Programación 2

La previa:
Estructuras Arborescentes
(parte 1)

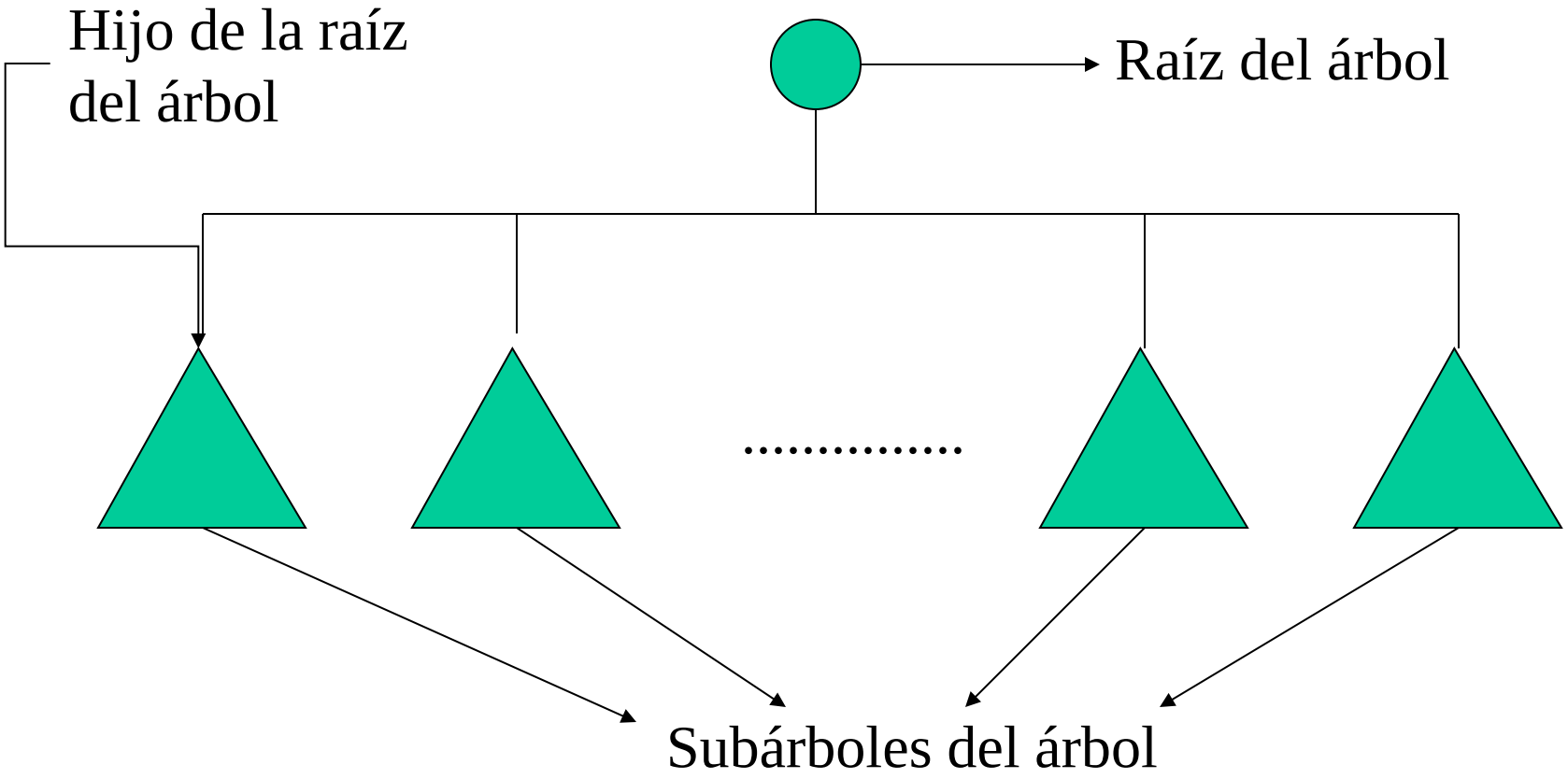
Definición

La recursión puede ser utilizada para la definición de estructuras realmente sofisticadas.

Una estructura *árbol* (*árbol general o finitario*) con tipo base T es,

1. O bien la estructura vacía
2. O bien un elemento de tipo T junto con un número finito de estructuras de árbol, de tipo base T, disjuntas, llamadas *subárboles*

Conceptos básicos (cont.)



Los elementos se ubican en *nodos* del árbol.

Arboles n-arios y binarios

La descripción de la noción de árbol dada antes es casi una definición inductiva.

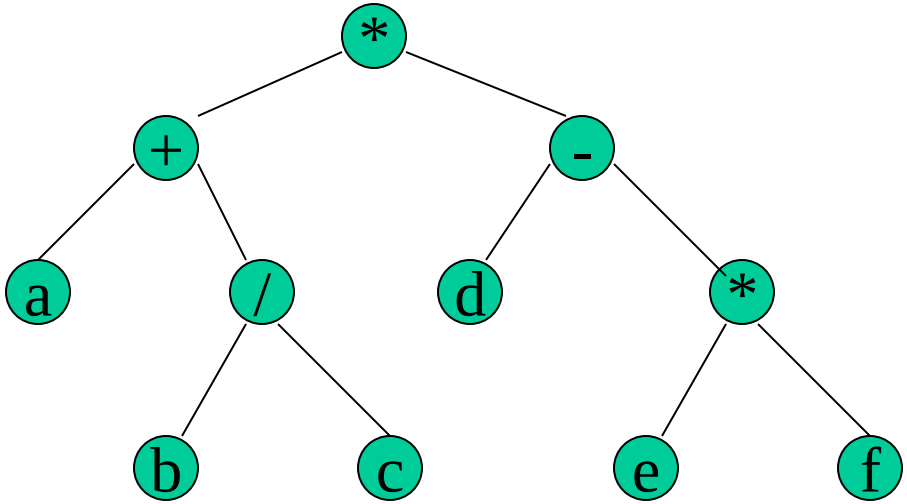
Falta precisar qué se quiere decir con “un número finito ...” en la segunda cláusula de construcción de árboles.

