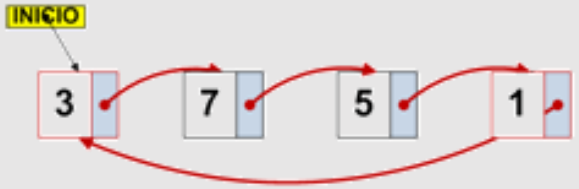
Técnico Informático

ESTRUCTURA DE DATOS

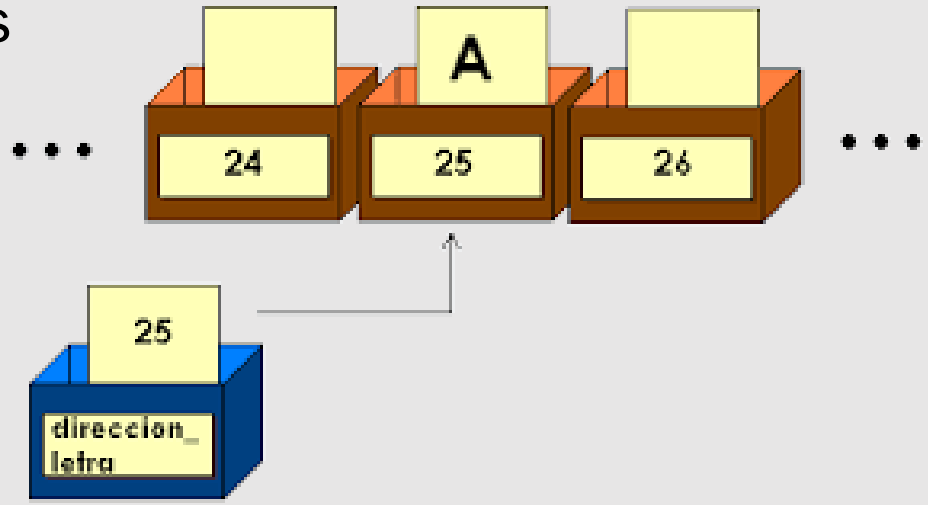
UNIDAD I: LISTAS SIMPLES



Escuela de Minas "Dr. Horacio Carrillo"
Universidad Nacional de Jujuy

Índice

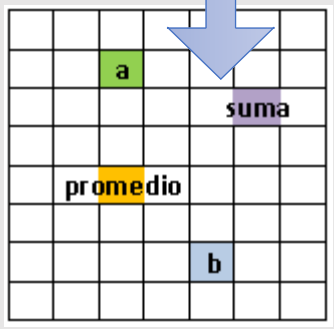
- Variables Estáticas y Dinámicas
- Definición del Tipo Puntero y sus operaciones
- Definición de Lista Simple
- Operaciones fundamentales
- Implementación



Variables Estáticas y Dinámicas

- Una **variable estática** se crea en la memoria al ejecutar un programa. Ésta existe hasta que el programa finaliza.
- El espacio de memoria reservado no está disponible para otras aplicaciones hasta que el programa finalice.

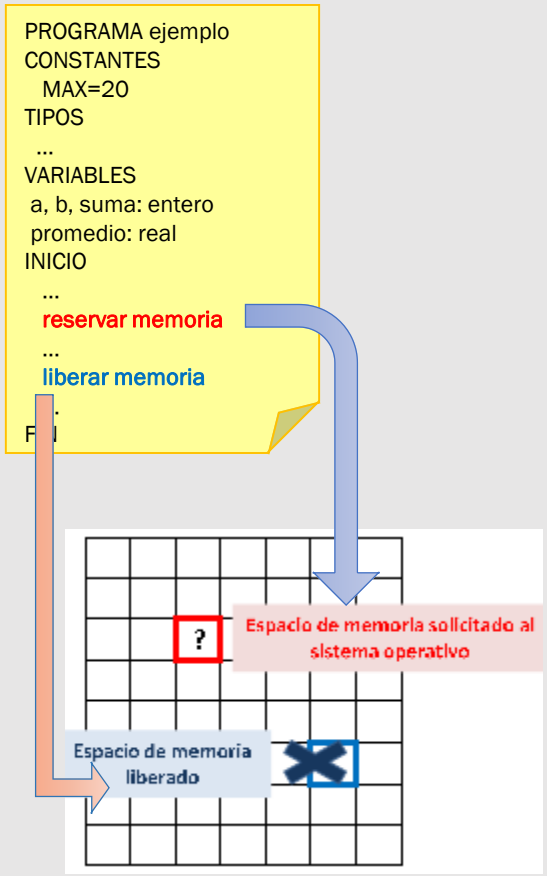
```
PROGRAMA ejemplo
CONSTANTES
   MAX=20
TIPOS
  ...
VARIABLES
 a, b, suma: entero
 promedio: real
INICIO
   ...
   instrucciones
   ...
FIN
```



