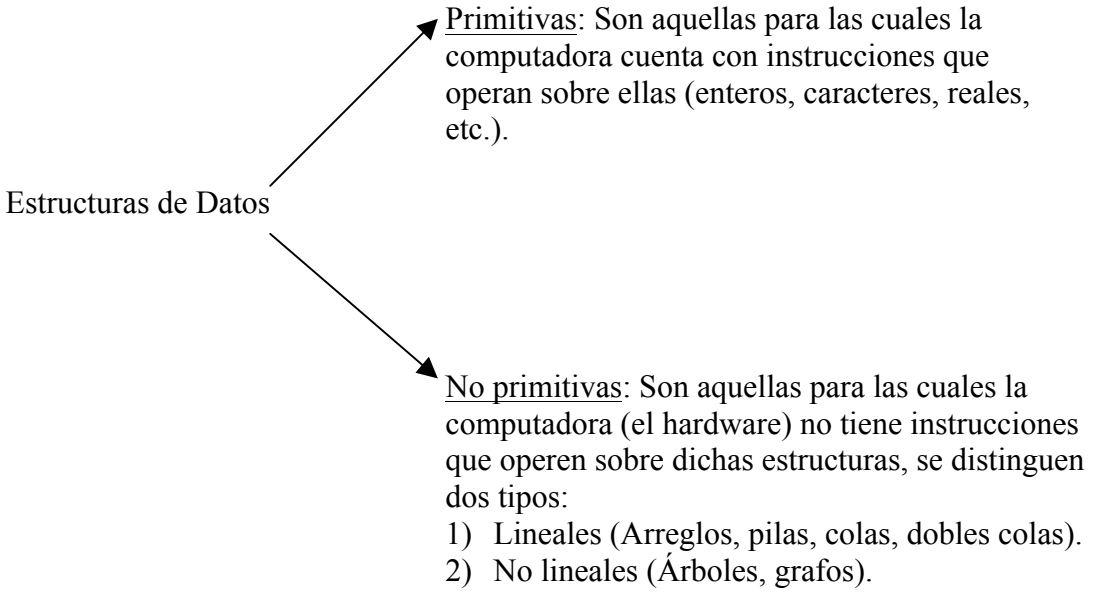
Unidad 3

ESTRUCTURAS LINEALES ESTÁTICAS Y DINAMICAS

- **Pilas.**
- **Colas.**
- **Listas enlazadas.**

Primero, veamos una breve clasificación las estructuras de datos:



Toda estructura de datos generalmente es dinámica en cuanto a sus elementos, es decir siempre se están agregando o quitando elementos (nodos) y en el peor de los casos se modifican contenidos.

- **Pilas.**

Algunos ejemplos de pilas:

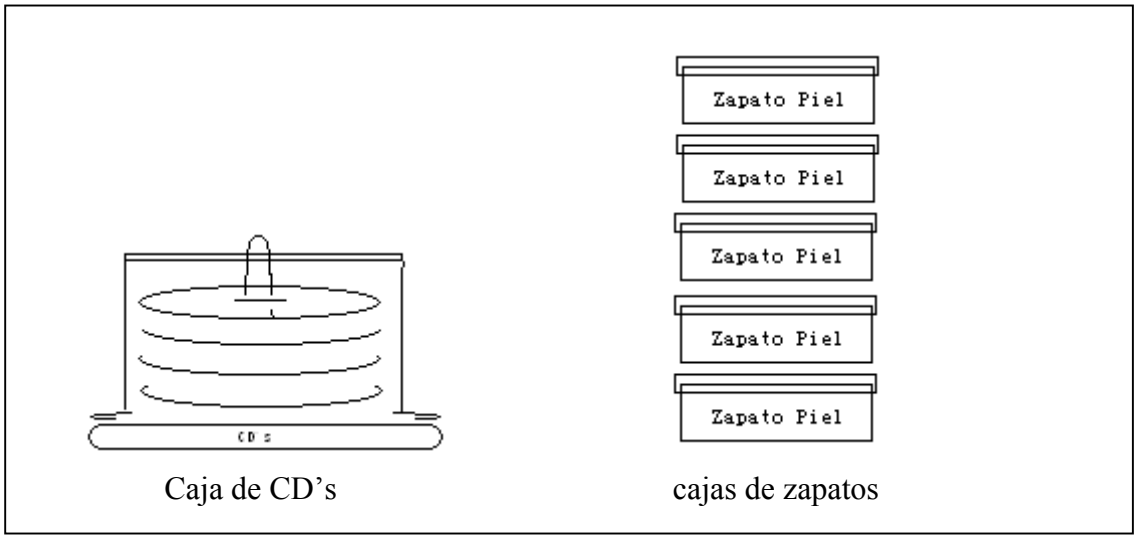


figura 3.1

En programación una pila es una estructura lineal de elementos donde las altas (inserciones) se realizan por un extremo llamado **tope** de la pila y las bajas (extracciones) también se realizan por el mismo extremo. También se le conoce con el nombre de estructura LIFO (Last Input First Output -último en entrar primero en salir-).

Funciones básicas asociadas con las pilas:

- Crear la pila.
- Insertar elementos (push).
- Extraer elementos (pop).
- Mostrar los elementos.

El elemento más recientemente almacenado se dice que esta en el tope de la pila, si extrae este elemento, desaparece de la pila, por lo que la extracción se considera una operación destructiva, ya que el valor del tope de la pila es decrementado. Si se inserta un elemento el valor del tope de la pila se deberá incrementar. En la figura 3.2 podemos observar el comportamiento de una pila.

