



Taller de Programación



AGENDA



Estructuras de Datos vistas: arreglos - listas

Concepto de Ordenación



ARREGLOS - Características



Un **arreglo** es una estructura de datos compuesta que permite acceder a cada componente por una variable índice.

Dicho índice da la posición del componente dentro de la estructura de datos.

La estructura arreglo se almacena en posiciones contiguas de memoria

CARACTERISTICAS

Homogénea

Estática

Acceso directo

Indexada

Lineal

Dimensión física

Dimensión lógica



ARREGLOS - Características

Type

elem	elem	elem	elem		
------	------	------	------	--	--

```
arregloEntero array [array] of 20 of integer;
```

Var

```
v:arregloEntero;
```

OPERACIONES

- Agregar un elemento
- Insertar un elemento
- Eliminar un elemento
- Recorrer la estructura
- Buscar un elemento

Ordenar la estructura



LISTAS - Características



Una **lista** es una estructura de datos lineal compuesta por nodos.

Cada nodo de la lista posee el dato que almacena la lista y la dirección del siguiente nodo.

Toda lista puede recorrerse a partir de su primer elemento.

Los elementos no necesariamente están en posiciones contiguas de memoria.

Para generar nuevos elementos en la lista, o eliminar alguno se deben utilizar las operaciones de new y dispose respectivamente.

CARACTERÍSTICAS

Homogénea

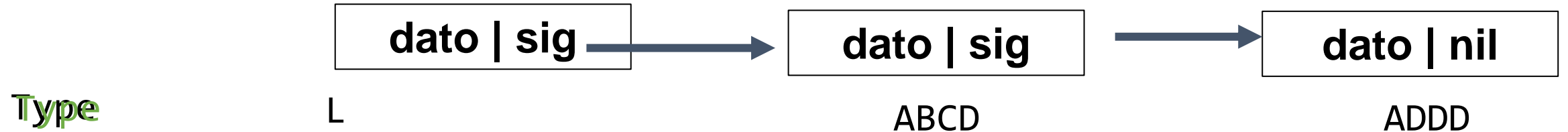
Dinámica

Acceso secuencial

Lineal



LISTAS - Características



```
lista = ^nodo;
```

```
nodo = record  
  dato: integer;  
  sig: lista;  
end;
```

```
Var  
  L: lista;
```

OPERACIONES

- Crear una lista vacía
- Agregar un elemento adelante
- Agregar un elemento atrás
- Insertar un elemento
- Eliminar un elemento
- Recorrer la estructura
- Buscar un elemento

Ordenar la estructura



ARREGLOS - Ordenación

Cuál sería el beneficio de tener una estructura ordenada?

23	1	100	4		
1	4	23	100		



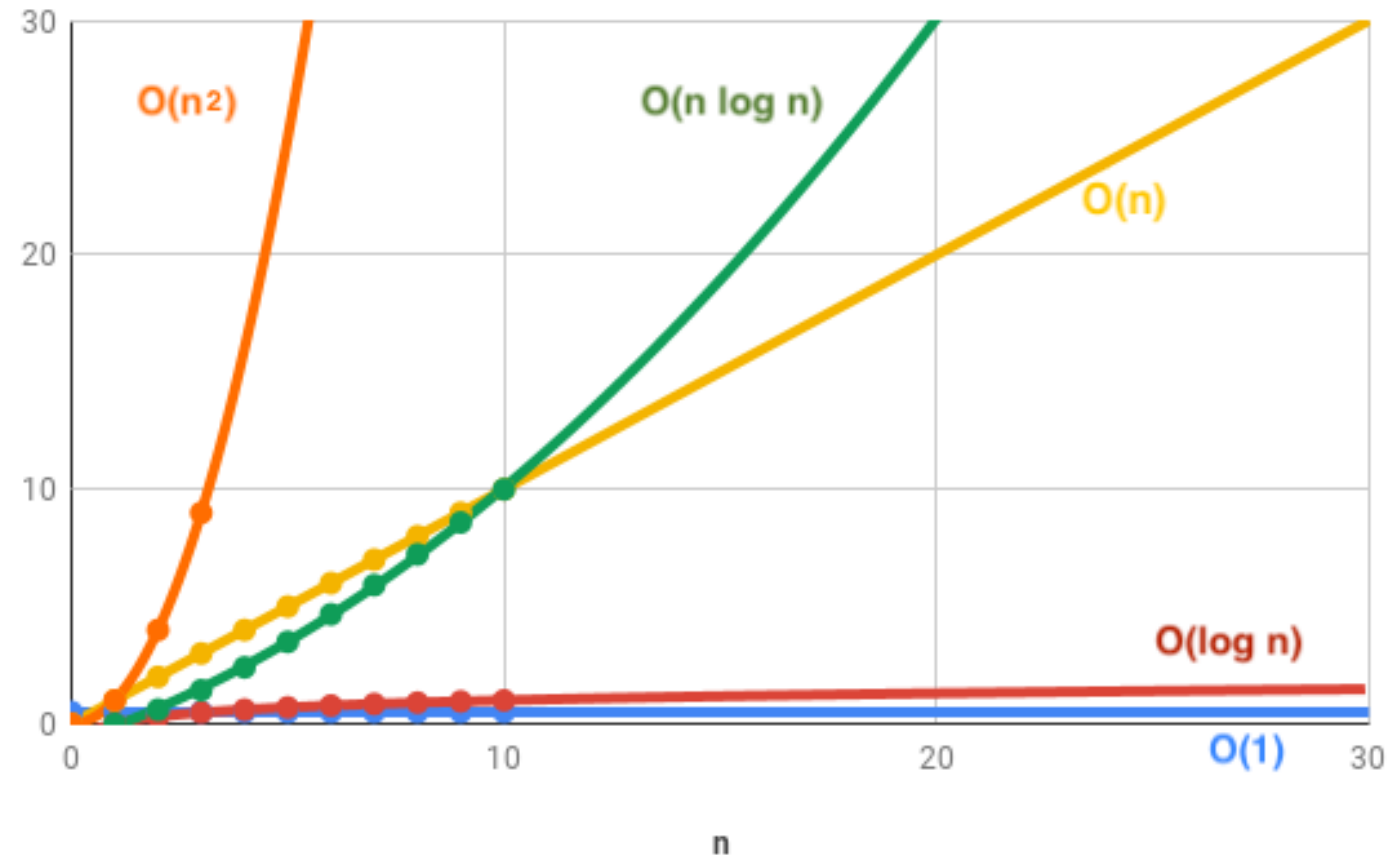
Un **algoritmo de ordenación** es un proceso por el cual un conjunto de elementos puede ser ordenado.

Existe una gran variedad de algoritmos para ordenar arreglos cada uno con características diferentes (facilidad de escritura, memoria utilizada, tiempo de ejecución)



ARREGLOS - Ordenación

ALGORITMO	ORDEN de EJECUCION
Selección	$O(N^2)$
Intercambio	$O(N^2)$
Inserción	$O(N^2)$
Heapsort	$O(N(\log N))$
Mergesort	$O(N(\log N))$
Quicksort	$O(N(\log N))$





ARREGLOS - Ordenación

CONSIDERACIONES al momento de implementar un algoritmo de Ordenación:

- Tiempo de ejecución.
- Facilidad para la escritura del mismo.
- Memoria utilizada en su ejecución.
- Complejidad de las estructuras auxiliares que necesite.
- Requiere el mismo tiempo si los datos ya están ordenados, si están al azar, si se encuentran en el orden exactamente inverso al que yo los quiero tener.

Selección - Inserción



Taller de Programación



AGENDA



Método de ordenación: selección



ARREGLOS – Ordenación - SELECCION

Este algortimo consta de N vueltas, donde N es la dimension lógica del arreglo.

En la **primera Vuelta**, se recorre todo el arreglo desde la posición 1 hasta el final (dimL) y se guarda en que posición se encuentra el elemento mas chico del arreglo. Al terminar el recorrido se intercambia el elemento de la posición 1 con el elemento ubicado en la posición en la cual se encontró el menor valor.

En la **segunda vuelta (ya se sabe, que en la posición 1 quedó ubicado el menor)**, y por lo tanto se busca a partir de la posición 2 hasta el final y se guarda en que posición se encuentra el elemento mas chico del arreglo. Al terminar el recorrido, se intercambia el elemento de la posición 2 con el elemento ubicado en la posición en la cual se encontró el menor valor.

Esto se repite hasta recorrer todo el arreglo, dando así un total de N vueltas (N= dimension logica del arreglo -1), ya que el último elemento en .la última vuelta ya está ubicado en su posición.



ARREGLOS – Ordenación - SELECCION

QUE NECESITAMOS CONOCER?

- Dimensión lógica del arreglo.
- Posición donde va el elemento mínimo.
- Rango en donde se busca el mínimo desde la que vamos a buscar el mínimo.
- Posición del elemento mínimo.

Selección



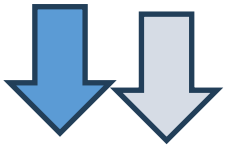


ARREGLOS – Ordenación - SELECCION

23	1	100	4	7	
----	---	-----	---	---	--

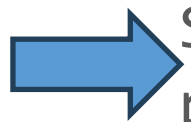
dimF = 6
dimL = 5

VUELTA 1



23	1	100	4	7	
----	---	-----	---	---	--

mínimo = 1
pos = 2



Se intercambia el elemento de la posición 1 (23) con el de la posición 2 (1).

1	23	100	4	7	
---	----	-----	---	---	--

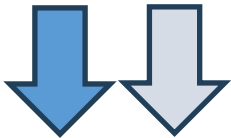


ARREGLOS – Ordenación - SELECCION

1	23	100	4	7	
---	----	-----	---	---	--

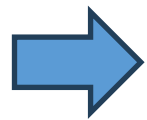
dimF = 6
dimL = 5

VUELTA 2



1	23	100	4	7	
---	----	-----	---	---	--

mínimo = 4
pos = 4



Se intercambia el elemento de la posición 2 (23) con el de la posición 4 (4).

1	4	100	23	7	
---	---	-----	----	---	--

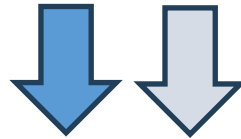


ARREGLOS – Ordenación - SELECCION

1	4	100	23	7	
---	---	-----	----	---	--

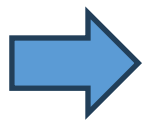
dimF = 6
dimL = 5

VUELTA 3



1	4	100	23	7	
---	---	-----	----	---	--

mínimo = 7
pos = 5



Se intercambia el elemento de la posición 3(100) con el de la posición 5 (7).

1	4	7	23	100	
---	---	---	----	-----	--

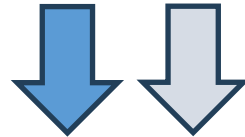


ARREGLOS – Ordenación - SELECCION

1	4	7	23	100	
---	---	---	----	-----	--

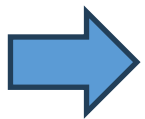
dimF = 6
dimL = 5

VUELTA 4



1	4	7	23	100	
---	---	---	----	-----	--

mínimo = 23
pos = 4



Se intercambia el elemento de la posición 4(23) con el de la posición 4 (23).

1	4	7	23	100	
---	---	---	----	-----	--



ARREGLOS – Ordenación - SELECCION

Program ordenar;

Const dimF =... *{máxima longitud del arreglo}*

Type

TipoElem = ... *{ tipo de datos del vector }*

Indice = 0.. dimF;

Tvector = **array** [1..dimF] **of** TipoElem;

Var

a:Tvector;

dimL:integer;

Begin

cargarVector (a, dimL);

seleccion (a, dimL);

End.



ARREGLOS – Ordenación - SELECCION

```
Procedure seleccion ( var v: tVector; dimLog: indice );  
  
var i, j, pos: indice; item : tipoElem;  
  
begin  
  for i:=1 to dimLog-1 do begin {busca el mínimo y guarda en pos la posición}  
    pos := i;  
    for j := i+1 to dimLog do  
      if v[ j ] < v[ pos ] then pos:=j;  
  
      {intercambia v[i] y v[p]}  
      item := v[ pos ];  
      v[ pos ] := v[ i ];  
      v[ i ] := item;  
    end;  
  end;  
end;
```



ARREGLOS – Ordenación - SELECCION

Program ordenar;

Const dimF = 200

Type

vectorEnteros = array [1..dimF] of integer;

Var

vE:vectorEnteros;

dimL:integer;

Begin

cargarVector (vE, dimL);

seleccion (vE, dimL);

End.



ARREGLOS – Ordenación - SELECCION

```
Procedure seleccion ( var v: vectorEnteros; dimLog: integer);  
  
var i, j, pos: integer; item : integer;  
  
begin  
  for i:=1 to (dimLog-1) do begin {busca el mínimo y guarda en pos la posición}  
    pos := i;  
    for j := i+1 to dimLog do  
      if v[ j ] < v[ pos ] then pos:=j;  
  
      {intercambia v[i] y v[p]}  
      item := v[ pos ];  
      v[ pos ] := v[ i ];  
      v[ i ] := item;  
    end;  
  end;  
end;
```



ARREGLOS – Ordenación - SELECCION

CONSIDERACIONES

- Tiempo de ejecución.
- Facilidad para la escritura del mismo.
- Memoria utilizada en su ejecución.
- Complejidad de las estructuras auxiliares que necesite.
- Requiere el mismo tiempo si los datos ya están ordenados, si están al azar, si se encuentran en el orden exactamente inverso al que yo los quiero tener.
- N^2 .
- Muy fácil.
- El arreglo y variables.
- No requiere
- Siempre requiere el mismo tiempo de ejecución.



Taller de Programación



AGENDA



Método de ordenación: inserción



ARREGLOS – Ordenación - INSERCION

Este algoritmo consta de N vueltas, donde N es la dimension lógica del arreglo. La idea general de este algoritmo es ir considerando subconjuntos de datos del vector e ir ordenándolos.

En la **primera Vuelta**, se trabaja el subconjunto formado por el primer elemento del arreglo que obviamente se considera ordenado.

En la **segunda vuelta**, se trabaja el subconjunto formado por el primer y segundo elemento del arreglo y se ordena ese subconjunto de manera de encontrar en que posición debe estar el segundo elemento para que el arreglo siga ordenado.

En la **tercera vuelta**, se trabaja el subconjunto formado por el primer, segundo y tercer elemento del arreglo y se ordena ese subconjunto de manera de encontrar en que posición debe estar el tercer elemento para que el arreglo siga ordenado (ya se sabe que el primero y segundo están ordenados).

En la **cuarta vuelta**, se trabaja el subconjunto formado por el primer, segundo, tercer y cuarto elemento del arreglo y se ordena ese subconjunto de manera de encontrar en que posición debe estar el cuarto elemento para que el arreglo siga ordenado (ya se sabe que el primero, el segundo y el tercero están ordenados).