Variables del programa almacenadas en memoria

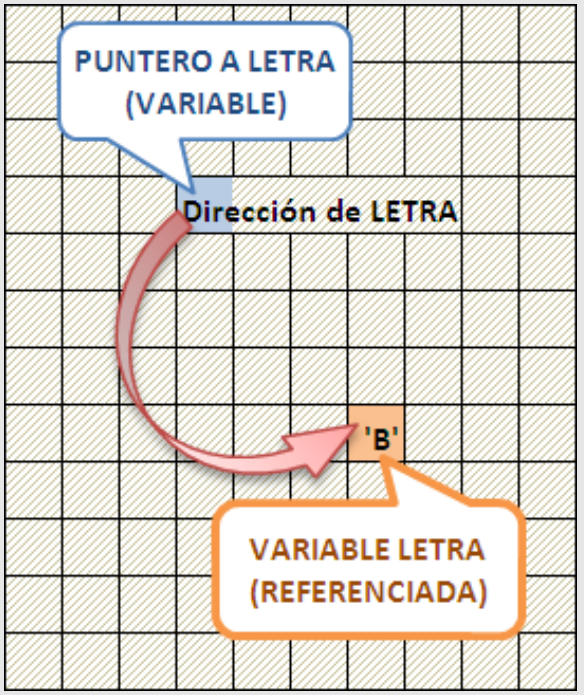
Variables Estáticas y Dinámicas

- Una **variable dinámica** es aquella que se crea durante la ejecución de un programa, a partir del espacio de memoria libre. Para ello, se **reserva** un espacio de memoria y una etiqueta. Una variable dinámica puede **eliminarse** en cualquier momento (liberando espacio de memoria).



Punteros

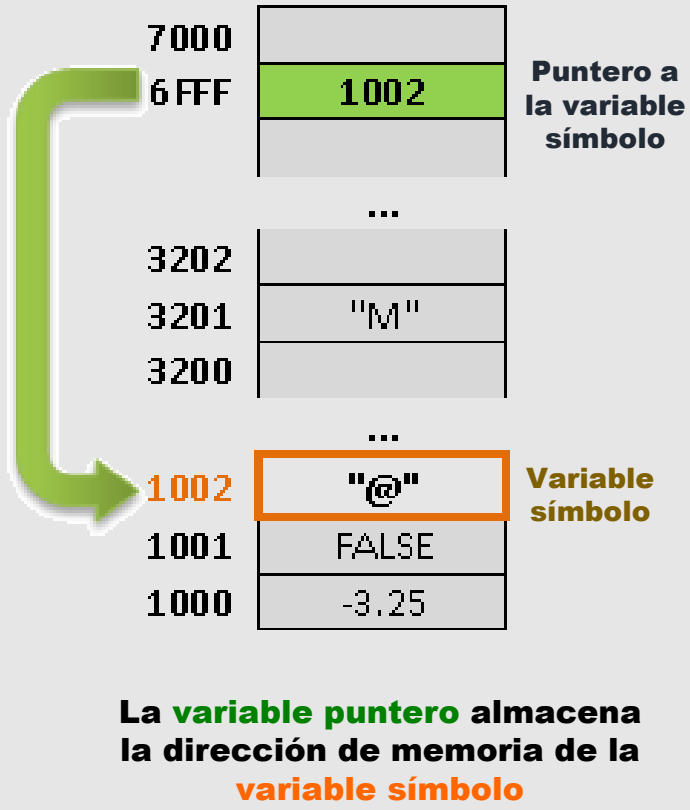
- El TDA puntero es un tipo de dato simple que almacena la **dirección** de memoria de otra variable, denominada variable referenciada.
- Un puntero “apunta” al espacio de memoria ocupado por otra variable.



MEMORIA DE LA COMPUTADORA

Operaciones sobre punteros (1)

