Departamento
de Ingeniería de la
Información y las
Comunicaciones
UMU

Tecnología de la Programación

Título de Grado en Ingeniería Informática

Curso 2024/25

Grupos 2, 3 y 4

Tema 5. Estructuras de Datos Enlazadas Arborescentes

Dept. Ingeniería de la Información y las Comunicaciones

Universidad de Murcia

TEMA 5. ESTRUCTURAS DE DATOS ENLAZADAS ARBORESCENTES

5.1 Estructuras de datos enlazadas arborescentes (1/19)

Algunos problemas requieren el uso de estructuras de datos enlazadas en las cuales no se asume la relación sucesor - predecesor de los datos, sino otro tipo de jerarquías que establecen ***relaciones de paternidad*** (por ejemplo inclusión) entre los datos como son las ***estructuras arborescentes***.

Cada uno de los elementos relacionados en el árbol se le denomina ***nodo*** (o ***vértice***). La relación entre los nodos se representa mediante una ***arista***.

Las estructuras arborescentes pueden definirse típicamente de dos formas: como ***árboles binarios*** o como ***árboles generales*** (***n-arios***). Mientras que en los primeros cada ***nodo*** del árbol tiene un ***hijo izquierdo*** y/o un ***hijo derecho***, en los generales cada nodo tiene un número indeterminado de hijos ordenados desde el ***hijo de más a la izquierda*** hasta el ***hijo de más a la derecha***.

Formalmente, un árbol se define como un ***grafo no dirigido conexo acíclico***. Un ***árbol con raíz*** es un árbol en el cual un nodo ha sido designado como raíz. En este caso, las aristas tienen una orientación natural desde la raíz hacia el resto de los nodos.

5.1 Estructuras de datos enlazadas arborescentes (2/19)

Un nodo puede tener cero o más nodos hijos conectados a él. Se dice entonces que un nodo *a* es ***padre*** de un nodo *b*, y el nodo *b* es ***hijo*** del nodo *a*, si existe una arista desde el nodo *a* hasta el nodo *b*. Existe un único nodo sin padre que se denomina ***raíz***. Un nodo sin hijos se denomina ***hoja***.

Un ***camino*** es una sucesión de nodos $n_1, n_2, \ldots, n_k$ en el árbol de forma que el nodo n_i es padre del nodo n_{i+1} para todo $i=1,\ldots,k-1$. La ***longitud*** del camino es el número de nodos del camino menos uno (número de aristas). Si existe un camino entre un nodo *a* y un nodo *b*, se dice que *a* es ***ancestro*** de *b* y *b* es ***descendiente*** de *a*.

Un ***subárbol de un árbol*** es el árbol compuesto por cualquier nodo del árbol y todos sus descendientes.

La ***altura de un nodo*** es la longitud del camino más largo desde ese nodo a cualquier hoja. La ***altura de un árbol*** es la altura de su raíz

La ***profundidad de un nodo*** es la longitud del camino desde la raíz al nodo.

En un árbol con altura *h*, existen *h+1* ***niveles*** (desde *0* hasta *h*) en donde cada nivel *i* es el conjunto de nodos que están a profundidad *i*.

5.1 Estructuras de datos enlazadas arborescentes (3/19)

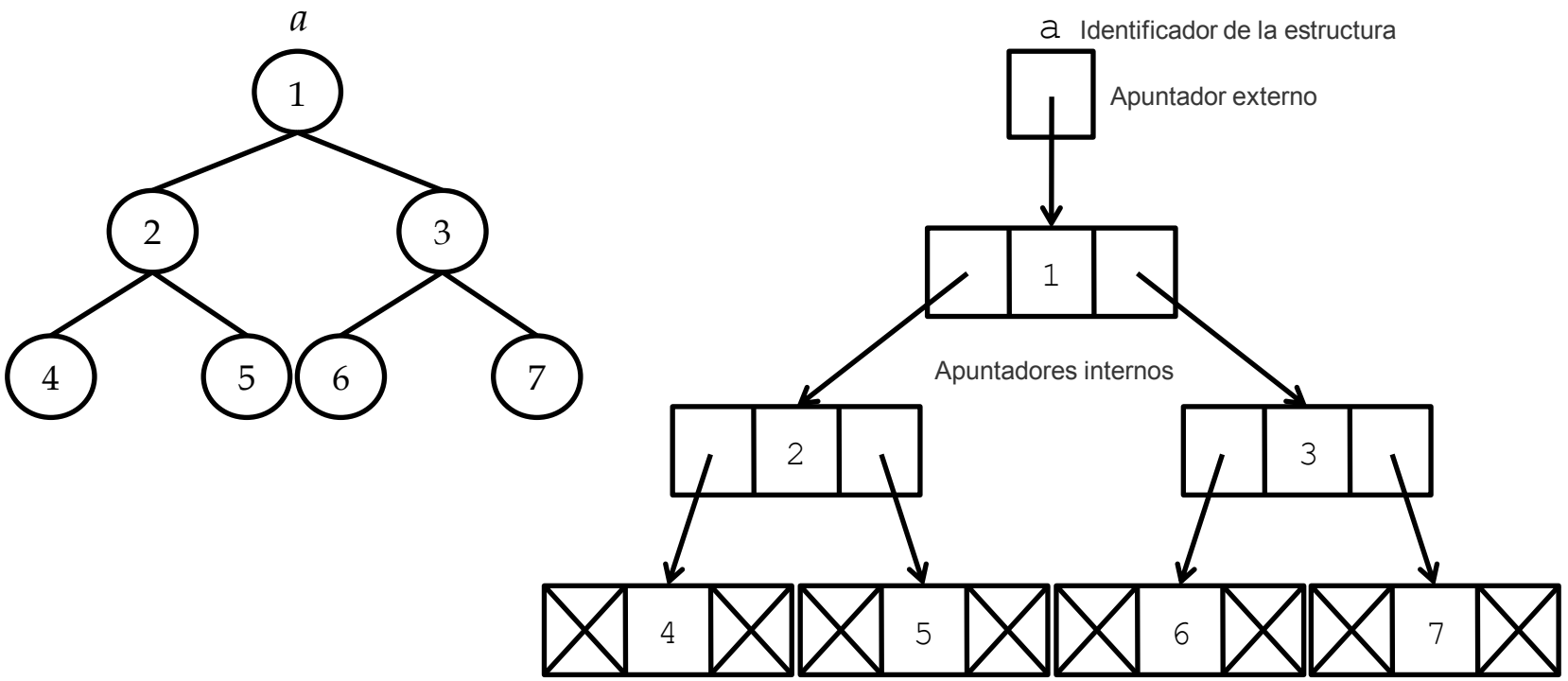
El siguiente tipo `ArbolBinario` define una ***estructura de datos enlazada arborescente binaria***:

```
struct Nodo
{
     Elemento elem;
     struct Nodo * hijoIzquierdo, * hijoDerecho;
};
typedef struct Nodo * ArbolBinario;
```