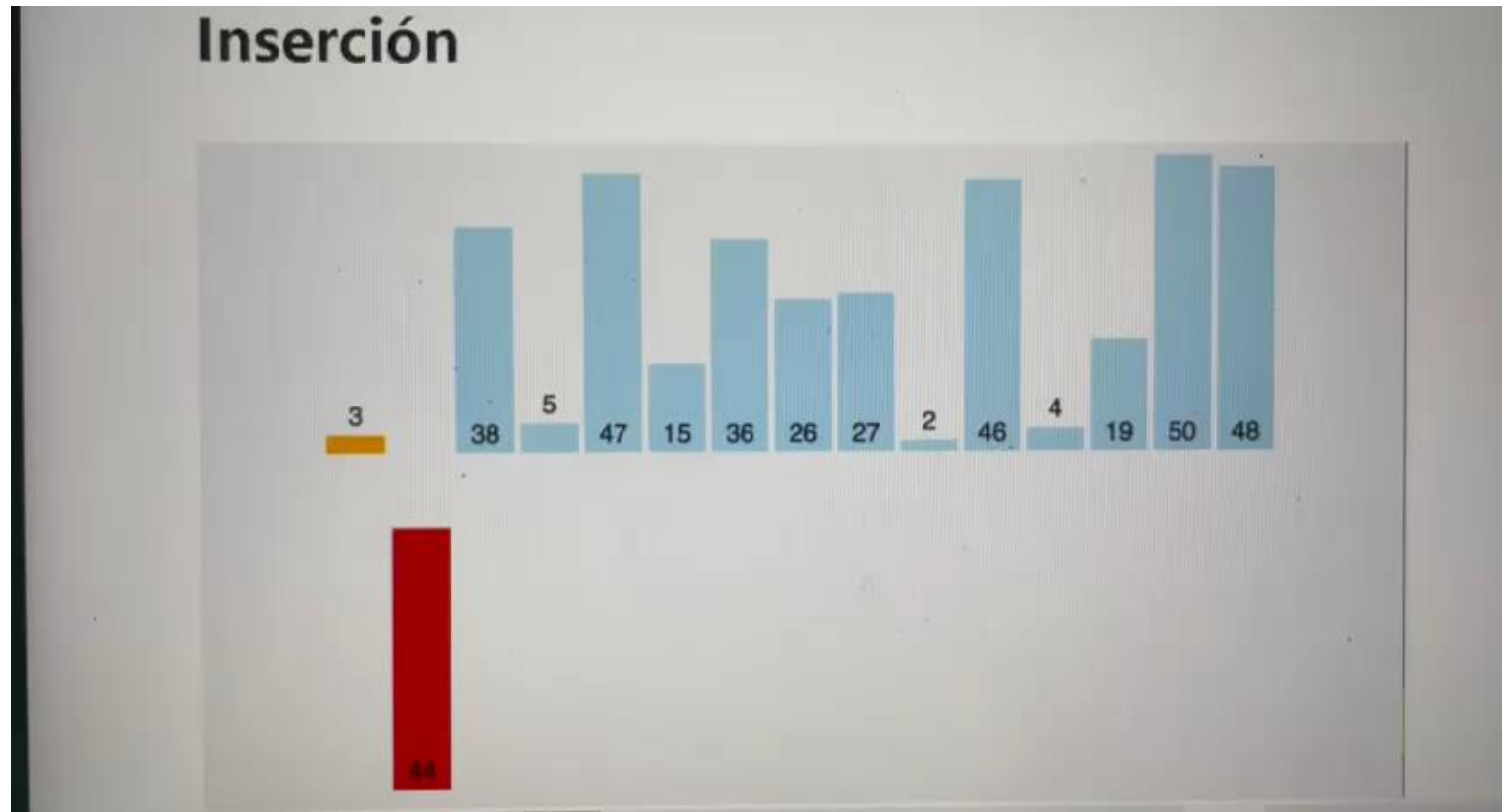
Esto se repite hasta recorrer todo el arreglo, dando así un total de N vueltas.



ARREGLOS – Ordenación - INSERCION

QUE NECESITAMOS CONOCER?

- Dimensión lógica del arreglo.
- Posición que se debe comparar
- Cuántos elementos ya se encuentran ordenados

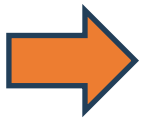




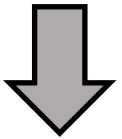
ARREGLOS – Ordenación - INSERCION

5	3	2	1	4	6	
---	---	---	---	---	---	--

VUELTA 1

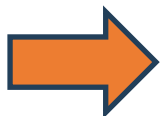


Tomo el elemento ubicado en la posición 2 (3) y se compara desde la posición 1 hasta la 1 para ver en qué posición debe insertarse.



5	3	2	1	4	6	
---	---	---	---	---	---	--

Debe insertarse
en la posición
1



Se encuentra que el valor debe insertarse en la posición 1, por lo tanto, se realiza un corrimiento desde la posición 1 hasta la 2 para “hacer” lugar en el vector.

5	3	2	1	4	6	
---	---	---	---	---	---	--

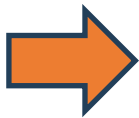


ARREGLOS – Ordenación - INSERCION

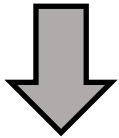
3	5	2	1	4	6	
---	---	---	---	---	---	--

$\text{dimF} = 7$
 $\text{dimL} = 6$

VUELTA 2

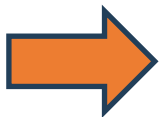


Tomo el elemento ubicado en la posición 3 (2) y se compara desde la posición 1 hasta la 2 para ver en qué posición debe insertarse.



3	5	2	1	4	6	
---	---	---	---	---	---	--

Debe insertarse
en la posición
1



Se encuentra que el valor debe insertarse en la posición 1, por lo tanto, se realiza un corrimiento desde la posición 1 hasta la 3 para “hacer” lugar en el vector.

3	5	2	1	4	6	
---	---	---	---	---	---	--

2

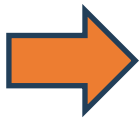


ARREGLOS – Ordenación - INSERCION

2	3	5	1	4	6	
---	---	---	---	---	---	--

$\text{dimF} = 7$
 $\text{dimL} = 6$

VUELTA 3

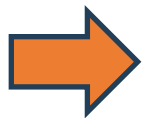


Tomo el elemento ubicado en la posición 4 (1) y se compara desde la posición 1 hasta la 3 para ver en qué posición debe insertarse.



2	3	5	1	4	6	
---	---	---	---	---	---	--

Debe insertarse
en la posición
1



Se encuentra que el valor debe insertarse en la posición 1, por lo tanto, se realiza un corrimiento desde la posición 1 hasta la 3 para “hacer” lugar en el vector.

2	3	5	1	4	6	
---	---	---	---	---	---	--



ARREGLOS – Ordenación - INSERCION

1	2	3	5	4	6	
---	---	---	---	---	---	--

$\text{dimF} = 7$
 $\text{dimL} = 6$

VUELTA 4  Tomo el elemento ubicado en la posición 5 (4) y se compara desde la posición 1 hasta la 4 para ver en qué posición debe insertarse.



1	2	3	5	4	6	
---	---	---	---	---	---	--

Debe insertarse
en la posición
4

 Se encuentra que el valor debe insertarse en la posición 4, por lo tanto, se realiza un corrimiento desde la posición 4 hasta la 5 para “hacer” lugar en el vector.

1	2	3	5	4	6	
---	---	---	---	---	---	--

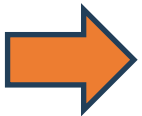


ARREGLOS – Ordenación - INSERCION

1	2	3	4	5	6	
---	---	---	---	---	---	--

$\text{dimF} = 7$
 $\text{dimL} = 6$

VUELTA 5

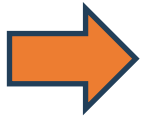


Tomo el elemento ubicado en la posición 6 (6) y se compara desde la posición 1 hasta la 5 para ver en qué posición debe insertarse.



1	2	3	4	5	6	
---	---	---	---	---	---	--

Debe insertarse
en la posición
6



Se encuentra que el valor debe insertarse en la posición 6, por lo tanto, se realiza un corrimiento desde la posición 6 hasta la 6 para “hacer” lugar en el vector.

1	2	3	4	5	6	
---	---	---	---	---	---	--



ARREGLOS – Ordenación - INSERCION

Program ordenar;

Const dimF =... *{máxima Longitud del arreglo}*

Type

TipoElem = ... *{ tipo de datos del vector }*

Indice = 0.. dimF;

Tvector = **array** [1..dimF] **of** TipoElem;

Var

a:Tvector;

dimL:integer;

Begin

cargarVector (a, dimL);

insercion (a, dimL);

End.



ARREGLOS – Ordenación - INSERCION

```
Procedure insercion ( var v: tVector; dimLog: indice );
```

```
Var
```

```
  i, j: indice; actual: tipoElem;
```

```
begin
```

```
  for i:= 2 to dimLog do begin
```

```
    actual:= v[i];
```

```
    j:= i-1;
```

```
    while (j > 0) y (v[j] > actual) do
```

```
      begin
```

```
        v[j+1]:= v[j];
```

```
        j:= j - 1;
```

```
      end;
```

```
    v[j+1]:= actual;
```

```
  end;
```

```
end;
```



ARREGLOS – Ordenación - INSERCION

Program ordenar;

Const dimF = 200

Type

Indice = 0.. dimF;

vectorEnteros = **array** [1..dimF] **of** integer;

Var

a:vectorEnteros;

dimL:integer;

Begin

cargarVector (a, dimL);

insercion (a, dimL);

End.



ARREGLOS – Ordenación - INSERCION

```
Procedure insercion ( var v: vectorEnteros; dimLog: indice );
```

```
Var
```

```
  i, j: indice; actual: integer;
```

```
begin
```

```
  for i:= 2 to dimLog do begin
```

```
    actual:= v[i];
```

```
    j:= i-1;
```

```
    while (j > 0) y (v[j] > actual) do
```

```
      begin
```

```
        v[j+1]:= v[j];
```

```
        j:= j - 1;
```

```
      end;
```

```
    v[j+1]:= actual;
```

```
  end;
```

```
end;
```



ARREGLOS – Ordenación - INSERCION

CONSIDERACIONES

- Tiempo de ejecución.
- Facilidad para la escritura del mismo.
- Memoria utilizada en su ejecución.
- Complejidad de las estructuras auxiliares que necesite.
- Requiere el mismo tiempo si los datos ya están ordenados, si están al azar, si se encuentran en el orden exactamente inverso al que yo los quiero tener.
- N^2
- No estan fácil de implementar.
- El arreglo y variables.
- No require.
- Si los datos están ordenados de menor a mayor el algoritmo solo hace comparaciones, por lo tanto, es de orden (N). Si los datos están ordenados de mayor a menor el algoritmo hace todas las comparaciones y todos los intercambios, por lo tanto, es de orden (N^2). comparaciones.



Taller de Programación



AGENDA



Recursión



AGENDA



Existen un conjunto de problemas que pueden resolverse siempre de la misma manera con la característica que el problema debe ir “achicandose” en cada instancia a resolver, hasta que en alguna instancia la solución es “trivial”.



La **recursividad** es una técnica de resolución de problemas que consiste en dividir un problema en instancias más pequeñas del mismo problema (también llamados subproblemas) hasta que obtengamos un subproblema lo suficientemente pequeño que tenga una solución trivial o directa.



Recursión - MOTIVACION



Contar las páginas de los siguientes libros

```
function cantidad (L:libros):integer;  
var  
    cantidadTotal:integer;  
  
Begin  
    cantidadTotal:=0;  
  
    mientras (hayaLibros)  
        cuentoPaginas de L  
        cantidadTotal:= cantidadTotal + cuentoPaginas  
  
    cantidad:= cantidadTotal;  
End;
```

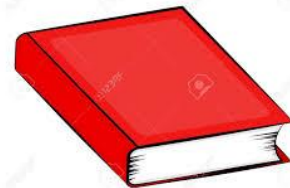
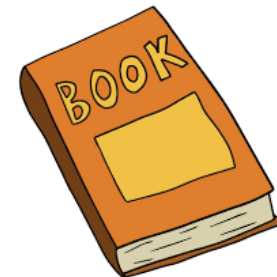
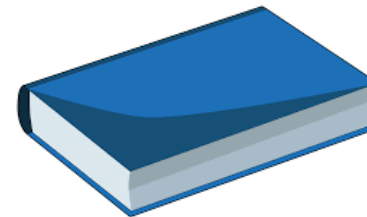
Cantidad Total = 100

30

15

55

10





Recursión - MOTIVACION



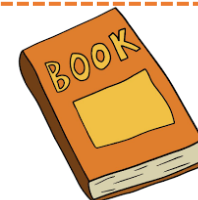
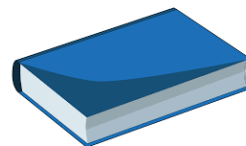
Contar las páginas de los siguientes libros

Cantidad Total = ~~1000~~

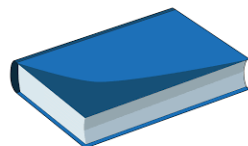


30

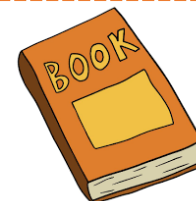
Cantidad Total



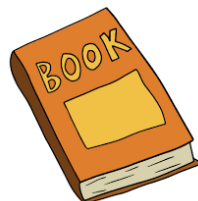
15



Cantidad Total

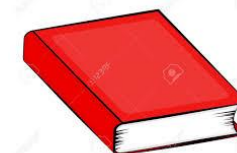
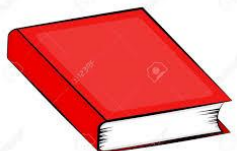


55



Cantidad Total

10

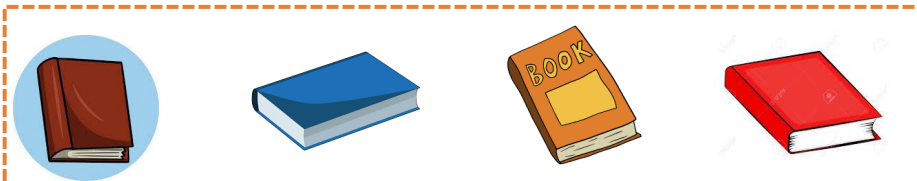
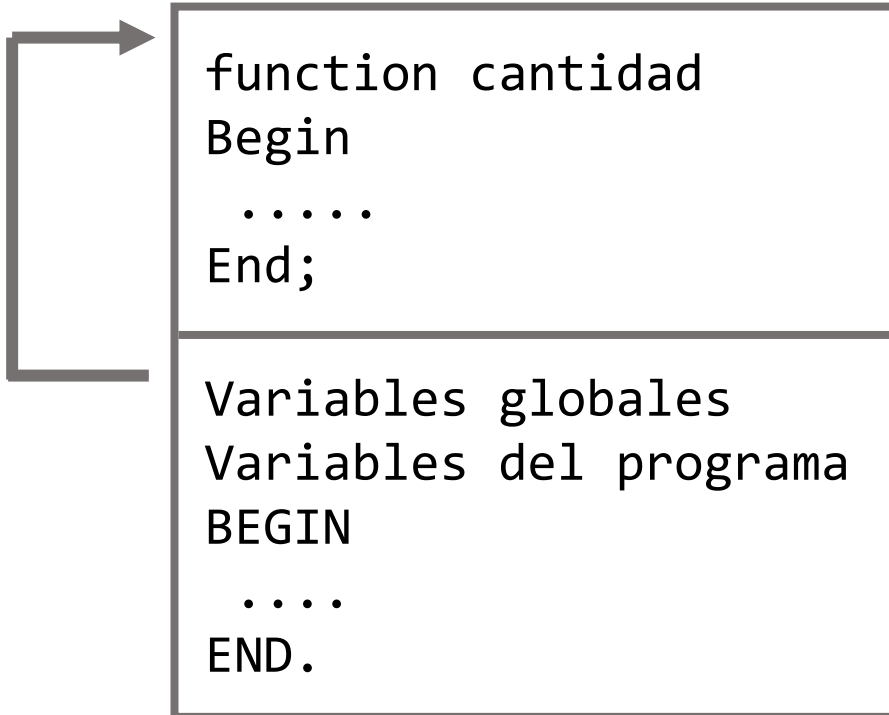