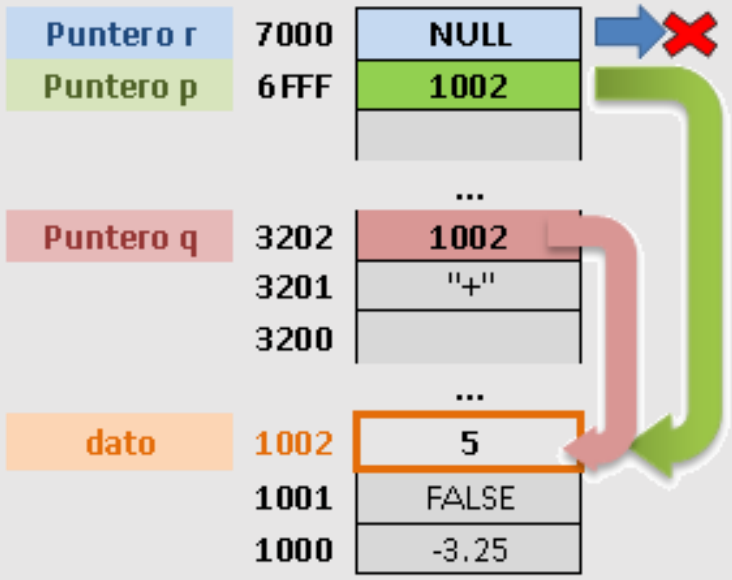
- El TDA puntero tiene asociadas las operaciones de **asignación** y **comparación** de punteros.
- La asignación almacena una **dirección de memoria** en una **variable** de tipo puntero. El valor **NULL** indica que el puntero no guarda ninguna dirección.



La variable puntero almacena la dirección de memoria de la variable símbolo

Operaciones sobre punteros (2)

- La operación de comparación permite determinar si 2 variables de tipo puntero referencian o no la misma **dirección de memoria**, o si un puntero tiene valor **NULL**.



Las variables punteros p y q referencian (almacenan la dirección de memoria) la variable dato.
El puntero r tiene el valor NULL, no apunta a ninguna posición de memoria.

Operaciones sobre punteros (3)

```
#include <iostream>
#include <stdlib.h>
using namespace std;
typedef int *p_entero;
main()
{ int dato;
  p_entero p,q;
  cout << "Ingrese un dato:";
  cin >> dato;
  cout << "Valor ingresado: " << dato << endl;
  p=&dato;
  cout << endl << "Valor del puntero p: " << p << endl;
  cout << "Valor apuntado por el puntero p: " << *p << endl;
  q=p;
  cout << endl << "Valor del puntero q: " << q << endl;
  cout << "Valor apuntado por el puntero q:" << *q << endl;
  system("pause");
}
```

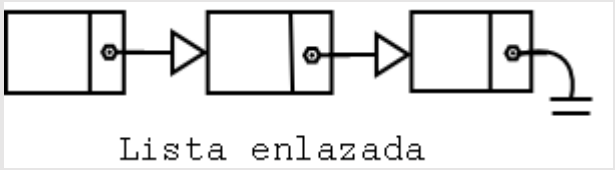
Guardando la dirección de la variable dato en el puntero p

Copiando la dirección que se guardó en p al puntero q

Acceso al valor de la variable apuntada por p y q

Listas Simples (1)

- Una lista simple es una colección de elementos (***nodos***) ordenada según su posición, cuyo acceso/recorrido se realiza mediante ***punteros*** que enlazan los nodos.
- Una lista es una **estructura lineal** en la que los elementos (nodos) se disponen de tal forma que cada uno tiene un **predecesor** y un **sucesor**, salvo el primero y el último.



Lista enlazada

Listas Simples (2)

- Un *nodo* es un registro con 2 campos esenciales:
  - Campo de **datos** (tipos de datos simples o compuestos)
  - Campo **puntero** (un puntero hacia otro nodo del mismo tipo.)

Listas Simples (3)

nodo=REGISTRO

datos: tipo_dato (simple, compuesto)

siguiente: puntero a nodo

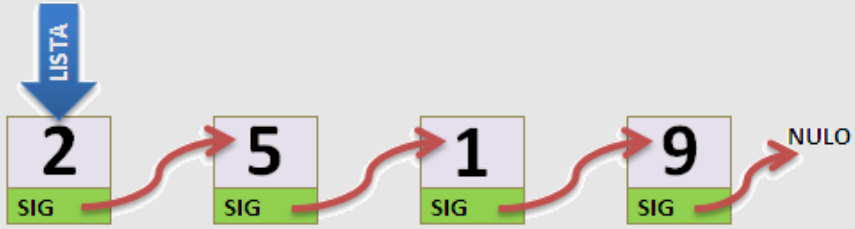
FIN_REGISTRO

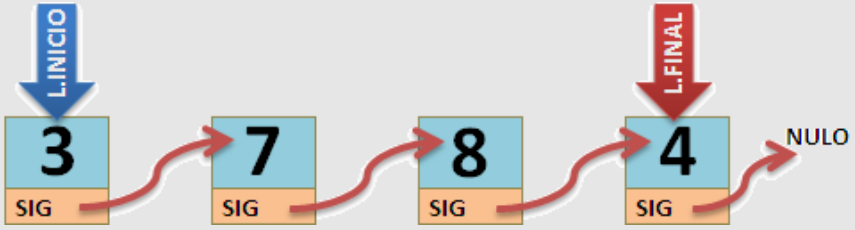
Operaciones Fundamentales

- Sobre una lista simple definen las siguientes operaciones:
  - Iniciar lista
  - Crear nodo
  - Agregar nodo
    - ✓ agregar_inicio
    - ✓ agregar_final
    - ✓ agregar en orden
  - Quitar nodo
    - ✓ quitar_inicio
    - ✓ quitar_final
    - ✓ quitar_nodo_especifico
  - Mostrar (recorrido de la lista)
  - Buscar un valor en la lista