En esta estructura, cada nodo tiene un apuntador a su hijo izquierdo y un apuntador a su hijo derecho, los cuales puede tomar el valor `NULL` cuando uno de estos hijos no exista. Un nodo sin hijo izquierdo ni hijo derecho es un nodo **hoja**. Una variable de tipo `ArbolBinario` es un apuntador al nodo **raíz,** situado en lo más alto de la estructura. A este tipo de representación se le denomina ***representación hijo izquierdo – hijo derecho***.

5.1 Estructuras de datos enlazadas arborescentes (4/19)

Representación gráfica *Árbol Binario a*



5.1 Estructuras de datos enlazadas arborescentes (5/19)

Construcción de un árbol binario con un solo nodo raíz

```
ArbolBinario crea_arbol(Elemento elem)
{
     ArbolBinario nuevo = malloc(sizeof(struct Nodo));
     nuevo->elem = elem;
     nuevo->hijoIzquierdo = NULL;
     nuevo->hijoDerecho = NULL;
     return nuevo;
}
```

Programa que construye un árbol binario a partir de tres subárboles de un solo nodo

```
int main()
{
     ArbolBinario raiz = crea_arbol(0);
     raiz->hijoIzquierdo = crea_arbol(1);
     raiz->hijoDerecho = crea_arbol(2);
     return 0;
}
```

5.1 Estructuras de datos enlazadas arborescentes (6/19)

Recorridos en profundidad en un árbol binario

```
void preorden(ArbolBinario a){
      if (a == NULL) return;
      printf("%d\n", a->elem);
      preorden(a->hijoIzquierdo);
      preorden(a->hijoDerecho);
}
```

```
void inorden(ArbolBinario a){
      if (a == NULL) return;
      inorden(a->hijoIzquierdo);
      printf("%d\n", a->elem);
      inorden(a->hijoDerecho);
}
```

```
void postorden(ArbolBinario a){
      if (a == NULL) return;
      postorden(a->hijoIzquierdo);
      postorden(a->hijoDerecho);
      printf("%d\n", a->elem);
}
```

5.1 Estructuras de datos enlazadas arborescentes (7/19)

Recorrido en anchura de un árbol binario

```
#include "Cola.h"
void recorrido_anchura( ArbolBinario a ) {
     if (a==NULL) return;
     Cola c = Cola_crea();
     Cola_inserta( c, a );
     while ( !Cola_vacia( c ) ) {
          ArbolBinario aux = Cola_recupera( c );
          printf( "%d\n", aux->elem );
          Cola_suprime( c );
          if ( aux->hijoIzquierdo ) {
               Cola_inserta( c, aux->hijoIzquierdo );
          }
          if ( aux->hijoDerecho ) {
               Cola_inserta( c, aux->hijoDerecho );
          }
     }
     Cola_libera( c );
}
```

5.1 Estructuras de datos enlazadas arborescentes (8/19)

Liberación de un árbol binario

```
void libera(ArbolBinario a) // postorden
{
     if (a == NULL) return;
     libera(a->hijoIzquierdo);
     libera(a->hijoDerecho);
     free(a);
}
```

Búsqueda de un valor en un árbol binario

```
int busca(ArbolBinario a, Elemento elem)
{
     if (a == NULL) return 0;
     if (a->elem == elem ) return 1;
     return (busca(a->hijoIzquierdo,elem))||
            (busca(a->hijoDerecho,elem));
}
```

5.1 Estructuras de datos enlazadas arborescentes (9/19)

Copia de un árbol binario

```
ArbolBinario copia(ArbolBinario a){
     if (a == NULL) return NULL;
     ArbolBinario c = crea_arbol(a-> elem);
     c->hijoIzquierdo = copia(a->hijoIzquierdo);
     c->hijoDerecho = copia(a->hijoDerecho);
     return c;
}
```

Comparación de igualdad de dos árboles binarios

```
int iguales(ArbolBinario a, ArbolBinario b) {
     if (a == NULL && b == NULL) return 1;
     if (a == NULL || b == NULL) return 0;
     return (a->elem == b->elem) &&
            iguales(a->hijoIzquierdo,b->hijoIzquierdo) &&
            iguales(a->hijoDerecho,b->hijoDerecho);
}
```

5.1 Estructuras de datos enlazadas arborescentes (10/19)

Altura de un árbol binario

```
int altura(ArbolBinario a)
{
     if (a == NULL) return -1;
     int ahi = altura(a->hijoIzquierdo);
     int ahd = altura(a->hijoDerecho);
     if (ahi > ahd) return ahi + 1;
     return ahd + 1;
}
```

Número de hojas de un árbol binario

```
int nHojas(ArbolBinario a)
{
     if (a == NULL) return 0;
     if (a->hijoIzquierdo==NULL && a->hijoDerecho==NULL) return 1;
     return nHojas(a->hijoIzquierdo) + nHojas(a->hijoDerecho);
}
```