Existe un caso particular de árbol en que esta cláusula se hace precisa fácilmente: “... junto con exactamente 2 subárboles binarios.”

Este es un caso particular de **árboles n-arios**, que son a su vez un caso particular de árboles generales o finitarios.

Ejemplo: árbol de expresiones
Sintaxis concreta vs. sintaxis abstracta

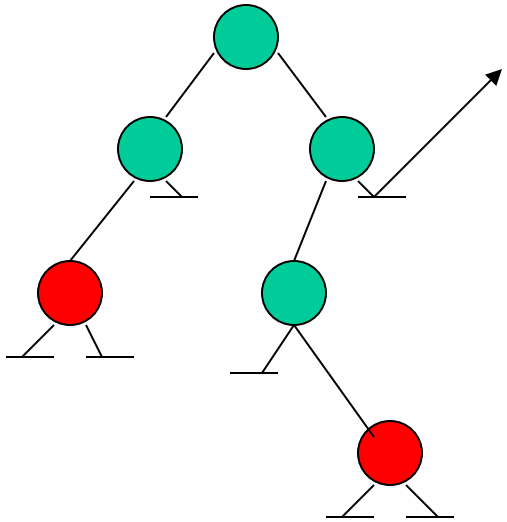
Representación no ambigua de expresiones aritméticas.



Representación arborescente de la fórmula:

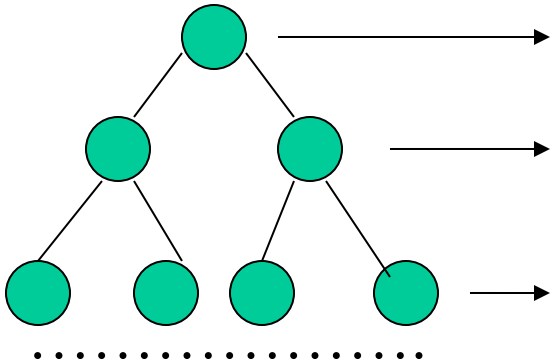
(a + b / c) * (d - e * f)

Hojas



Def: *Hojas* son los nodos cuyos (ambos) subárboles son vacíos

Niveles y altura



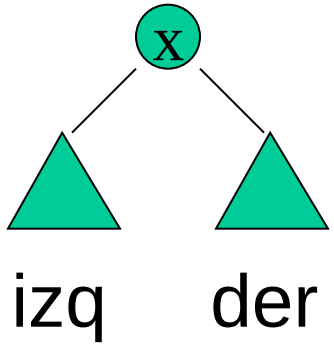
Def.
La *altura* de un árbol es:
la cantidad de niveles que tiene, o
la cantidad de nodos en el camino más largo de la raíz a una hoja.

La altura del árbol binario vacío es 0.

Altura (cont.)

Definición Inductiva de Árboles Binarios

$$\frac{}{() : \text{ArbBin}} \qquad \frac{izq : \text{ArbBin} \quad x : T \quad der : \text{ArbBin}}{(izq\ ,\ t\ ,\ der) : \text{ArbBin}}$$




```
altura (()) = 0
altura ((izq,a,der)) =
          1 + max(altura(izq), altura(der))
```

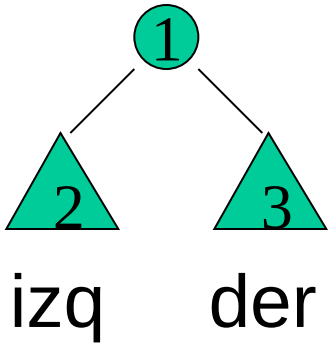
Recorridas de árboles binarios

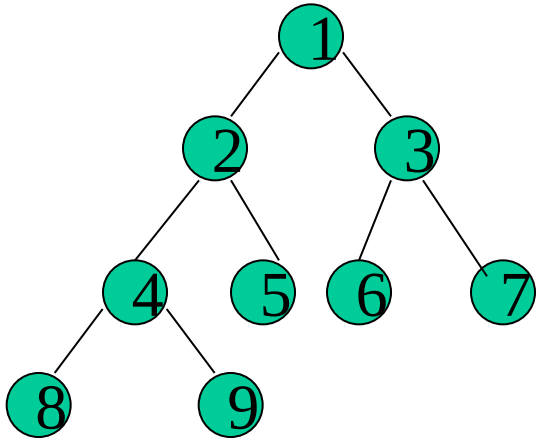
Para recorrer un árbol no vacío hay tres órdenes naturales, según la raíz sea visitada:

- antes que los subárboles
  (PreOrden - preorder)
- entre las recorridas de los subárboles
  (EnOrden - inorder)
- después de recorrer los subárboles
  (PostOrden - postorder)

Recorridas de árboles binarios (cont.)

Antes que los subárboles (preorder)

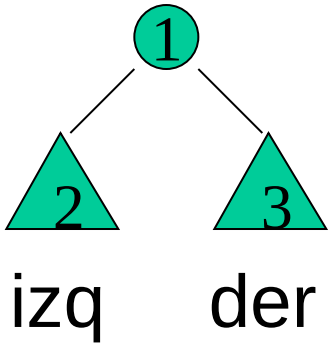


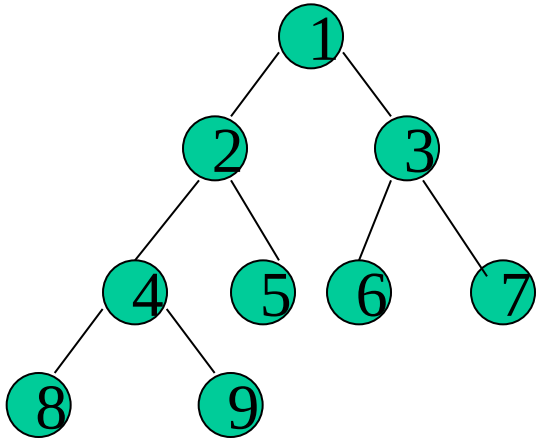


Preorder: …

Recorridas de árboles binarios (cont.)