Alternativas de Implementación

- Básicamente, la implementación del TDA lista requiere de la definición de un registro (datos y puntero a próximo elemento) y punteros que permitan acceder a la lista. La implementación puede presentar las siguientes variantes:
  - Un puntero al inicio de la lista
  - Un puntero al inicio y otro al final de la lista





Implementación (1)

- TDA lista: Implementación del tipo nodo y del puntero a éste.

```
typedef struct tnodo *pnodo;
typedef struct tnodo{
                        int dato;
                        pnodo sig;
                      };
```

  - ***pnodo***: es un tipo puntero que permite referenciar registros *tnodo*.
  - ***sig***: es un campo tipo puntero (*pnodo*) que permite enlazar los nodos de una lista.

Implementación (2)

- *Iniciar lista*
  - Propósito: inicializar la lista (esto genera una lista vacía).
  - Entrada: una lista (puntero de inicio de la lista).
  - Salida: una lista vacía (puntero de inicio de la lista en valor NULO).
  - Restricciones: ninguna.

- Operación *iniciar lista*

```
void inicia_lista(pnodo &lista)
{
      lista=NULL;
}
```

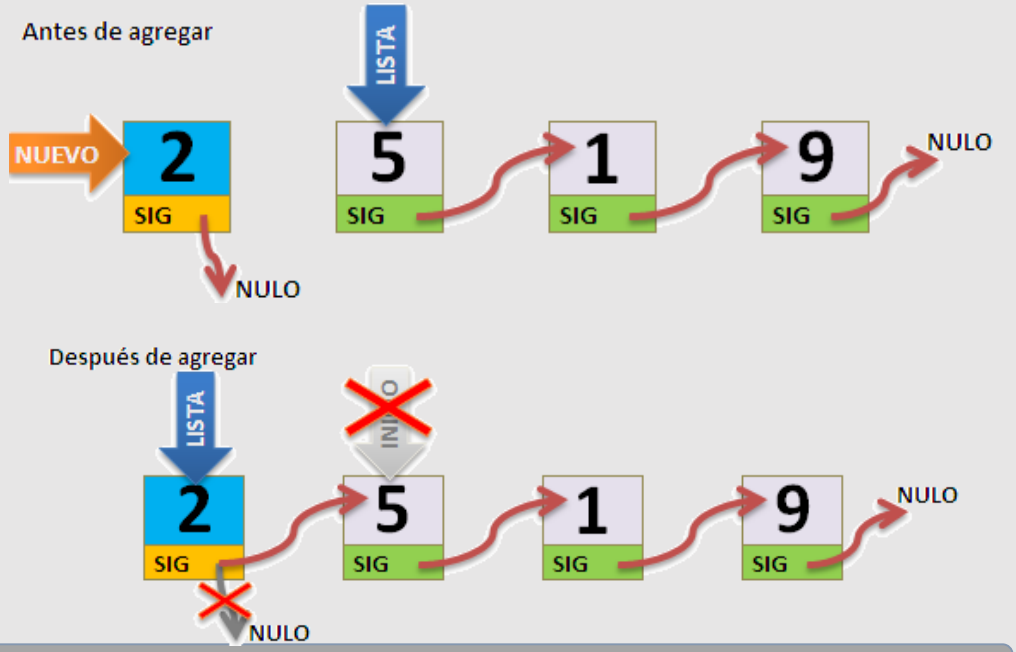
SE GENERA UNA LISTA VACÍA

Implementación (3)

- Operación *agregar al inicio*

- *Agregar un nodo al inicio*
  - Propósito: agregar un nodo al inicio de la lista.
  - Entrada: una lista y un nuevo dato.
  - Salida: una lista con un nuevo nodo al principio.
  - Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

```
void agregar_inicio(pnodo &lista,pnodo nuevo)
{   nuevo->sig=lista;
    lista=nuevo;
}
```



Implementación (4)

