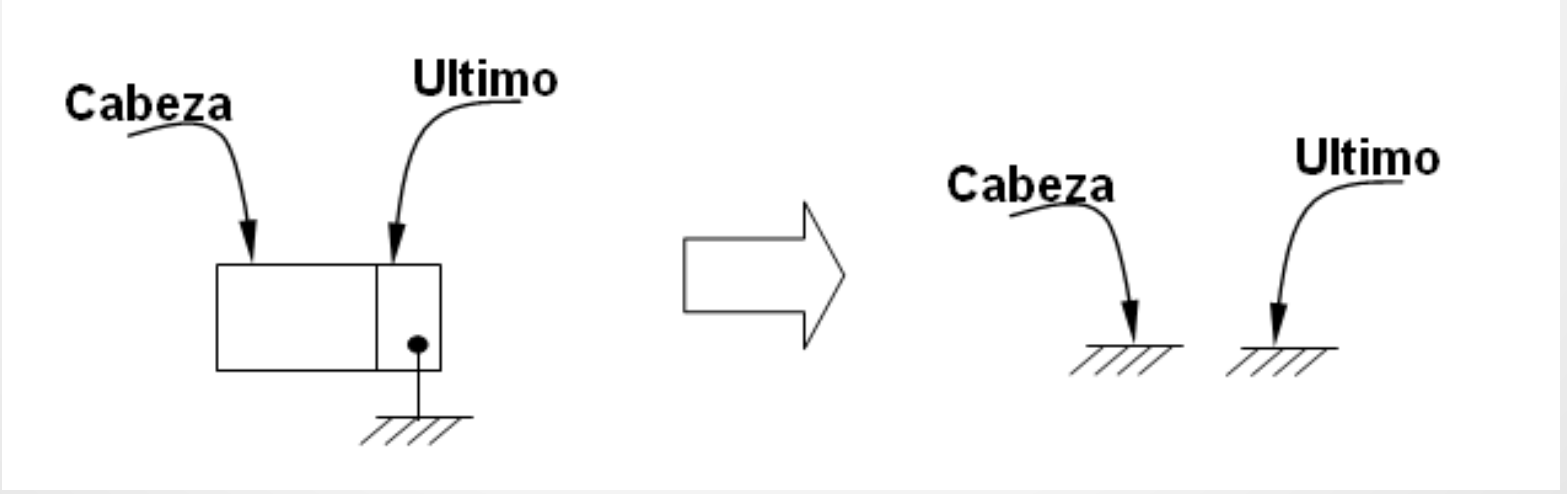
- Guardar el **nuevo dato** en el campo **info** de un nuevo nodo.
- Guardar **NULL** en el campo **sucesor** del nuevo nodo.
- Guardar en el **sucesor** del ultimo nodo la dirección del **nuevo nodo**.
- Guardar en la variable **ultimo** la dirección del nuevo nodo.
  - **Caso Especial:** Cuando la lista está vacía, la dirección del nuevo nodo se debe guardar en la variable **cabeza**.



3.3 Listas Encadenadas

sale()

- Recuperar el dato contenido en el primer nodo.
- Guardar en la variable **cabeza** la dirección que está en el campo **sucesor** del primer nodo.
  - **Caso especial:** Si solo resta un nodo en la lista, a la variable **ultimo** deberá asignarse **NULL**.



- Liberar el nodo que ya no se usa.

3.3 Listas Encadenadas

vacia()

Si cabeza de lista contiene el valor **NULL.**

llena()

Si no hay memoria suficiente para otro nodo.

3.3 Listas Encadenadas

```
class Cola{
        private:
            struct Nodo
               {
               char info;
               Nodo *sucesor;
               };
            Nodo *cabeza,*ultimo;
        public:
            Cola();
            void entra(char x);
            char sale();
            bool vacia();
            bool llena();
            ~Cola();
};
```

3.3 Listas Encadenadas

```
Cola::Cola(){
    cabeza=NULL;
    ultimo=NULL;
};

void Cola::entra(char x) {
    Nodo *temp;
    temp=new Nodo;
    temp->info = x;
    temp->sucesor = NULL;
    if (cabeza==NULL)
       cabeza=temp;
    else
       ultimo->sucesor=temp;
    ultimo=temp;
};
```

