Programación 2

Estructuras Arborescentes

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*

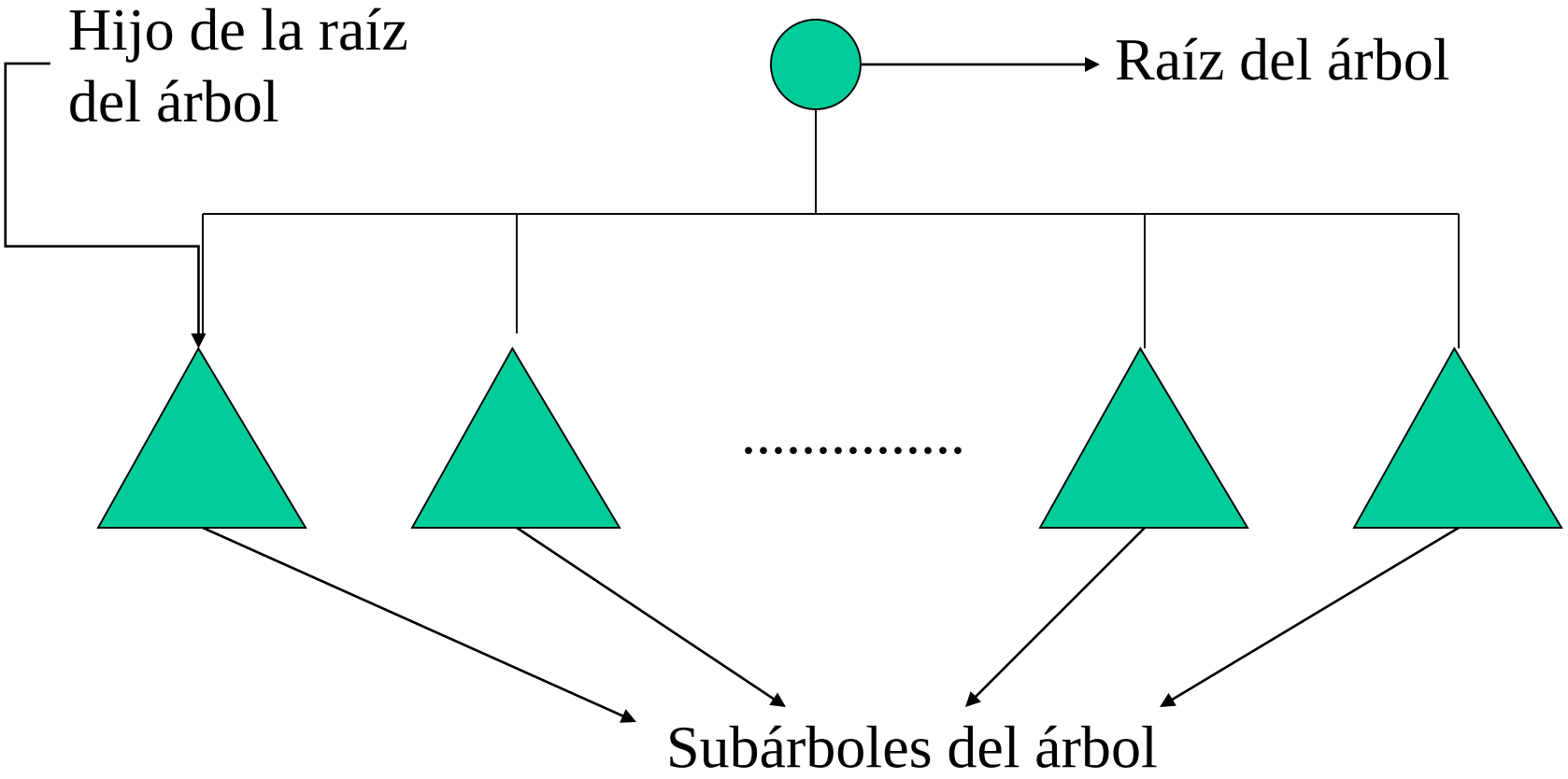
Conceptos básicos

Resulta evidente, a partir de la definición de secuencias (listas) y estructura árbol, que la lista es una estructura árbol en la cual cada nodo tiene a lo sumo un subárbol.

La lista es por ello también conocida por el nombre de árbol degenerado.

Veremos a continuación un poco de nomenclatura:

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 (dos) subárboles binarios.”

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

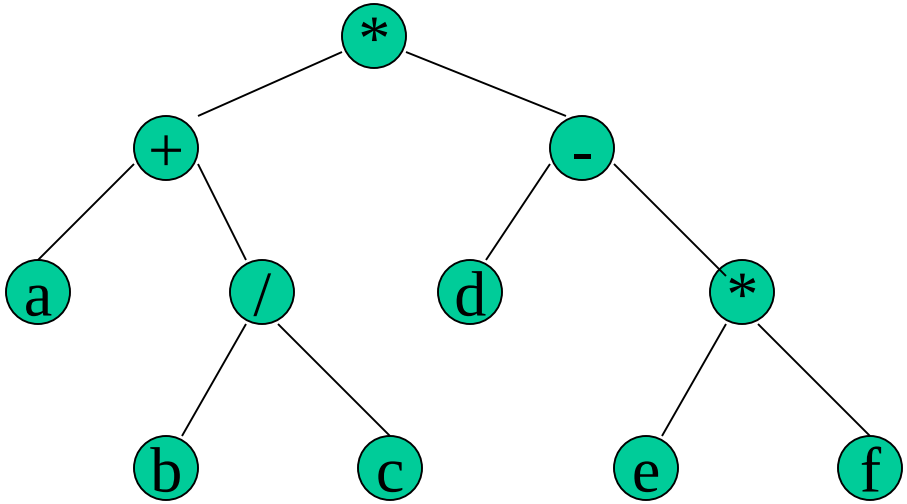
Arboles binarios

NOTA:

Comenzaremos primero a trabajar con árboles binarios, ya que al final de este módulo veremos que es posible representar árboles generales usando árboles binarios (con una semántica particular).

Ejemplo: árbol de expresiones
Sintaxis concreta vs. sintaxis abstracta

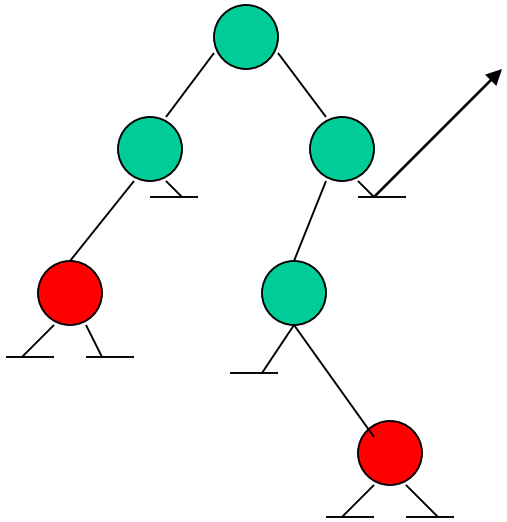
Representación no ambigua de expresiones aritméticas.



Representación arborescente de la fórmula:

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

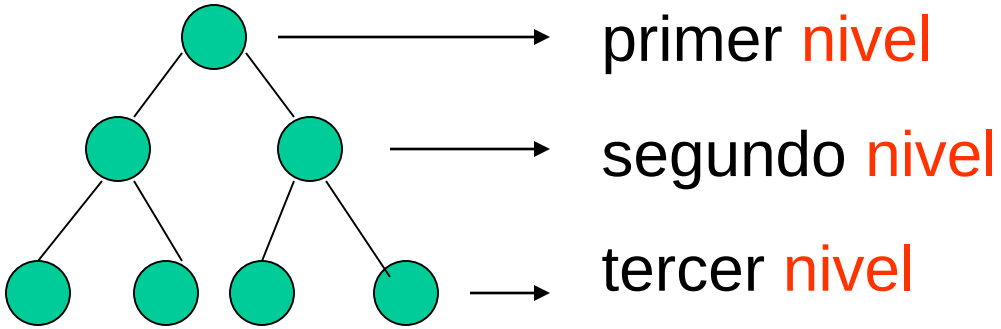
Hojas



Subárbol vacío

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

Niveles y altura

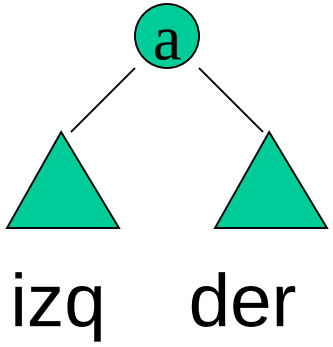


Def.
La *altura* de un árbol es:
la cantidad de niveles que tiene, o
la cantidad de nodos en el camino más largo de la raíz a una hoja.