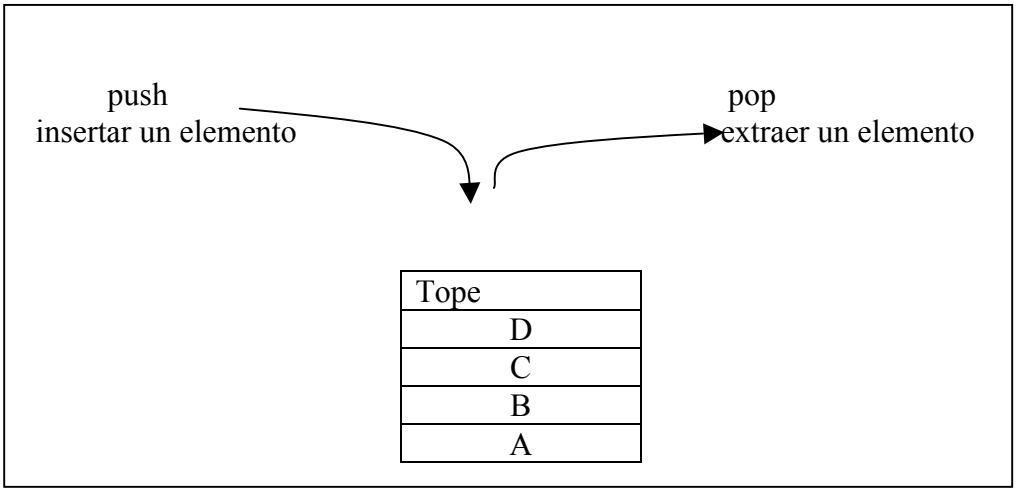


figura 3.2

Cuando estamos manipulando una pila se pueden presentar dos situaciones especiales:

- Si se intenta extraer un elemento cuando la pila esta vacía se debe indicar o mostrar esta situación (UNDERFLOW).
- Si se intenta insertar un elemento cuando la pila esta llena se debe indicar o mostrar dicha situación (OVERFLOW).

Algoritmos que muestran el comportamiento de una pila:

```
Push(Pila, MAX, tope, dato)
INICIO
  Si tope < MAX
  Entonces
       Hacer   tope <- tope + 1
               Pila[tope] <- dato
  Sino
      Escribir 'OVERFLOW –Desbordamiento-'
FIN

Pop(Pila, MAX, tope, dato)
INICIO
    Si tope > 0 y tope < MAX
    Entonces
          Hacer   dato <- Pila[tope]
                  tope <- tope -1
    Sino
          Escribir 'UNDERFLOW –Subdesbordamiento-'
FIN
```

Práctica # 6. Implementación de una pila.

Las pilas se pueden manejar internamente de distintas formas, siempre y cuando se mantengan las funciones que implementen la disciplina de acceso. Dos alternativas muy empleadas son: El uso de arreglos (estáticos ó dinámicos) y las listas enlazadas.

- Usando un arreglo estático: El siguiente código presenta el diseño básico de una pila empleando un arreglo con tamaño fijo, además de una sencilla aplicación para probar nuestra clase Pila.

```
//Pila.h

#ifndef PilaH
#define PilaH
const int MAX = 5;
enum {tam = MAX};
class Pila
{
  int datos[tam];
  int tope;
 public:
    Pila();
    bool push(int valor);
    bool pop (int &valor);
    void mostrar();
};
#endif


//Pila.cpp

//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
#include "Pila.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
```

```
Pila::Pila()
{
  tope = -1;
  for(int i = 0; i < tam; i++)
      datos[i] = 0;
}

bool Pila::push(int valor)
{
 if(tope == tam -1)
   return false;
 else
 {
   datos[++tope]= valor;
   return true;
 }
}

bool Pila::pop(int &valor)
{
 if(tope == -1)
  return false;
 else
 {
   valor = datos[tope--];
   return true;
 }
}


void Pila::mostrar()
{
  cout<<"\nDatos en la pila : \n";
  for(int i = 0; i <= tope; i++)
    cout<<"\n Dato[ "<<i<<" ] ->  "<<datos[i];
}
```

Como se pretende que la aplicación tenga un comportamiento interactivo, diseñamos una clase menú para obtener dicho comportamiento.

```
//Menu.h
#ifndef MenuH
#define MenuH
//---------------------------------------------------------------------------
class Menu
{
  public:
    int opciones();
};
#endif
//---------------------------------------------------------------------------

//Menu.cpp
#include <iostream.h>
#pragma hdrstop
#include "Menu.h"
#pragma package(smart_init)

int Menu::opciones()
{
   int op;
   cout<<"\nTrabajando con una pila : "<<endl;
   cout<<"\n Que deseas hacer : ";
   cout<<"\n 1) Insertar Dato : ";
   cout<<"\n 2) Extraer Dato : ";
   cout<<"\n 3) Salir : ";
   cout<<"\n Opcion : ";
   cin>>op;
   return op;
}

//Aplicación para usar nuestra pila
#include <vcl.h>
#include <iostream.h>
#include <conio.h>
```

```
#include "Pila.h"
#include "Menu.h"
#pragma hdrstop
#pragma argsused


int main(int argc, char* argv[])
{
  int d,op;
  Pila pila;
  Menu menu;
  bool band=true;
  while(band)
  {
   clrscr();
   op = menu.opciones();
   switch(op)
   {
    case 1 :  {
              cout<<" Dato : ";
              cin>>d;
              if(pila.push(d))
              {
               cout<<"  OK ";
               clrscr();
               pila.mostrar();
              }
              else
               cout<<" Desbordamiento (Overflow) ";
             }
             getch();
             break;
    case 2: {
               if(pila.pop(d))
               {
                   cout<<"\n Dato eliminado : "<<d;
                   pila.mostrar();
               }
               else
                    cout<<" Pila vacia  (Underflow)";
              }
              getch();
              break;
    case 3: band = false;// o solo usamos exit(0);
```

```
    }
  }
  return 0;
}
```

Aplicación visual para usar nuestra clase pila.

Crear una aplicación visual (ver figura 3.3) para usar nuestra clase pila es muy sencillo, ya que con el programa anterior ya fue probada y ahora podemos aprovechar las facilidades que ofrece el entorno visual y así simular una pila como la siguiente:

figura 3.3

En esta aplicación vamos a colocar nuestro código principal en un solo método, con el fin de usar el objeto Sender (mensajero).