3.3 Listas Encadenadas

```
char Cola::sale(){
    Nodo *temp=cabeza;
    char result=cabeza->info;
    cabeza=cabeza->sucesor;
    if (cabeza==NULL)
       ultimo=NULL;
    delete temp;
    return result;
};

bool Cola::vacia() {
    return (cabeza==NULL);
};
```

3.3 Listas Encadenadas

```
bool Cola::llena() {
    return (false);   // asumimos que siempre
                      // habrá espacio en el Heap
};

Cola::~Cola(){
    Nodo *temp;
    while (cabeza!=NULL){
        temp=cabeza;
        cabeza=cabeza->sucesor;
        delete temp;
    };
};
```

3.3 Listas Encadenadas

Ejercicio:

Modificar la clase Cola para que se resuelva sin necesidad del apuntador *último*, la solución correcta es usar ese apuntador, sin embargo, se hace esta solicitud como ejercicio para reforzar el conocimiento respecto al uso del apuntador *último*.

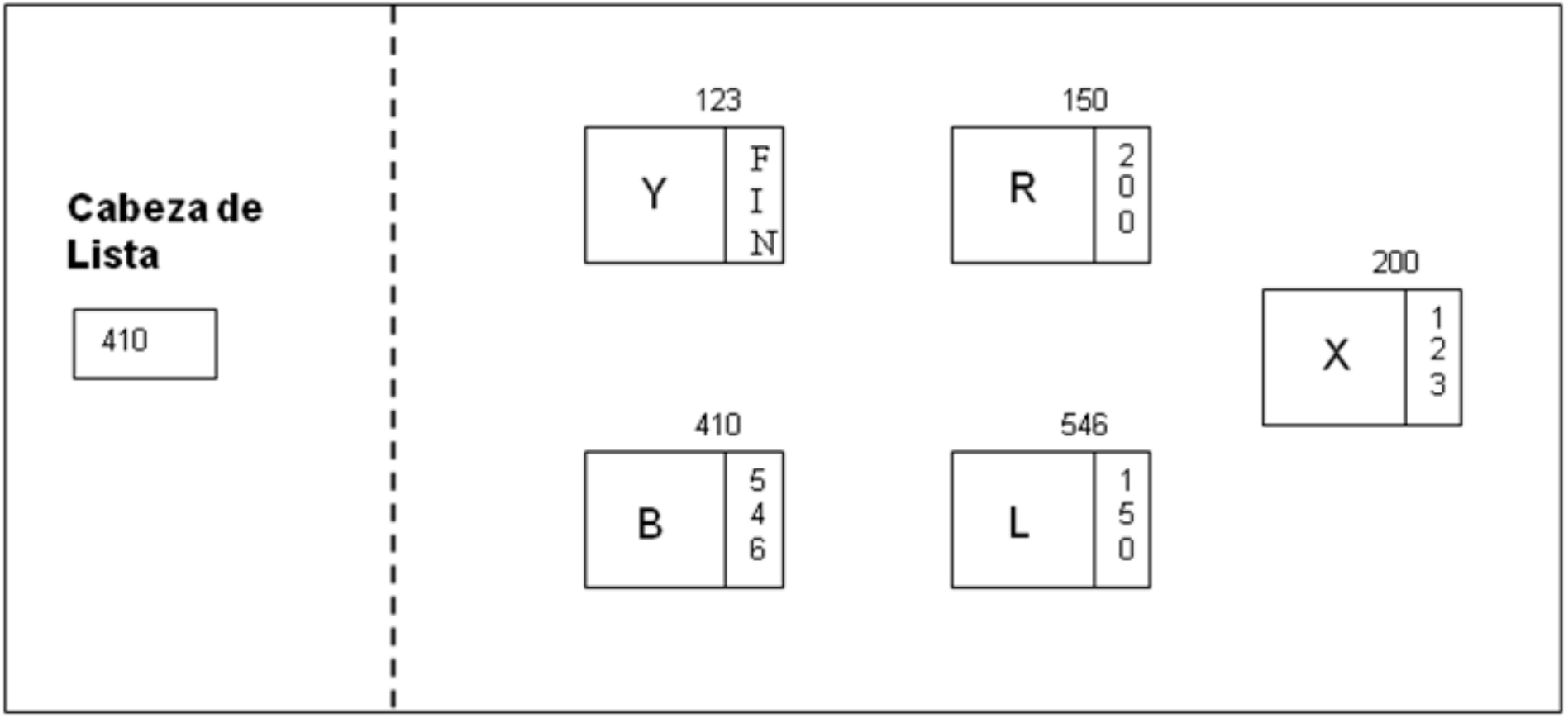
3.3 Listas Encadenadas

Tercera Aplicación de Listas Encadenadas

Lista Ordenada
y
Colas de Prioridades

- Cuando usamos una L. E. para **pilas** o **colas**:
  1. El orden en que los nodos se colocan en la lista, no depende del valor del dato sino de la secuencia de entrada a la lista.
  2. El proceso que se sigue durante las altas y las bajas, siempre es el mismo.
- Al añadir datos a una **lista que debe permanecer ordenada** el nodo para el nuevo dato puede quedar en cualquier parte de la lista.

3.3 Listas Encadenadas



Con la figura de arriba (una Lista Ordenada) razone que procesos hay que seguir para dar de alta una D, luego una Z, luego una T y finalmente una A y la lista siga permaneciendo ordenada.