- Operación *crear nodo*

- *Crear nodo*
  - Propósito: crear un nuevo nodo (se reserva memoria para un nuevo elemento).
  - Entrada: un puntero a nodo.
  - Salida: un puntero con la dirección del nodo creado. Si el nodo no puede crearse, retorna NULO.
  - Restricciones: ninguna.

```
void crear(pnodo &nuevo)
{
  nuevo=new tnodo;
  if (nuevo!=NULL)
     { cout << "Ingrese valor: ";
       cin >> nuevo->dato;
       nuevo->sig=NULL;
     }
  else
     cout << "MEMORIA INSUFICIENTE" << endl;
}
```

¿Cómo se usa *crear_nodo*?

```
crear(nuevo);
if (nuevo!=NULL)
   agregar_inicio(milista,nuevo);
```

Implementación (5)

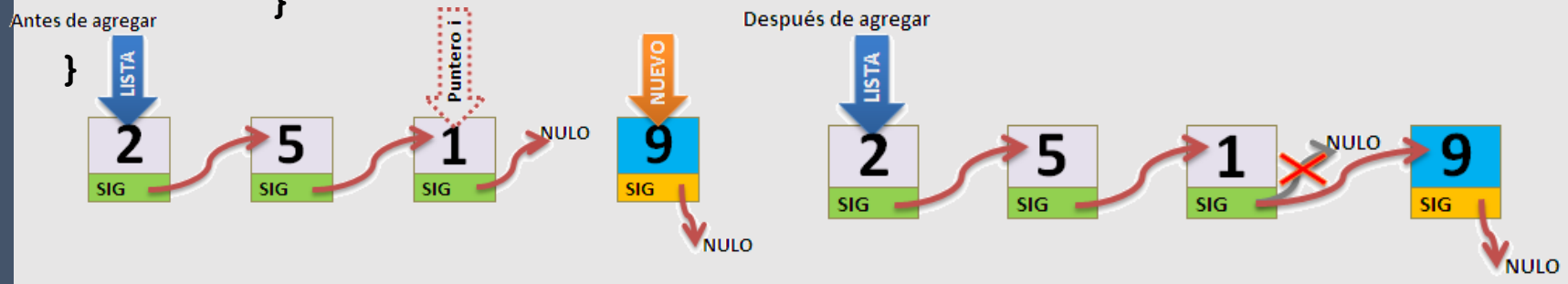
- *Agregar un nodo al final*
  - Propósito: agregar un nodo al final de la lista.
  - Entrada: una lista y un nuevo dato.
  - Salida: una lista con un nuevo nodo al final.
  - Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

- Operación *agregar al final*

```
void agregar_final(pnodo &lista,pnodo nuevo)
{ pnodo i;
      if (lista==NULL)
         lista=nuevo;
      else
        { for(i=lista;i->sig!=NULL;i=i->sig);
          i->sig=nuevo;
        }
}
```

Inserción en lista vacía

Inserción al final de una lista con elementos



Implementación (6)

- Operación *agregar en orden*

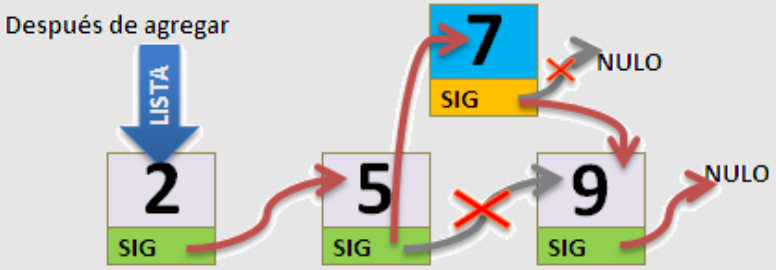
- *Agregar un nodo en orden*
  - Propósito: agregar, en orden, un nodo a la lista.
  - Entrada: una lista y un nuevo dato.
  - Salida: una lista, ordenada, con un nuevo nodo.
  - Restricciones: una lista inicializada y espacio en memoria para la creación del nuevo nodo.

```
void agregarorden(pnodo &lista,pnodo nuevo)
{ pnodo i;
  if (lista==NULL)
   lista=nuevo;
  else
   {if (nuevo->dato < lista->dato)
     {nuevo->sig=lista;
      lista=nuevo;}
    else
     {for(i=lista;i->sig!=NULL && nuevo->dato > (i->sig)->dato;i=i->sig);
        nuevo->sig=i->sig;
        i->sig=nuevo; }
   }
}
```

Inserción en lista vacía

Inserción al inicio

Inserción al medio o final



Implementación (7)

- Operación *quitar del inicio*

```
pnodo quitar_inicio(pnodo &lista)
{ pnodo borrado;
  if (lista==NULL)
     borrado=NULL;
  else
   { borrado=lista;
     lista=lista->sig;
     borrado->sig=NULL;
   }
  return borrado;
}
```