5.1 Estructuras de datos enlazadas arborescentes (11/19)

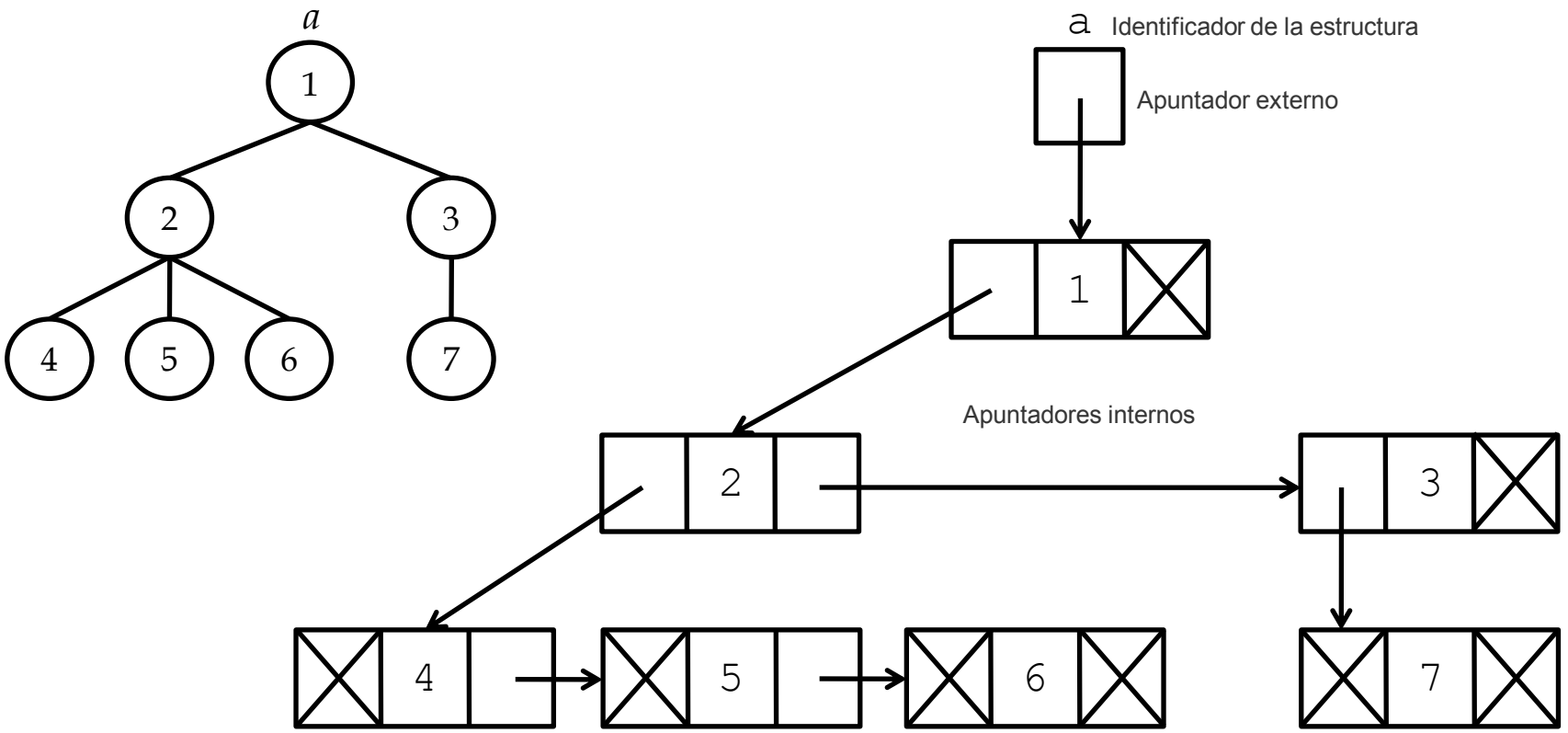
El siguiente tipo `ArbolGeneral` define una ***estructura de datos enlazada arborescente general***:

```
struct Nodo
{
     Elemento elem;
     struct Nodo * hijoIzquierdo, * hermanoDerecho;
};
typedef struct Nodo * ArbolGeneral;
```

En esta estructura cada nodo tiene un apuntador a su hijo de más a la izquierda y un apuntador a su hermano derecho. Cuando un nodo no tiene hijos (es un nodo hoja) el apuntador `hijoIzquierdo` toma el valor `NULL`, mientras que cuando un nodo no tiene hermano derecho es el apuntador `hermanoDerecho` el que toma el valor `NULL`. A esta representación se le denomina ***representación hijo izquierdo – hermano derecho***.

5.1 Estructuras de datos enlazadas arborescentes (12/19)

Representación gráfica *Árbol General a*



5.1 Estructuras de datos enlazadas arborescentes (13/19)

Construcción de un árbol general con un solo nodo raíz

```
ArbolGeneral crea_arbol(Elemento elem)
{
     ArbolGeneral nuevo = malloc(sizeof(struct Nodo));
     nuevo->elem = elem;
     nuevo->hijoIzquierdo = NULL;
     nuevo->hermanoDerecho = NULL;
     return nuevo;
}
```

Programa que construye un árbol general a partir de tres subárboles de un solo nodo

```
int main()
{
     ArbolGeneral raiz = crea_arbol(0);
     raiz->hijoIzquierdo = crea_arbol(1);
     raiz->hijoIzquierdo->hermanoDerecho = crea_arbol(2);
     return 0;
}
```

5.1 Estructuras de datos enlazadas arborescentes (14/19)

Recorridos en profundidad en un árbol general

```
void preorden(ArbolGeneral a) {
     if (a == NULL) return;
     printf("%d\n", a->elem);
     ArbolGeneral aux = a->hijoIzquierdo;
     while (aux != NULL) {
          preorden(aux);
          aux = aux->hermanoDerecho;
     }
}
```

```
void postorden(ArbolGeneral a) {
     if (a == NULL) return;
     ArbolGeneral aux = a->hijoIzquierdo;
     while (aux != NULL) {
          postorden(aux);
          aux = aux->hermanoDerecho;
     }
     printf("%d\n", a->elem);
}
```