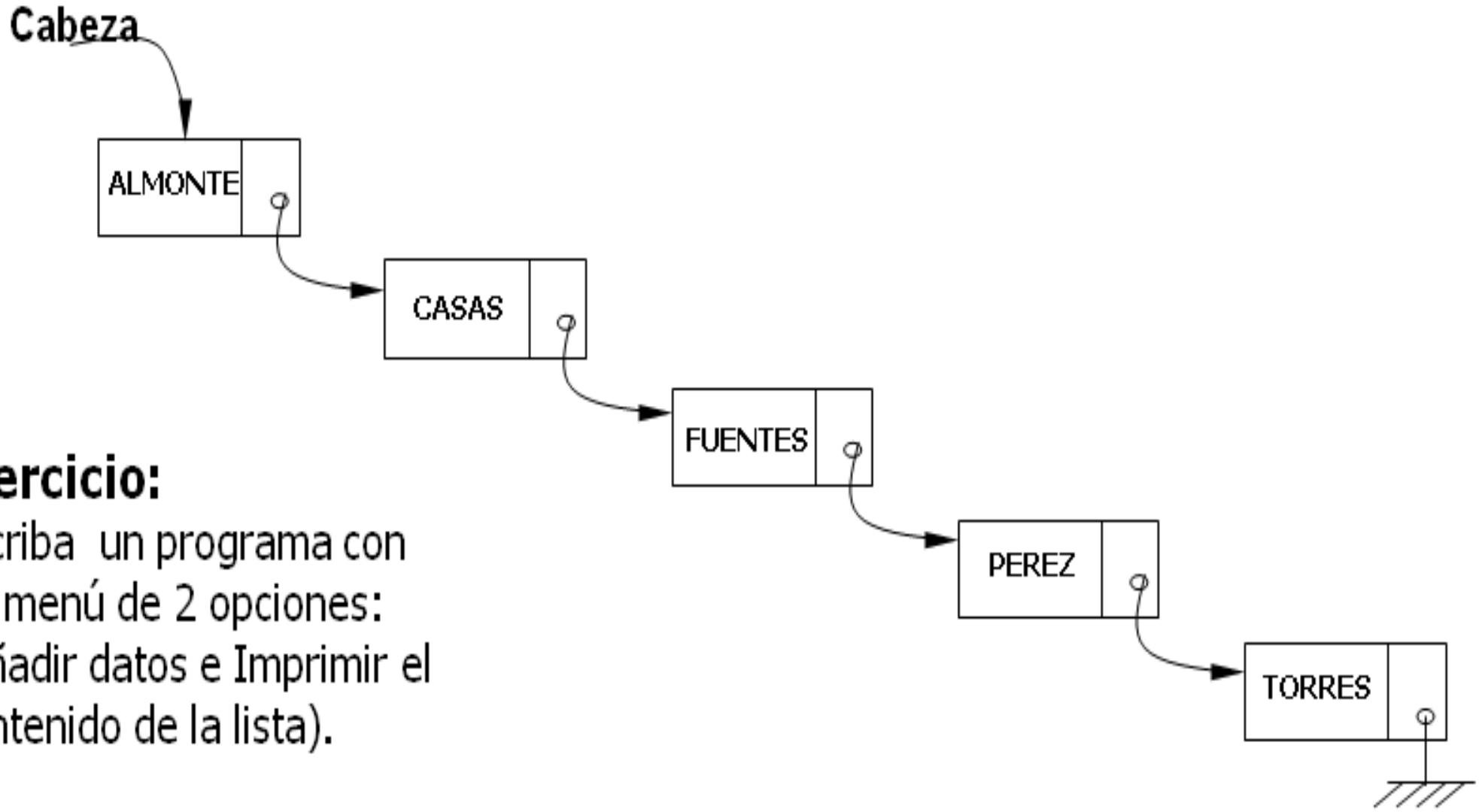
3.3 Listas Encadenadas

Procedimiento para Añadir

1. Solicitar un nuevo nodo y guardar el dato en el campo **info**.
2. Determinar en que parte de la lista debe quedar. Algoritmo:
   - Recorrer la lista hasta encontrar un nodo con un dato **mayor o igual** que el que se va a añadir.
   - Use dos variables: **temp2** para guardar la dirección del nodo localizado (dato mayor) y **temp1** para el nodo anterior.
   - El nuevo nodo quedará entre ambos nodos.
   - Es posible que **temp2** valga **NULL**.
   - Es posible que no haya un nodo **anterior** y por lo tanto **temp1** también valga **NULL**.
3. Enlazar el nuevo nodo.
   - Si **temp1** contiene **NULL**, el nuevo nodo deberá quedar al inicio de la lista.
   - No importa que **temp2** valga **NULL**, basta aplicar el procedimiento para enlazar un nodo "entre" dos nodos y se obtiene el resultado esperado.

3.3 Listas Encadenadas

Tome como base la siguiente lista para probar el algoritmo anterior.



Ejercicio:

Escriba un programa con un menú de 2 opciones: (Añadir datos e Imprimir el contenido de la lista).

3.3 Listas Encadenadas

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
struct Nodo {
    string info;
    Nodo *sucesor;
    };
int main(){
  char opcion;
  Nodo *cabeza,*temp,*nuevo,*nodo1,*nodo2;
  cabeza=NULL;
  do {
       cout << "\nLista Ordenada \n";
       cout << "1.- Altas\n" << "2.- Imprimir\n";
       cout << "0.- Terminar\n\n";
       cout << "Opcion: ";opcion = getch(); cout << opcion;
       cout << endl;
       switch (opcion){
```

3.3 Listas Encadenadas

```
case '0':
        while (cabeza!=NULL) {
          temp=cabeza;
          cabeza=cabeza->sucesor;
          delete temp;
          };
        break;
```

3.3 Listas Encadenadas

```
case '1':
          nuevo=new Nodo;
          cout<<"Nuevo Dato: ";getline(cin,nuevo->info);
          nodo1=NULL;
          nodo2=cabeza;
          while ((nodo2!=NULL)and(nuevo->info>nodo2->info)) {
                nodo1=nodo2;
                nodo2=nodo2->sucesor;
                };
          if (nodo1==NULL)
             cabeza=nuevo;
          else
             nodo1->sucesor=nuevo;
          nuevo->sucesor=nodo2;
          break;
```

3.3 Listas Encadenadas