Antes que los subárboles (preorder)

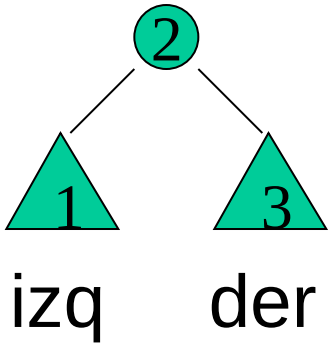


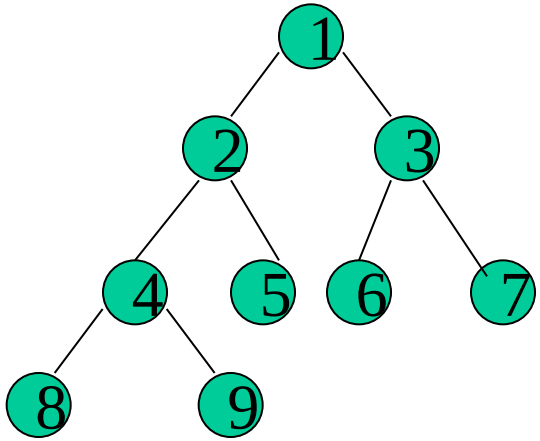


Preorder: 1, 2, 4, 8, 9, 5, 3, 6, 7

Recorridas de árboles binarios (cont.)

Antes que los subárboles (inorder)

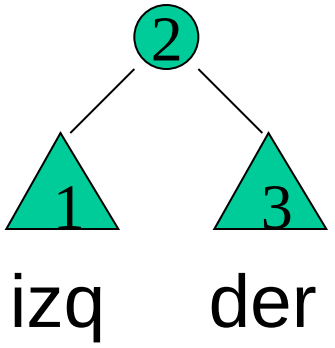


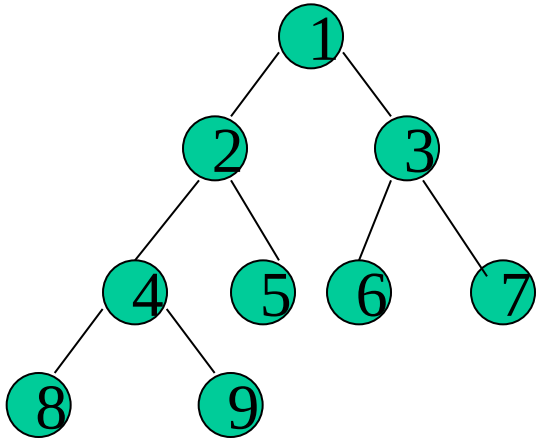


Inorder: …

Recorridas de árboles binarios (cont.)

Antes que los subárboles (inorder)

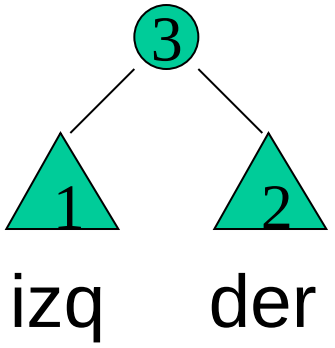


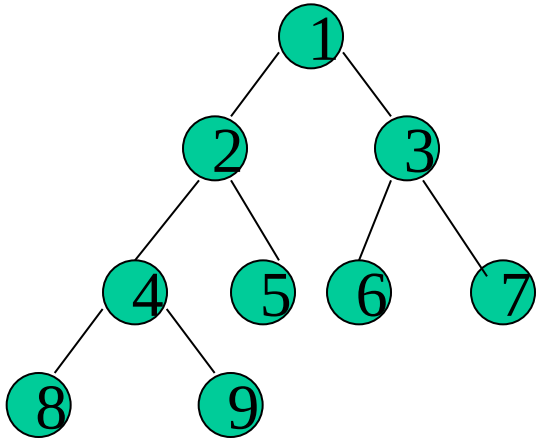


Inorder: 8, 4, 9, 2, 5, 1, 6, 3, 7

Recorridas de árboles binarios (cont.)

Antes que los subárboles (postorder)