5.1 Estructuras de datos enlazadas arborescentes (15/19)

Recorridos en profundidad en un árbol general

```
void inorden(ArbolGeneral a)
{
     if (a == NULL) return;
     inorden(a->hijoIzquierdo);
     printf("%d\n", a->elem);
     ArbolGeneral aux = a->hijoIzquierdo;
     while (aux != NULL)
     {
          aux = aux->hermanoDerecho;
          inorden(aux);
     }
}
```

5.1 Estructuras de datos enlazadas arborescentes (16/19)

Liberación de un árbol general

```
void libera(ArbolGeneral a)
{
     if (a == NULL) return;
     libera(a->hijoIzquierdo);
     libera(a->hermanoDerecho);
     free(a);
}
```

Búsqueda de un valor en un árbol general

```
int busca(ArbolGeneral a, Elemento elem)
{
     if (a == NULL) return 0;
     if (a->elem == elem ) return 1;
     return (busca(a->hijoIzquierdo,elem))||
            (busca(a->hermanoDerecho,elem));
}
```

5.1 Estructuras de datos enlazadas arborescentes (17/19)

Copia de un árbol general

```
ArbolGeneral copia(ArbolGeneral a){
     if (a == NULL) return NULL;
     ArbolGeneral c = crea_arbol(a-> elem);
     c->hijoIzquierdo = copia(a->hijoIzquierdo);
     c->hermanoDerecho = copia(a->hermanoDerecho);
     return c;
}
```

Comparación de igualdad de dos árboles generales

```
int iguales(ArbolGeneral a, ArbolGeneral b) {
     if (a == NULL && b == NULL) return 1;
     if (a == NULL || b == NULL) return 0;
     return (a->elem == b->elem) &&
            iguales(a->hijoIzquierdo,b->hijoIzquierdo) &&
            iguales(a->hermanoDerecho,b->hermanoDerecho);
}
```

5.1 Estructuras de datos enlazadas arborescentes (18/19)

Altura de un árbol general

```
int altura(ArbolGeneral a)
{
    if (a == NULL) return -1;
    int max = 0;
    ArbolGeneral aux = a->hijoIzquierdo;
    while (aux != NULL)
    {
        int h = altura(aux);
        if (h > max) max = h;
        aux = aux->hermanoDerecho;
    }
    return max + 1;
}
```

5.1 Estructuras de datos enlazadas arborescentes (19/19)

Número de hojas de un árbol general

```
int nHojas(ArbolGeneral a)
{
    if (a == NULL) return 0;
    if (a->hijoIzquierdo == NULL) return 1;
    int cont = 0;
    ArbolGeneral aux = a->hijoIzquierdo;
    while (aux != NULL)
    {
        cont += nHojas(aux);
        aux = aux->hermanoDerecho;
    }
    return cont;
}
```

5.2 Variantes y aplicaciones de Árboles (1/13)

Árboles balanceados

Se dice que un árbol está balanceado cuando su altura es mínima. Esta es una propiedad importante ya que la complejidad de las operaciones sobre árboles dependen de la altura de éstos.

Árboles parcialmente ordenados

Son árboles balanceados donde el contenido de los nodos de los subárboles de un nodo no es mayor que el contenido del nodo. Típicamente se utilizan como un TDA intermedio para el diseño de métodos de ordenación (*heapsort*), y para implementar *colas de prioridad*.

Árboles binarios de búsqueda

El contenido de los nodos del subárbol izquierdo de un nodo son menores o iguales que el contenido del nodo y el contenido de los nodos del subárbol derecho son mayores que el contenido del nodo. Pueden utilizarse para representar conjuntos de elementos en los que existe una relación de orden.

Árboles AVL

Árboles binarios de búsqueda balanceados.

Árboles B

Árboles cuyas hojas tienen todas las misma altura.