```
        case '2':
                    temp=cabeza;
                    while (temp!=NULL) {
                          cout << temp->info << endl;
                          temp=temp->sucesor;
                          }
                          system("pause");
                    break;
          }
      }
    while (opcion!='0');
    return 0;
}
```

Eliminación de Datos de Lista Ordenada

Debe considerarse la posibilidad de eliminar cualquier nodo de la lista ordenada, sin importar en donde se encuentra.

- Se usan 2 variables temporales tipo apuntador: **nodoant** y **temp**.
- **Temp** señala siempre al nodo que se "visita" (empezando en el primer nodo) y **nodoant** al nodo anterior (empezando en **NULL**).
- Recorrido.
  - Cuando temp llegue a **NULL** o el nodo **temp** contenga un dato mayor al que buscamos eliminar, significa que la búsqueda terminó y el dato no está en la lista.
  - Por el contrario, si encontramos en un nodo el dato a eliminar, hay que aplicar el algoritmo siguiente:
    - Si **nodoant**=**NULL**, aplicar el procedimiento para eliminar el **primer nodo**.
    - Si **nodoant!**=**NULL** aplicar el algoritmo para eliminar un nodo entre otros dos (si se trata de eliminar el último nodo incluso).

Este razonamiento asume que el dato que se pretende eliminar es un dato único en la lista...en caso de no serlo, este proceso de eliminación elimina el primero que se encuentre (estarán consecutivos).

Ejercicio:

Complemente el programa para que se puedan efectuar bajas.

3.3 Listas Encadenadas

Añada al programa 2 opciones más:

4.- Imprimir la lista en el orden inverso
(programa recursivo).
5.- Imprimir la lista en el orden inverso
(programa iterativo).

Para la opción 5, utilice la clase Pila (***tipo plantilla***) que se incluye en las diapositivas siguientes.

3.3 Listas Encadenadas

Enseguida se incluye el archivo de cabecera de la clase Pila usando una plantilla (*template*) para usar en la aplicación de *lista ordenada*, una pila para imprimir a la inversa usando un programa iterativo (se guardan las direcciones de los nodos en una pila mientras se recorre la lista y al final se imprime la información de los nodos mientras se retiran de la pila las direcciones).

3.3 Listas Encadenadas