La altura del árbol binario vacío es 0.

Altura (cont.)

La altura de un árbol de la forma:



es 1 + max(p_i, p_d), donde p_i es la altura de *izq* y p_d es la altura de *der*.

Más formalmente:

Altura (cont.)

Definición Inductiva de Árboles Binarios

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

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

Recorridas de árboles binarios

- En general, son procedimientos que visitan todos los nodos de un árbol binario efectuando cierta acción sobre cada uno de ellos
- La forma de estos procedimientos depende del orden que se elija para visitar los nodos
- Obviamente recorrer un árbol vacío es trivial

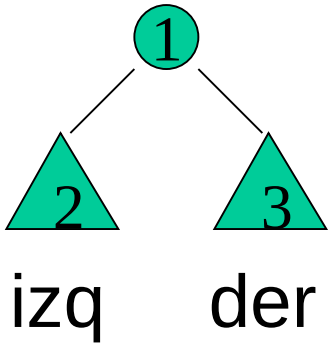
Recorridas de árboles binarios (cont.)

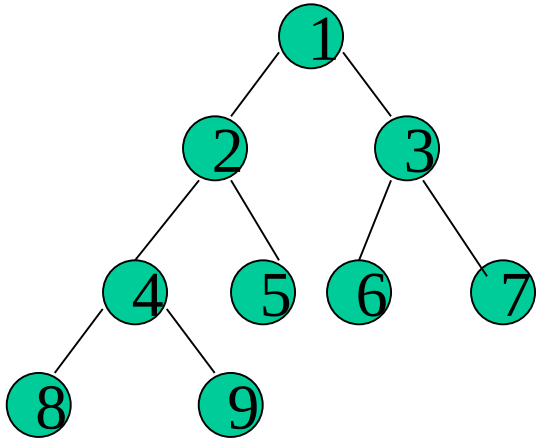
Para recorrer un árbol no vacío hay tres órdenes naturales, según la raíz sea visitada:

- 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)

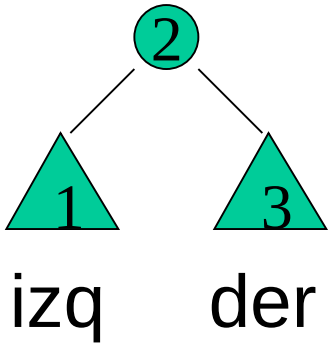


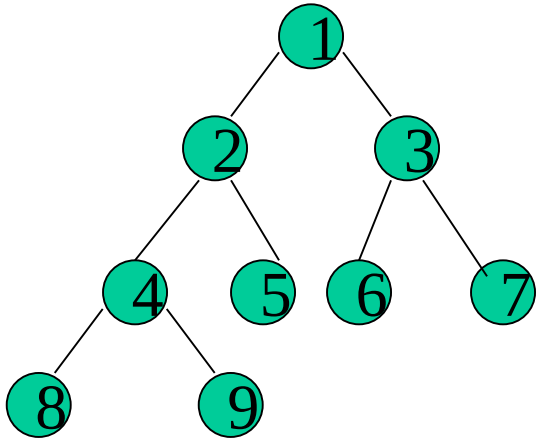


Preorder: 1, 2, 4, 8, 9, 5, 3, 6, 7

Recorridas de árboles binarios (cont.)

Antes que los subárboles (inorder)

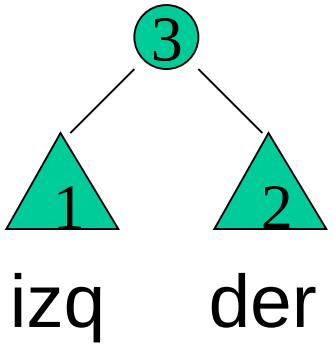


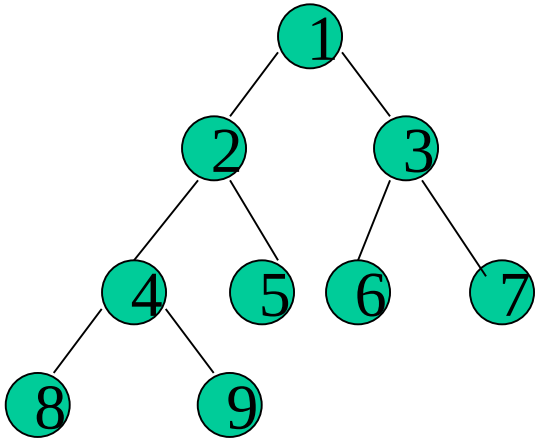


Inorder: 8, 4, 9, 2, 5, 1, 6, 3, 7

Recorridas de árboles binarios (cont.)

Antes que los subárboles (postorder)





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

Ejemplo

Generar la lista de elementos de un árbol binario que se obtiene al recorrerlo en orden:

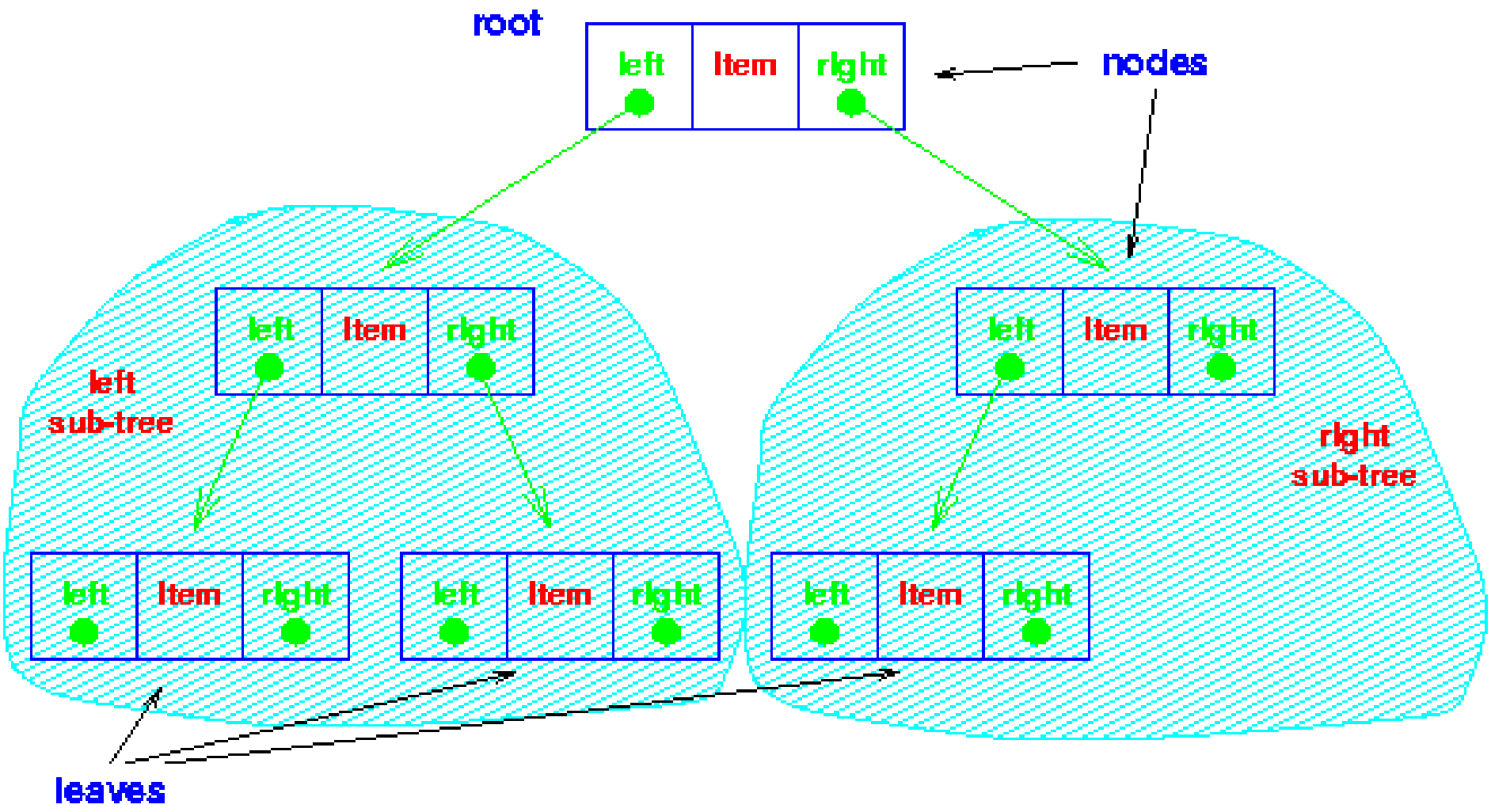
```
enOrden (()) = []
enOrden ((izq,a,der))
    = enOrden(izq) ++ (a . enOrden(der))

    = enOrden(izq) ++ [a] ++ enOrden(der)
```