5.2 Variantes y aplicaciones de Árboles (2/13)

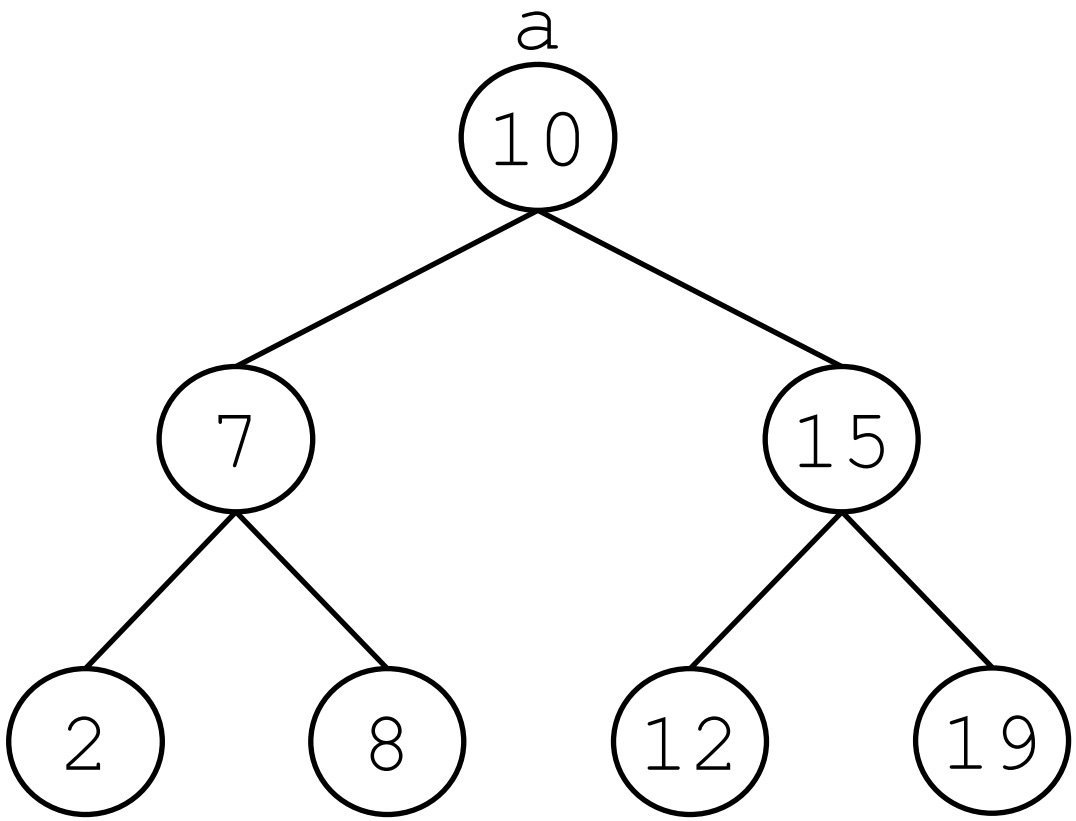
Aplicaciones

Son múltiples las aplicaciones que utilizan árboles en su diseño. Algunas de éstas son:

- Muchas aplicaciones usan los árboles para representar datos jerárquicos. Por ejemplo, los sistemas de directorios y ficheros gestionados por los sistemas operativos pueden representarse con árboles generales.
- Los árboles son usados para realizar búsquedas eficientes en colecciones de datos.
- Una de las aplicaciones más importantes de los árboles en el campo de la *inteligencia artificial* es la construcción de *árboles de decisión* para la toma de decisiones en sistemas complejos (algoritmos *C4.5* y *Random Forest*).

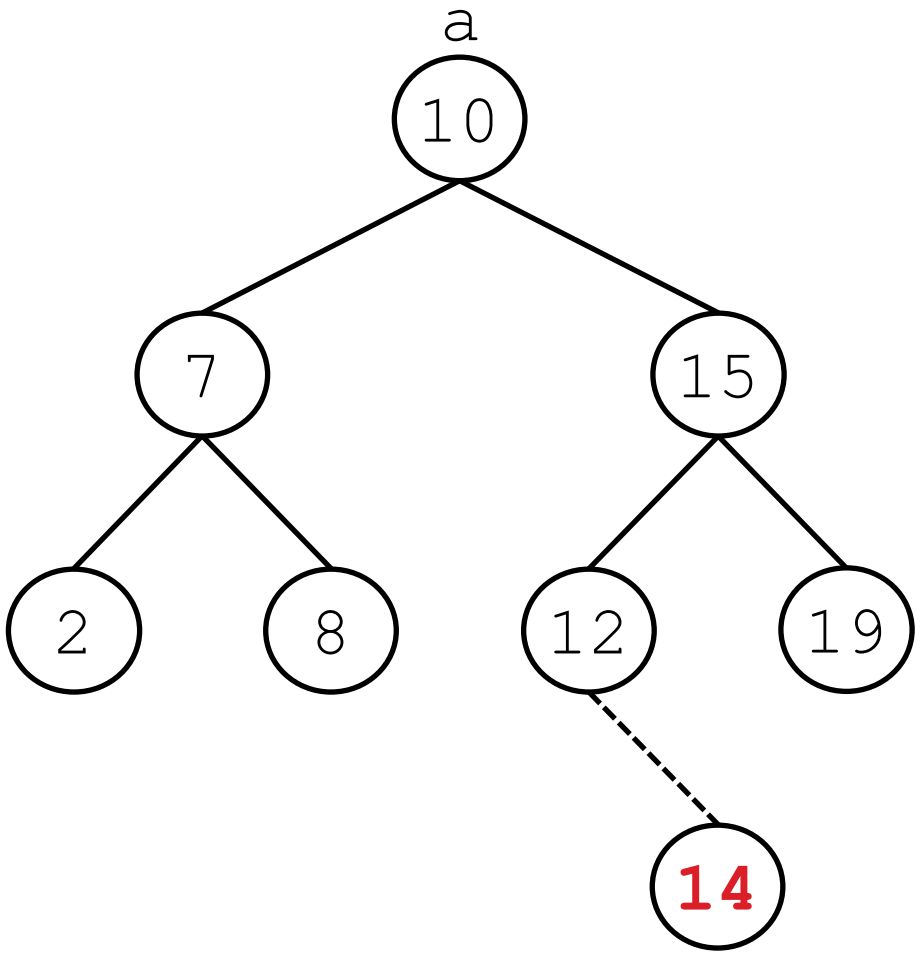
5.2 Variantes y aplicaciones de Árboles (3/13)