Postorder: 8, 9, 4, 5, 2, 6, 7, 3, 1

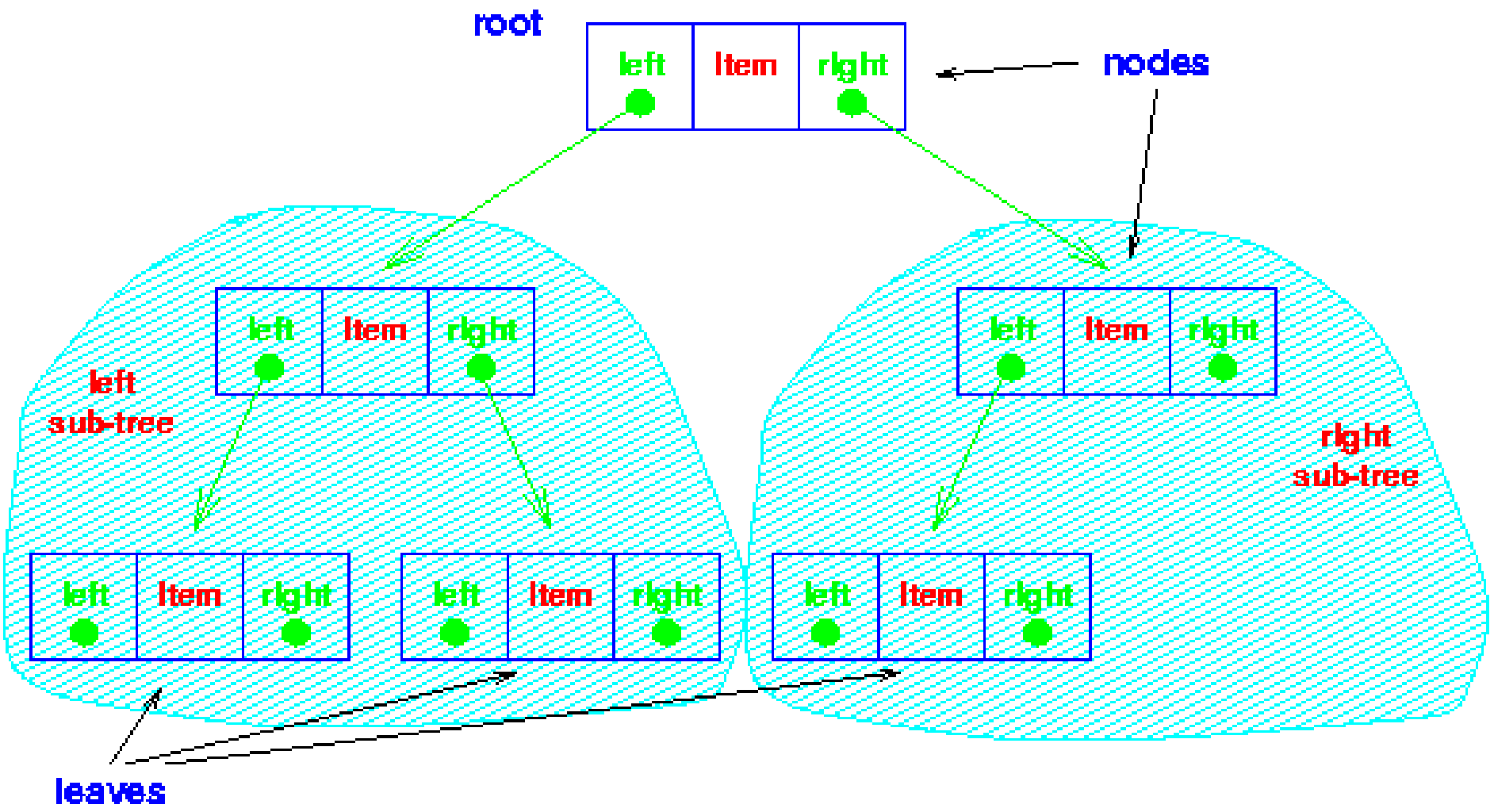
Implementación en C y C++

Ahora presentaremos una posible implementación del tipo de dato árbol binario (ArbBin) en C++. Los elementos del árbol son de tipo T en la siguiente definición:

```
typedef NodoAB* AB;

struct NodoAB{
    T item;
    AB left, right;
};
```

Arbol Binario (AB)



Procedimiento preOrden

```
void preOrden (AB t){
      if (t != NULL){
            P(t -> item);
            preOrden(t -> left);
            preOrden(t -> right);
      }
}
```

Procedimiento enOrden

```
void enOrden (AB t){
      if (t != NULL){
            enOrden(t -> left);
            P(t -> item);
            enOrden(t -> right);
      }
}
```

Procedimiento postOrden

```
void postOrden (AB t){
      if (t != NULL){
            postOrden(t -> left);
            postOrden(t -> right);
            P(t -> item);
      }
}
```

Cantidad de nodos de un árbol

```
cantNodos (()) = 0
cantNodos ((izq,a,der)) =
      1 + cantNodos(izq) + cantNodos(der)
```

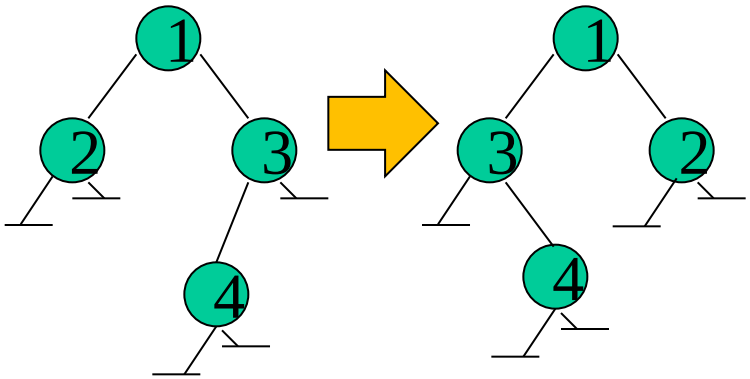
```
int cantNodos (AB t){
      if (t == NULL) return 0;
      else
         return 1 + cantNodos(t->left)
                  + cantNodos(t->right);
}
```

¿Cuál es la diferencia con la función altura?

Cantidad de nodos de un árbol

```
int altura (AB t){
      if (t == NULL) return 0;
      else
         return 1 + max(altura (t->left),
                        altura (t->right));
}
```

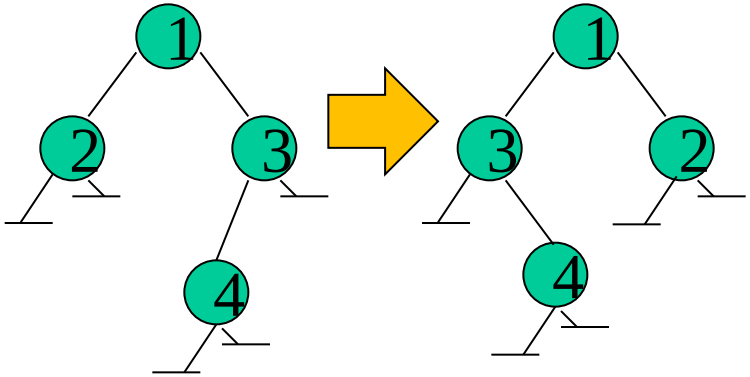
Espejo




```
espejo (()) = ()
espejo ((izq,a,der)) = (der,a,izq)    ???
```