```
template <class Tipo>   // Tipo es un identificador
                        // correspondiente a cualquier tipo
class Pila  {
       private:
            struct Nodo {
               Tipo info;
               Nodo *sucesor;
               };
            Nodo *cabeza;
       public:
            Pila();
            void push(Tipo x);
            Tipo pop();
            Tipo ver();
            bool vacia();
            bool llena();
            ~Pila();
    };
```

3.3 Listas Encadenadas

```
template <class Tipo>
Pila<Tipo>::Pila()
     {
     cabeza=NULL;
     };

template <class Tipo>
void Pila<Tipo>::push(Tipo x)
     {
     Nodo *temp;
     temp=new Nodo;
     temp->info = x;
     temp->sucesor = cabeza;
     cabeza = temp;
     };
```

3.3 Listas Encadenadas

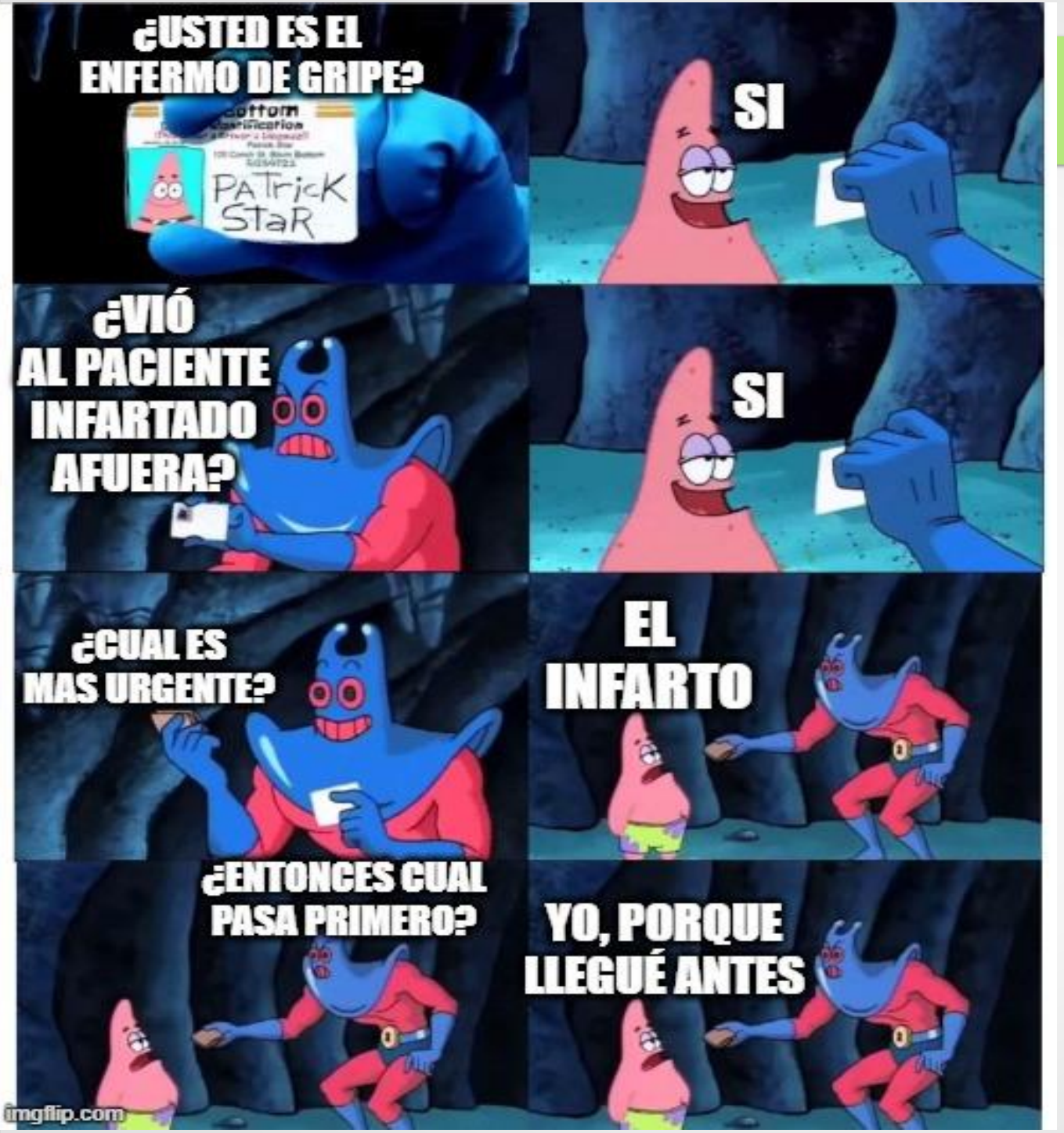
```
template <class Tipo>
Tipo Pila<Tipo>::pop()
     {
     Nodo *temp=cabeza;
     Tipo result=cabeza->info;
     cabeza=cabeza->sucesor;
     delete temp;
     return result;
     };