Árbol Binario de Búsqueda *a*



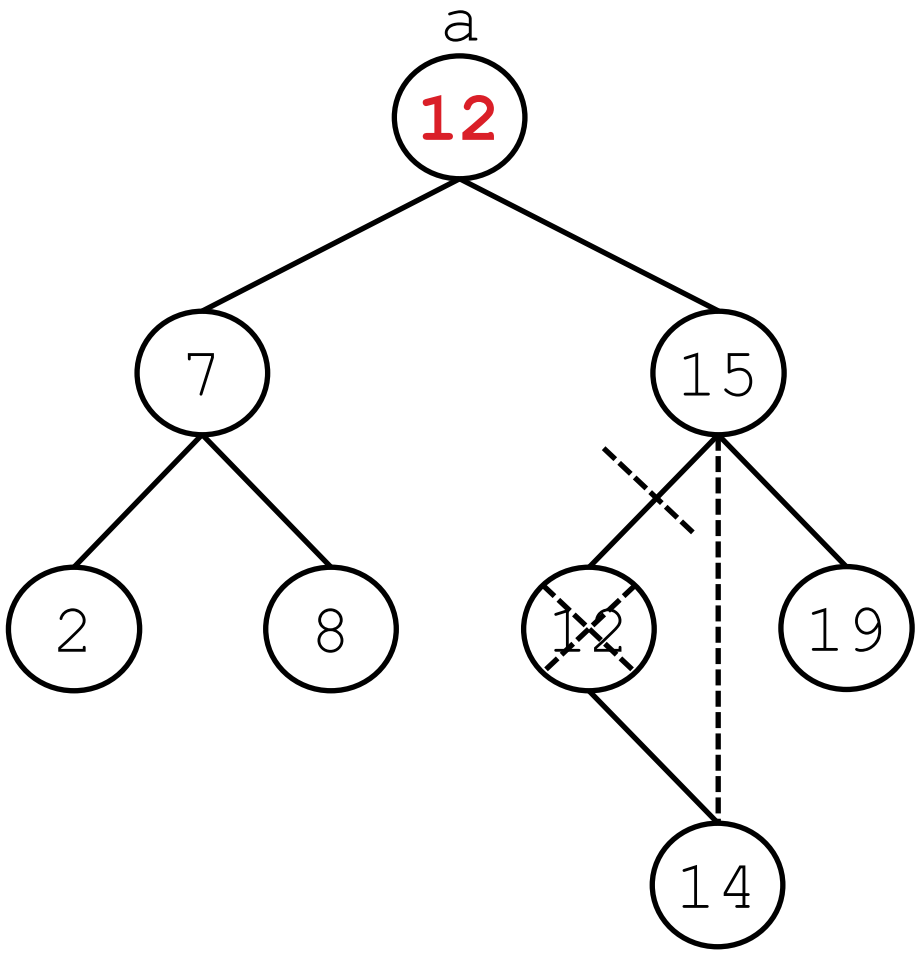
5.2 Variantes y aplicaciones de Árboles (4/13)

Árbol Binario de Búsqueda *a* tras insertar el elemento `14`



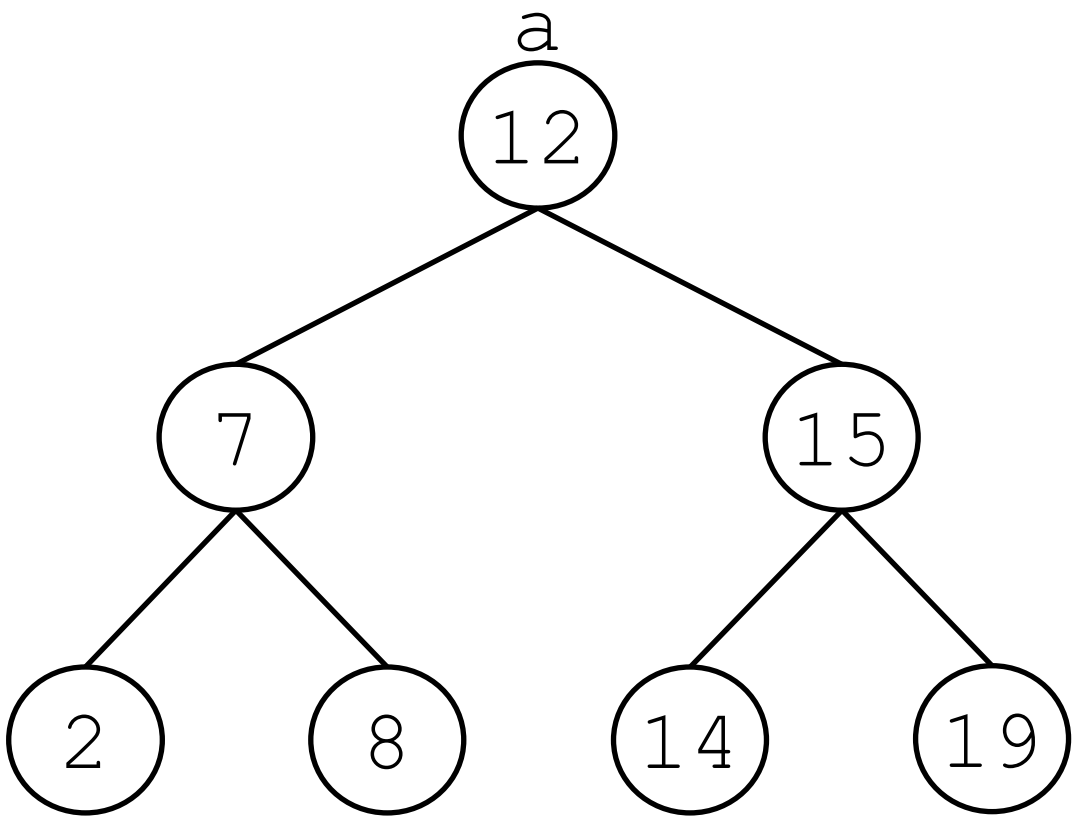
5.2 Variantes y aplicaciones de Árboles (5/13)

Árbol Binario de Búsqueda *a* tras suprimir el elemento `10`



5.2 Variantes y aplicaciones de Árboles (6/13)

Árbol Binario de Búsqueda *a* tras insertar el elemento `14` y suprimir el elemento `10`, disposición final



5.2 Variantes y aplicaciones de Árboles (7/13)

ABB.h

```
#ifndef __ABB_H
#define __ABB_H

typedef struct ABBRep * ABB;
typedef int Elemento;

ABB ABBCrea();
void ABBLibera(ABB a);
int ABBPertenece(ABB a, Elemento e);
void ABBInserta(ABB * a, Elemento e);
void ABBSuprime(ABB * a, Elemento e);
char * ABBToString(ABB a);

#endif
```

5.2 Variantes y aplicaciones de Árboles (8/13)

ABB.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "ABB.h"
#define STRING_LONG 100

struct ABBRep
{
     Elemento elem;
     ABB izq, der;
};

ABB ABBCrea()
{
     return NULL;
}
```

5.2 Variantes y aplicaciones de Árboles (9/13)

ABB.c