Lo primero que necesitamos es declarar un objeto pila, sentencia que colocaremos en la definición de la clase forma, en su parte privada. Por lo tanto debemos abrir el archivo .h de nuestra aplicación, incluir nuestro archivo donde escribimos la definición de nuestra clase pila, y listo.

```
#ifndef AppPilaH
#define AppPilaH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
```

```
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Grids.hpp>
#include "Pila.h"
#include <Buttons.hpp>
//---------------------------------------------------------------------------


class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TLabel *Label1;
        TEdit *Edit1;
        TStringGrid *StringGrid1;
        TLabel *Label2;
        TLabel *Label3;
        TBitBtn *BitBtn1;
        TBitBtn *BitBtn2;
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall BitBtn1Click(TObject *Sender);
private:          // User declarations
     Pila pila;
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Componentes a usar:

Componente	Propiedad	Asignar
Form1	Caption	Simulación de una pila
Label1	Caption	Dato a insertar
Label2	Caption	Dato extraido
Label3	Caption	Cadena vacia
Edit1	Text	Borramos el texto (cadena vacia)
StringGrid1	FixedCols FixedRows ColCount RowCount	1 0 2 5
BitBtn1	Caption	Push

	Glyph	Incrustar un mapa de bits adecuado
BitBtn2	Caption Glyph	Pop Incrustar un mapa de bits adecuado.

Escribir el siguiente código en los métodos que se indican:

- Doble click en la forma y coloca la siguiente línea de código

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  StringGrid1->Cells[0][4]="Tope";
}
```

- Doble click en el BitBtn1 y escribimos el código que se muestra a continuación, después seleccionar el BitBtn2, nos desplazamos a la ficha events del Object Inspector, seleccionamos su evento OnClick y en este escogemos el BitBtn1Click.

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
   static pos = 4;
   int dato;
   if(Sender == BitBtn1)
   {
     if(pila.push(Edit1->Text.ToIntDef(0)))
     {
       StringGrid1->Cells[0][pos]= "";
       StringGrid1->Cells[1][pos--]=Edit1->Text.ToIntDef(0);
       if(pos >= 0)
          StringGrid1->Cells[0][pos]= "Tope";
      // else
        //  StringGrid1->Cells[0][0]="";// "Tope";
     }
     else
     {
      ShowMessage("Pila Llena -Overflow- ");
      //return;
     }
```

```
   }
   else if(Sender == BitBtn2)
   {
     if(pila.pop(dato))
     {
       Label3->Caption = dato;



      StringGrid1->Cells[1][++pos]= "";
       StringGrid1->Cells[0][pos]= "Tope";
       if(pos > 0)
          StringGrid1->Cells[0][pos-1]= "";
     }
     else
     {
        ShowMessage("Pila Vacia -Underflow-");
        Label3->Caption = "";
     }
   }
   Edit1->Clear();
   Edit1->SetFocus();
}
```

- Usando un arreglo dinámico: La siguiente implementación de la clase Pila, además de utilizar memoria dinámica, presenta algunos cambios, analízalos y detecta dichos cambios, ya que crearemos una nueva aplicación para probar nuestra clase Pila.

```
//PilaDin.h
#ifndef PilaDinH
#define PilaDinH
//---------------------------------------------------------------------------
class Pila
{
  int *vec;
  int tam;
  int tope;
  bool vacia;
  bool llena;
 public:
```

```
   Pila(int n = 10);
   void push(int valor);
   int pop();
   bool esta_vacia(){return vacia;}
   bool esta_llena(){return llena;}
};
#endif



//PilaDin.cpp
#include <assert.h>
#pragma hdrstop
#include "PilaDin.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

Pila::Pila(int n)
{
  tam = n;
  vec = new int[tam];
  tope= 0;
 // assert(vec);
  vacia = true;
  llena = false;
}

void Pila::push(int valor)
{
     vacia = false;
     vec[tope++]= valor;
     if( tope == tam)
       llena = true;

}

int Pila::pop()
{
   if(--tope == 0)
      vacia = true;
   llena = false;
```

```
   return vec[tope];
}
//---------------------------------------------------------------------------

//Aplicación para probar nuestra clase Pila
//AppPilaDin.cpp
#include <vcl.h>
#include <iostream.h>



#include "PilaDin.h"

#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused

int main(int argc, char* argv[])
{

  Pila pila(10);//Construimos una Pila con 10 elementos
                //Podemos solicitar este dato al usuario
  cout<<"\t Introduciendo datos en la pila "<<endl;
  for(int i = 0; !pila.esta_llena();i++)
  {
     cout<<"Valor introducido en la pila >>> "<<i<<endl;
     pila.push(i);
  }
  cout<<"\n\t Extrayendo datos de la pila "<<endl;


  while(!pila.esta_vacia())
  {
     cout<<pila.pop()<<" <<< Dato extraido de la pila "<<endl;
  }
  cin.get();
  return 0;
}
//---------------------------------------------------------------------------
```

Ahora vamos a hacerle unos pequeños cambios a nuestra clase Pila, en su versión dinámica para aplicar el concepto de Plantillas (templates) en C++, y utilizamos nuestra aplicación para probar los cambios que realizamos. El siguiente código muestra los cambios a realizar:

```
//PilaDin.h
#ifndef PilaDinH
#define PilaDinH



template <class T>
class Pila
{
  T *vec;
  int tam;
  int tope;
  bool vacia;
  bool llena;
 public:
    Pila(int n = 10);
    void push(T valor);
    T pop();
    bool esta_vacia(){return vacia;}
    bool esta_llena(){return llena;}
    T tope_pila(){return vec[tope-1];}
};
#endif

//PilaDin.cpp
//---------------------------------------------------------------------------
#include <assert.h>
#include <mem.h>
#pragma hdrstop

#include "PilaDin.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

template<class T>
```

```
Pila<T>::Pila(int n)
{
 tam = n;
 vec = new T[tam];
 assert(vec);
 memset(vec,0,tam);
 tope= 0;
 vacia = true;
 llena = false;
}



template<class T>
void Pila<T>::push(T valor)
{
     vacia = false;
     vec[tope++]= valor;
     if( tope == tam)
       llena = true;
}