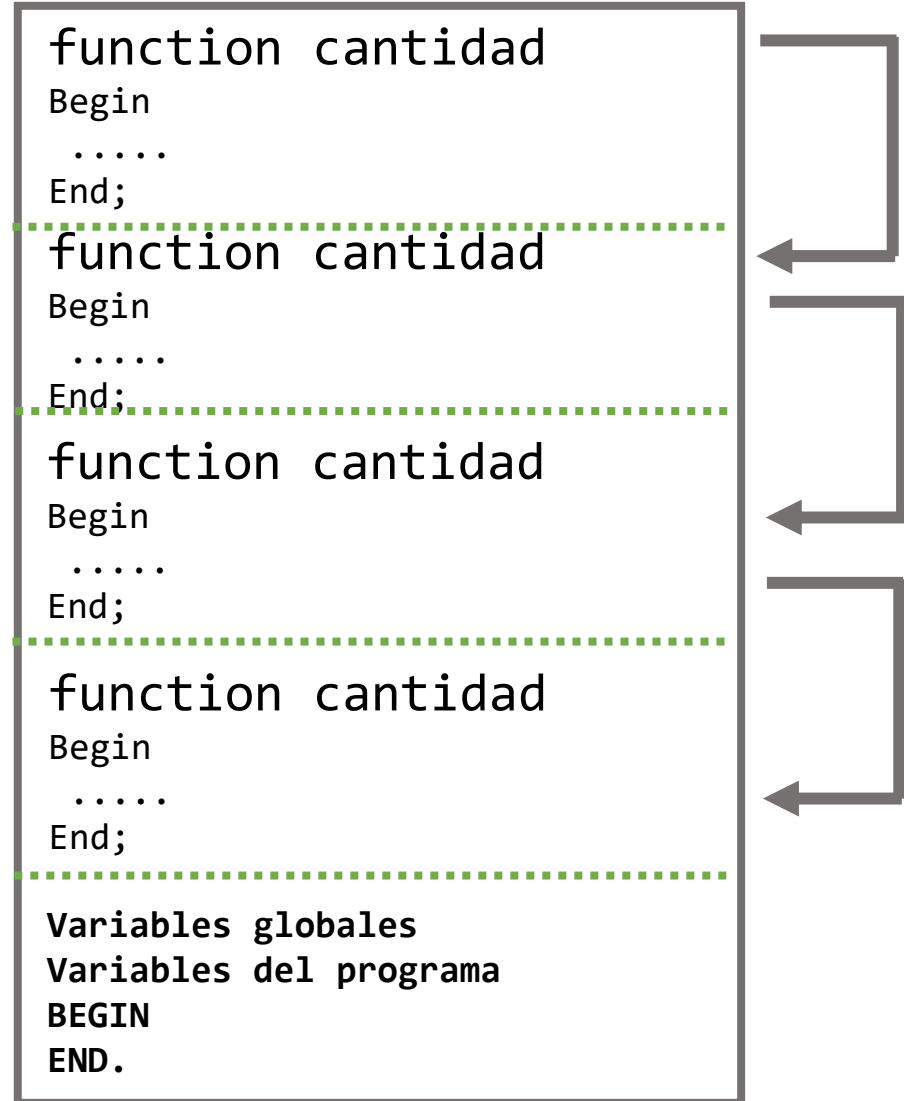
Recursión - MOTIVACION



ITERATIVA



RECURSIVA





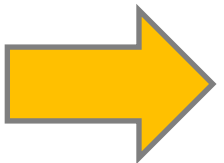
RECURSION



Existen un conjunto de problemas que pueden resolverse siempre de la misma manera con la característica que el problema debe ir “achicandose” en cada instancia a resolver, hasta que en alguna instancia la solución es “trivial”.



La **recursividad** es una técnica de resolución de problemas que consiste en dividir un problema en instancias más pequeñas del mismo problema (también llamados subproblemas) hasta que obtengamos un subproblema lo suficientemente pequeño que tenga una solución trivial o directa.



La recursividad consiste en resolver un problema por medio de un módulo (procedimientos o funciones) que se llama a sí mismo, evitando el uso de bucles y otros iteradores.

Cuando el problema se va achicando llega a un punto que no puede achicarse más, esa instancia se denomina **caso base**.

Hay problemas en los cuales debe realizarse alguna tarea cuando se alcanza el caso base y otros que no. Hay problemas que pueden tener más de un caso base.



Taller de Programación



AGENDA



Recursión

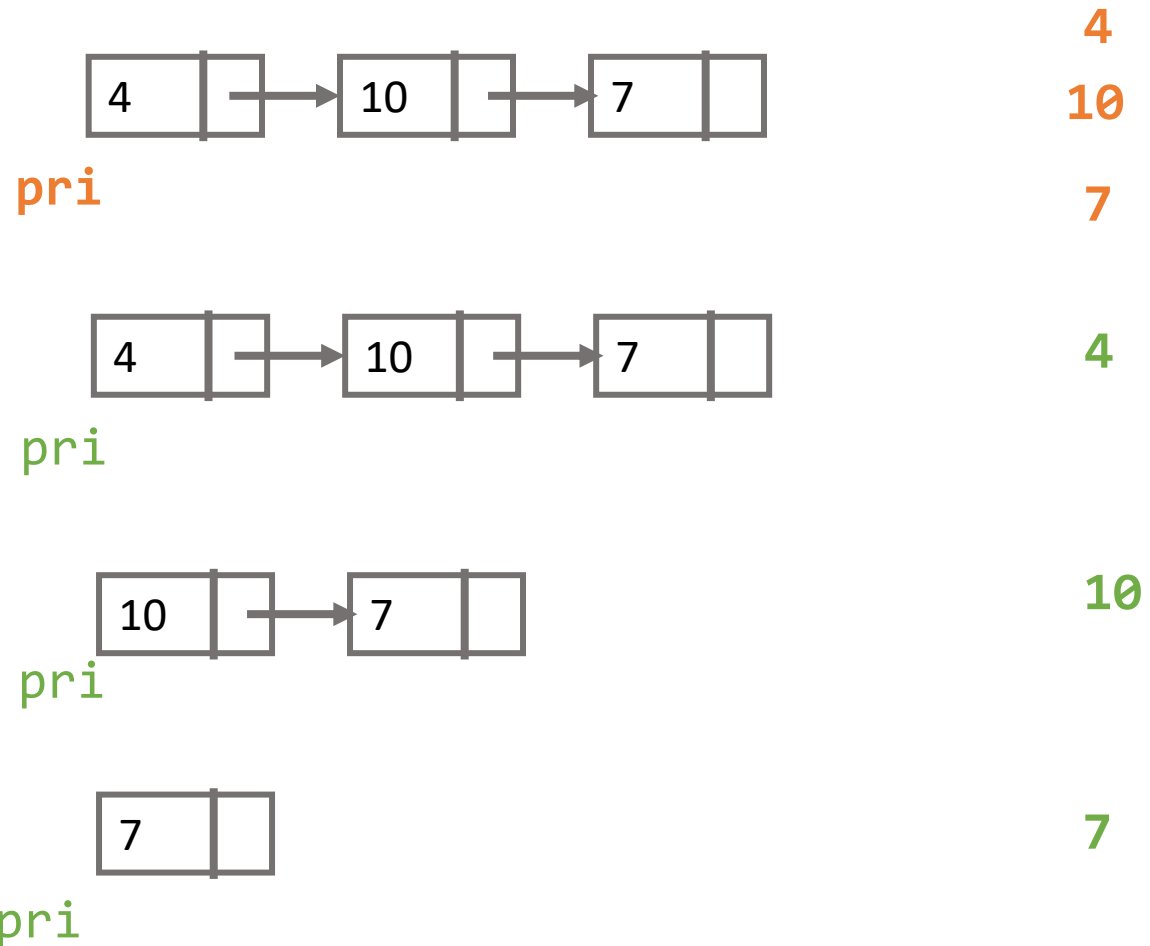


Recursión - MOTIVACION

Suponga que debe realizar un módulo que imprima una los elementos de una lista de enteros.

```
Procedure imprimir (pri:lista);  
Begin  
  while (pri <> nil) do  
    begin  
      write(pri^.dato);  
      pri:= pri^.sig;  
    end;  
End;
```

Se presenta el mismo problema cada vez más chico hasta llegar a una instancia que no se debe resolver nada





Recursión - MOTIVACION



Suponga que debe realizar un módulo que retorne el factorial de un número entero recibido. $\text{Fac}(n) = n * (n-1) * \dots * 1$ n veces

$$5 = 5 * 4 * 3 * 2 * 1 = 120$$

```
Procedure factorial (num:integer; var fac:integer);  
Var  
  i:integer;
```

```
Begin  
  fac:= 1;  
  for i:= num downto 1 do  
    begin  
      fac:= fac * i;  
    end;  
End;
```

Se presenta el mismo problema cada vez más chico hasta llegar a una instancia que se resuelve de manera directa

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$

$$\text{factorial}(4) = 4 * \text{factorial}(3)$$

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(2) = 2 * \text{factorial}(1)$$

$$\text{factorial}(1) = 1$$



RECURSIÓN - EJEMPLOS

Suponga que debe realizar un módulo que imprima los elementos de una lista de enteros que recibe como parámetro.

SOLUCIÓN ITERATIVA

```
Procedure imprimir (pri:lista);
Begin
  while (pri <> nil) do
    begin
      write (pri^.dato);
      pri:= pri^.sig;
    end;
  End;
```

Cómo achíco el problema?

Hasta cuando achíco el problema?

Qué hago cuando llego al caso base?

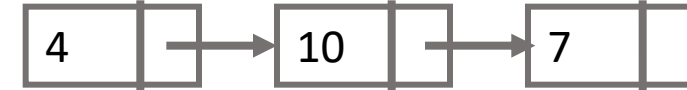
SOLUCIÓN RECURSIVA

```
Procedure imprimir (pri:lista);
Begin
  if (pri <> nil) do
    begin
      write (pri^.dato);
      pri:= pri^.sig;
      imprimir (pri);
    end;
  End;
```

Cómo funciona?



RECURSIÓN – EJEMPLOS – Cómo funciona?



```
Procedure imprimir (pri:lista);  
Begin  
  if (pri <> nil) then  
    begin  
      write (pri^.dato);  
      pri:= pri^.sig;  
      imprimir (pri);  
    end;  
  End;
```

*Cuál es la diferencia
con la solución
secuencial?*

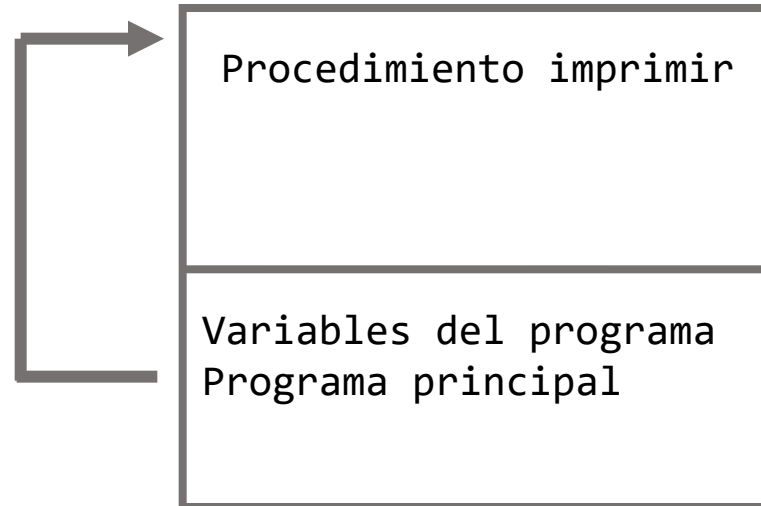
Procedimiento imprimir	pri= 4	4	3
Procedimiento imprimir	pri= 10	10	3
Procedimiento imprimir	pri= 7	7	3
Procedimiento imprimir	pri= nil	En este caso no se hace nada	
Variables del programa Programa principal			



RECURSIÓN – EJEMPLOS – Cómo funcionan?

SOLUCIÓN ITERATIVA

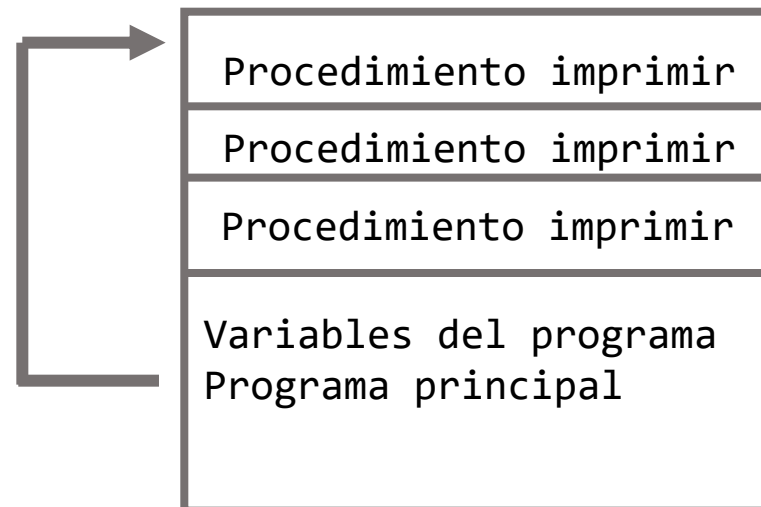
```
Procedure imprimir (pri:lista);  
Begin  
  while (pri <> nil) do  
    begin  
      write (pri^.dato);  
      pri:= pri^.sig;  
    end;  
  End;
```



Cuál cree que es más eficiente en cuanto al uso de la memoria?

SOLUCIÓN RECURSIVA

```
Procedure imprimir (pri:lista);  
Begin  
  IF (pri <> nil) then  
    begin  
      write (pri^.dato);  
      pri:= pri^.sig;  
      imprimir (pri);  
    end;  
  End;
```



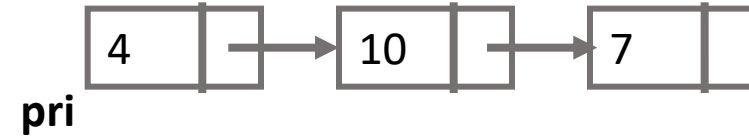
Qué pasa con los parámetros?




RECURSIÓN – EJEMPLOS – Cómo funciona?