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)



Recorridas

Formularemos ahora los 3 métodos de recorrida de árbol previamente presentados como procedimientos C++.

Usaremos un parámetro explícito t, que denotará el árbol sobre el cual se efectuará la recorrida.

Usaremos también un parámetro implícito P, que denotará la operación a ser aplicada sobre los elementos del árbol.

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

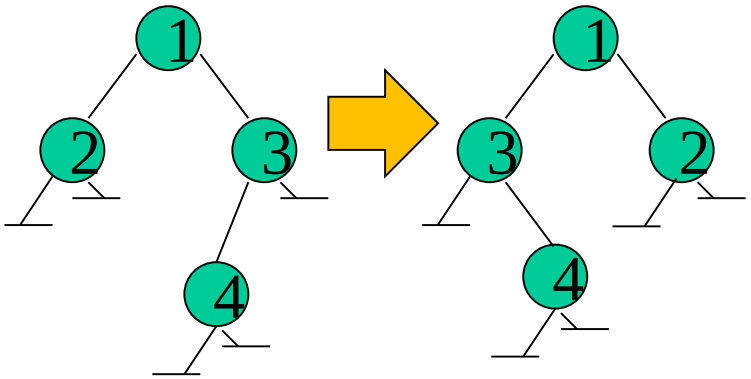
```
cantNodos (()) = 0
cantNodos ((izq,a,der)) =
      1 + cantNodos(izq) + cantNodos(der)
```

```
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?

Espejo

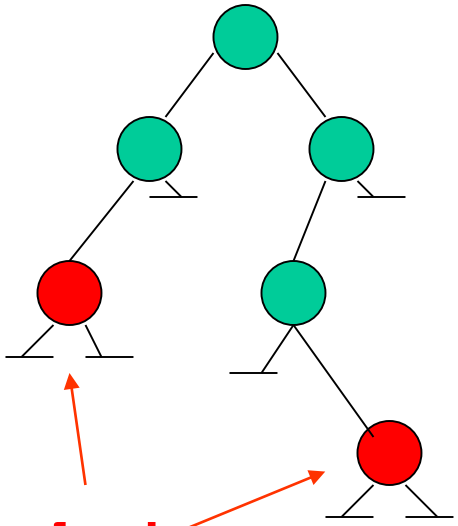
```
espejo (()) = ()
espejo ((izq,a,der)) =
  (espejo(der),a,espejo(izq))
```




```
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?

Hojas de un árbol




```
hojas (()) = []

hojas ((),a,())) = [a]

hojas ((izq,a,der)) =
                  hojas(izq) ++ hojas(der),
                  Si izq≠() o der≠()
```

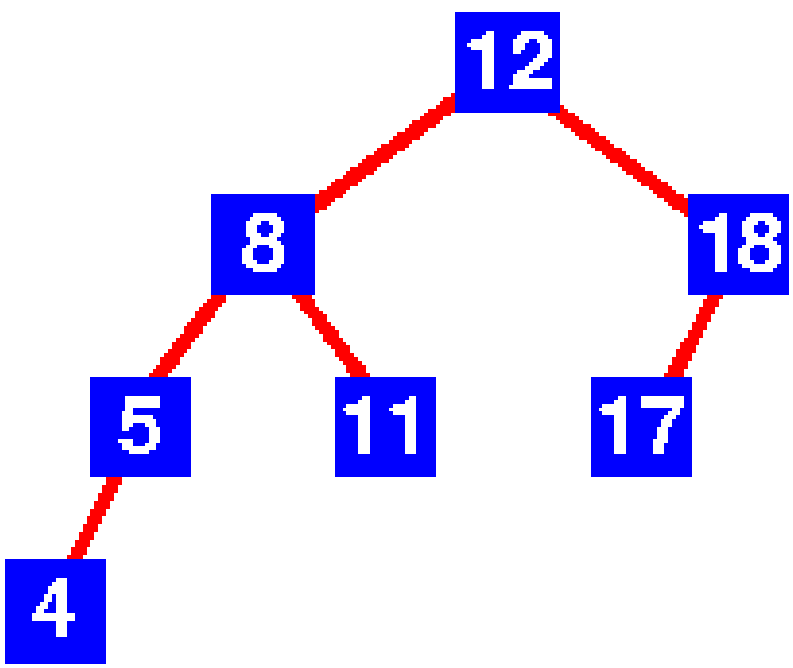
```
Lista hojas (AB t){
 Lista p;
 if (t == NULL) return NULL;
 else
    if ((t -> left == NULL)
         && (t -> right == NULL))
        { p = new nodoLista;
          p -> info = t -> item;
          p -> sig  = NULL;
          return p;
        }
    else return Concat(hojas(t -> left),
                       hojas(t -> right));
} // Nota: Concat es aquí una función (no un proced.)
```

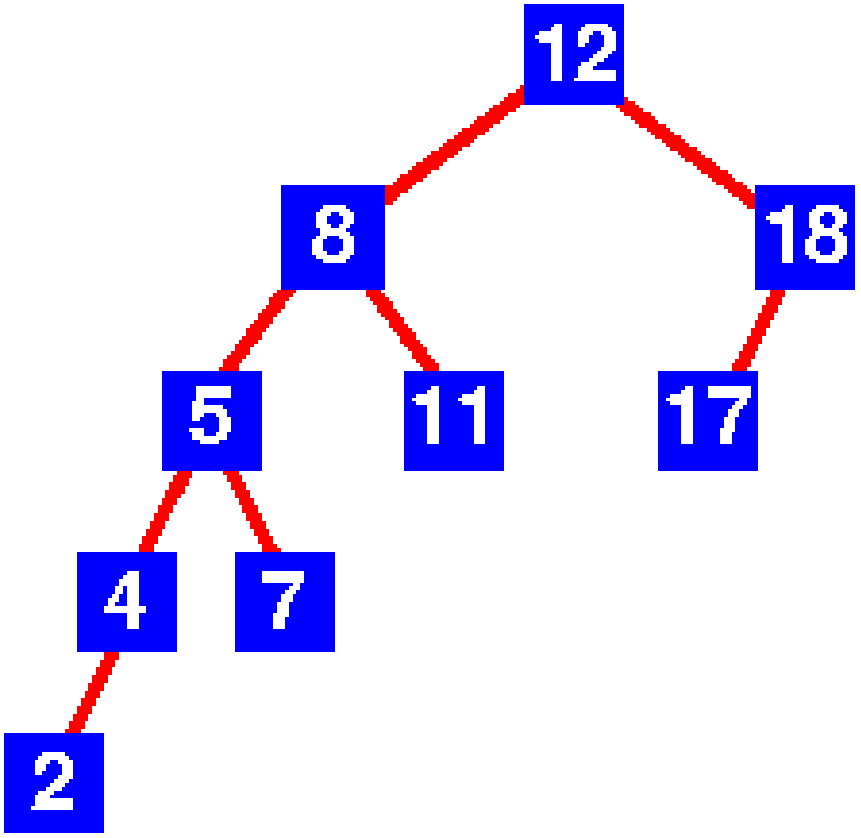
Arbol binario de búsqueda (ABB)

Arboles binarios son frecuentemente usados para representar conjuntos de datos cuyos elementos pueden ser recuperables (y por lo tanto identificables) por medio de una clave única.