template<class T>
T Pila<T>::pop()
{
   if(--tope == 0)
      vacia = true;
   llena = false;
   return vec[tope];
}
```

Ejercicio: Diseñe una aplicación visual, para probar sus clases implementadas.

- **Colas.**

Algunos ejemplos de colas:

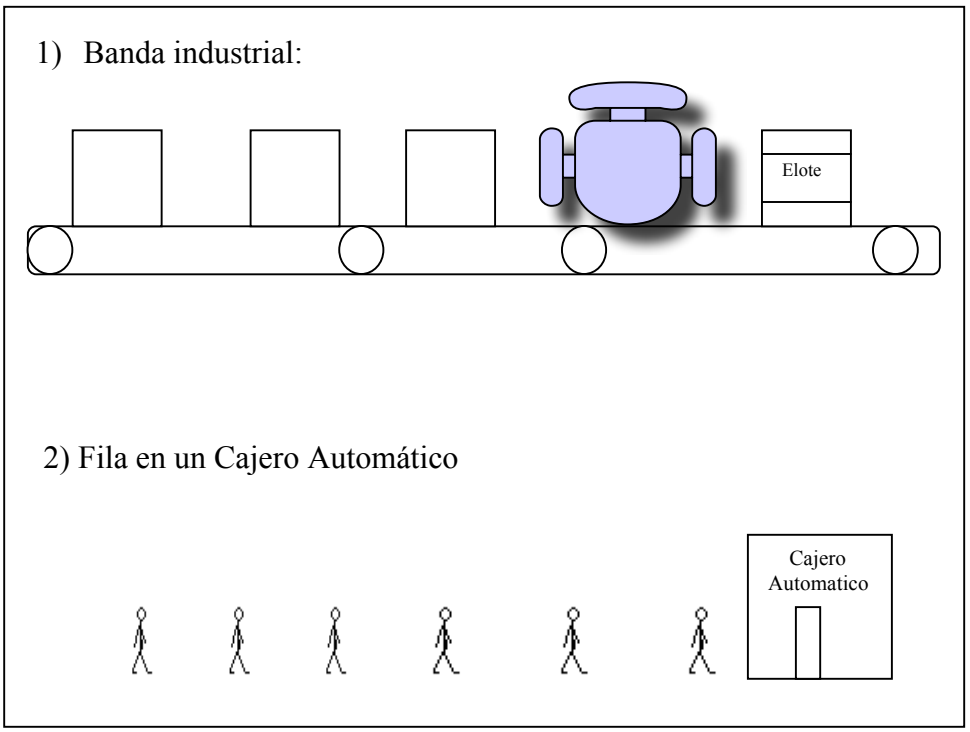
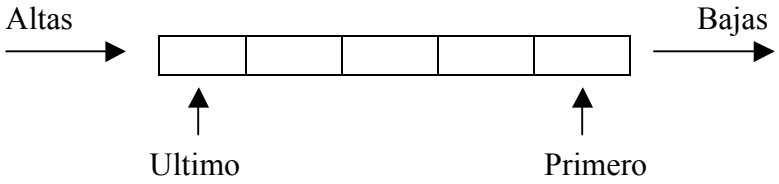


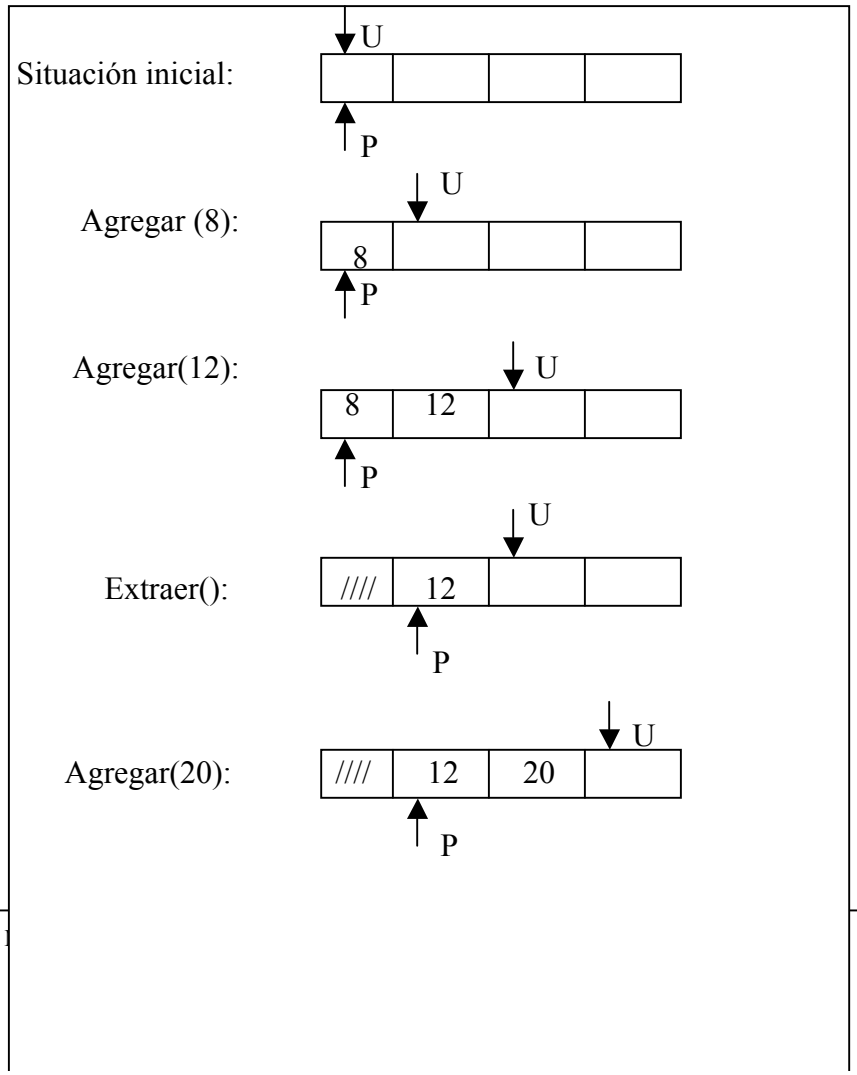
figura 3.4

La representación física en la memoria de una computadora, podría ser:



Las colas son estructuras lineales que pertenecen a la disciplina FIFO (First In, First Out – primero en entrar, primero en salir-). En este tipo de estructuras las altas se realizan por un extremo y las bajas por el otro.

Se debe definir el tamaño máximo de la cola y 2 variables auxiliares. Una de ellas para almacenar la posición del primer elemento de la cola (Primero – p -), y la otra para guardar la posición del último elemento (Ultimo – u -). Podemos observar el comportamiento de una cola en la figura 3.5



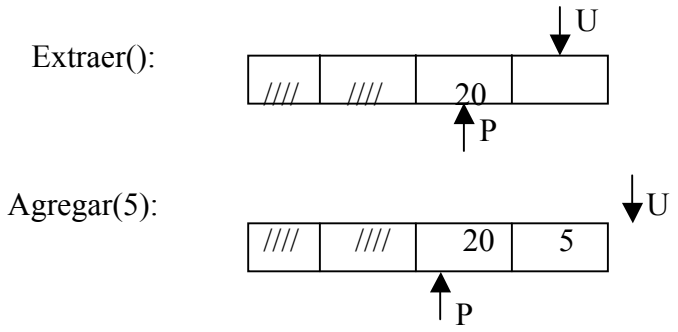
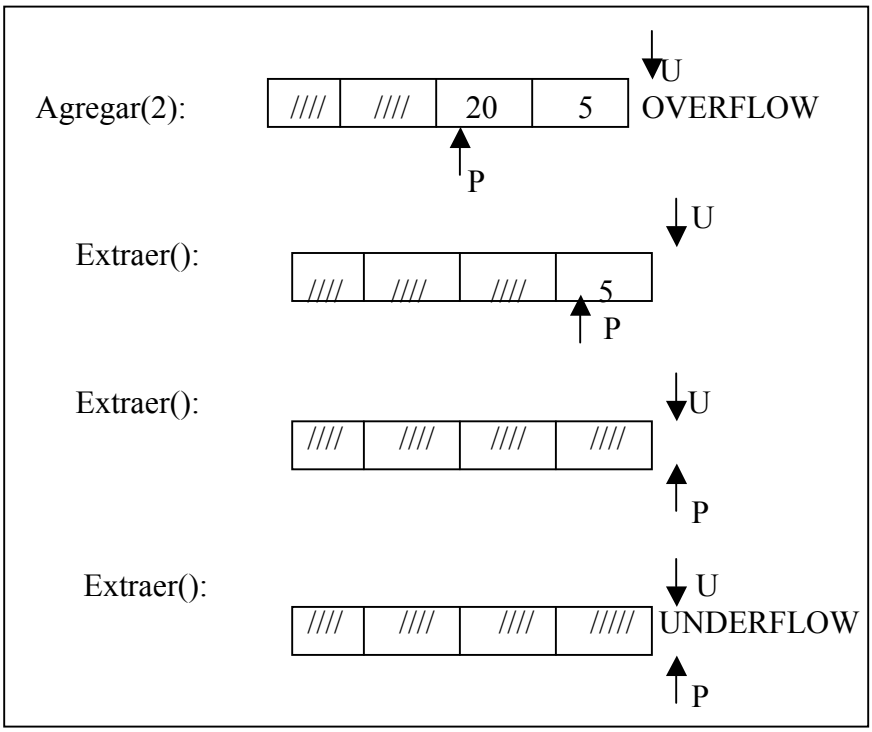


Figura 3.5



cont. Figura 3.5

La condición de cola vacía siempre es P = U.

Se presenta el desbordamiento (OVERFLOW) cuando U apunta a un elemento fuera del arreglo, y subdesbordamiento (UNDERFLOW) cuando intentamos extraer un elemento y la cola esta vacía.

Algoritmos que muestran el comportamiento de una cola:

Inserta_en_Cola(Cola, MAX, P, U, dato)