SOLUCIÓN RECURSIVA

```
Procedure imprimir (pri:lista);
Begin
  if (pri <> nil) then
    begin
      write (pri^.dato);
      pri:= pri^.sig;
      imprimir (pri);
    end;
End;
```



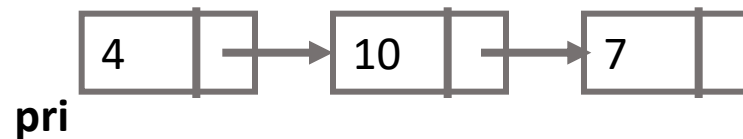
	Procedimiento imprimir	pri= 4	4	3
	Procedimiento imprimir	pri= 10	10	3
	Procedimiento imprimir	pri= 7	7	3
	Procedimiento imprimir	pri= nil	En este caso no se hace nada	
	Variables del programa Programa principal pri=4			

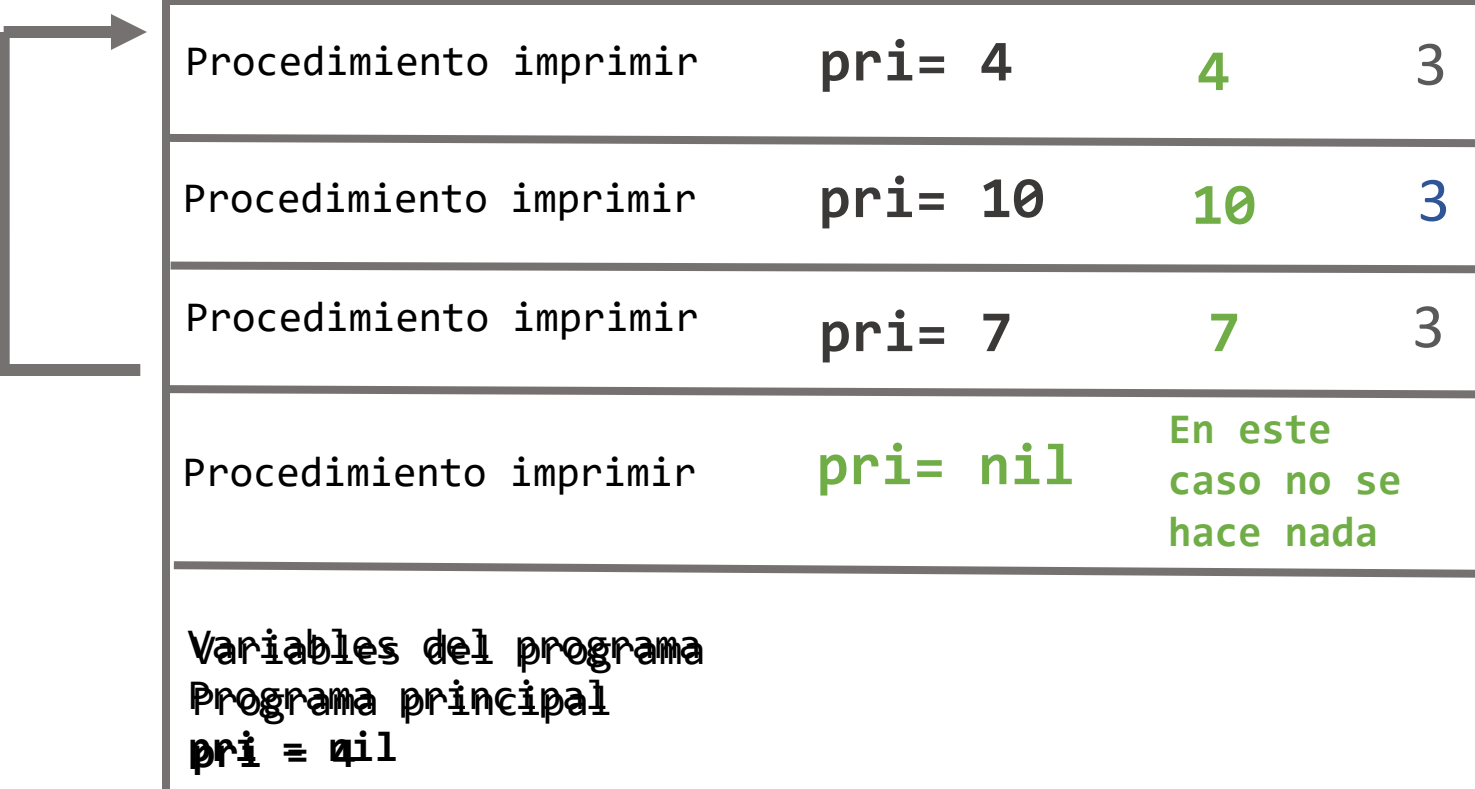


RECURSIÓN – EJEMPLOS – Cómo funciona?

SOLUCIÓN RECURSIVA

```
Procedure imprimir (VAR pri:lista);
Begin
  IF (pri <> nil) then
    begin
      write (pri^.dato);
      pri:= pri^.sig;
      imprimir (pri);
    end;
End;
```



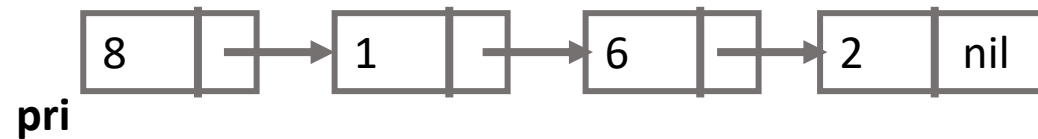


Procedimiento imprimir	pri= 4	4	3
Procedimiento imprimir	pri= 10	10	3
Procedimiento imprimir	pri= 7	7	3
Procedimiento imprimir	pri= nil	En este caso no se hace nada	
Variables del programa Programa principal pri = nil			



RECURSIÓN – EJEMPLOS - DESAFIO

Dada la siguiente lista que cree que imprime cada modulo?.



SOLUCIÓN UNO

```
Procedure imprimir (pri:lista);
Begin
  if (pri <> nil) do
    begin
      write (pri^.dato);
      pri:= pri^.sig;
      imprimir (pri);
    end;
End;
```

SOLUCIÓN DOS

```
Procedure imprimir (VAR pri:lista);
Begin
  if (pri <> nil) do
    begin
      write (pri^.dato);
      pri:= pri^.sig;
      imprimir (pri);
    end;
End;
```

SOLUCIÓN TRES

```
Procedure imprimir (pri:lista);
Begin
  if (pri <> nil) do
    begin
      pri:= pri^.sig;
      imprimir (pri);
      write (pri^.dato);
    end;
End;
```



RECURSIÓN - EJEMPLOS

Suponga que debe realizar un módulo que calcular la potencia de un número x a la n , que es $= x^n = x * x * x$ (n veces).

SOLUCIÓN ITERATIVA

```
Procedure potencia (x,n:integer;  
                  var pot:integer);
```

```
Var  
  i:integer;  
Begin  
  if (n = 0) then pot:= 1  
  else if (n = 1) then pot:= x  
  else begin  
    pot:= 1;  
    for i:= 1 to n do  
      pot:= pot * x;  
    end;  
  End;
```

Con una
función?

Cómo lo
pienso
recursivo?

SOLUCIÓN ITERATIVA

```
Function potencia (x,n:integer):integer;  
Var
```

```
  i,pot:integer;  
Begin  
  if (n = 0) then pot:= 1  
  else if (n = 1) then pot:= x  
  else begin  
    pot:= 1;  
    for i:= 1 to n do  
      pot:= pot * x;  
    end;  
  potencia:=pot;  
End;
```



RECURSIÓN - EJEMPLOS

SOLUCIÓN ITERATIVA

```
Function potencia (x,n:integer):integer;  
Var  
  i,pot:integer;  
Begin  
  if (n = 0) then pot:= 1  
  else if (n = 1) then pot:= x  
  else begin  
    pot:= 1;  
    for i:= 1 to n do  
      pot:= pot * x;  
    end;  
    potencia:=pot;  
  End;
```

SOLUCIÓN RECURSIVA

```
Function potencia (x,n:integer):integer;  
Begin  
  if (n = 0) then potencia:= 1  
  
  else if (n = 1) then potencia:= x  
  
  else  
    potencia:= x * potencia(x, n-1));  
  end;  
End;
```

Cuántos casos
base hay?

Cómo
funciona?



RECURSIÓN - Características

SOLUCIÓN RECURSIVA

```
Function potencia (x,n:integer);
```

```
Begin
```

```
  if (n = 0) then potencia:= 1
```

```
  else if (n = 1) then potencia:= x
```

```
  else
```

```
    potencia:= x * potencia(x, n-1));
```

```
  end;
```

```
End;
```

Supongamos $x = 4$ $n=3$



potencia x= 4,n=3

4 * **potencia** (4,2)

164

potencia x= 4,n=2

4 * **potencia** (4,1)

potencia x= 4,n=1

4

Alguna vez entrará
por el caso (n=0)?



Taller de Programación



AGENDA



Estructura de datos arbol

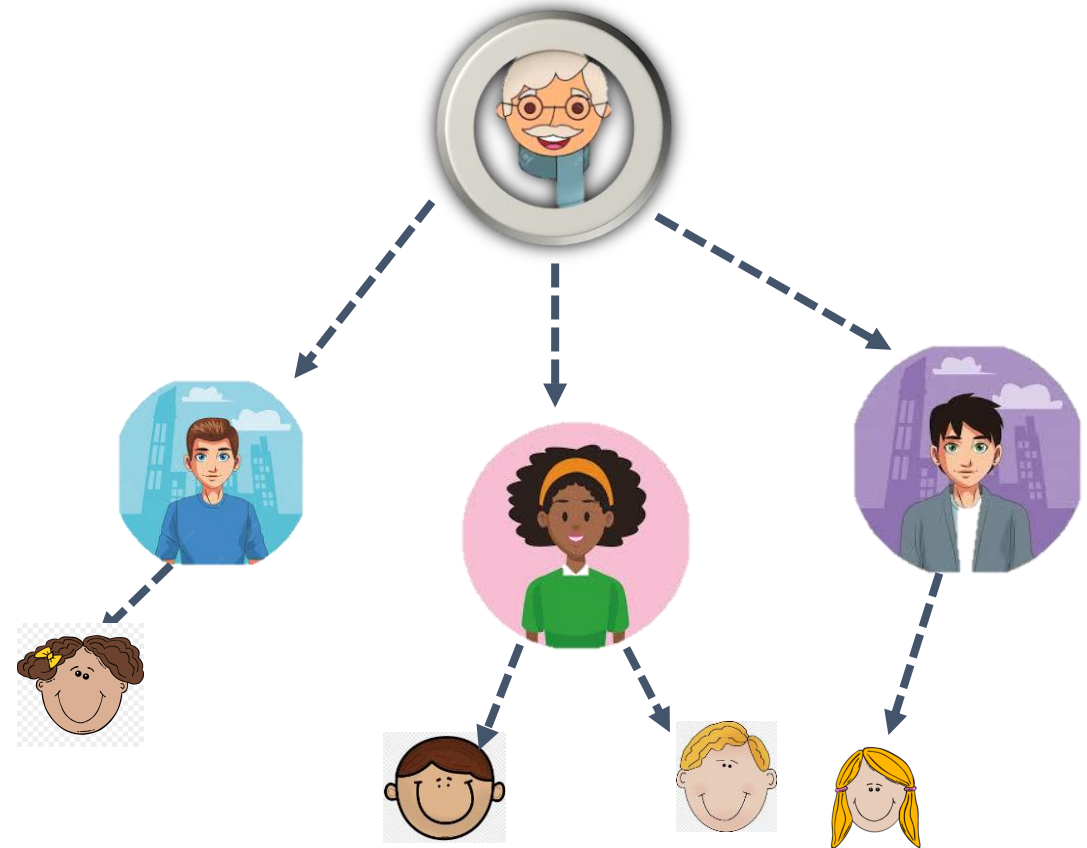


MOTIVACION

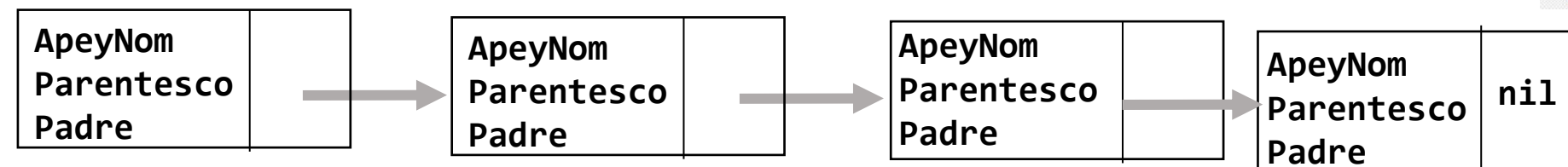
Supongamos que queremos representar el árbol genealógico de una familia a partir de un integrante (por ejemplo un abuelo).

El abuelo tiene hijos y a su vez esos hijos también pueden tener hijos (nietos del abuelo).

Para simplificar supongamos que cada individuo de la familia puede tener a lo sumo 2 hijos.



Vector → IMPOSIBLE



Pri

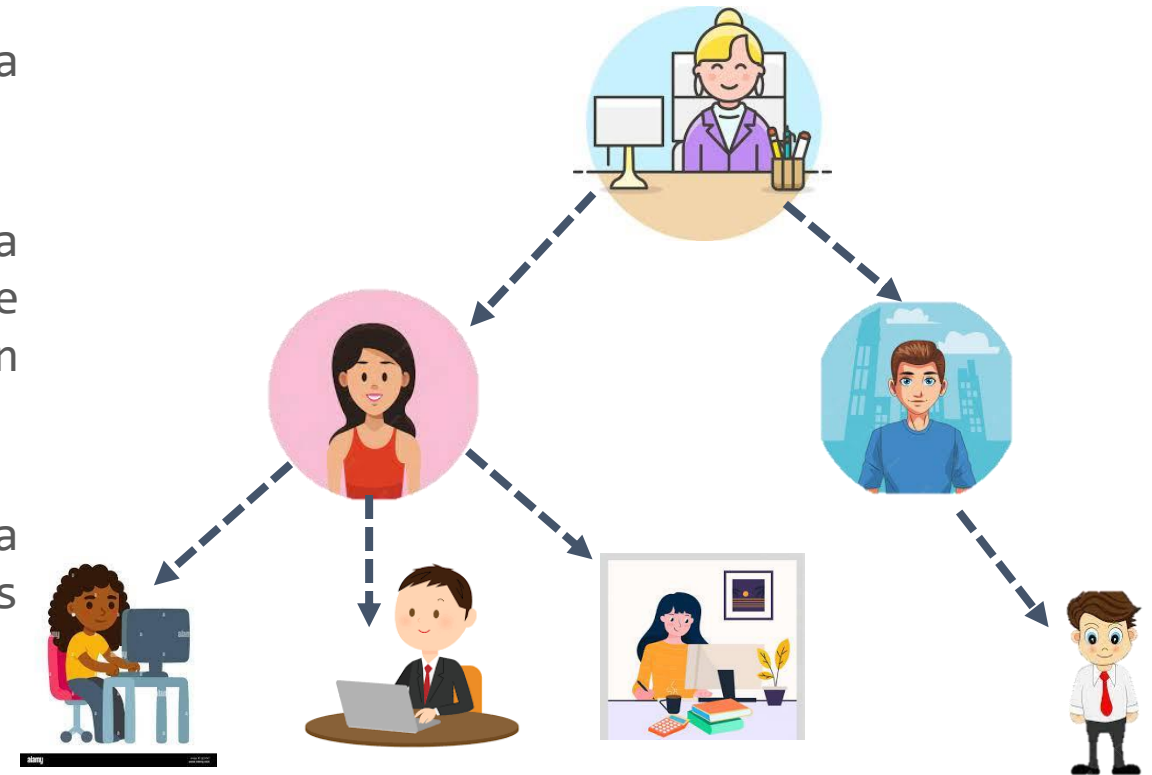


MOTIVACION

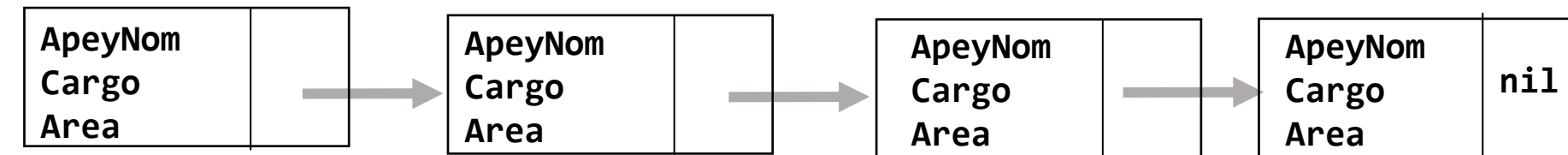
Supongamos que queremos representar la organización de una empresa.