Si un árbol está organizado de forma tal que para cada nodo n_i, todas las claves en el subárbol izquierdo de n_i son menores que la clave de n_i, y todas las claves en el subárbol derecho de n_i son mayores que la clave de n_i, entonces este árbol es un *ABB*.

Arbol binario de búsqueda (ABB): Ejemplos

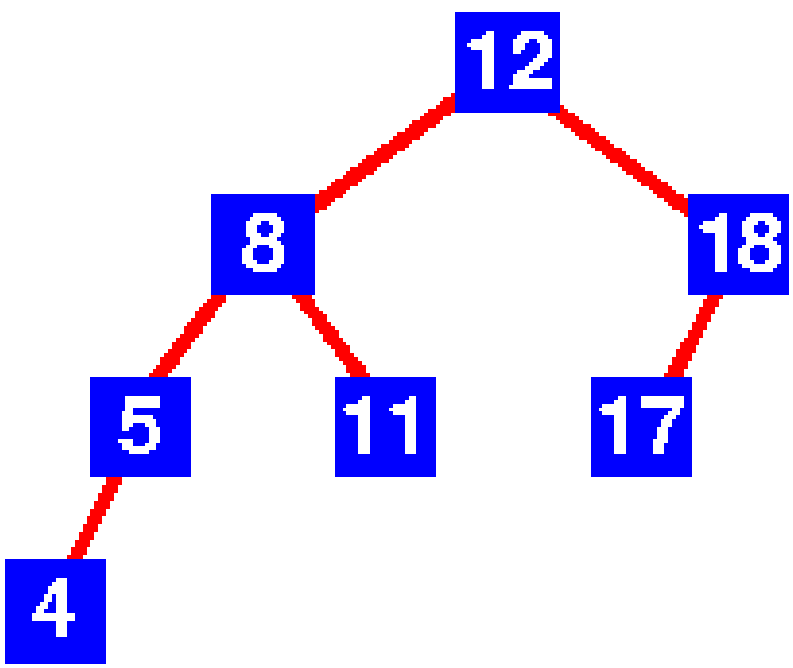


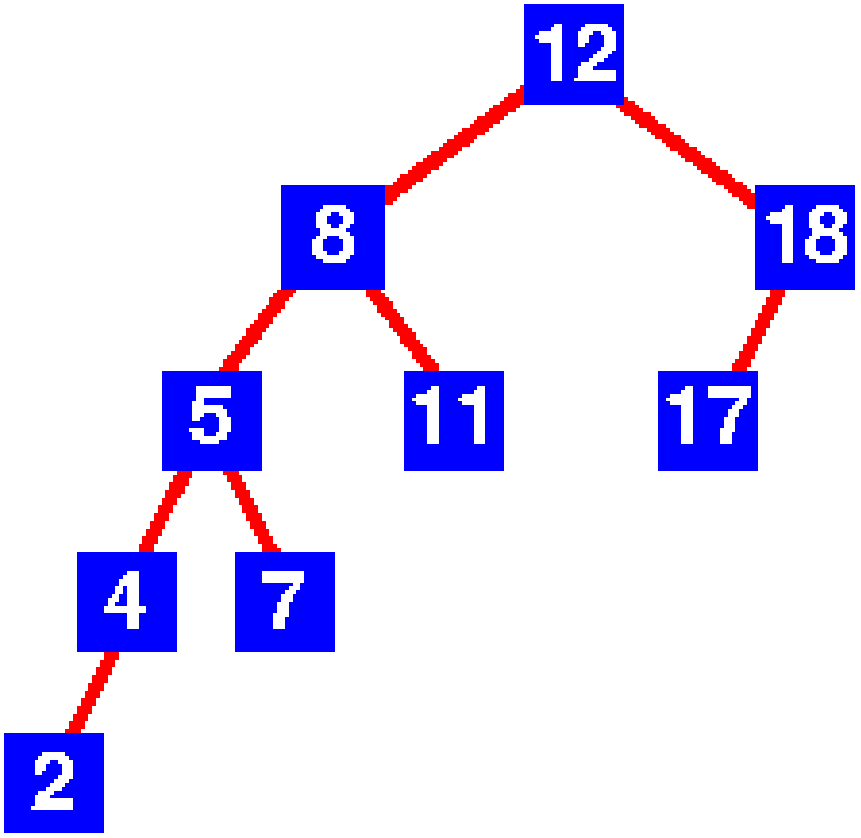


Analizar la relación con la búsqueda binaria sobre un arreglo ordenado (y sobre una lista).

4	5	8	11	12	17	18	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;
}
```

Búsqueda binaria

En un ABB es posible localizar (encontrar) un nodo de clave arbitraria comenzando desde la raíz del árbol y recorriendo un camino de búsqueda orientado hacia el subárbol izquierdo o derecho de cada nodo del camino dependiendo solamente del valor de la clave del nodo.

Como esta búsqueda sigue un único camino desde la raíz, puede ser fácilmente implementada en forma iterativa, como veremos a continuación:

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;
}
```

Buscar recursivo

```
ABB buscarRecursivo(Ord x, ABB t){
  ABB res;
  if (t == NULL)
   res = NULL;
  else
   if (x == t->key)
     res = t;
   else
     if (x < t->key)
           res = buscarRecursivo (x,t -> left);
     else
           res = buscarRecursivo (x,t-> right);
  return res;
}
```

¿Usaría esta versión recursiva?

Pertenencia

Hemos dicho que ABB se usan para representar conjuntos, o colección de elementos que son identificados únicamente por su clave.

Otra de las funciones características definidas sobre este tipo de árboles es la función que determina si existe un nodo del árbol cuya clave sea igual a una arbitraria dada (si el elemento pertenece al conjunto).

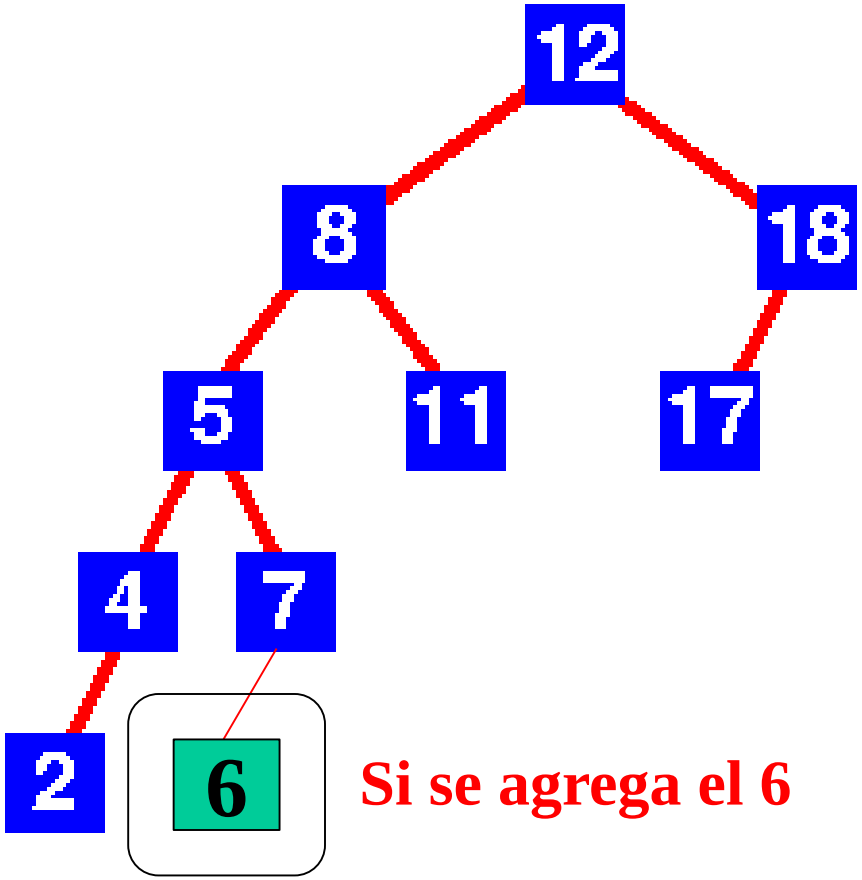
La función Miembro

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

¿Es eficiente la función *miembro*?