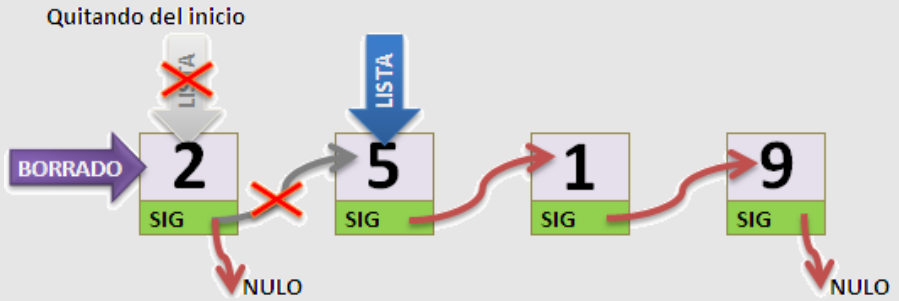
Extracción de lista vacía

Extracción del primer nodo

- *Quitar un nodo del inicio*
  - Propósito: quitar el primer nodo de la lista.
  - Entrada: una lista.
  - Salida: una lista con un nodo menos (extraído del inicio) y la dirección del elemento extraído.
  - Restricciones: una lista inicializada y no vacía.

¿Cómo se usa *quitar_inicio*?

```
eliminado=quitar_inicio(milista);
 if (eliminado!=NULL)
   { cout << "Eliminado: " << eliminado->dato;
     delete(eliminado); }
 else
   cout << "NO PUEDE ELIMINAR, LISTA VACIA";
```

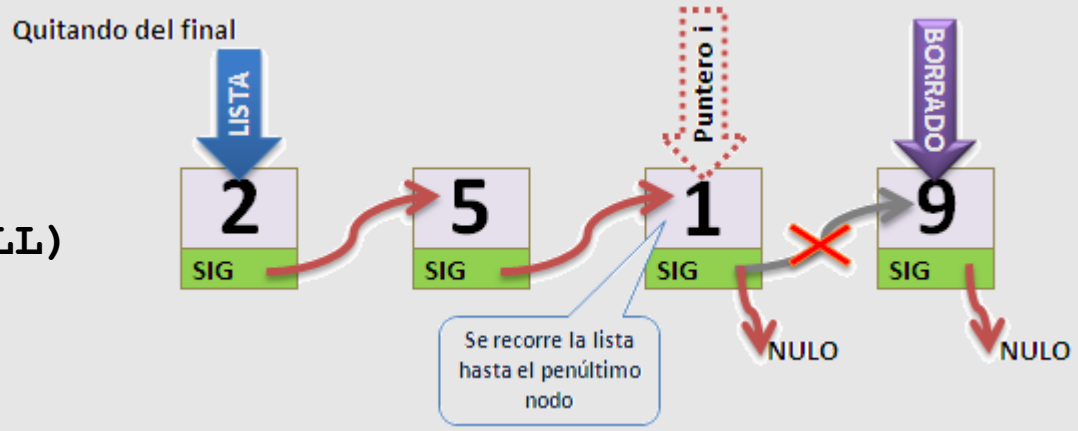


Implementación (8)

- *Quitar un nodo del final*
  - Propósito: quitar el último nodo de la lista.
  - Entrada: una lista.
  - Salida: una lista con un nodo menos (extraído del final) y la dirección del elemento extraído.
  - Restricciones: una lista inicializada y no vacía.

- Operación *quitar del final*

```
pnodo quitar_final(pnodo &lista)
{ pnodo borrado,i;
  if (lista==NULL)                          // Extracción de lista vacía
     borrado=NULL;
  else
   {if (lista->sig==NULL)                   // Extracción del único nodo
      { borrado=lista;
        lista=NULL; }
    else
      { for(i=lista;(i->sig)->sig!=NULL;i=i->sig);   // Extracción del último nodo
        borrado=i->sig;
        i->sig=NULL; }
   }
  return borrado;
}
```



Implementación (9)

- Operación *quitar un nodo segú*

- *Quitar un nodo según un valor especificado*
  - Propósito: quitar un nodo con valor específico.
  - Entrada: una lista y el valor a extraer.
  - Salida: una lista con un nodo menos (extraído el valor solicitado) y la dirección del nodo extraído.
  - Restricciones: una lista inicializada y no vacía.