La empresa tiene un área generencial (con una jefa), la cual está compuesta por varias áreas de trabajo (área de personal, área contable, etc con diferentes personas a cargo).

Cada una de estas áreas a su vez también podría estar compuesta por alguna/s subárea (con varias personas a cargo).



Vector → IMPOSIBLE

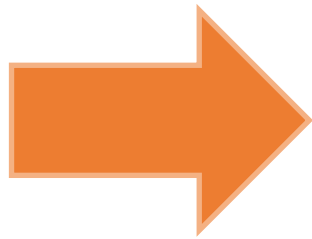


Pri



MOTIVACION

Por todo lo mencionado es importante notar que existen un conjunto de problemas que necesitan expresarse de una manera jerárquica característica que no permiten las estructuras vistas hasta el momento (arreglos y listas).



Dinámica
Homogénea
NO lineal

ARBOL



Taller de Programación



AGENDA



Estructura de datos arbol

Arboles binarios de búsqueda



ESTRUCTURA DE DATOS ARBOL

Por todo lo mencionado es importante notar que existen un conjunto de problemas que necesitan expresarse de una manera jerárquica característica que no permiten las estructuras vistas hasta el momento (arreglos y listas).



ARBOL

Es una estructura de datos jerárquica.

Está formada por nodos.

El nodo principal del árbol se denomina raíz y los nodos que no tienen hijos se denominan hojas del árbol.

Compuesta

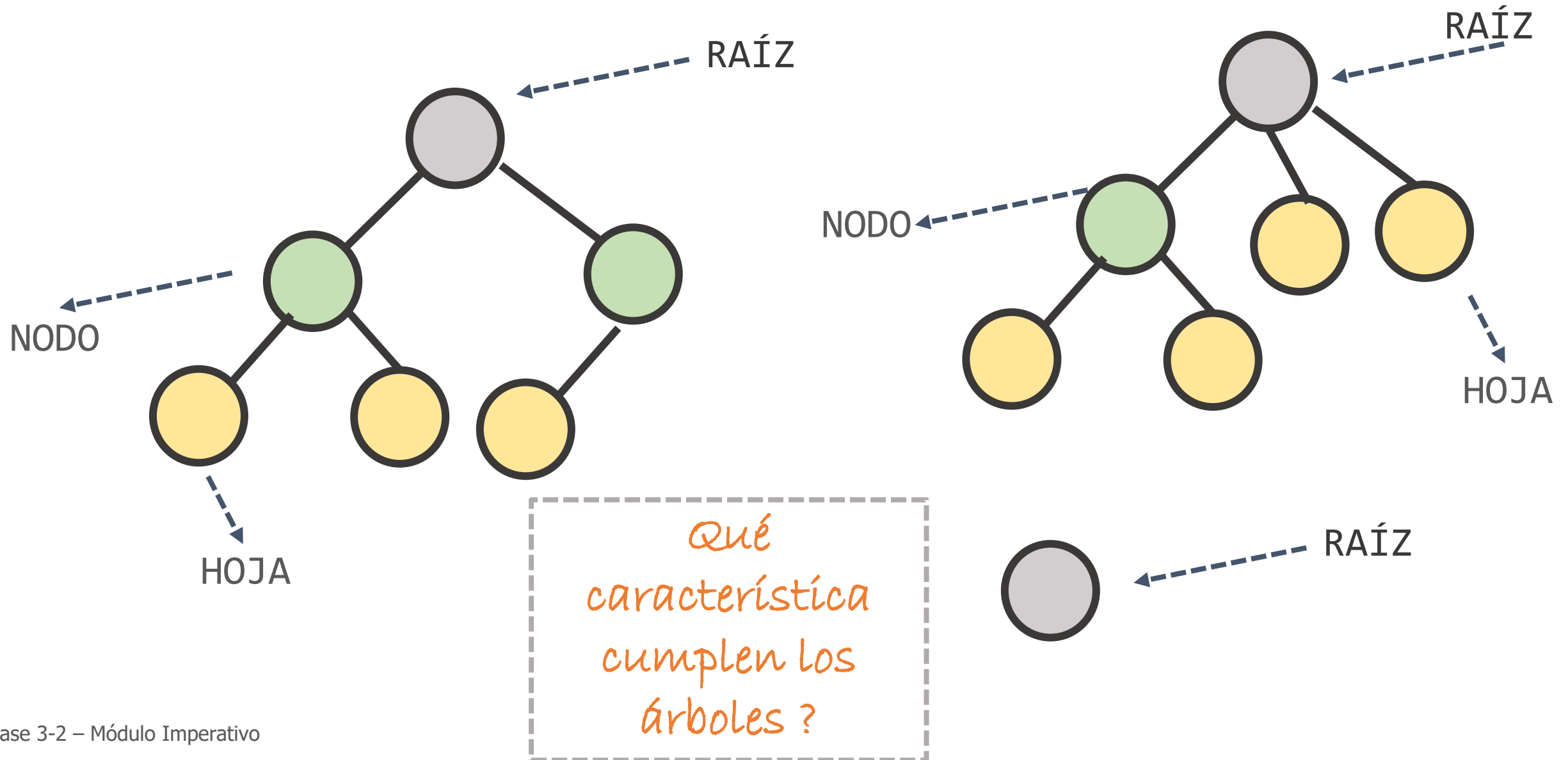
Dinámica

Homogénea

NO lineal



ESTRUCTURA DE DATOS ARBOL





ÁRBOLES - Características

Todo árbol es una estructura jerárquica

Todo árbol es una estructura dinámica

Todo árbol es una estructura homogénea

Un árbol vacío se representa con el valor nil

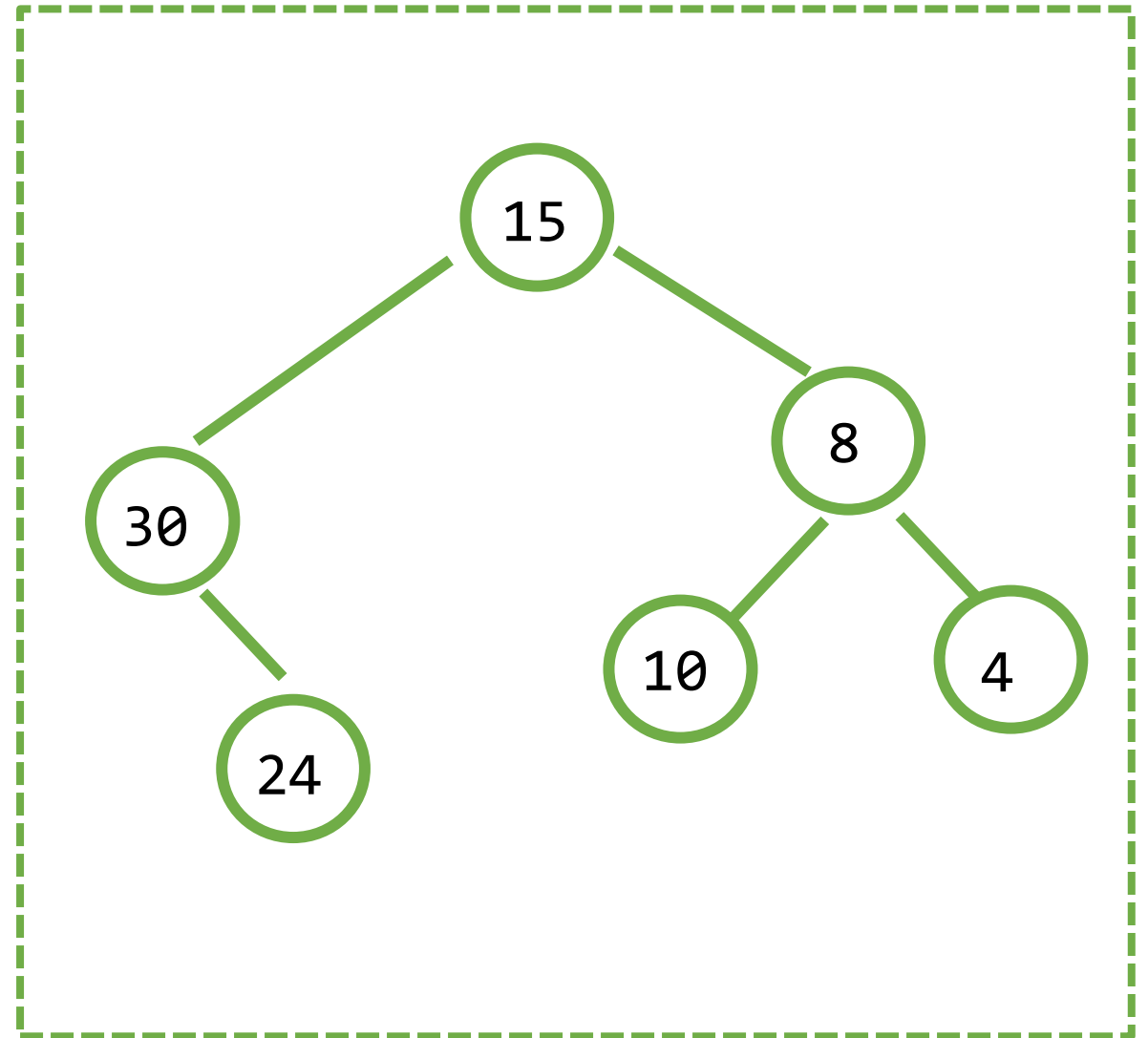
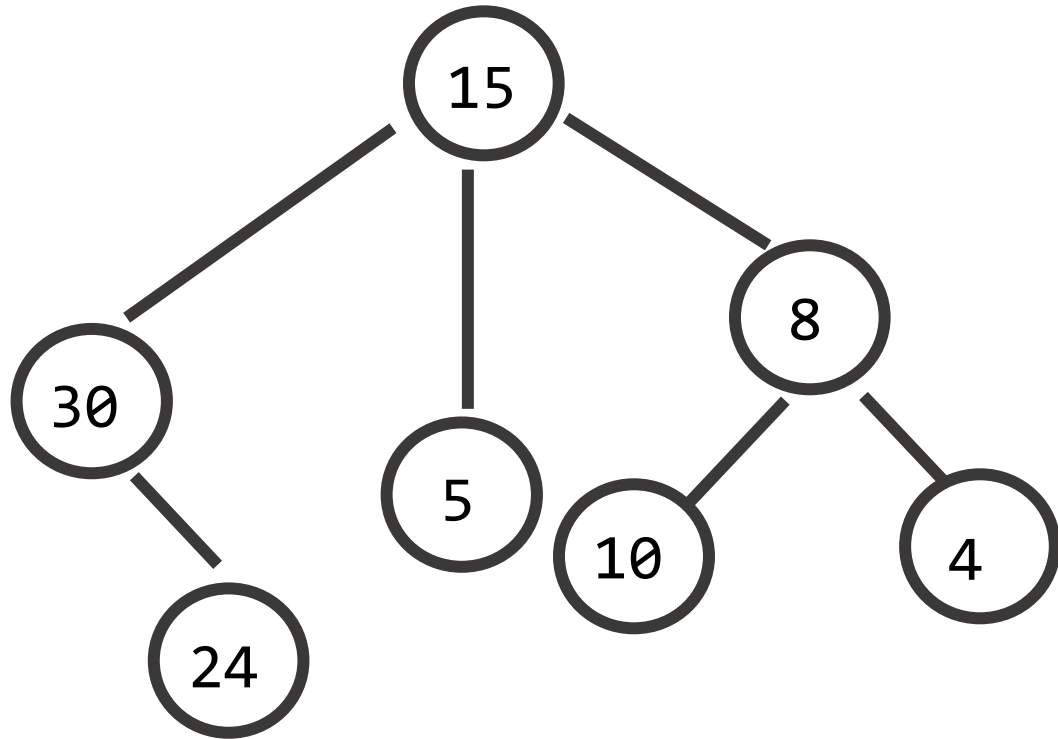
Un nuevo dato siempre se inserta como una hoja

Cómo se declara?

*Con qué árboles
vamos a
trabajar?*



ÁRBOLES – Con cual trabajaremos?





ÁRBOLES BINARIOS - Declaración

Programa arboles;

Type

arbol = ^nodo;

nodo = record

dato: tipodeElemento;

HI: arbol;

HD: arbol;

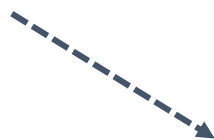
end;

}

}



elemento del árbol



Cada nodo
puede tener a
lo sumo dos
hijos

Var

a:arbol;

Begin

... •

End.