Espejo

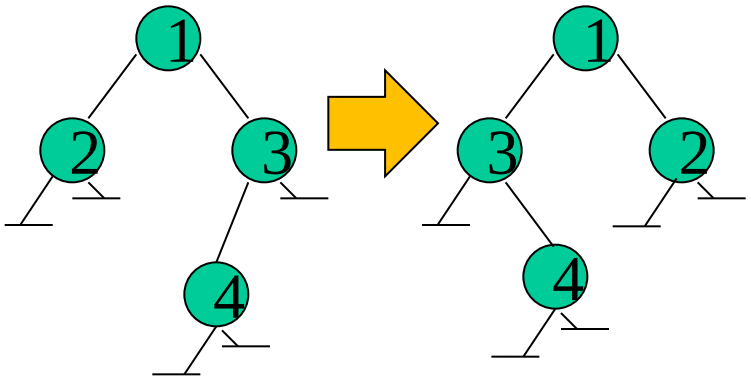
```
espejo (()) = ()
espejo ((izq,a,der)) =
  (espejo(der),a,espejo(izq))
```




```
AB espejo (AB t){…}
```

Espejo

```
espejo (()) = ()
espejo ((izq,a,der)) =
  (espejo(der),a,espejo(izq))
```



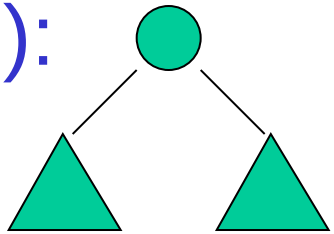

```
AB espejo (AB t){
     if (t == NULL) return NULL;
     else
        { AB rt = new NodoAB;
          rt -> item = t -> item;
          rt -> left = espejo (t -> right);
          rt -> right = espejo (t -> left);
          return rt;
        }
}
```

La función que retorna una copia idéntica de un árbol, sin compartir memoria, ¿se parece a la función espejo?

Copia

```
AB copia (AB t){
    if (t == NULL) return NULL;
    else
        { AB rt = new NodoAB;
          rt -> item = t -> item;
          rt -> left = copia (t -> left);
          rt -> right = copia (t -> right);
          return rt;
        }
}
```

Árboles binarios de enteros (AB):



struct nodoAB {int dato; nodoAB *izq, *der;};

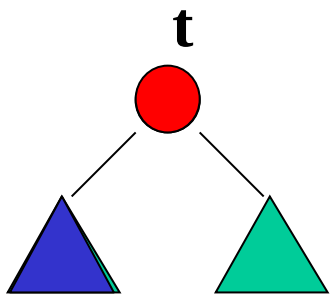
typedef struct nodoAB * AB;

Implemente una función recursiva (sin usar iteración) máximo que dado un árbol binario t de tipo AB retorne un puntero al nodo de t que tenga el dato mayor (el máximo), o NULL si t es NULL. Asumimos que t no tiene elementos repetidos, aunque no puede asumirse que los elementos en t están ordenados como en un árbol binario de búsqueda.

No defina ni asuma la existencia de operaciones auxiliares para implementar maximo. Además, la función maximo no deberá visitar cada nodo de t más de una vez (solo se puede recorrer el árbol una vez).

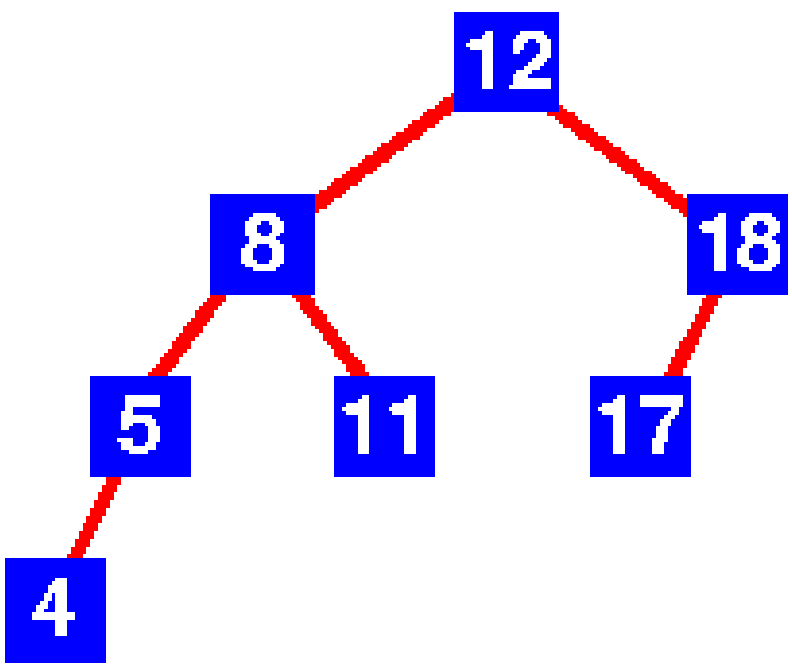
AB maximo(AB t){ … }

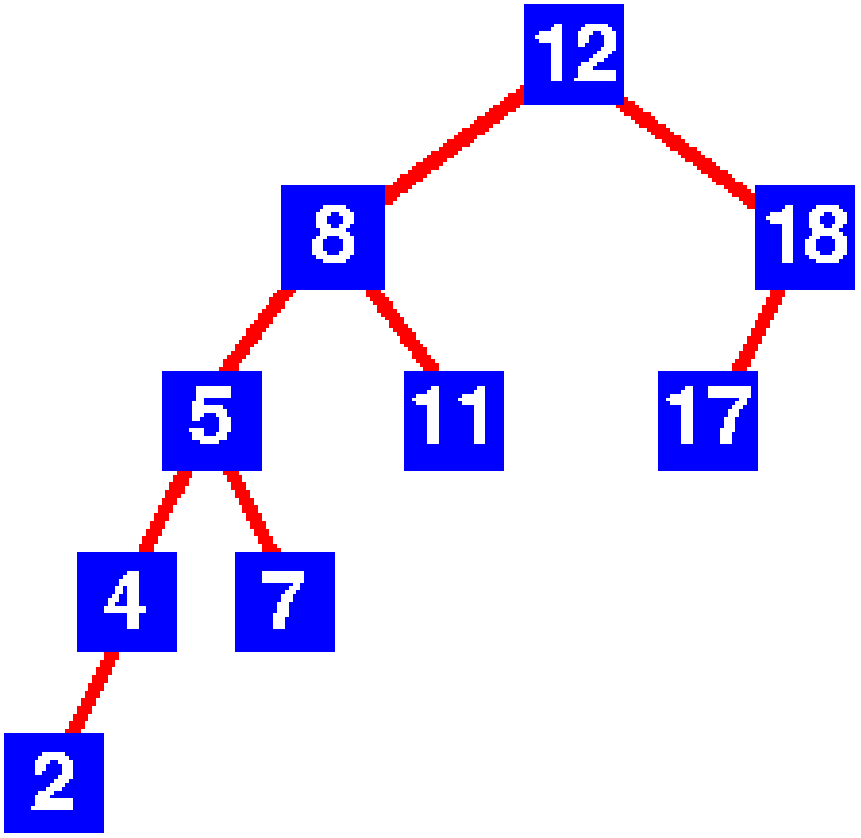
Árboles binarios de enteros (AB):




```
PRE: t no tiene elementos repetidos
AB maximo(AB t){
    if (t==NULL) return NULL;
    else{ AB max = t; // inicializamos con la raíz
          AB maxRec = maximo(t->izq);
          if (maxRec!=NULL && maxRec->dato > max->dato)
                max = maxRec; // se considera el max de t->izq
          maxRec = maximo(t->der);
          if (maxRec!=NULL && maxRec->dato > max->dato)
                max = maxRec; // se considera el max de t->der
          return max;
    }
}
```