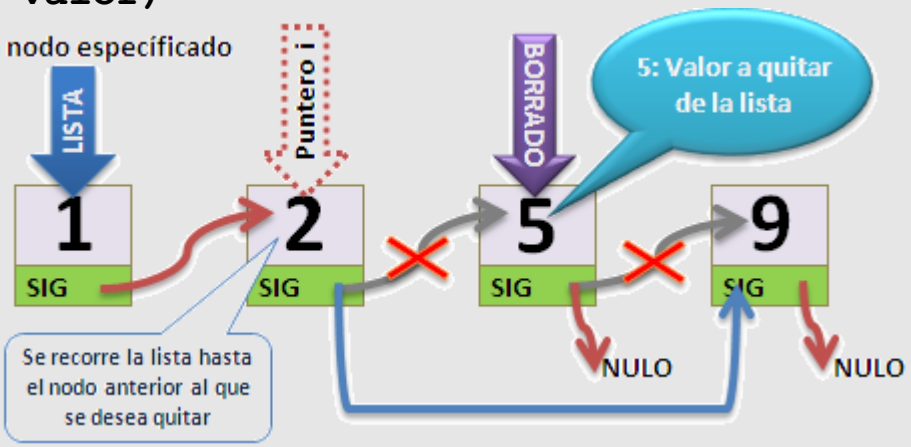
```
pnodo quitar_nodo(pnodo &lista,int valor)
{ pnodo borrado,i;
  if (lista==NULL)
    borrado=NULL;
  else
    if (lista->dato==valor)
     { borrado=lista;
       lista=borrado->sig;
       borrado->sig=NULL; }
    else
     {for(i=lista;i->sig!=NULL && valor!=(i->sig)->dato;i=i->sig);
      if (i->sig!=NULL)
       {borrado=i->sig;
        i->sig=borrado->sig;
        borrado->sig=NULL; }
      else
       borrado=NULL; }
  return borrado;
}
```

Extracción de lista vacía

Extracción del primer nodo

Extracción del medio o final

Valor no encontrado



Implementación (10)

- Operación *mostrar datos de la*

- *Mostrar lista*
  - Propósito: mostrar el contenido de la lista.
  - Entrada: una lista (puntero de inicio de la lista).
  - Salida: se muestran por pantalla los datos almacenados en los nodos.
  - Restricciones: una lista inicializada y no vacía.

```
void mostrar(pnodo lista)
{ pnodo i;
  if (lista!=NULL)
     for(i=lista;i!=NULL;i=i->sig)
     cout << "Nodo: " << i->dato << endl;
  else
     cout << "LISTA VACIA";
}
```

Se recorre la lista, nodo a nodo, hasta que el puntero *i* sea NULO

Implementación (11)

- *Buscar un dato en la lista*
  - Propósito: buscar un valor específico en la lista
  - Entrada: una lista y el valor a buscar.
  - Salida: valor true si el dato buscado se encuentra en la lista, caso contrario, false.
  - Restricciones: una lista inicializada y no vacía.

- Operación *buscar un dato en la*

```
bool buscar_nodo(pnodo lista, int valor)
{ pnodo i;
  bool encontrado=false;
  if (lista!=NULL)
    for(i=lista;i!=NULL && encontrado==false;i=i->sig)
       if (i->dato==valor)
         encontrado=true;
  return encontrado;
}
```

Se recorre la lista, nodo a nodo, hasta que el puntero *i* sea NULO o se detecte el valor buscado.

Implementación (12)

- TDA lista: Implementación del tipo nodo y del puntero a éste.

```
typedef struct tnodo *pnodo;
typedef struct tnodo{
                        int dato;
                        pnodo sig;
                       };
typedef struct tlista{
                        pnodo inicio;
                        pnodo final;
                       };
```

  - *inicio*: es el puntero que indica el **primer nodo** de la lista.
  - *final*: es el puntero que indica el **último nodo** de la lista.

Implementación (13)

- ¿Cómo se modifican las operaciones fundamentales al utilizar 2 punteros?
  - Se debe inicializar 2 punteros.
  - Al agregar el primer nodo deben actualizar ambos punteros.
  - Se simplifica la inserción de elementos al final de la lista.
  - Si la lista está ordenada, se simplifica la búsqueda de datos.
  - Al eliminar un nodo único se deben actualizar ambos punteros.
  - Al eliminar un nodo del final de la lista debe actualizarse el puntero correspondiente.