template <class Tipo>
Tipo Pila<Tipo>::ver(){
      return cabeza->info;
      };
```

3.3 Listas Encadenadas

```
template <class Tipo>
bool Pila<Tipo>::vacia()
     {
     return (cabeza==NULL);
     };

template <class Tipo>
bool Pila<Tipo>::llena()
     {
     return (false);
     };
```

3.3 Listas Encadenadas

```
template <class Tipo>
Pila<Tipo>::~Pila()
    {
    Nodo *temp;
    while (cabeza!=NULL)
        {
        temp=cabeza;
        cabeza=cabeza->sucesor;
        delete temp;
        };
    };
```



Patricio tendría razón si se usara una cola para atender a los pacientes en un hospital.

Afortunadamente Patricio está equivocado.

3.3 Listas Encadenadas (colas de prioridades)

Estado de una **lista** de pacientes **esperando** que se desocupe una sala de operaciones en cierto hospital:

Paciente	Edad	Tipo de Cirugía
Hugo Patito	7	Corregir pie plano
Pato Donald	31	Cuerdas vocales
Paco Patito	7	Corregir pie plano
Luis Patito	7	Corregir pie plano
Cerdo Porky	29	Lipoescultura

¿En que lugar se debería colocar en espera a ....

Tío Rico	50	Transplante de riñón

3.3 Listas Encadenadas (colas de prioridades)

Como la cirugía de Tío Rico es **más importante** que las cinco que están en espera antes que el, pongámosla **al frente** (antes que las demás).

Paciente	Edad	Tipo de Cirugía
Tío Rico	50	Transplante de riñón
Hugo Patito	7	Corregir pie plano
Pato Donald	31	Cuerdas vocales
Paco Patito	7	Corregir pie plano
Luis Patito	7	Corregir pie plano
Cerdo Porky	29	Lipoescultura

... transcurre el tiempo y siguen ocupadas las salas ... los pacientes siguen en espera ... y llega:

Ciro Peraloca	50	Transplante de riñón

3.3 Listas Encadenadas (colas de prioridades)

Si seguimos el mismo razonamiento, lo colocaríamos al frente, porque es una cirugía importante como la de Tío Rico ...

Es decir, quedaría adelante del Tío Rico ... ¿es correcto?