Arbol binario de búsqueda (ABB): Ejemplos

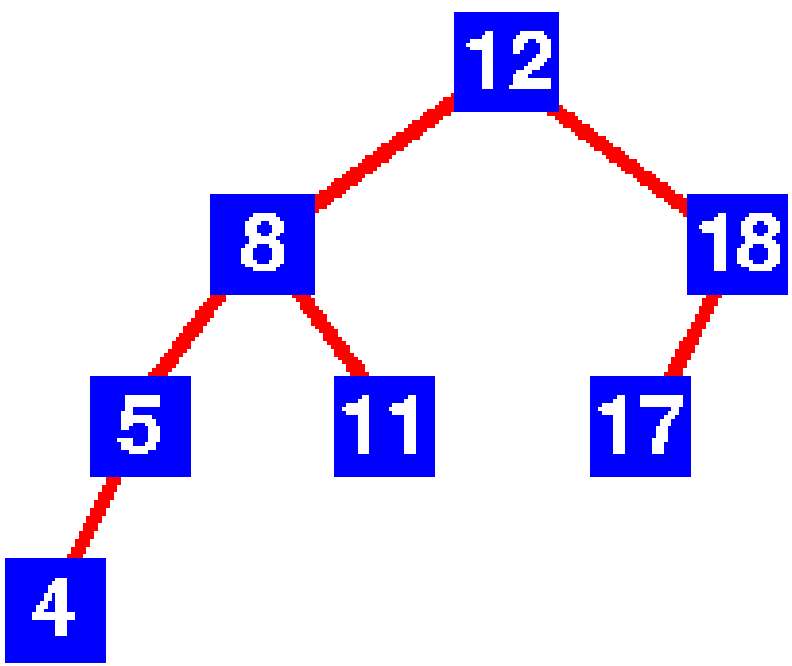


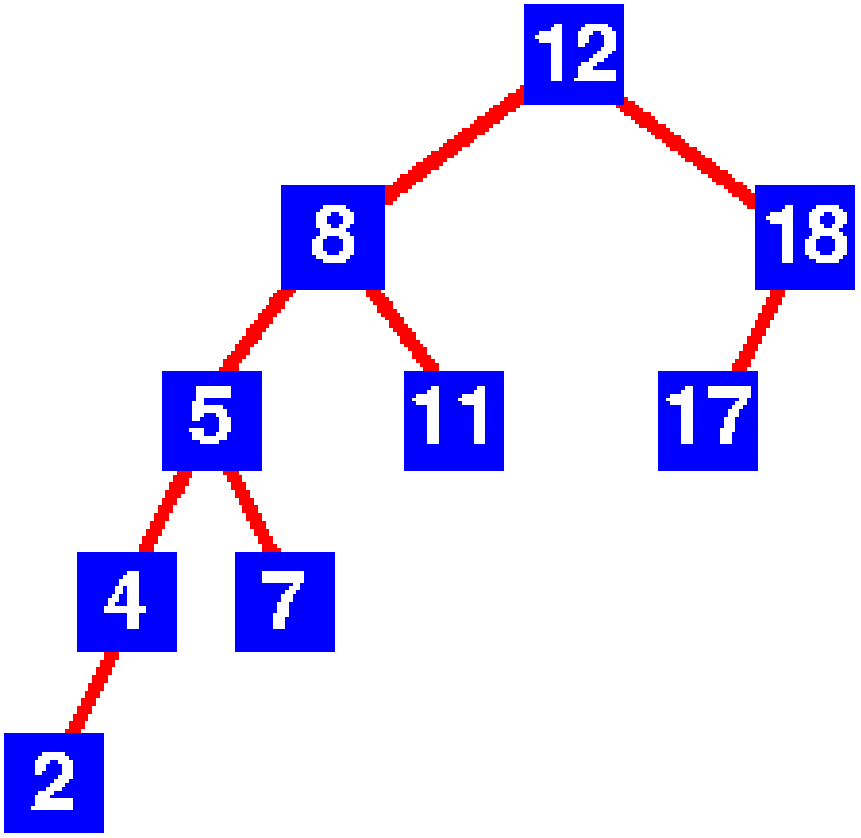


Analizar la relación con la búsqueda binaria sobre un arreglo ordenado (y sobre una lista).

3	5	8	10	14	22	29	45	77

Arbol binario de búsqueda (ABB): Ejemplos





¿Dónde está el mínimo y el máximo en un ABB?

¿Qué pasa con un recorrido en orden de un ABB?