¿Podría optimizarse?

Inserción en un ABB

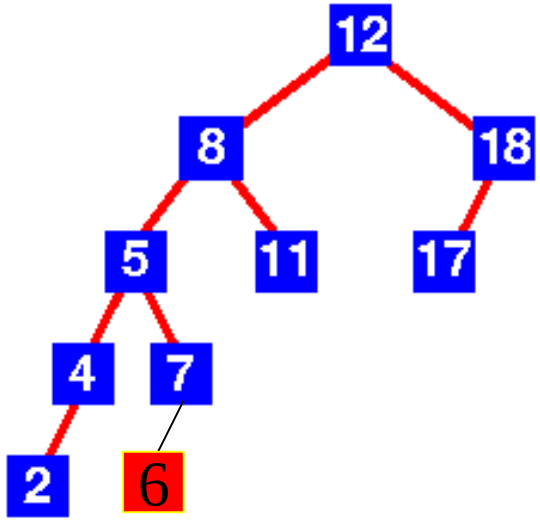


¿Siempre se agrega (eventualmente) un elemento como una hoja?

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

Inserción en un ABB

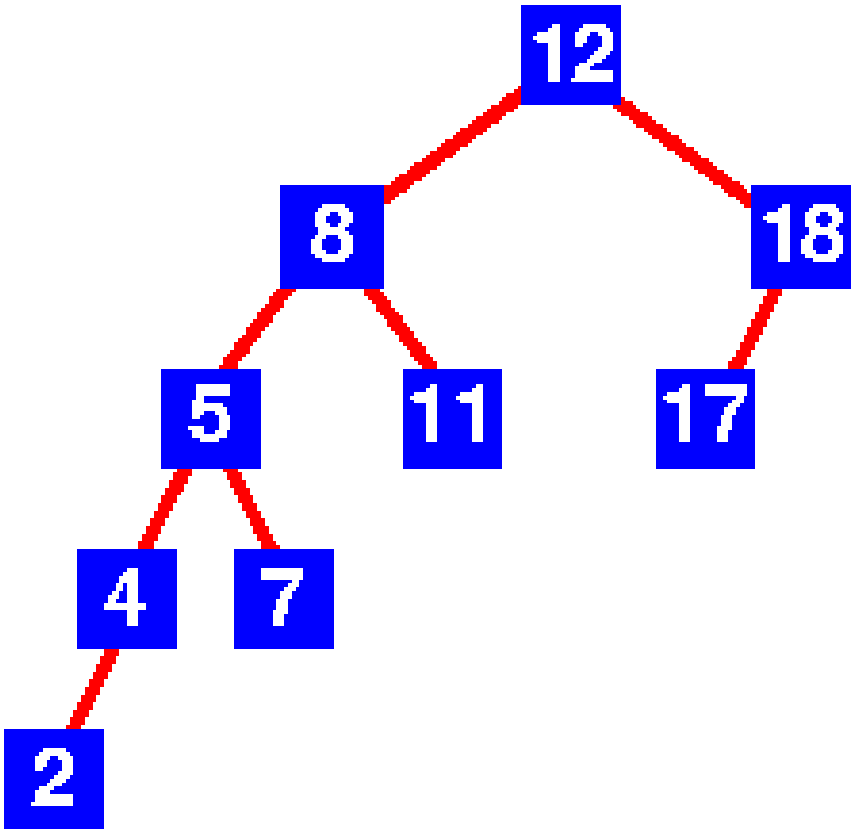
```
void insABB (Ord clave, T dato, ABB & t){
  if (t == NULL){
     t = new NodoABB ;
     t->key = clave;
     t->info = dato;
     t->left = t->right = NULL;
  }
  else if (clave < t->key)
      insABB (clave, dato, t->left);
  else if (clave > t->key)
      insABB (clave, dato, t->right);
}
```



Eliminación en un ABB

¿Cómo podría ser la eliminación de un elemento de un ABB?

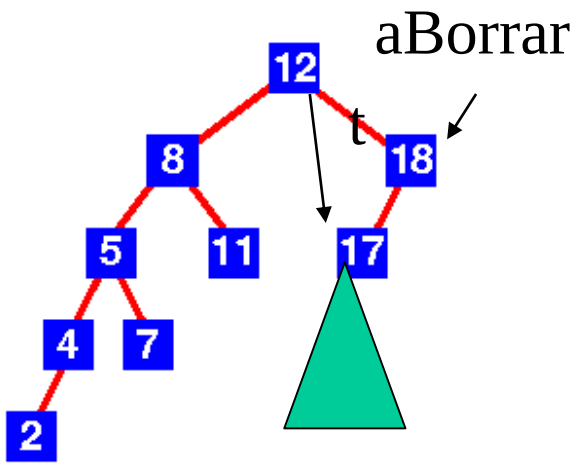
```
void elimABB (Ord clave, ABB & t){ ... }
```



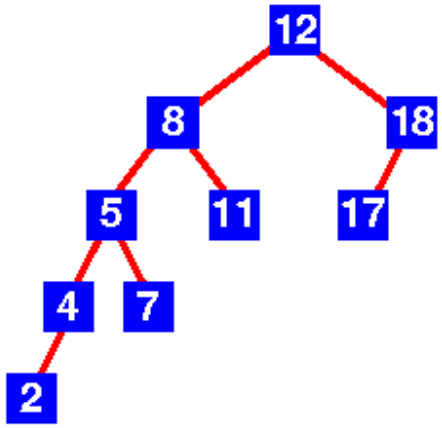
Eliminación en un ABB

Completar los casos

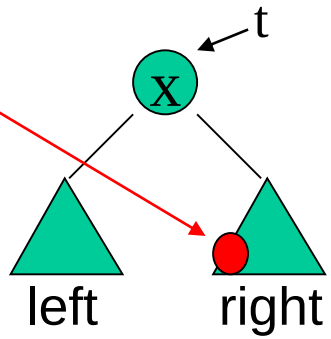
```
void elimABB (Ord clave, ABB & t){
   if (t!=NULL){
        if (clave < t->key)
              elimABB(…);
        else if (clave > t->key)
              elimABB(…);
        else { \\ clave == t->key
              if (t->right == NULL){
                    ABB aBorrar = t;
                    t = t->left;
                    delete aBorrar;
              }
              else … // ver la diapositiva siguiente
```



Eliminación en un ABB (cont.)

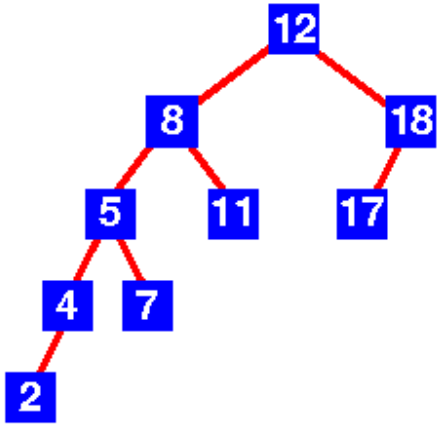



```
            else if (t->left == NULL){
            …
            }
            else {
                  ABB min_t_der = minimo(t->right);
                  t->key = …
                  t->info = …
                  elimABB(t->key,t->right);
            }
      }
   }
}
\\ retorna un puntero al nodo con el mínimo de t; NULL si es vacío.
ABB minimo (ABB t)
```



Eliminación en un ABB (cont.)

```
\\ retorna un puntero al nodo con el mínimo de t; NULL si es vacío.
ABB minimo (ABB t){
   if (t==NULL) return NULL;
   else{
        while(t->left!=NULL){
               t = t->left;
        }
        return t;
   }
}
```



Sobre el orden de tiempo de ejecución del peor caso y del caso promedio en ABBs

Sobre un ABB de *n* nodos:

- Si un algoritmo recorre todo el árbol, haciendo cosas de tiempo constante, tiene O(*n*) para el peor caso.
- En particular, si un algoritmo recorre solo un camino del árbol, su orden es *n* en el peor caso (árbol degenerado; como una lista).
- No obstante, si un algoritmo recorre solo un camino del árbol, su orden es log(*n*) en el caso promedio (asumiendo que todos los árboles son igualmente probables).