Como se puede ver, hay casos de colas en las que se requiere que los datos tengan asociado otro dato:

se le llama PRIORIDAD ....

La prioridad determina el orden dentro de la lista:

- El dato del frente será el de mayor prioridad (valor numérico menor).
- El dato de atrás será el de prioridad menor (valor numérico mayor).

3.3 Listas Encadenadas (colas de prioridades)

Pregunta:

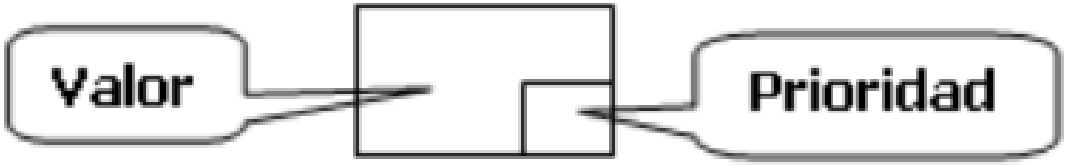
¿Que debe pasar con los datos que tienen la misma prioridad, como en el caso de Tío Rico y Ciro? ….

La respuesta es: deben quedar juntos pero adelante el que entró primero (por esta razón es una **COLA**).

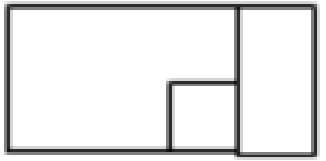
El caso de las cirugías es una de muchas aplicaciones. Por ejemplo, en los bancos los ejecutivos de cuenta son atendidos en las cajas antes que las personas que esperan, si no fuera así, pasarían mucho tiempo en espera ya que hacen varias operaciones de apertura de cuenta por mencionar uno de los casos.

3.3 Listas Encadenadas (colas de prioridades)

El miembro INFO de Cada Nodo debe ser así:



La representación de los nodos sería de la siguiente manera:



Cada nodo debe contener al menos dos miembros **"info"**, uno para el dato en sí (que en este caso es una estructura con 3 datos) y otro para la prioridad del dato en la cola.

3.3 Listas Encadenadas (colas de prioridades)

Como la prioridad es el valor que determina principalmente el orden de la lista, nuestro ejemplo, ya con prioridades, quedaría:

Paciente	Edad	Tipo de Cirugía	Prioridad
Tío Rico	50	Transplante de riñón	4
Ciro Peraloca	50	Transplante de riñón	4
Cerdo Porky	29	Lipoescultura	8
Hugo Patito	7	Corregir pie plano	10
Pato Donald	31	Cuerdas vocales	10
Paco Patito	7	Corregir pie plano	10
Luis Patito	7	Corregir pie plano	10

¿Que algoritmo usaríamos para encontrar lugar para el siguiente paciente?

Pata Daisy	26	Transplante de córnea	4