```
INICIO
      Si  U < MAX
      Entonces
            Hacer    U <- U +1
                     Cola[U] <- dato
             Si U = 1    // Se inserto el primer elemento en la Cola
             Entonces
             Hacer     P = 1
      Sino
            Escribir 'Desbordamiento'
FIN
```

Elimina_de_Cola(Cola, P, U, dato)

```
INICIO
   Si P  ≠ 0  // Verificamos si la cola no esta vacía
   Entonces
         Hacer   dato<- Cola[P]
         Si P = U        //Hay un solo elemento
         Entonces
              Hacer   P <- 0   //indica cola vacía
                      U <- 0
         Sino
              Hacer  P <- P + 1
    Sino
         Escribir 'Subdesbordamiento'
FIN
```

Implementación de una cola.

Las colas se pueden manejar internamente de distintas formas, siempre y cuando se mantengan las funciones que implementen su disciplina de acceso. Al igual que con las pilas existen dos alternativas muy empleadas : Utilizando arreglos (estáticos ó dinámicos) y las listas enlazadas.

- Usando un arreglo estático: El siguiente código muestra el diseño básico de una cola empleando un arreglo con tamaño fijo, además de una sencilla aplicación para probar nuestra clase Cola.

```
//Cola.h

#ifndef ColaH
#define ColaH
//---------------------------------------------------------------------------
const int MAX= 5;
enum {tam=MAX};
class Cola
{
   int vec[tam];
   int p,u;
  public:
    Cola();



    bool agregar(int dato);
    bool extraer(int &dato);
    bool esta_llena();
    bool esta_vacia();
};
#endif

//Cola.cpp
#include <mem.h>
#include <assert.h>
#pragma hdrstop
```

```
#include "Cola.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

Cola::Cola()
{
 p=u=-1;
}
bool Cola::esta_llena()
{
  if( u >= MAX-1)
    return true;
  return false;
}


bool Cola::esta_vacia()
{
  if( p == -1)
    return true;
  return false;
}


bool Cola::agregar(int dato)
{
 if(!esta_llena())
 {
    vec[++u]= dato;
    if(u == 0)
       p = 0;
    return true;
 }
  return false;
}

bool Cola::extraer(int &dato)
{
 if(!esta_vacia())
 {
   dato = vec[p];
```

```
    if(p == u)
    {
      p = -1;
      u = p;
    }
    else
      p++;
    return true;
  }
  return false;
}
```

```
//Aplicación

#include <vcl.h>
#include <iostream.h>
#include "Cola.h"
#pragma hdrstop

#pragma argsused



int main(int argc, char* argv[])
{
  Cola cola;
  int i;
  cout<<"\tAgregando datos a la Cola : "<<endl;
  for( i = 0; i < 5; i++)
   if(cola.agregar(i+5))
      cout<<"\nDato agregado : "<<i+5;
   else
      cout<<"\nDesbordamiento, Cola llena : ";
  int d;
  cout<<"\tExtrayendo datos de la Cola : ";
  while(1)//Para probar la extracción cuando ya no
   {         //hay datos
     if(cola.extraer(d))
         cout<<"\nDato extraido : "<< d;
     else
     {
      cout<<"\nSubdesbordamiento, cola vacia";
      break;
     }
   }
   cin.get();
   return 0;
}
```

- Usando un arreglo dinámico: La siguiente implementación de la clase Cola, presenta algunos cambios, principalmente en la creación de un arreglo dinámico, y

la necesidad de un destructor para eliminar la memoria reservada. Además, Analiza los cambios realizados en la aplicación.