```
void ABBLibera(ABB a)
{
     if (a==NULL) return;
     ABBLibera(a->izq);
     ABBLibera(a->der);
     free(a);
}

int ABBPertenece(ABB a, Elemento e)
{
     if (a==NULL) return 0;
     if (e==a->elem) return 1;
     if (e<a->elem) return ABBPertenece(a->izq,e);
     else return ABBPertenece(a->der,e);
}
```

5.2 Variantes y aplicaciones de Árboles (10/13)

ABB.c

```
void ABBInserta(ABB * a, Elemento e)
{
     if ((*a)==NULL)
     {
          (*a) = malloc(sizeof(struct ABBRep));
          (*a)->elem = e;
          (*a)->izq = NULL;
          (*a)->der = NULL;
          return;
     }
     if (e<(*a)->elem)
          ABBInserta(&((*a)->izq),e);
     else if (e>(*a)->elem)
          ABBInserta(&((*a)->der),e);
}
```

5.2 Variantes y aplicaciones de Árboles (11/13)

ABB.c

```
void ABBSuprime(ABB * a, Elemento e)
{
     if ((*a) == NULL) return;
     if (e < (*a)->elem) ABBSuprime(&((*a)->izq),e);
     else if (e > (*a)->elem) ABBSuprime(&((*a)->der),e);
     else if (((*a)->izq == NULL) && ((*a)->der == NULL)){
          free(*a);
          *a = NULL;
     }
     else if ((*a)->izq == NULL){
          ABB der = (*a)->der;
          free(*a);
          (*a) = der;
     }
     else if ((*a)->der == NULL){
          ABB izq = (*a)->izq;
          free(*a);
          (*a) = izq;
     }
     else (*a)->elem = ABBSuprimeMin(&((*a)->der));
}
```

5.2 Variantes y aplicaciones de Árboles (12/13)

ABB.c

```
Elemento ABBSuprimeMin(ABB * a)
{
    if ((*a)->izq==NULL)
    {
        int min=(*a)->elem;
        ABB der = (*a)->der;
        free(*a);
        (*a)=der;
        return min;
    }
    return ABBSuprimeMin(&((*a)->izq));
}
```

5.2 Variantes y aplicaciones de Árboles (13/13)

ABB.c

```
char * ABBToString(char * s, ABB a)
{
    if (a!=NULL)
    {
        ABBToString(s,a->izq);
        char aux[STRING_LONG];
        sprintf(aux," %d",a->elem);
        strcat(s,aux);
        ABBToString(s,a->der);
    }
    return s;
}
```

5.3 Ejercicios resueltos (1/16)

Objetivos

- Recorrer árboles tanto binarios como generales para llevar a cabo cualquier operación sobre ellos.

1) Las siguientes definiciones en C definen un tipo árbol binario que representa una expresión aritmética con operadores binarios (ver figura 1):

```
typedef struct ArbolRep * Arbol;
struct ArbolRep {
    char * str;
    Arbol izq, der;
};
```

5.3 Ejercicios resueltos (2/16)

Implementar en *C* las siguientes funciones:

1.1) Imprime en pantalla la expresión aritmética representada por el árbol binario `a`. La expresión y las sub-expresiones se imprimen entre paréntesis. Los operandos no deben imprimirse entre paréntesis.

```
void imprime(Arbol a){...}
```

1.2) Devuelve el número de operadores de la expresión aritmética representada por el árbol binario `a`.

```
int operadores(Arbol a){...}
```

1.3) Devuelve el valor de la expresión aritmética representada por el árbol binario `a`. El árbol binario a debe ser distinto de `NULL` como precondición.

```
double ExpresionValor(Arbol a) {...}
```

5.3 Ejercicios resueltos (3/16)

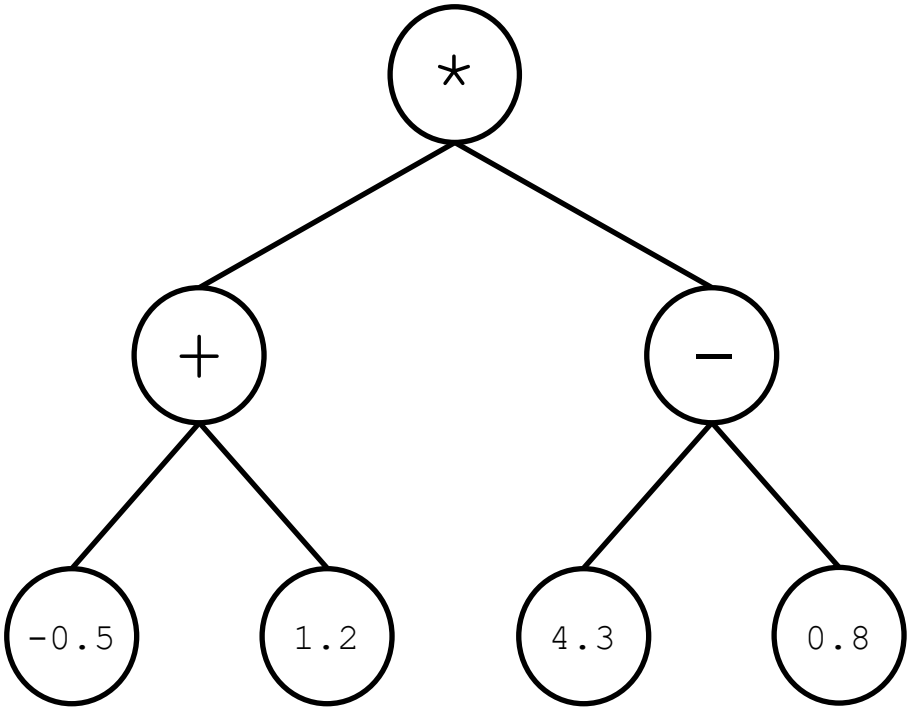


Figura 1: Ejemplo de un árbol binario para representar una expresión aritmética con operadores binarios. La función `imprime` (ejercicio 1.1) imprime en pantalla `((-0.5+1.2)*(4.3-0.8))`. La función `operadores` (ejercicio 1.2) devuelve `3`. La función `ExpresionValor` devuelve el valor `2.45`.

5.3 Ejercicios resueltos (4/16)