ÁRBOLES BINARIOS - Declaración

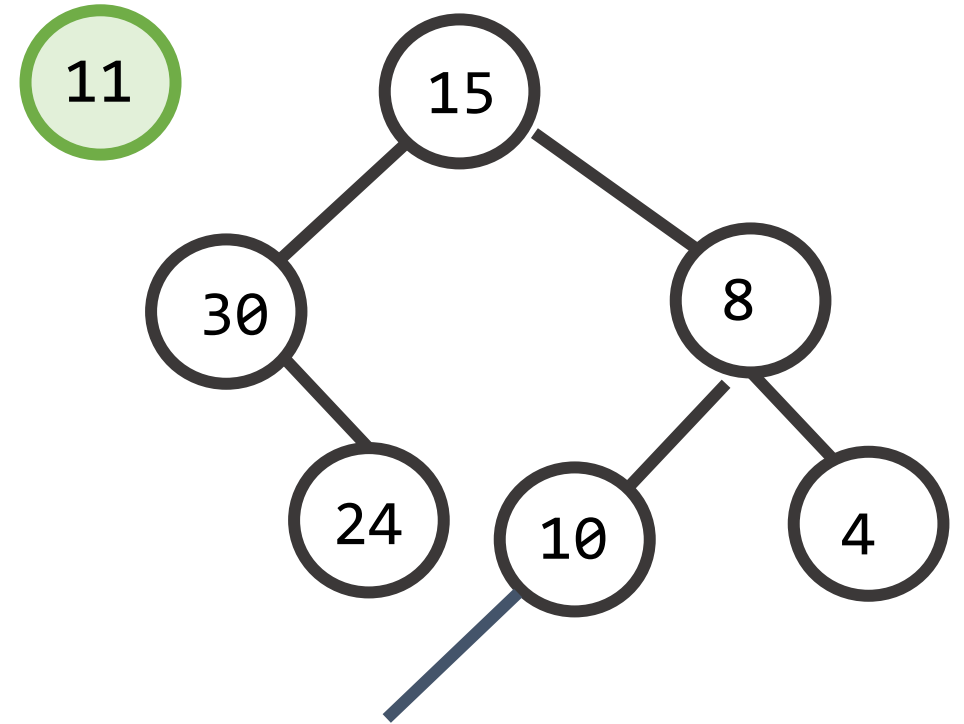
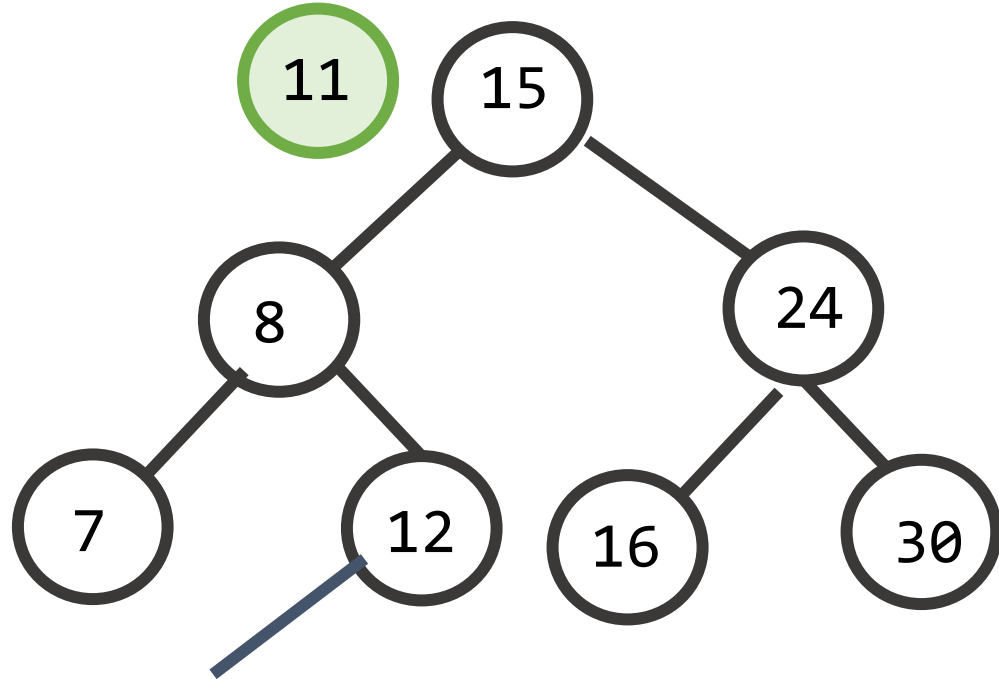
```
Programa arbolesEnteros;  
Type  
  arbol = ^nodo;  
  nodo = record  
    dato: integer;  
    HI: arbol;  
    HD: arbol;  
  end;  
  
Var  
  a:arbol;  
Begin  
  ...  
End.
```

*Qué característica
cumplen los
árboles BB?*

```
Programa arbolesPersonas;  
Type  
  persona = record  
    nombre:string;  
    dni:integer;  
  end;  
  arbol = ^nodo;  
  nodo = record  
    dato: persona;  
    HI: arbol;  
    HD: arbol;  
  end;  
  
Var  
  a:arbol;  
Begin  
  ...  
End.
```



ÁRBOLES BINARIOS DE BÚSQUEDA- Característica



Un **árbol binario de búsqueda** (ABB) agrega los elementos por sus hojas.

Dichos elementos quedan ordenados (todos por el mismo criterio). Esta operación lleva un tiempo de ejecución de $O(\log n)$.



Taller de Programación



AGENDA



Estructura de datos arbol de búsqueda

Operaciones – CREACION – CARGA DE DATOS



ÁRBOLES BINARIOS DE BÚSQUEDA- CREACION

Programa arboles;

Type

```
arbol = ^nodo;  
nodo = record  
    dato: integer;  
    HI: arbol;  
    HD: arbol;  
end;
```

Var

```
a:arbol;  
num:integer;
```

Begin

...

End.

Begin

```
a:= nil; //indico que el árbol está vacío
```

```
read (num); //leo un valor  
while (num <> 50) do
```

```
begin
```

```
    agregar (a,num); //agrego el valor al arbol
```

```
    read (num);
```

```
end;
```

End..

Suponga que se leen los siguientes valores y se quieren ir agregando en a (9, 18, 22, 19, 7,50). Cómo quedarán guardados?



ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB
(9, 18, 22, 19, 7, 50)

Programa arboles;

Type

```
arbol = ^nodo;  
nodo = record  
    dato: integer;  
    HI: arbol;  
    HD: arbol;  
end;
```

Var

a:arbol; **num:integer;**

Begin

```
a:= nil;  
read (num);  
while (num <> 50) do  
    begin  
        agregar (a,num);  
        read (num);  
    end;
```

End.

*Cómo quedarán
guardados los valores en
el ABB?*

a = nil

Se **lee el valor 9 (num)** y se invoca
al procedimiento agregar (a,num)

Como a = nil, el primer valor leído será la
raíz del arbol.

Para agregarlo al ser una estructura dinámica
debe reservarse memoria.

Luego se asigna en el campo dato el valor de
num, y como por ahora este nodo no tiene hijos,
en los campos HI e HD debe asignarse nil

El procedimiento agregar termina y vuelve al
programa principal en donde **a** ahora apunta a un
nodo con valor 9 y sus hijos en nil.





ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)

Programa arboles;

Type

```
arbol = ^nodo;  
nodo = record  
    dato: integer;  
    HI: arbol;  
    HD: arbol;  
end;
```

Var

```
a:arbol; num:integer;
```

Begin

```
a:= nil;  
read (num);  
while (num <> 50) do  
    begin  
        agregar (a,num);  
        read (num);  
    end;
```

End.

a = 9



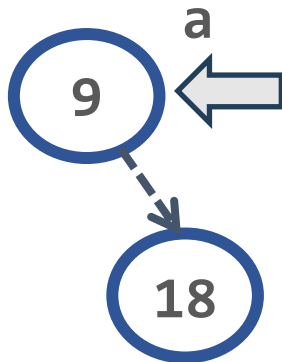
^a Se lee el valor 18 (num) y se invoca al procedimiento agregar (a,num)

Como el árbol NO es vacío, tengo que recorrer desde la raíz hasta el lugar correspondiente respetando el orden. Siempre se inserta en una hoja.

Comparo num (18) con lo que está apuntado a (9), como $18 > 9$ se determina que hay que agregarlo a la derecha de 9.

Como HD de 9 es =nil, ya se encontró el lugar (hoja), reservo memoria dinámica y asigno los valores correspondientes

El procedimiento agregar termina y vuelve al programa principal en donde a ahora apunta a un nodo con valor 9 y su HI= nil y HD = 18.

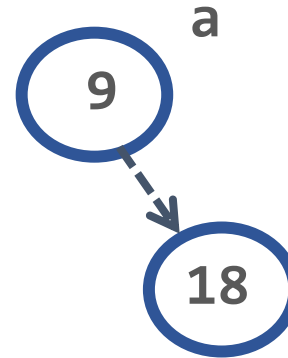




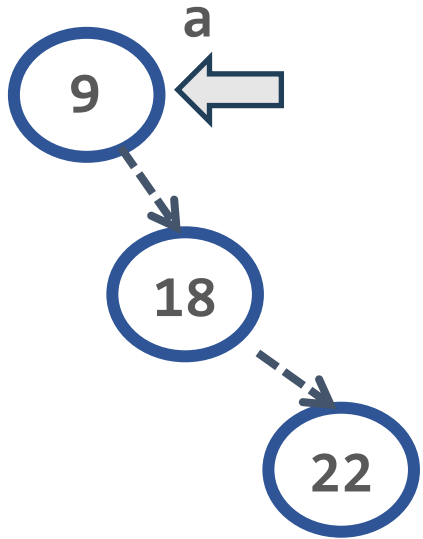
ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)

$a = 9$



Se lee el valor 22 (num) y se invoca al procedimiento agregar (a,num)



Como el árbol $a \neq \text{nil}$, tengo que recorrer desde la raíz hasta el lugar correspondiente respetando el orden. Siempre se inserta en una hoja.

Comparo num (22) con lo que está apuntado a (9), $22 > 9$ se determina que hay que agregarlo a la derecha de 9. Como HD de 9 $\neq \text{nil}$, sigo recorriendo hacia la derecha. Luego se compara y $22 > 18$ y como HD 18 = nil se encontró el lugar donde agregar el 22.

Reservo memoria dinámica y asigno los valores correspondientes

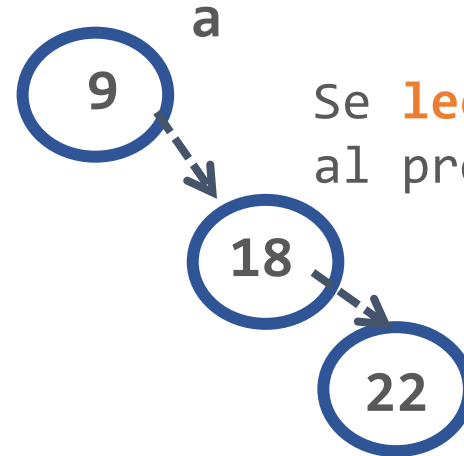
El procedimiento agregar termina y vuelve al programa principal en donde a ahora apunta a un nodo con valor 9 y su HI= nil y HD = 18 y 18 con su HD= 22.



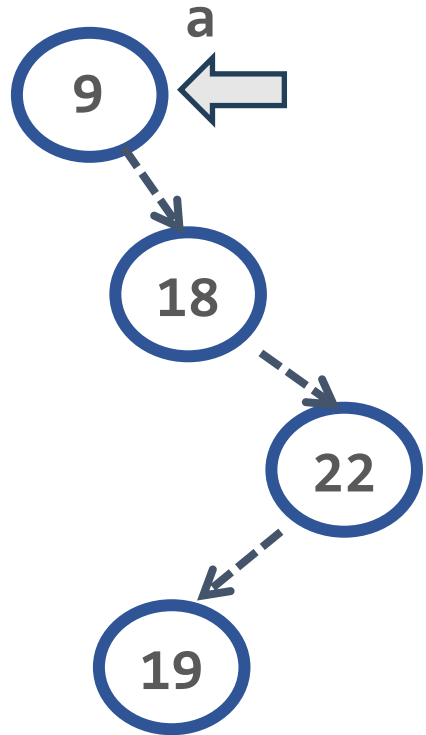
ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)

$a = 9$



Se lee el valor 19 (num) y se invoca al procedimiento agregar (a,num)



Como $a \neq \text{nil}$, tengo que recorrer desde la raíz hasta el lugar correspondiente respetando el orden. Siempre se inserta en una hoja.

Comparo num (19) con lo que está apunta a (9) y como $18 > 9$, hay que agregarlo a la derecha 9. Luego $18 \leq 19$ entonces hay que agregarlo a la derecha. Luego $19 \leq 22$, hay que agregarlo a la izquierda de 22. Como es nil, se encontró el lugar.

Reservo memoria dinámica y asigno los valores correspondientes

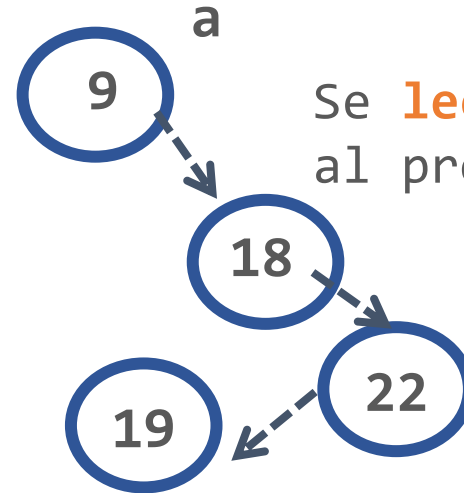
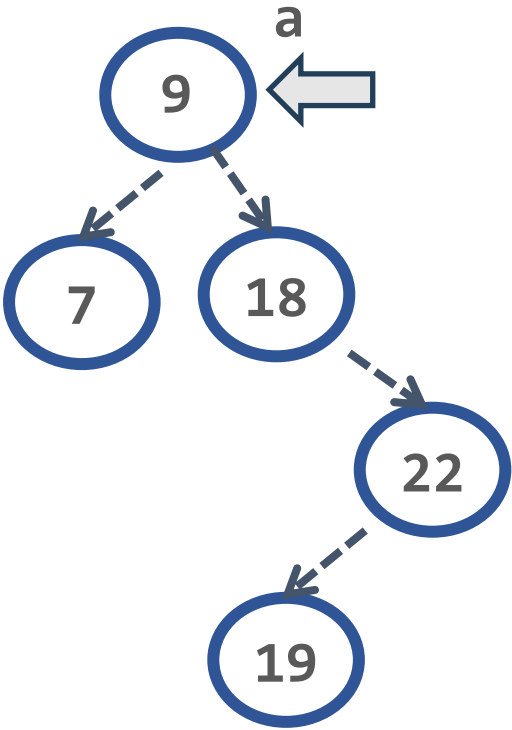
El procedimiento agregar termina y vuelve al programa principal en donde $a = 9$, su HD =18, a su vez su HD=22 y el HI de 22 = 19.



ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)

$a = 9$



Se lee el valor 7 (num) y se invoca al procedimiento agregar (a,num)

Como $a \neq \text{nil}$, tengo que recorrer desde la raíz hasta el lugar correspondiente respetando el orden. Siempre se inserta en una hoja.

Comparo num (7) con lo que está apunta a (9) y como es $7 \leq 9$ se determina que hay que agregarlo a la izquierda de 9, que como 9 no tiene HI se encontró el lugar.

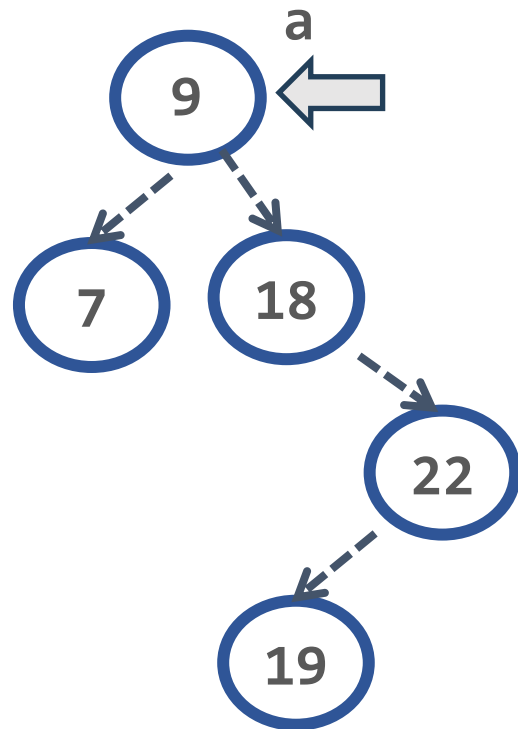
Reservo memoria dinámica y asigno los valores correspondientes

El procedimiento agregar termina y vuelve al programa principal en donde $a = 9$, su HD =18, y su **HI=7**.