Implementación de ABB

La implementación del tipo *ABB* es muy similar a la de ArbBin. La diferencia es que ahora diferenciamos un campo, **`key`**, cuyo tipo es un ordinal (**`Ord`**), del resto de la información del nodo:

```
typedef NodoABB* ABB;

struct NodoABB {
    Ord key;
    T info;
    ABB left, right;
};
```

Buscar iterativo

```
ABB buscarIterativo(Ord x, ABB t){
  while ((t != NULL) && (t -> key != x)){
        if (t -> key > x)
           t = t -> left;
        else t = t -> right;
  }
  return t;
}
```

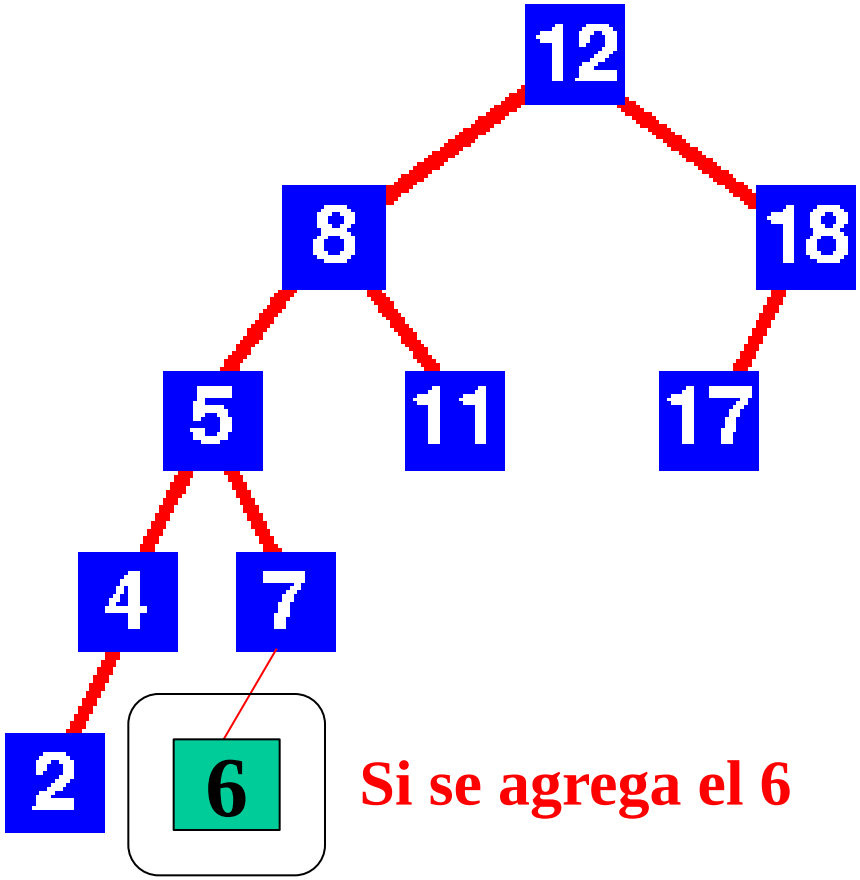
La función Pertenece

```
bool pertenece(Ord x, ABB t){
  if (t == NULL) return false;
  else
     if (x == t->key)
       return true;
     else
       return (pertenece(x,t->left)
            || pertenece(x,t->right));
}
```

¿Es eficiente la función *pertenece*?

¿Podría optimizarse?

Inserción en un ABB

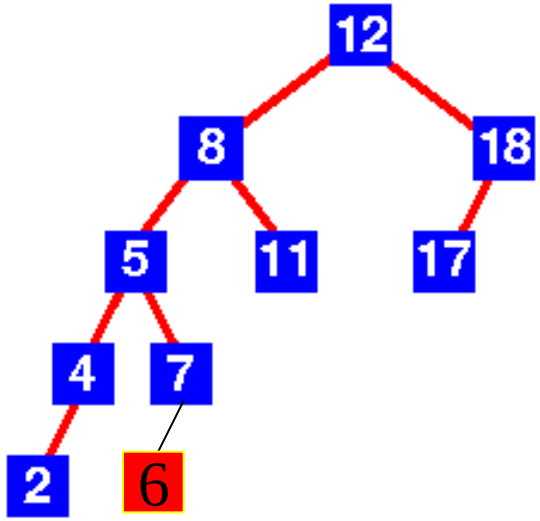


¿Siempre se agrega (eventualmente) un elemento como una hoja?

```
void insABB (Ord clave, T dato, ABB & t)
```

Inserción en un ABB

```
void insABB (Ord clave, T dato, ABB & t){
  if (t == NULL){
     t = new NodoABB ;
     t->key = clave;
     t->info = dato;
     t->left = t->right = NULL;
  }
  else if (clave < t->key)
      insABB (clave, dato, t->left);
  else if (clave > t->key)
      insABB (clave, dato, t->right);
}
```



Ejercicio: Aplanar eficientemente un ABB

Completar el siguiente código para obtener una lista ordenada con los elementos de un ABB, implementando *aplanarEnLista*, de tal manera que el árbol se recorra solo una vez y que la lista no se recorra al agregar cada elemento:

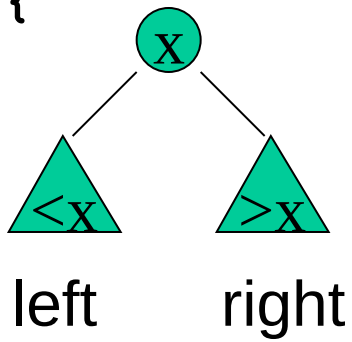
```
Lista aplanar (ABB t){
  Lista l = NULL;
  aplanarEnLista (t, l);
  return l;
}
```

Compare esta versión de *aplanar* con la que se obtiene usando la concatenación de listas.

Ejercicio: Aplanar eficientemente un ABB

```
Lista aplanar (ABB t){
   Lista l = NULL;
   aplanarEnLista (t, l);
   return l;
}
```

```
void aplanarEnLista (ABB t, Lista & l){
   if ( t!= NULL){
        aplanarEnLista(t->right,l);
        insComienzo(t->dato,l);
        aplanarEnLista(t->left,l);
   }
}
```



Árboles binarios de búsqueda (ABB)

typedef nodoABB * ABB

struct nodoABB { int dato; ABB izq; ABB der; }

Implemente una función recursiva (sin usar iteración) delMin que dado un árbol binario de búsqueda t de tipo ABB no vacío, elimine de t el mínimo elemento y retorne su valor. El árbol resultante deber ser también binario de búsqueda. No implemente ni asuma la existencia de operaciones auxiliares para implementar delMin.