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:

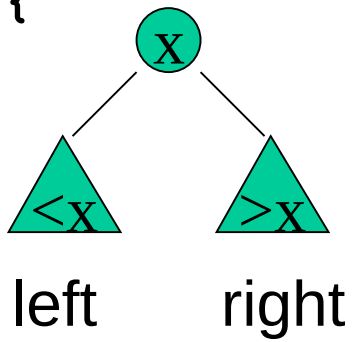
```
Lista aplanar (ABB t){
  Lista l = NULL;
  aplanarEnLista (t, l);
  return l;
}
```

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);
   }
}
```



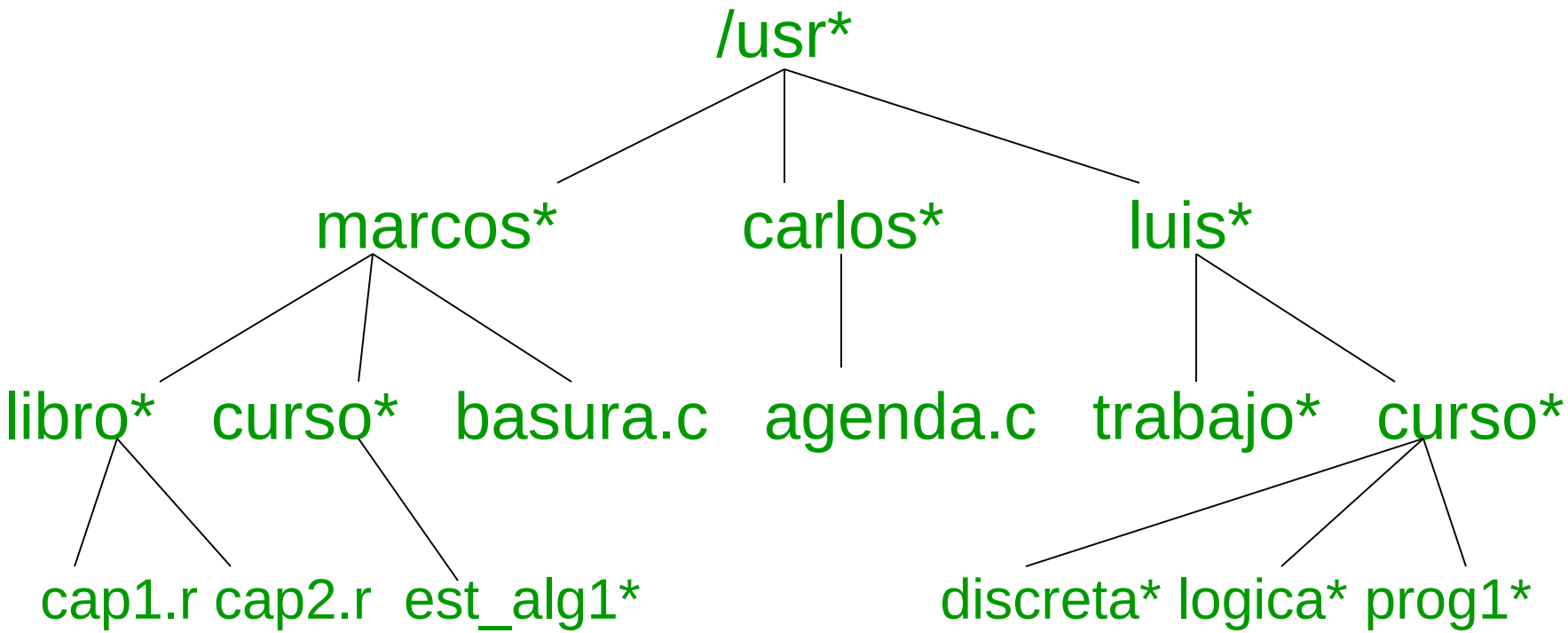
Arboles Generales.
Cómo implementarlos?

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

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

¿cómo representamos árboles generales?

Un ejemplo de árbol general: “estructura de directorios”



Una aplicación: listar un directorio

(recorridos de árboles)

¿Cómo representamos árboles generales?

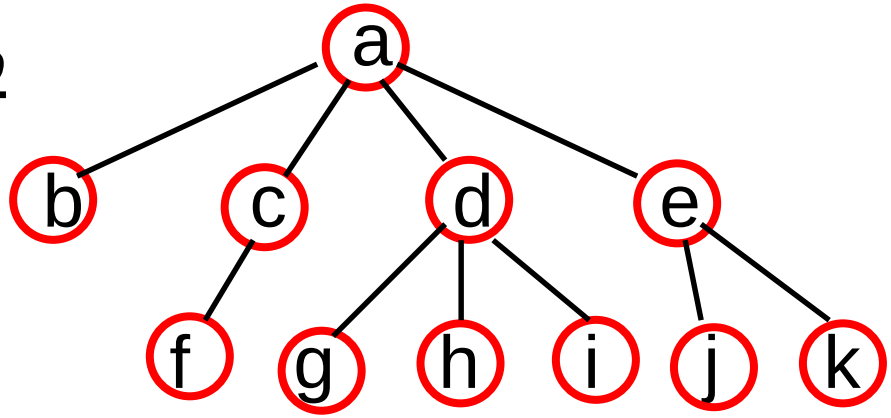
En un árbol general (o finitario) el número de hijos por nodo puede variar. Luego, una idea de representación consiste en pensar que cada nodo tiene una "**lista**" de árboles asociados (sus subárboles).

Una manera de hacer esto es considerando que cada nodo se relaciona con su "primer hijo (pH)" y con su "siguiente hermano (sH)", conformando una estructura de árbol binario.

Esta forma de representación establece una **equivalencia entre árboles generales y árboles binarios**: todo árbol general puede representarse como uno binario y, todo árbol binario corresponde a un determinado árbol general.

Arbol general - Arbol binario

Cada nodo tiene un núnero *finitio* de subárboles

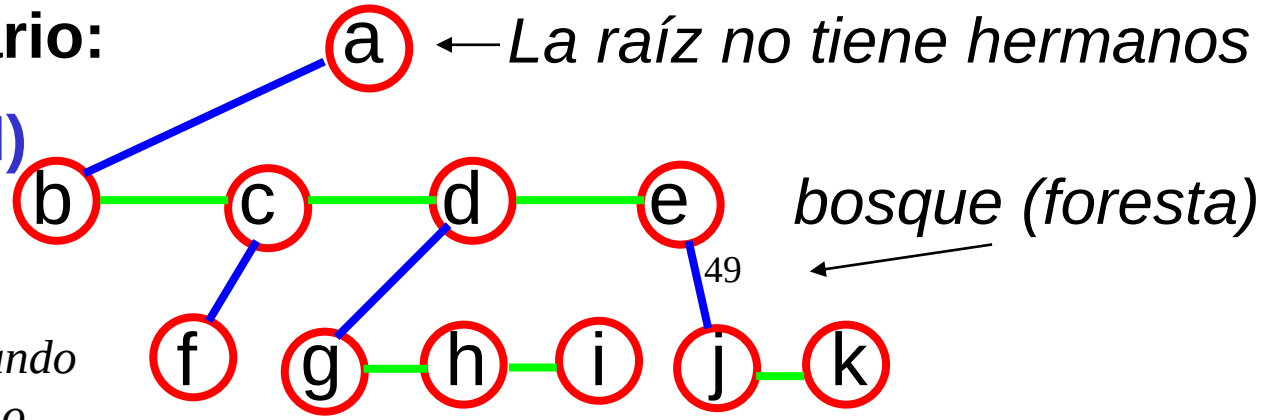


. . . . . .

Representación como árbol binario:

*** primer hijo (pH)**

*** siguiente hermano (sH)**



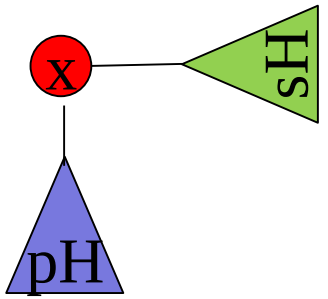
. . . . . .

Notar que estamos representando tanto bosques de árboles como árboles (bosques de un sólo árbol; la raíz).

Ejemplo 1

- La función contar nodos de un árbol general (recordar que la raíz no tiene hermanos).

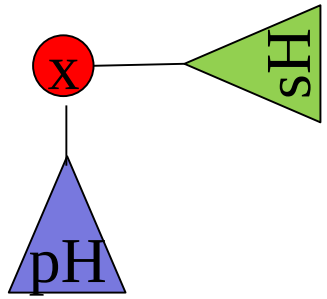
```
typedef NodoAG* AG;
struct NodoAG { T item; AG pH; AG sH; };