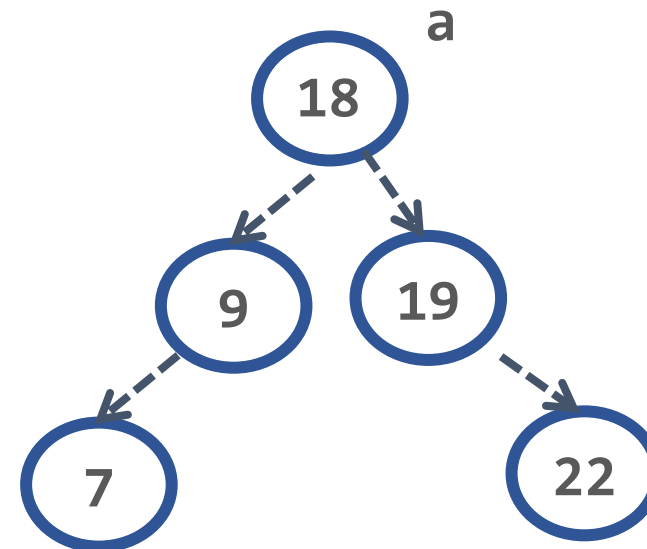


ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)



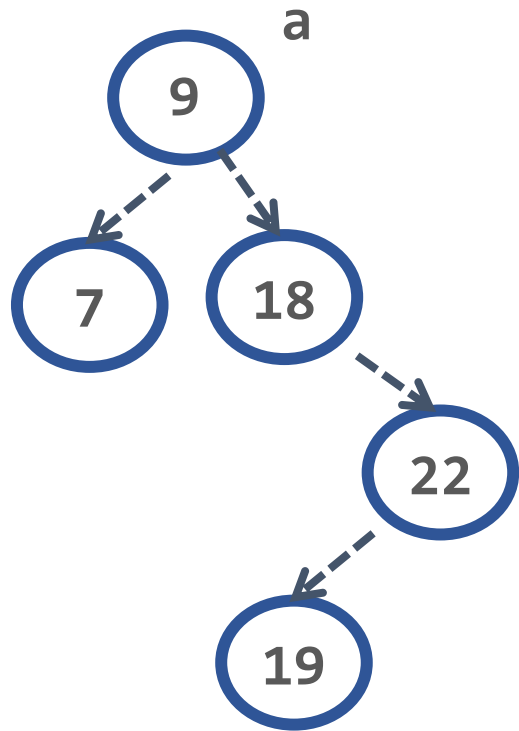
Si se leyeran los mismos valores pero en otro orden quedaría formado el mismo arbol? (18, 9, 7, 19, 22, 50)



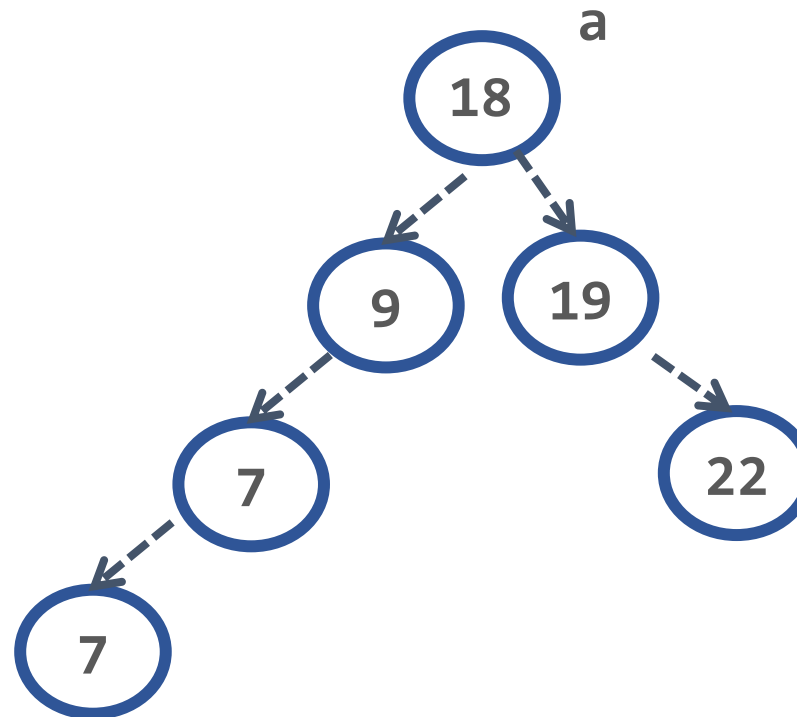


ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Suponga que se leen los siguientes valores y se quieren ir agregando en un ABB (9, 18, 22, 19, 7, 50)



Qué ocurre si se leen valores repetidos?
(18, 9, 7, 7, 19, 22, 50)



Cómo lo implementamos?

Cuál sería el caso base?



ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

Programa arboles;

Type

arbol = ^nodo;

nodo = record
dato: integer;
HI: arbol;
HD: arbol;
end;

Var

abb:arbol; x:integer;

Begin

abb:=nil;
read (x);
while (x<>50)do
begin
AGREGAR(abb,x);
read(x);
end;
End.

Procedure agregar (var a:árbol; num:integer);

Begin

if (a = nil) then
begin

new(A);

a^.dato:= num; a^.HI:= nil; a^.HD:= nil;

end

else

if (num <= A^.dato) then agregar(a^.HI,num)

else agregar (a^.HD,num)

End;

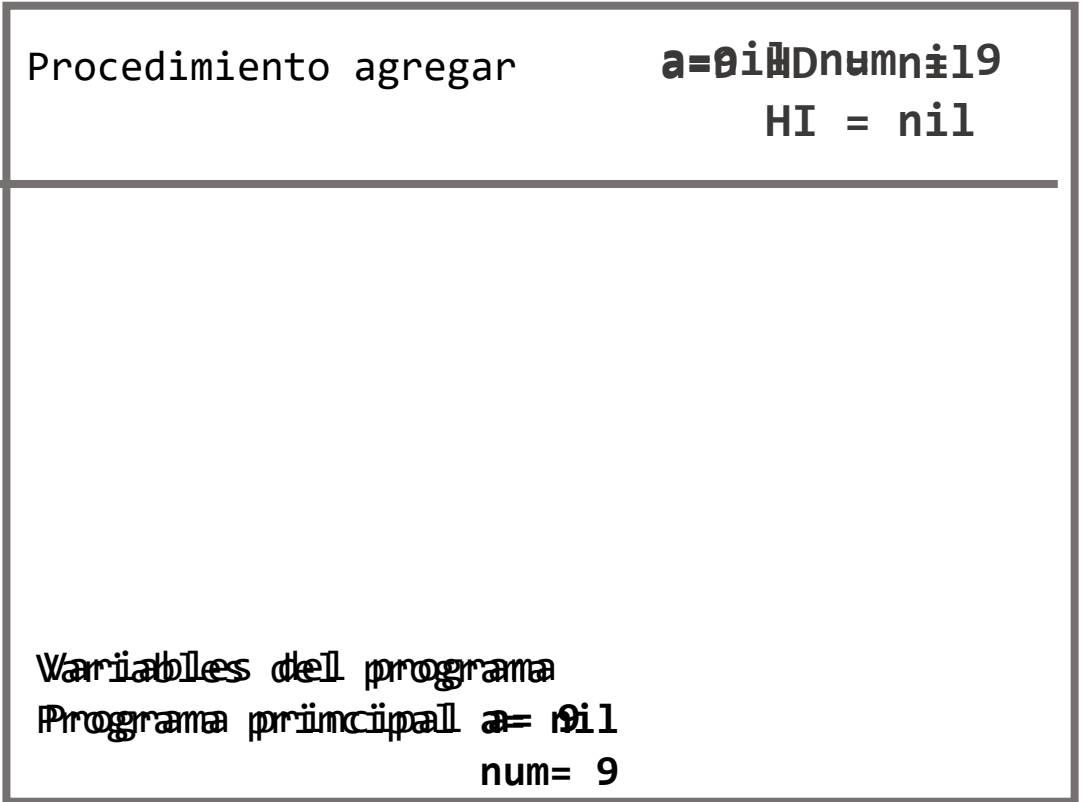
Cómo funciona?



ÁRBOLES BINARIOS DE BÚSQUEDA- CARGA

(9, 18, 22, 7)

```
Procedure agregar (var a:arbol; num:integer);
Begin
  if (a = nil) then
    begin
      new(a);
      a^.dato:= num; a^.HI:= nil; a^.HD:= nil;
    end
  else
    if (num <= a^.dato) then agregar(a^.HI,num)
    else agregar(a^.HD,num)
  End;
```



9

a



Diagram illustrating a node with value 9 and label 'a' pointing to a node with value 18.

Clase 3-3 – Módulo Imperativo



```
graph TD; a1((9)) -.-> b1((18)); a2((9)) -.-> b2((18)); b2 -.-> c((22));
```

```
Variables del programa
Programa principal a= 9
                    num=22
```



```
graph TD; N9_1((9)) --> N18_L((18)); N9_1 --> N18_R((18)); N18_L --> N22_L((22)); N18_R --> N22_R((22)); N7((7)) -.-> N9_1;
```

```
Variables del programa
Programa principal a = 9
                        num = 7
```



Taller de Programación



AGENDA

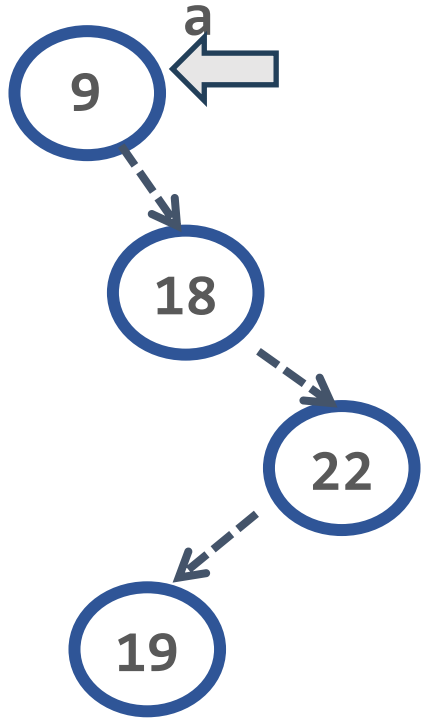


Operaciones ABB- RECORRIDOS

BUSQUEDA



ESTRUCTURA DE DATOS ARBOL - RECORRIDO



Para poder recorrer un ABB siempre debe comenzarse el recorrido desde la raíz.

Una vez que se esta parado en la raíz debe hacerse la acción que se quiera con el valor (imprimir, agregar en otro arbol, agregarlo en una lista, modificarlo, etc).

Luego debe tomarse uno de sus hijos y realizar la misma acción que para el nodo padre y luego el otro de sus hijos.

Cuál es el caso base?

Cuántos llamados recursivos se hacen en cada nodo?

Cómo se implementa?



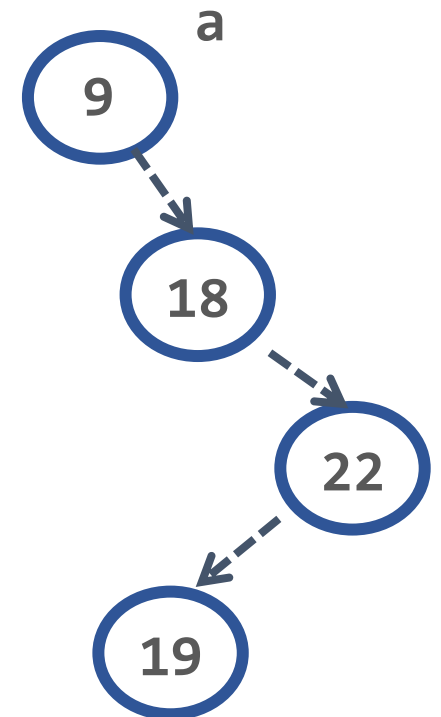
ESTRUCTURA DE DATOS ARBOL - RECORRIDO

```
Programa arboles;  
Type  
  arbol = ^nodo;  
  nodo = record  
    dato: integer;  
    HI: arbol;  
    HD: arbol;  
  end;  
Var  
  a:arbol;  num:integer;  
Begin  
  a:= nil;  
  read (num);  
  while (num <> 50) do  
    begin  
      agregar (a,num);  
      read (num);  
    end;  
  recorrido_enOrden(a);  
End.
```

```
Procedure enOrden ( a : arbol );  
begin  
  if ( a <> nil ) then begin  
    enOrden (a^.HI);  
    write (a^.dato); //o cualquier otra acción  
    enOrden (a^.HD);  
  end;  
end;
```

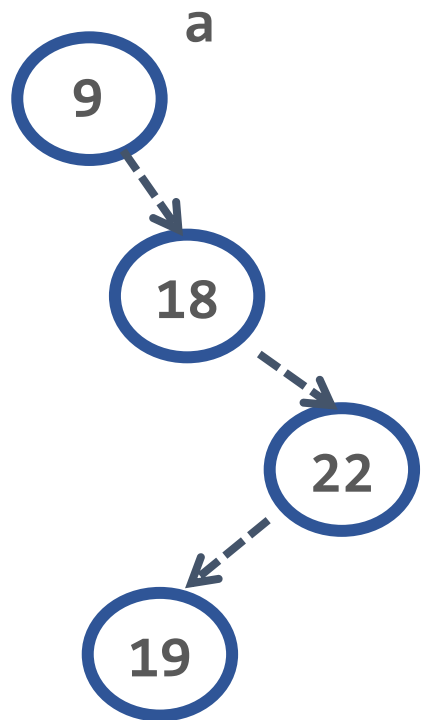
Es lo mismo pasar
a por referencia?

Cómo funciona?