```
//ColaDinámica.h
#ifndef ColaH
#define ColaH
//---------------------------------------------------------------------------
class Cola
{
   int *vec;
   int p,u,tam;
  public:
    Cola(int n=5);
    ~Cola(){delete []vec;}
    bool agregar(int dato);
    bool extraer(int &dato);
    bool esta_llena();
    bool esta_vacia();
};
#endif

//ColaDinámica.cpp
#include <mem.h>
#include <assert.h>
#pragma hdrstop
#include "Cola.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
Cola::Cola(int n)
{
 p=u=-1;
 tam = n;
 vec = new int[tam];
 assert(vec);
}
bool Cola::esta_llena()
{
  if( u >= tam-1)
    return true;
  return false;
}
```

```
bool Cola::esta_vacia()
{
  if( p == -1)
    return true;
  return false;
}

bool Cola::agregar(int dato)
{
 if(!esta_llena())
 {
    vec[++u]= dato;
    if(u == 0)
       p = 0;
    return true;
 }
 return false;
}


bool Cola::extraer(int &dato)
{
 if(!esta_vacia())
 {
   dato = vec[p];
   if(p == u)
   {
     p = -1;
     u = p;
   }
   else
     p++;
   return true;
 }
 return false;
}


//Aplicación
//---------------------------------------------------------------------------
```

```
#include <vcl.h>
#include <conio.h>
#include <iostream.h>
#include "Cola.h"
#pragma hdrstop



int main(int argc, char* argv[])
{
  int i,n,dato;
  cout<<"\n Cuantos elementos : ";
  cin>>n;
  Cola cola(n);
  randomize();
  cout<<"\n\tInsertando Datos en la cola "<<endl;
  for( i = 0; i < n; i++)
  {
   dato = random(100);
   if(cola.agregar(dato))
      cout<<"\nDato agregado : "<<dato;
   else
      cout<<"\nDesbordamiento, Cola llena : ";
  }
  cout<<"\n\n\tExtrayendo Datos de la cola "<<endl;
  while(cola.extraer(dato))
  {
      cout<<"\nDato extraido : "<< dato;
  }
  getch();
  return 0;
}
//---------------------------------------------------------------------------
```

Ejercicio: Reescriba el código de la clase cola, aplicando el concepto de plantillas. Además diseñe una aplicación para comprobar los cambios hechos.

Los algoritmos de agregar y extraer datos de una cola simple presentan falsos "Overflows", ya que no se puede utilizar el espacio liberado en las bajas.

Solución → Corrimientos: * Al momento de las altas.
* Al momento de las bajas.

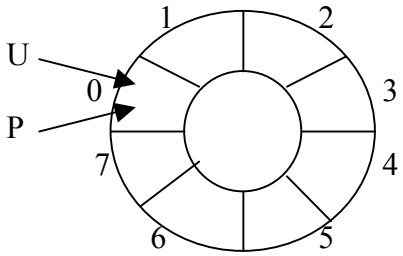
Solución → Colas circulares

- **Cola Circular.** El problema de desbordamiento en las colas secuenciales tienen una fácil solución: Haciendo uso de las colas circulares, las cuales una vez que llegan al final del arreglo o de la lista, comienzan a insertar datos por el principio, siempre y cuando la cola no este llena.

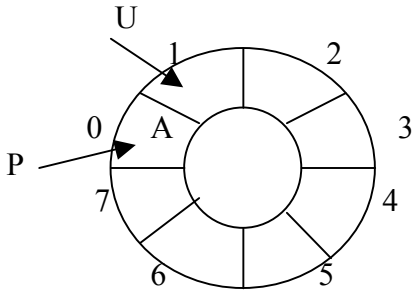
En este tipo de colas cuando Primero a llegado a la última posición, su incremento deberá situarlo en la primera posición del arreglo.
De forma gráfica podemos analizar el comportamiento de una cola circular:

Situación inicial:

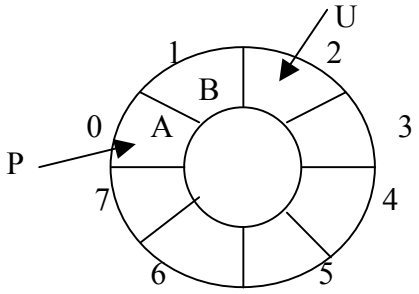
P : Primero
U : Ultimo



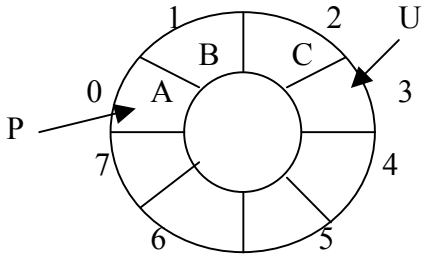
1) .- Insertar (A):



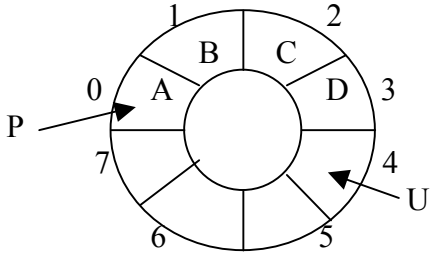
2).- Insertar (B)



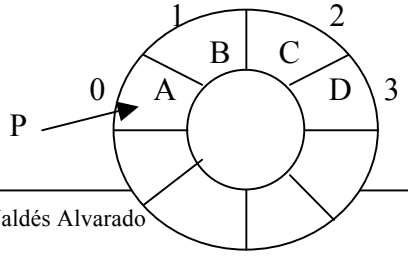
3).- Insertar (C)

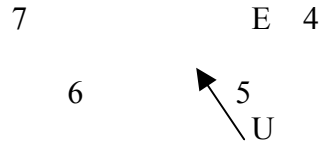


4).- Insertar (D)

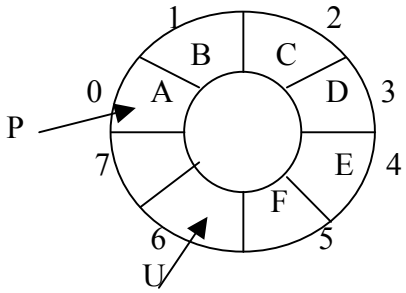


5).- Insertar (E)

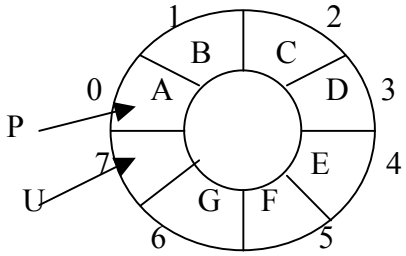




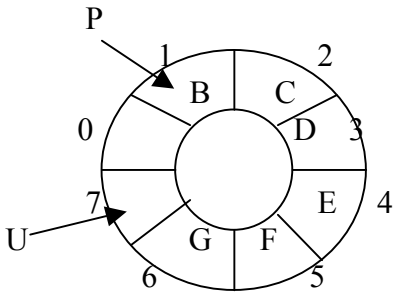
6).- Insertar (F)



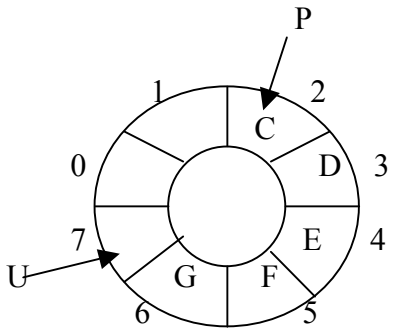
7).- Insertar (G)



8) .- Extraer ( )



9).- Extraer ( )



Algoritmos que muestran el comportamiento de una cola circular:

```
Insertar_en_cola_Circ(Datos, MAX, P, U, dato)
INICIO
    Si (U = MAX) y (P = 1) ó   ((U+1) = P )
       Entonces
            Escribir  "Desbordamiento "
       Sino
            Si (U = MAX)
               Entonces
                   Hacer U <- 1
            Sino
                Hacer U<- U + 1
       Hacer Datos[U] <- dato
       Si P = 0
           Entonces
                  P = 1
FIN
```

```
Eliminar_de_cola_Circ(Datos, MAX, P, U, dato)
INICIO
    Si P = 0
       Entonces
             Escribir "Subdesbordamiento"
       Sino
            Hacer  dato <- Datos[P]
            Si P = U      //Si hay un solo elemento
                  Entonces
                       Hacer   P = 0
                               U = 0
                  Sino
                       Si P = MAX
                         Entonces
                            Hacer P = 1
                         Sino
                            Hacer P = P + 1
                       Fin si
             Fin si
    Fin si
FIN
```

A continuación se presenta la implementación de una cola circular, en la cual utilizamos el concepto de plantillas:

```
//ColaCirc.h

#ifndef ColaCirc1H
#define ColaCirc1H
//---------------------------------------------------------------------------
template <class T>
class ColaCirc
{
   T *datos;
   int tam;
   int P,U;
  public:
   ColaCirc(int n);
   bool insertar(T dato);
   bool eliminar(T &dato);
   bool esta_vacia();
   bool esta_llena();
};
#endif
```

```
//ColaCirc.cpp
#include <assert.h>
#pragma hdrstop
#include "ColaCirc1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

template <class T>
ColaCirc<T>::ColaCirc(int n)
{
  tam = n;
  datos = new T[tam];
  assert(datos);
  P = U = -1;
}



template <class T>
bool ColaCirc<T>::insertar(T dato)
{
  if((U == tam-1) && (P == 0) || (U+1 == P))
     return false;
  else
    if( U == tam-1)
        U =0;
    else
       U++;
  datos[U]=dato;
  if( P == -1)
     P = 0;
  return true;

}
```

```
template <class T>
bool ColaCirc<T>::eliminar(T &dato)
{
   if(P == -1)
     return false;

   dato = datos[P];
   if(P == U)    //Hay Un solo elemento
       P = U = -1;

   else if(P == tam-1)
   {
        datos[P] = -1;
         P = 0;
   }
   else
   {
     datos[P]=-1;
     P++;
   }
   return true;
}



template <class T>
bool ColaCirc<T>::esta_vacia()
{
  return (P == -1);
}

template <class T>
bool ColaCirc<T>::esta_llena()
{
   return (((U+1)%tam) == P);
}
```

```
//AppColaCirc.cpp
#include <vcl.h>
#include <iostream.h>
#include "ColaCirc1.h"
#include "ColaCirc1.cpp"

#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    ColaCirc<char>  cola(3);
    char car;
    for(car ='A'; car <= 'C';car++)
      if(cola.insertar(car))

        cout<<"\n Dato insertado "<<car;
      else
        cout<<"\n Overflow, cola llena ";
    int cont=0;

    while(1)
    {
     if(cola.eliminar(car))
       cout<<"\n Dato extraido : "<<car;
     else
     {
       cout<<"\n Underflow,Cola vacia : ";
       cont++;
     }
     if(cont==10)//condicion que usamos para probar nuestra clase
      break;
    }
    for(car ='A'; car <= 'D'; car++)
      if(cola.insertar(car))
        cout<<"\n Dato insertado "<<car;
      else
        cout<<"\n Overflow, cola llena ";

    cont=0;
```

```
    while(1)
    {
     if(cola.eliminar(car))
       cout<<"\n Dato extraido : "<<car;
     else
     {
       cout<<"\n Underflow,Cola vacia : ";
       cont++;
     }
     if(cont==10)
      break;
    }
    cin.get();
      return 0;
}
```

Realiza los cambios que se presentan a continuación, para probar nuestra clase ColaCirc usando la clase AnsiString.