Implementación (14)

- Operación *iniciar lista*

```
void inicia_lista(tlista &lista)
{
      lista.inicio=NULL;
      lista.final=NULL;
}
```

SE GENERA UNA LISTA VACÍA

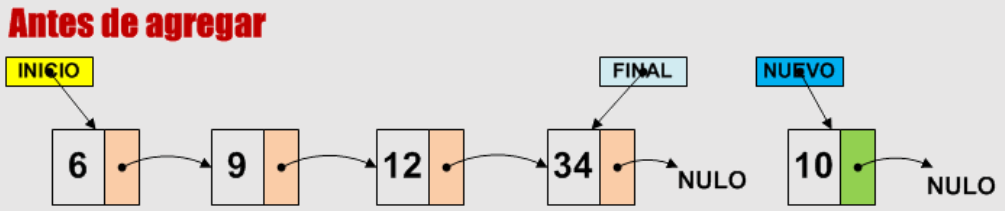
Implementación (15)

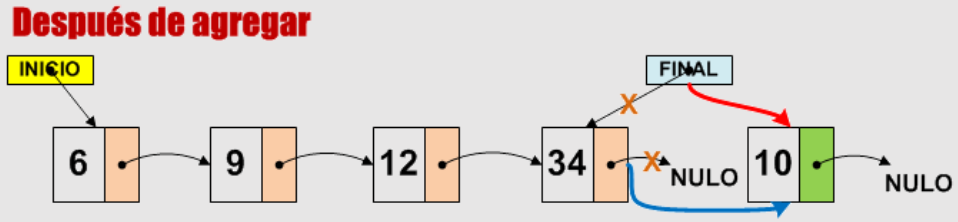
- Operación *agregar al final*

```
void agregar_final(tlista &lista,pnodo nuevo)
{
  if (lista.inicio==NULL)
     {lista.inicio=nuevo;
      lista.final=nuevo;
     }
   else
     {lista.final->sig=nuevo;
      lista.final=nuevo;
     }
}
```

Inserción en una lista vacía

Inserción al final de la lista





Implementación (16)

- Operación *quitar del final*

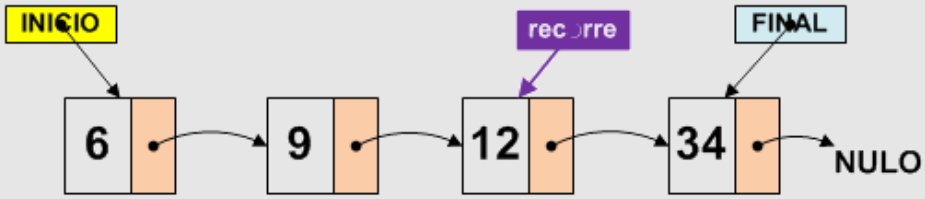
```
pnodo quitar_final(tlista &lista)
{ pnodo borrado,i;
  if (lista.inicio==NULL)
     borrado=NULL;
  else
   {if (lista.inicio==lista.final)
      { borrado=lista.inicio;
        lista.inicio=NULL;
        lista.final=NULL; }
    else
     { for(i=lista.inicio;(i->sig)->sig!=NULL;i=i->sig);
       borrado=lista.final;
       lista.final=i;
       lista.final->sig=NULL; }
   }
  return borrado;
}
```

Extracción de lista vacía

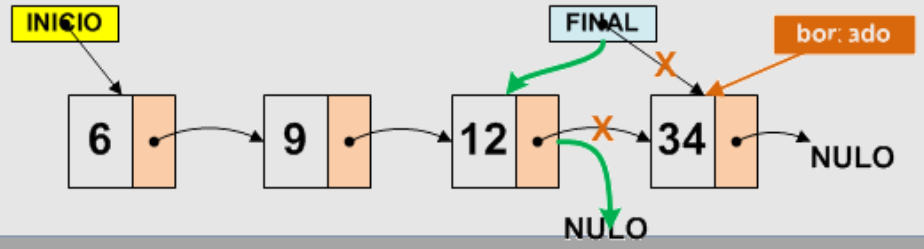
Extracción del único nodo

Extracción del último nodo

Antes de quitar último elemento



Después de quitar último elemento



Aplicaciones

- El TDA lista puede aplicarse para:
  - implementación de cualquier colección homogénea de elementos (por ejemplo, conjuntos).
  - implementación del TDA pila
  - implementación del TDA cola
- El TDA lista permite realizar una implementación dinámica que resulta útil cuando la variabilidad en la cantidad de elementos del problema es grande.
- El TDA lista permite implementar muchas de las estructuras utilizadas por los Sistemas Operativos.

Bibliografía

- Joyanes Aguilar *et al*. Estructuras de Datos en C++. Mc Graw Hill. 2007.
- De Giusti, Armando *et al.* Algoritmos, datos y programas, conceptos básicos. Editorial Exacta. 1998.
- Joyanes Aguilar, Luis. Fundamentos de Programación. Mc Graw Hill. 1996.
- Hernández, Roberto *et al*. Estructuras de Datos y Algoritmos. Prentice Hall. 2001.