int nodos (AG t){
   if (t == NULL) return 0;
   else return nodos(t->pH)+nodos(t->sH)+1;
}
```



Notar que:

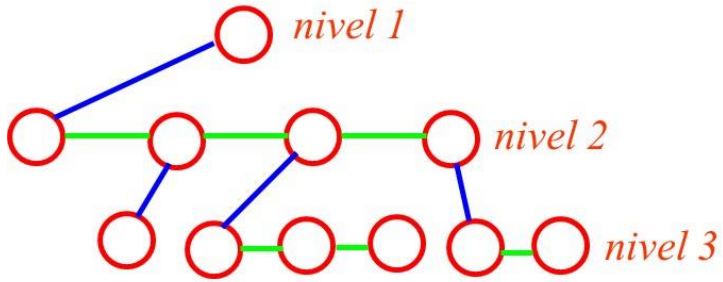
- **el código es idéntico al de contar nodos en un árbol binario tradicional.**
- **si se invoca a *nodos* con un bosque *t* (*t->sH!=NULL*), se tiene la cantidad de nodos del bosque.**

Ejemplo 2



- La función altura de un árbol general (recordar que la raíz no tiene hermanos).

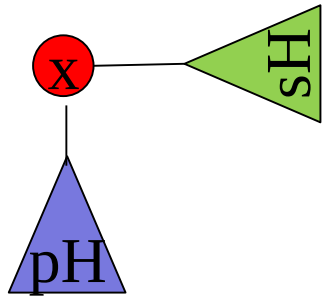
```
int altura (AG t){
   if (t == NULL) return 0;
   else return 1+MAX(altura(t->pH),
                     altura(t->sH));
}
```



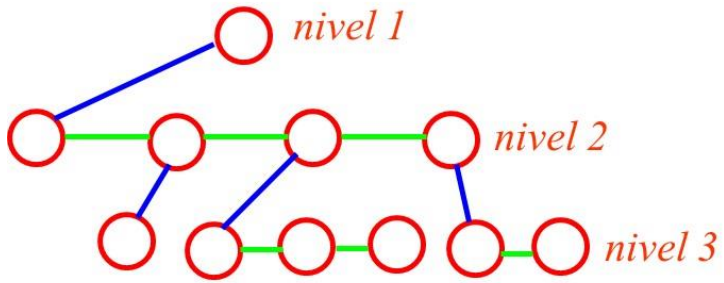
¿Es correcta? ¿Qué retorna para el árbol ejemplo?

Ejemplo 2

- La función altura de un árbol general (recordar que la raíz no tiene hermanos).




```
int altura (AG t){
   if (t == NULL) return 0;
   else return MAX(1+altura(t->pH),
                   altura(t->sH));
}
```



Notar que el código NO es idéntico al de la altura de un árbol binario tradicional:

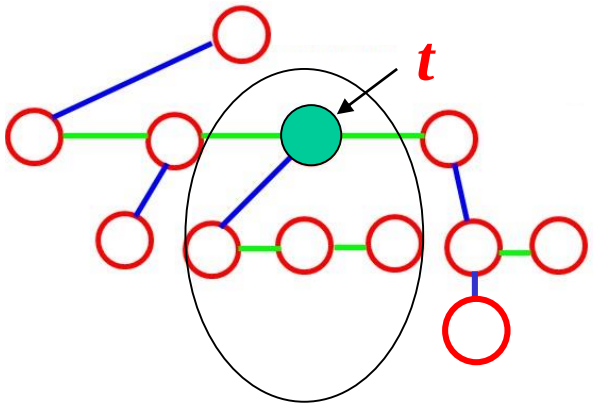
- "primer hijo" (**pH**) aumenta la altura.
- "siguiente hermano" (**sH**) no aumenta la altura, la mantiene (ver dibujo).

Si se invoca a la función *altura* con un bosque *t* (*t->sH!=NULL*), se tiene la altura del bosque.

Ejemplo 2 – Altura de un (sub)árbol

Para calcular la altura de un (sub)árbol con t como raíz en la representación pH-sH podemos hacer:

```
int alturaArbol(AG t){
   if (t == NULL) return 0;
   else return 1 + altura(t->pH);
}
```



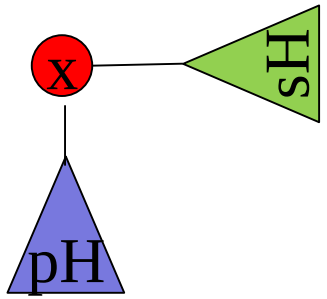
Notar que en el ejemplo:

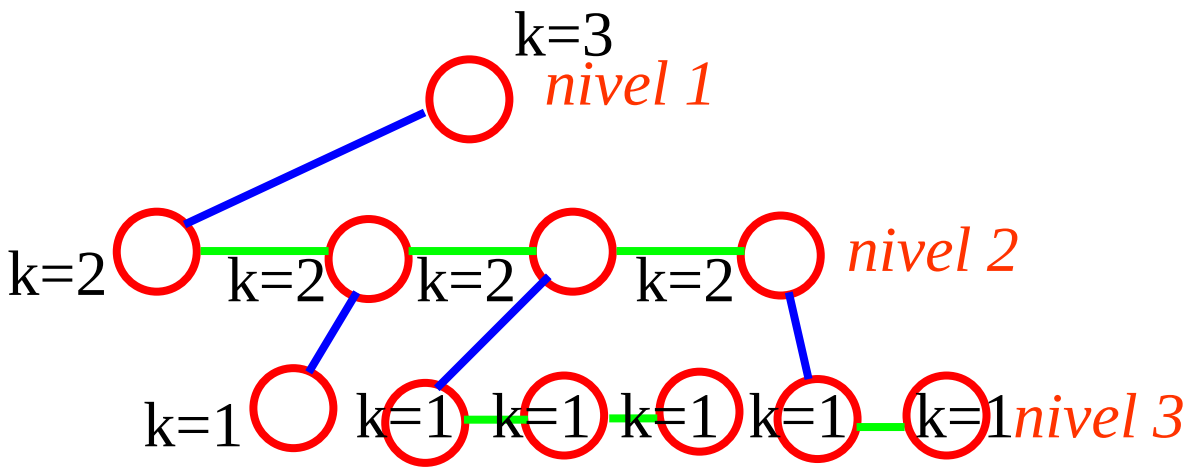
- **alturaArbol(t) retorna 2.**
- **altura(t) retorna 3.**

Ejemplo 3

- Imprime los elementos en el nivel k de un árbol general, asumiendo que la raíz de un árbol general no vacío está en el nivel 1 (recordar que la raíz no tiene hermanos).