```
int main(int argc, char* argv[])
{
    AnsiString nombres[4]={"Ana","Beto","Maria","Alma"};
    ColaCirc<AnsiString>  cola(3);
    int i;
    cout<<"\nInsertamos datos en la cola circular : ";
    for(i =0; i <= 3;i++)
      if(cola.insertar(nombres[i]))
        cout<<"\n Dato insertado "<<nombres[i].c_str();
      else
        cout<<"\n Overflow, cola llena ";
    int cont=0;
    cout<<"\n\nEliminamos datos de la cola circular : ";
    while(1)
    {
     if(cola.eliminar(nombres[i]))
       cout<<"\n Dato extraido : "<<nombres[i].c_str();
     else
     {
       cout<<"\n Underflow,Cola vacia : ";
       cont++;
     }
```

```
      if(cont==5)//condicion que usamos para probar nuestra clase
       break;
    }
    cin.get();
    return 0;
}
```

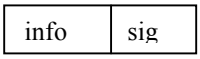
Ejercicio: Escriba una aplicación Visual para probar su clase Cola Circular.

- **Listas Enlazadas.** Una lista enlazada simple o abierta, es una forma sencilla de organizar nodos, de tal modo que cada uno apunta al siguiente, y el último apunta a nada –NULL- (ver fig. 3.6), es decir, el puntero del nodo siguiente vale NULL.
  En estas listas existe un nodo especial: el primero. Normalmente diremos que nuestra lista es un puntero a ese primer nodo o llamaremos a ese nodo la cabeza de la lista. Esto es porque mediante ese único puntero podemos acceder a toda la lista.
  Cuando el puntero que usamos para acceder a la lista vale NULL, diremos que la lista está vacia.

El nodo típico para construir listas enlazadas tiene la siguiente forma:

```
struct Nodo
{
   int info;
   Nodo *sig;
};
```

Gráficamente un nodo lo representaríamos como:



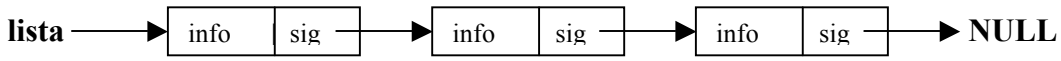


figura 3.6

a)Listas enlazadas simples. El siguiente programa muestra la implementación de una lista enlazada.

```
//Representación Básica de una Lista Enlazada

struct Nodo
{
   int info;
   Nodo *sig;
};

int main( )
{
    int i;
    Nodo lista, *nodo;
    lista.sig = NULL;  //Lista vacia
    nodo = &lista;     //Apunta al inicio de la lista
```

```
     for(i = 0; i <= 10; i++)
     {          //Asignar memoria dinamica
           nodo->sig = new Nodo;
           nodo = nodo->sig;


           nodo->info = i;
           nodo->sig = NULL;
     }

//Mostar la lista      Nota:  & = direccion
    //Le asigna a la & nodo la & de lista.sig  0
    //posiciona el nodo al inicio lista
    nodo = lista.sig;
    while(nodo)  // Mientras no sea NULL
    {
      cout <<"Dir: "<<hex<<&nodo->sig <<dec<<"   Valor: "<<nodo->info<<endl;
       nodo = nodo->sig;
    }
    delete nodo;
    getch();
    return 0;
}
/*NOTA: EN LA SALIDA DEL PROGRAMA OBSERVAR LAS DIRECCIONES
      ASIGNADAS A LA LISTA ENLAZADA
*/
```

Ahora escribamos una aplicación en ambiente visual para probar nuestra lista enlazada. Nuestro código lo colocaremos en el evento OnClick del boton “Crear Lista”.

fig. 3.6

```
struct Nodo
{
   int info;
   Nodo *sig;
};

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int i, ren = 0;
    AnsiString cad;
    Nodo inicio, *lista;
    Nodo lista, *nodo;
    lista.sig = NULL;  //Lista vacia
    cad.sprintf(" %p", &lista);
    Edit1->Text=cad;
    cad.sprintf(" %d", lista.info);
    Edit2->Text = cad;
    cad.sprintf(" %p", lista.sig);
```

```
    Edit3->Text = cad;
    nodo = &lista;     //Apunta al inicio de la lista
    cad.sprintf(" %p", nodo);
    Edit4->Text = cad;
    cad.sprintf(" %d", nodo->info);
    Edit5->Text = cad;
    for(i = 0; i <= 10; i++)
    {          //Asignar memoria dinamica
      nodo->sig = new Nodo;
      if(i == 0)
      {
        cad.sprintf(" %p", lista.sig);
        Edit3->Text = cad;
        cad.sprintf(" %p", nodo->sig);
        Edit6->Text = cad;
      }
      nodo = nodo->sig;
      nodo->info = i;
      nodo->sig = NULL;
    }



     //Mostar la lista
    //Le asigna a la & nodo la & de lista.sig  ó
    //posiciona nodo al inicio de la lista enlazada
    nodo = lista.sig;

    while(nodo)
    {
      cad.sprintf(" %p", nodo);
      Vector->Cells[0][ren]= cad;
      Vector->Cells[1][ren]=nodo->info;
      cad.sprintf( " %p", nodo->sig);
      Vector->Cells[2][ren++]= cad;
      nodo = nodo->sig;
    }
    delete nodo;

}
```

Existen múltiples alternativas para implementar listas enlazadas. Ahora que ya tenemos una idea clara de lo que es una lista enlazada y como funciona, vamos a aplicar nuestros conocimientos de POO para diseñar e implementar una clase que permita representar el comportamiento de una lista enlazada.

Lista enlazada simple utilizando POO.

```
class nodo_lista
{
   int dato;
   nodo_lista *sig;
  public:
   nodo_lista(int d);
   void inserta_s(nodo_lista *nodo);
   void muestra();
};
nodo_lista::nodo_lista(int d)
{
  dato = d;
  sig = NULL;
}

void nodo_lista::inserta_s(nodo_lista *nodo)
{
  if(sig == NULL) //0
  {
    sig = nodo;
    nodo->sig = NULL;//0
  }
  /*else  {
     nodo->sig = sig;
     sig = nodo;
  }*/
}
```

```
void nodo_lista::muestra()
{
  nodo_lista *p1;
  p1 = this;
  while(p1 != 0)
  {
    cout<< p1->dato<<" ";
    p1 = p1->sig;
  }
  cout<<'\n';
}
int main(int argc, char **argv)
{
  nodo_lista *p = new nodo_lista(5);
  nodo_lista *n= p;
  for(int i = 0; i < 5;i++)
  {
    nodo_lista *o = new nodo_lista(i*i);
    n->inserta_s(o);
    n = o;
  }
  p->muestra();
  n->muestra();
  getch();
  return 0;
}
```