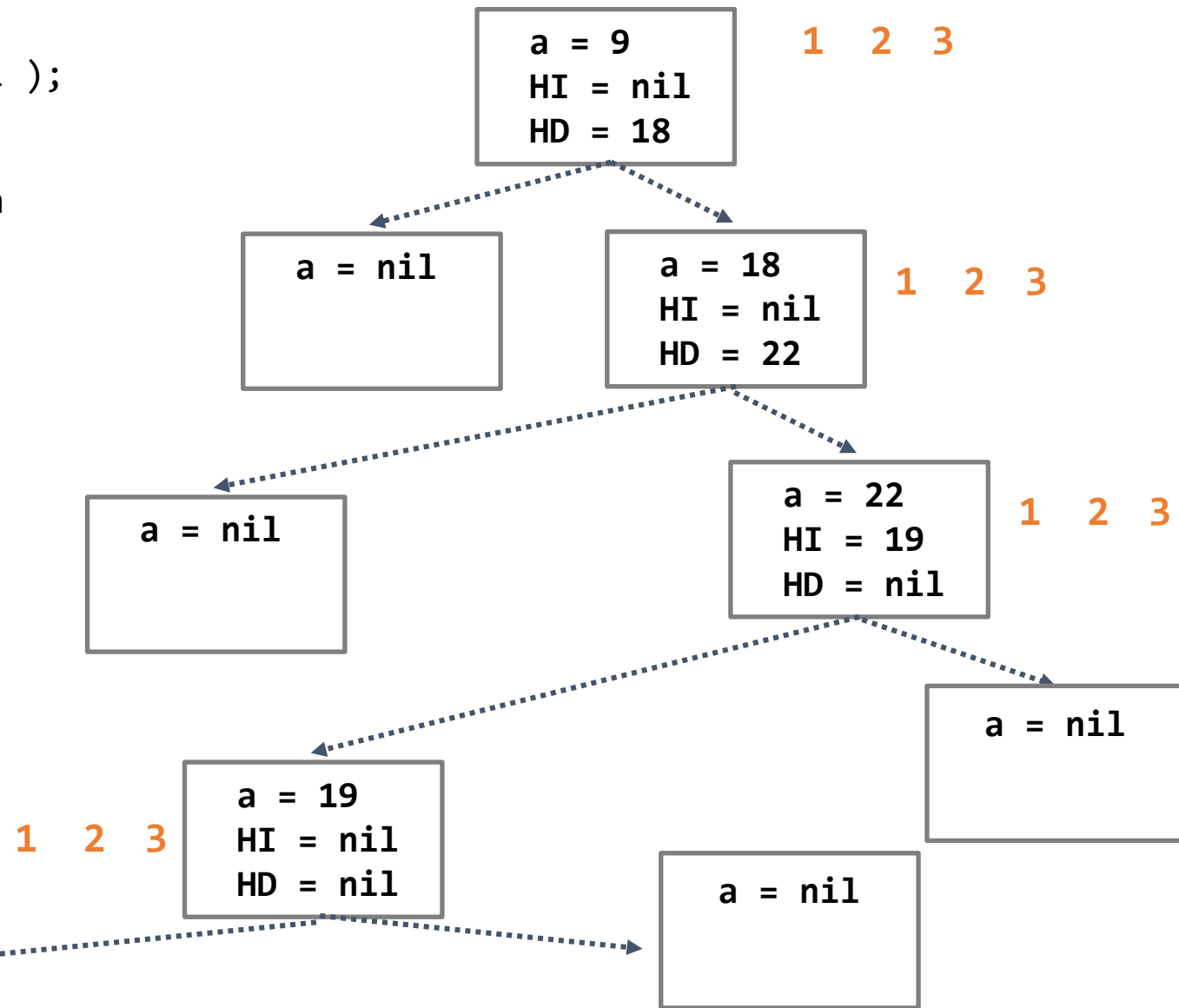


ESTRUCTURA DE DATOS ARBOL - RECORRIDO



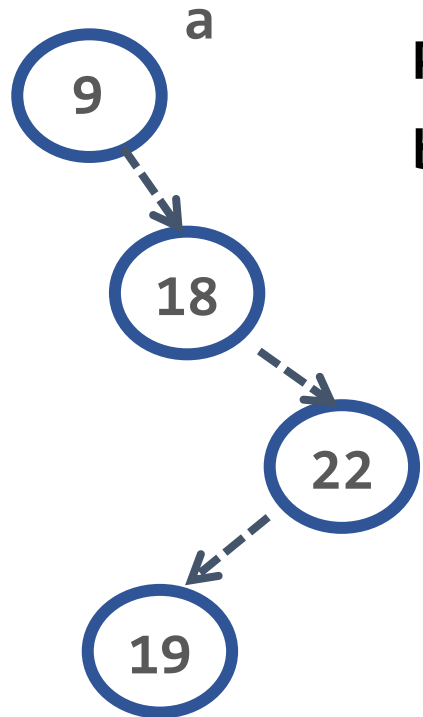
```
Procedure enOrden ( a : arbol );  
begin  
  if ( a <> nil ) then begin  
    1 enOrden (a^.HI);  
    2 write (a^.dato);  
    3 enOrden (a^.HD);  
  end;  
end;
```

Pantalla 9
Pantalla 18
Pantalla 19
Pantalla 22





ESTRUCTURA DE DATOS ARBOL - RECORRIDO



```
Procedure preOrden (a:arbol);  
begin  
  if ( a <> nil ) then  
    begin  
      write (a^.dato);  
      preOrden (a^.HI);  
      preOrden (a^.HD);  
    end;  
  end;  
end;
```

```
Procedure postOrden (a:arbol);  
begin  
  if ( a <> nil ) then  
    begin  
      postOrden (a^.HI);  
      postOrden (a^.HD);  
      write (a^.dato);  
    end;  
  end;  
end;
```

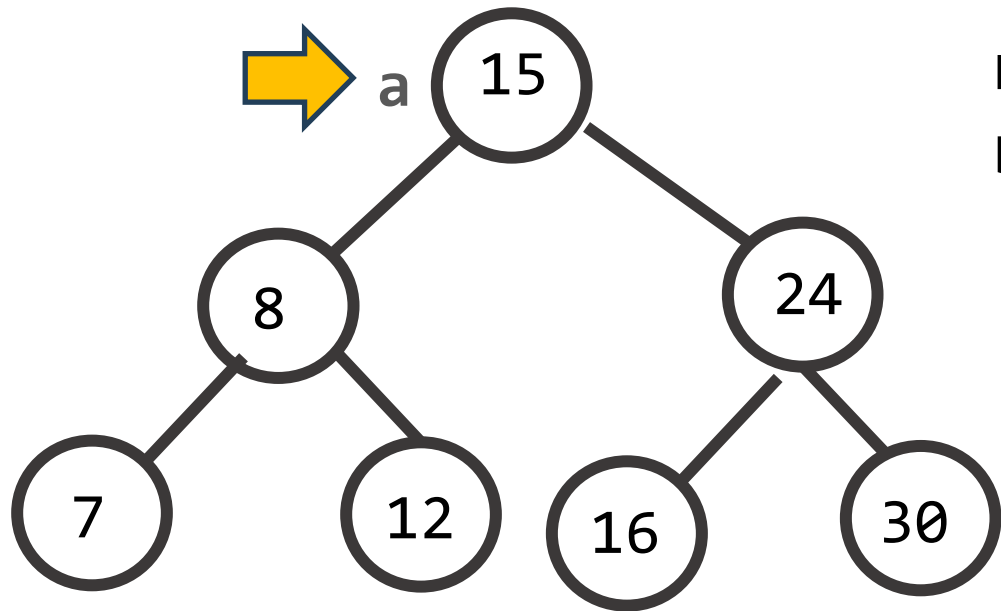
Qué imprímen?

Si *a* se pasa por
referencia que
imprime cada uno?



ARBOLES BINARIOS DE BUSQUEDA - BUSCAR

Supongamos que disponemos de un árbol binario de búsqueda, y queremos implementar un módulo que retorne un booleano si se encuentra un valor buscado que se recibe como parámetro.



```
Procedure Buscar (a:arbol;x:integer;var ok:boolean);  
begin  
  if ( a <> nil ) then begin  
    buscar (a^.HI,x,ok);  
    if (a^.dato = x) then ok:= true;  
    buscar (a^.HD,x,ok);  
  end;  
end;
```

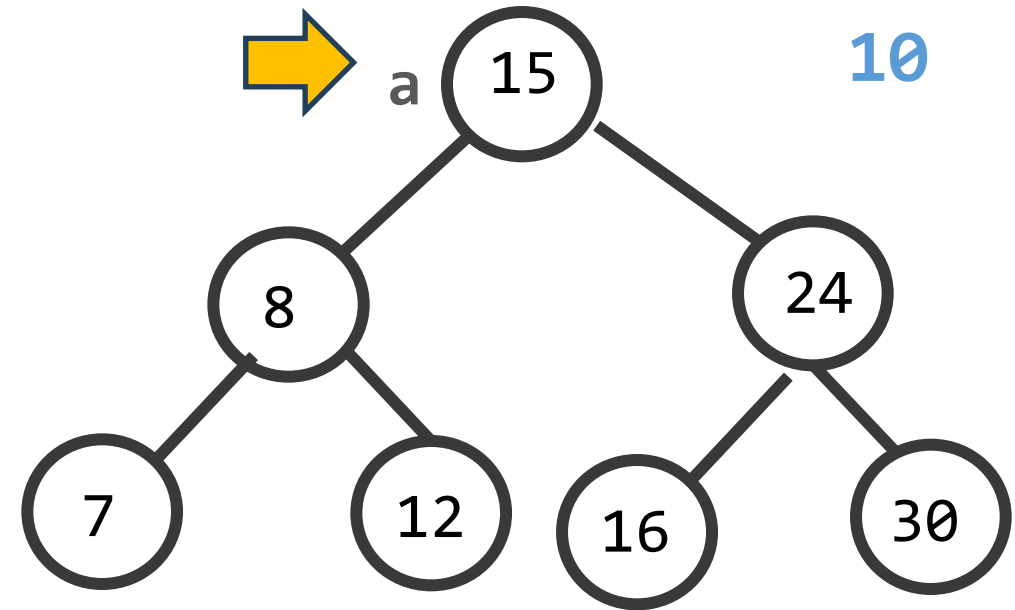
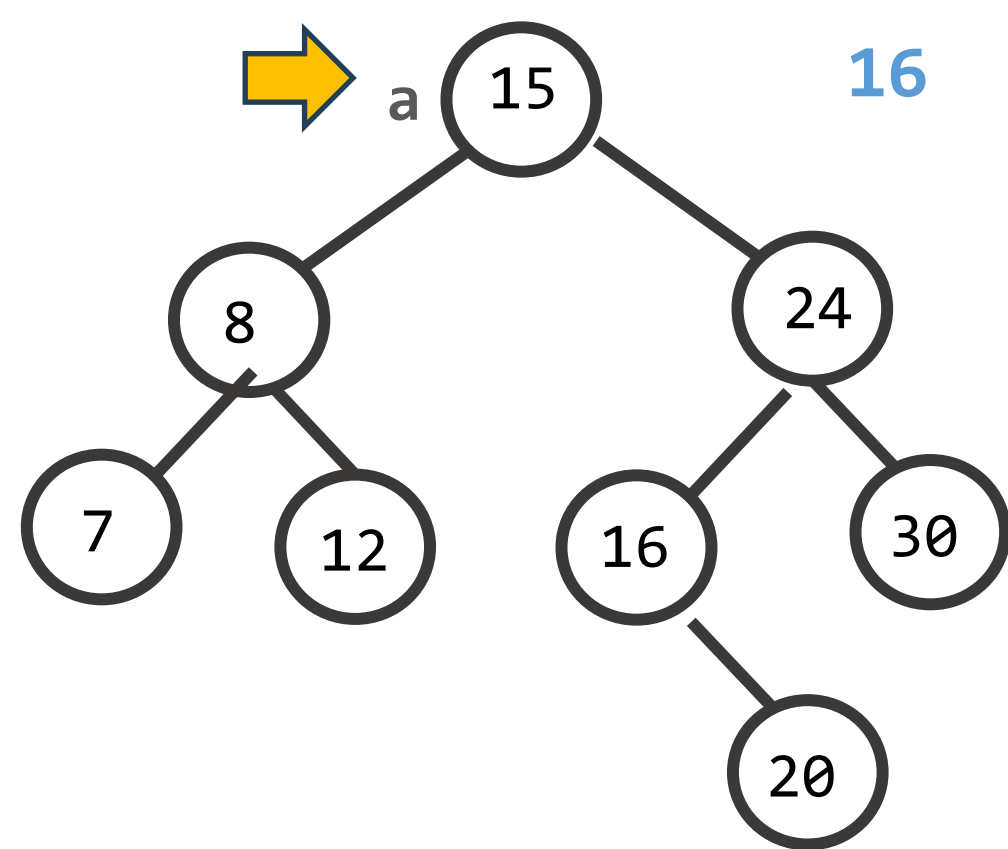


Para buscar un valor siempre se debe "aprovechar" el orden del ABB.



ARBOLES BINARIOS DE BUSQUEDA - BUSCAR

Supongamos que disponemos de un árbol binario de búsqueda, y queremos implementar un módulo que retorne el nodo donde se encuentra un valor buscado que se recibe como parámetro.



Cómo lo implemento?



ARBOLES BINARIOS DE BUSQUEDA - BUSCAR

DEVUELVE EL VALOR BOOLEAN

Programa arbolesEnteros;

Type

arbol = ^nodo;

nodo = record

dato: integer;

HI: arbol;

HD: arbol;

end;

Var

a:arbol;min:integer;ok:booleanx:integer

Begin

cargarArbol(a); read(x);

if (a <> nil) then

ok:= buscar(a,x);

write (ok);

End.

DEVUELVE EL NODO QUE CONTIENE EL BUSCADO

Programa arbolesEnteros;

Type

arbol = ^nodo;

nodo = record

dato: integer;

HI: arbol;

HD: arbol;

end;

Var

a:arbol; bus:arbol; x:integer;

Begin

cargarArbol(a); read(x);

bus:= buscarNodo(a,x);

if (bus <> nil) then write ("encontro");

End.



ARBOLES BINARIOS DE BUSQUEDA - BUSCAR

DEVUELVE EL BOOLEAN

```
Programa arbolesEnteros;
```

```
Type
```

```
    arbol = ^nodo;
```

```
    nodo = record
```

```
        dato: integer;
```

```
        HI: arbol;
```

```
        HD: arbol;
```

```
    end;
```

```
Var
```

```
    a:arbol; min:integer; x:integer;
```

```
Begin
```

```
    cargarArbol(a); read(x);
```

```
    if (a <> nil) then
```

```
        ok:= buscar(a,x);
```

```
        write (ok);
```

```
End.
```

```
Function buscar (a:arbol; x:integer): boolean;
```

```
begin
```

```
    if (a = nil) then buscar:= false
```

```
    else (a^.dato = x) then buscar:= true
```

```
    else if (x > a^.dato) then buscar:= buscar(a^.HD, x)
```

```
    else buscar:= buscar(a^.HI, x)
```

```
end;
```

Puede implementarse
de manera iterativa?



ARBOLES BINARIOS DE BUSQUEDA - BUSCAR

DEVUELVE EL NODO BUSCADO

Programa arbolesEnteros;

Type

arbol = ^nodo;

nodo = record

dato: integer;

HI: arbol;

HD: arbol;

end;

Var

a:arbol; bus:arbol; x:integer;

Begin

cargarArbol(a); read(x);

bus:= buscarNodo(a,x);

if (bus <> nil) then write ("encontro");

End.

function buscarNodo (a:arbol; x:integer): arbol;

Begin

if (a = nil) then buscarNodo:= nil

else (a^.dato = x) then buscarNodo:= a

else if (x > a^.dato) then

buscarNodo:= buscarNodo(a^.HD, x)

else buscarNodo:= buscarNodo(a^.HI, x);

End;

Puede implementarse
de manera iterativa?



ARBOLES BINARIOS DE BUSQUEDA -



Operaciones:

- Buscar valores dentro de un rango
- Obtener el valor máximo de un ABB
- Obtener el valor mínimo de un ABB
- Calcular la cantidad de nodos que tiene un ABB
- Calcular el nivel de un ABB
- Calcular la altura de un ABB