```
void impNivel(AG t, int k){
   if (t != NULL && k>0){
        if (k == 1) cout << t->item;
        else impNivel(t->pH, k-1);
        impNivel(t->sH, k);
   }
}
```

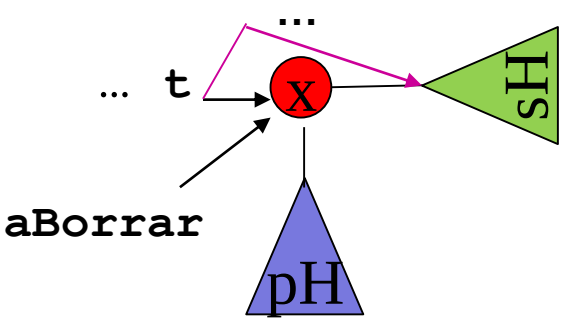




Ejemplo 4

- Elimina cada subárbol que tiene a x como raíz de un árbol general (recordar que en la representación *pH-sH*, el nodo raíz principal no tiene hermanos).

```
void elim(AG & t, T x){
   if (t != NULL){
        if (t->item == x){
                AG aBorrar = t;
                t = t->sH;
                elimTodo(aBorrar->pH);
                delete aBorrar;
                elim(t, x);
        }else{ elim(t->pH, x);
                elim(t->SH, x);
        }
   }
}
```




```
void elimTodo(AG & t){
  if (t != NULL){
    elimTodo(t->pH);
    elimTodo(t->sH);
    delete t;
    t = NULL;
  }
}
```

Ejemplo 4 (variante)

Considere y analice la siguiente solución alternativa del problema anterior, que distingue el tratamiento de árboles y bosques:

```
void destruirArbol(AG & a);
void destruirBosque(AG & a);

// Precondición t != NULL
void destruirArbol(AG & t) {
  destruirBosque(t->pH);
  delete t;
  t = NULL;
}

void destruirBosque(AG & t) {
  if (t != NULL) {
    destruirBosque(t->sH);
    destruirArbol(t);
  }
}
```

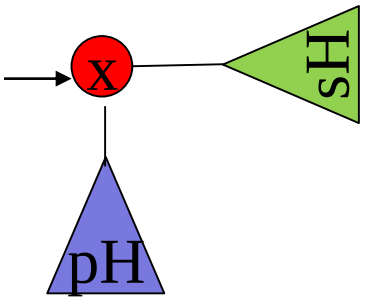
Ejemplo 4 (variante – cont.)

```
// Eliminan los subarboles que tengan x en la raíz
void elimArbol(AG & t, int x);
void elimBosque(AG & t, int x);

// Se trata a 't' como árbol, no se usa t->sH
// t está compuesto por un elemento y un bosque de subárboles
// Precondición t != NULL
void elimArbol(AG & t, T x) {
  if (t->dato == x) {
    destruirArbol(t);
    t = NULL; // por si no lo hiciera destruirArbol
  } else
    elimBosque(t->pH, x);
}

// Se trata a 't' como un bosque
// Si t no es vacío, está compuesto por un primer árbol y un bosque.
void elimBosque(AG & t, T x){
  if (t != NULL){
    elimBosque(t->sH, x);
    AG sig = t->sH; // se obtiene el siguiente porque t puede eliminarse
    elimArbol(t,x);
    if (t == NULL) {
      t = sig;
    }
  }
}
```

Ejemplo 5

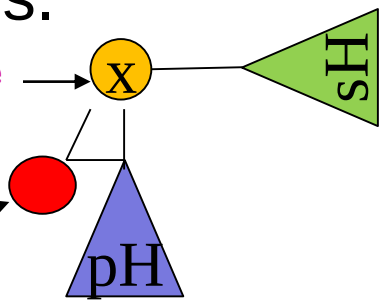
- Dados un árbol general (pH-sH) y un elemento, retorna un puntero al nodo del árbol que tiene dicho elemento o NULL si no está. Asumimos que el árbol no tiene repetidos.

```
AG buscar(AG t, T x){
   if (t == NULL)
        return NULL;
   else {
        if (t->item != x){
              AG esta_sH = buscar(t->sH, x);
              if (esta_sH != NULL)
                    return esta_sH;
              else return buscar(t->pH, x);
        }
        else return t;
   }
}
```



Ejemplo 6

- Dados un árbol general (pH-sH) y dos elementos “padre” e “hijo”, agregar a “hijo” como primer hijo de “padre” en el árbol si: “padre” está e “hijo” no está. Asumimos que el árbol no tiene repetidos.

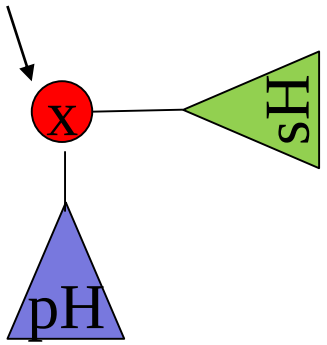


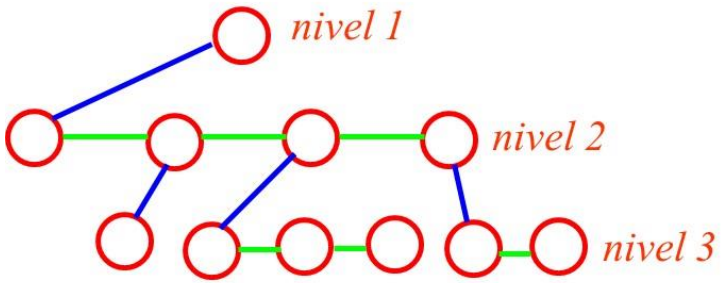

```
void insertarAG(AG & t, T padre, T hijo){
   if (buscar(t,hijo) == NULL){
        AG nodo_padre = buscar(t,padre);
        if (nodo_padre != NULL){
              AG nodo_nuevo = new nodoAG;
              nodo_nuevo->ítem = hijo;
              nodo_nuevo->pH = NULL;
              nodo_nuevo->sH = nodo_padre->pH;
              nodo_padre->pH = nodo_nuevo;
        }
   }
}
```

Ejemplo 7

- Dado un árbol general (pH-sH) y un entero no negativo k, retorna una copia del árbol, sin compartir memoria, hasta el nivel k inclusive.

```
AG copiaParcial (AG t, unsigned int k){
        if (k==0 || t==NULL)
                return NULL;
        else {  AG nuevo = new nodoAG;
                nuevo->item = t->ítem;
                nuevo->pH = copiaParcial (t->pH, k-1);
                nuevo->sH = copiaParcial (t->sH, k);
                return nuevo;
        }
}
```

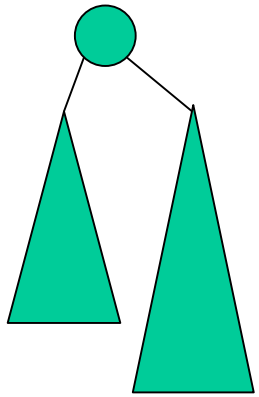




Ejemplo 7 – variante (en AB)

- Dado un árbol binario (left-right) y un entero no negativo k, retorna una copia del árbol, sin compartir memoria, hasta el nivel k inclusive.

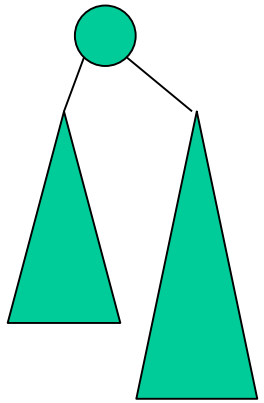
```
AB copiaParcial (AB t, unsigned int k){
        if (k==0 || t==NULL)
                return NULL;
        else {  AB nuevo = new nodoAB;
                nuevo->item = t->item;
                nuevo->left = copiaParcial (t->left, k-1);
                nuevo->right = copiaParcial (t->right, k-1);
                return nuevo;
        }
}
```



Ejemplo 7 – otra variante (en AB)

- Dado un árbol binario (left-right) retorna una copia del árbol, sin compartir memoria.

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



Bibliografía Recomendada

Todos los libros propuestos por la cátedra. *En particular el libro de Weiss*.