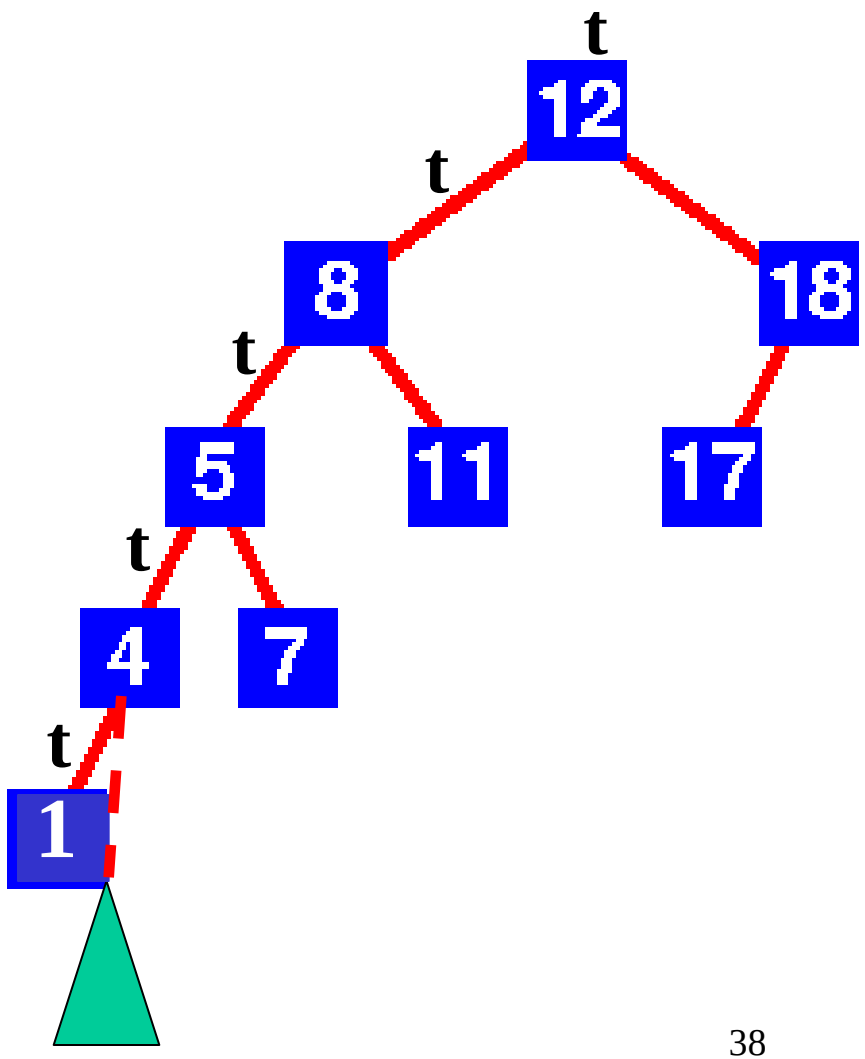
PRE: t no vacío

int delMin(ABB & t)

Explique muy brevemente el orden de tiempo de ejecución para el peor caso y para el caso promedio de delMin.

Árboles binarios de búsqueda (ABB)

```
PRE: t no vacío
int delMin(ABB & t){
    if (t->izq==NULL){
            int min = t->dato;
            ABB aBorrar = t;
            t = t->der;
            delete aBorrar;
            return min;
    }
    else return delMin(t->izq);
}
```



Árboles binarios de búsqueda (ABB)

Sea n la cantidad de nodos del árbol.

- **El peor caso es O(n), ya que el camino al mínimo puede involucrar a todos los nodos del árbol (árbol degenerado hacia la izquierda). Sobre cada nodo las acciones involucradas son de O(1).**
- **El caso promedio es O($\log_2$(n)), ya que en promedio la altura del árbol es $\log_2$(n). Luego, el camino más hacia la izquierda tiene en promedio $\log_2$(n) nodos. Sobre cada nodo las acciones involucradas son de O(1).**

Algo de la Tarea 2

```
#include "persona.h“
// El tipo TABBPersonas es un puntero a rep_abbPersonas.
typedef struct rep_abbPersonas *TABBPersonas;

// Función para crear un nuevo abb de personas vacío.
// Devuelve un nuevo árbol binario de búsqueda vacío.
// Requisitos específicos de la implementación solicitada:
// La funcion es Theta(1) peor caso.
TABBPersonas crearTABBPersonasVacio();
```

Algo de la Tarea 2

```
// Función para insertar una persona en el árbol, ordenada por CI.
// PRE: La persona con id no está en el árbol.
// Requisitos específicos de la implementación solicitada:
/* La función es Theta(n) peor caso, siendo n la cantidad de
   personas en el árbol. */

void insertarTPersonaTABBPersonas
   (TABBPersonas &abbPersonas, TPersona persona);
```

Algo de la Tarea 2

- **void imprimirTABBPersonas(TABBPersonas abbPersonas);**
- **void liberarTABBPersonas(TABBPersonas &abbPersonas);**
- **bool existeTPersonaTABBPersonas(TABBPersonas abbPersonas, int ciPersona);**
- **TPersona obtenerTPersonaTABBPersonas(TABBPersonas abbPersonas, int ciPersona);**
- **nat alturaTABBPersonas(TABBPersonas abbPersonas);**
- **TPersona maxCITPersonaTABBPersonas(TABBPersonas abbPersonas);**
- **int cantidadTABBPersonas(TABBPersonas abbPersonas);**
- **TPersona obtenerNesimaPersonaTABBPersonas(TABBPersonas abbPersonas, int n);**
- **TABBPersonas filtradoPorFechaDeNacimientoTABBPersonas(TABBPersonas abbPersonas, TFecha fecha, int criterio);**