... Recorrer la lista mientras la prioridad de los datos de la lista sea **MENOR O IGUAL** que la del nodo que va a entrar ...

3.3 Listas Encadenadas (colas de prioridades)

Paciente	Edad	Tipo de Cirugía	Prioridad
Tío Rico	50	Transplante de riñón	4
Ciro Peraloca	50	Transplante de riñón	4
Pata Daisy	26	Transplante de córnea	4
Cerdo Porky	29	Lipoescultura	8
Hugo Patito	7	Corregir pie plano	10
Pato Donald	31	Cuerdas vocales	10
Paco Patito	7	Corregir pie plano	10
Luis Patito	7	Corregir pie plano	10

Dos preguntas finales:

- De los 8 pacientes de esta lista, ¿quién llegó primero?
- Considerando solo a los 4 últimos, ¿quién llegó primero?

3.3 Listas Encadenadas (colas de prioridades)

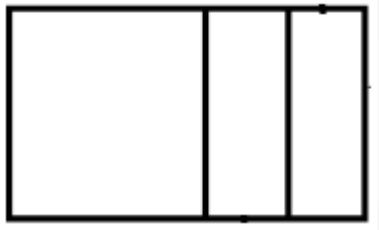
Modifique el programa Lista Encadenada Ordenada para que se implemente la Cola de Prioridades del Hospital.

Considere manejar toda la información que se presenta en las diapositivas.

La prioridad es un dato que el usuario capturará en el programa a su discreción.

3.3 Listas Doblemente encadenadas

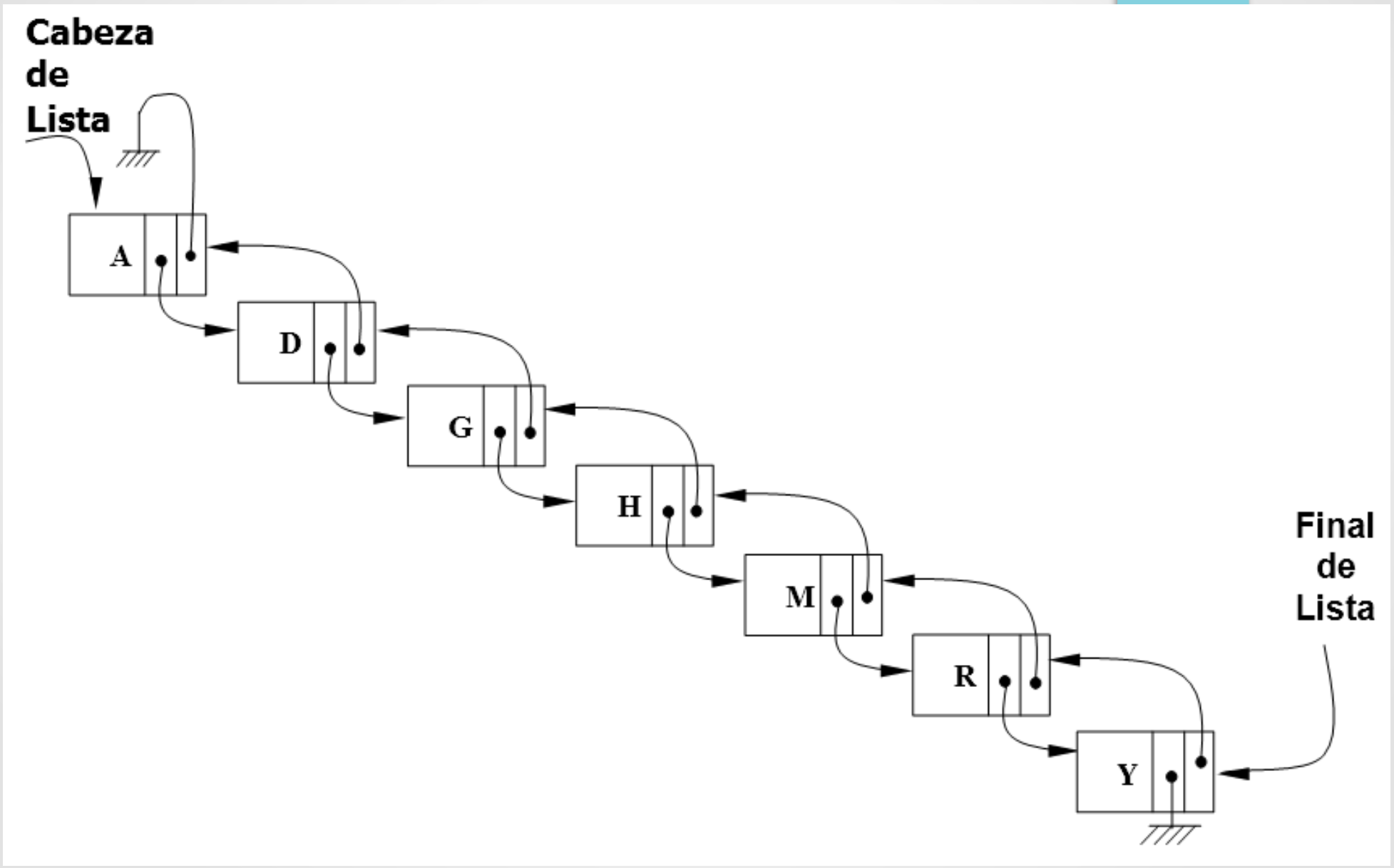
Las listas dobles son estructuras de datos en las que cada nodo contiene 2 apuntadores, uno al nodo siguiente y otro al nodo anterior en la lista. Esta variante aplica para el caso de **listas ordenadas**.



Se requiere también de un apuntador al final de la lista para que, junto con los apuntadores al nodo anterior, pueda hacerse un recorrido en el sentido inverso a la lista.

Los algoritmos de creación de la lista deben modificarse para mantener las direcciones de los apuntadores de cada nodo y las variables cabeza y final, siempre actualizadas correctamente.

3.3 Listas dobles.



3.3 Listas Doblemente encadenadas

Ejercicio

Modifique el programa de lista ordenada para que se use una lista doble y la impresión a la inversa se realice de manera muy simple:

- Solo se tiene que iniciar el recorrido por el final de la lista y seguir los apuntadores al nodo anterior.