```
void imprime(Arbol a)
{
     if (a != NULL)
     {
          if (a->izq == NULL)
                printf("%s",a->str);
          else
          {
               printf("(");
               imprime(a->izq);
               printf("%s",a->str);
               imprime(a->der);
               printf(")");
          }
     }
}
```

5.3 Ejercicios resueltos (5/16)

```
int operadores(Arbol a)
{
     if (a == NULL)
          return 0;
     if (a->izq==NULL)
          return 0;
     return 1+operadores(a->izq)+operadores(a->der);
}
```

5.3 Ejercicios resueltos (6/16)

```
double ExpresionValor(Arbol a)
{
     double v = 0;
     if (a->izq==NULL)
          v = atof(a->str);
     else switch(a->str[0]){
          case '+': v = ExpresionValor(a->izq)+ExpresionValor(a->der);
                    break;
          case '-': v = ExpresionValor(a->izq)-ExpresionValor(a->der);
                    break;
          case '*': v = ExpresionValor(a->izq)*ExpresionValor(a->der);
                    break;
          case '/': v = ExpresionValor(a->izq)/ExpresionValor(a->der);
     }
     return v;
}
```

5.3 Ejercicios resueltos (7/16)

Considérense las siguientes definiciones en *C*:

```
typedef struct ArbolRep * Arbol;
struct ArbolRep {
    int elemento;
    Arbol izquierdo, derecho;
};
```

NOTA: El árbol vacío se representa con `NULL`.

2) Implementar en *C* la siguiente función `minimo` que devuelve el mínimo elemento del árbol `a`, o el valor `INT_MAX` (máximo valor representable por un tipo `int`) si el árbol a es vacío:

```
int minimo(Arbol a){...}
```

suponiendo los siguientes escenarios:

1. El árbol `a` es un árbol binario (representación hijo izquierdo - hijo derecho).
2. El árbol `a` es un árbol general (representación hijo izquierdo - hermano derecho).
3. El árbol `a` es un árbol binario de búsqueda.

5.3 Ejercicios resueltos (8/16)

```
int minimo(Arbol a)
{
     if (a==NULL)
          return INT_MAX;
     int v = a->elemento;
     int v1 = minimo(a->izquierdo);
     int v2 = minimo(a->derecho);
     if (v1<v)
          v = v1;
     if (v2<v)
          v = v2;
     return v;
}
```

5.3 Ejercicios resueltos (9/16)

```
int minimo(Arbol a)
// puede hacerse también igual que el anterior
{
     if (a==NULL) return INT_MAX;
     int v = a->elemento;
     for(Arbol aux = a->izquierdo; aux!=NULL; aux=aux->derecho){
          int v1 = minimo(aux);
          if (v1<v) v = v1;
     }
     return v;
}
int minimo(Arbol a)
{
     if (a==NULL) return INT_MAX;
     while(a->izquierdo!=NULL) a = a->izquierdo;
     return a->elemento;
}
```

5.3 Ejercicios resueltos (10/16)

3) Implementar en *C* la siguiente función `suma` que devuelve la suma de todos los elementos del árbol binario `a`:

```
int suma(Arbol a)
{
     if (a==NULL)
          return 0;
     return a->elemento+suma(a->izquierdo)+suma(a->derecho);
}
```

5.3 Ejercicios resueltos (11/16)

4) Implementar en *C* la siguiente función `imprime_hojas` que imprime en pantalla las hojas del árbol binario `a`:

```
void imprime_hojas(struct NodoArbol * a)
{
     if (a!=NULL)
     {
          if ((a->izquierdo==NULL)&&(a->derecho==NULL))
               printf(" %d ",a->elemento);
          else
          {
               imprime_hojas(a->izquierdo);
               imprime_hojas(a->derecho);
          }
     }
}
```

5.3 Ejercicios resueltos (12/16)

5) Implementar en *C* la siguiente función `mayores` que devuelve el número de elementos en el árbol binario `a` que sean mayores que `n`.

```
int mayores(Arbol a, int n)
{
     if (a==NULL)
          return 0;
     int valor = mayores(a->izquierdo,n)+mayores(a->derecho,n);
     if (a->elemento>n)
          valor++;
     return valor;
}
```

5.3 Ejercicios resueltos (13/16)

6) Implementar en *C* la siguiente función `arbolTolista` que inserta los elementos del árbol binario `a` en la estructura enlazada `l`.

```
struct Nodo
{
      int elem;
      struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
void inserta_primero(NodoPtr l, int e){
      Lista aux = malloc(sizeof(struct ListaRep));
      aux->elem = e;
      aux->sig = l->sig;
      l->sig = aux;
}
void arbolTolista(Arbol a, NodoPtr l){ // preorden
      if (a!=NULL)
      {
            arbolTolista(a->derecho,l);
            arbolTolista(a->izquierdo,l);
            inserta_primero(l,a->elemento);
      }
}
```

5.3 Ejercicios resueltos (14/16)

7) Implementar en *C* la siguiente función `modifica` que cambia todos los elementos iguales a `e1` por `e2` en el árbol binario `a`.

```
void modifica(Arbol a, int e1, int e2)
{
     if (a!=NULL)
     {
          if (a->elemento==e1)
                a->elemento=e2;
          modifica(a->izquierdo,e1,e2);
          modifica(a->derecho,e1,e2);
     }
}
```

5.3 Ejercicios resueltos (15/16)

8) Implementar en *C* la siguiente función `nodos` que devuelve el número de elementos en el árbol binario `a` .

```
int nodos(Arbol a)
{
     if (a==NULL)
          return 0;
     return nodos(a->izquierdo)+nodos(a->derecho)+1;
}
```

5.3 Ejercicios resueltos (16/16)

9) Implementar en C la función `insertaEnhojas` que inserta, en cada nodo hoja del árbol binario `a`, dos nuevos nodos como hijo izquierdo e hijo derecho cuyos contenidos son el contenido del nodo hoja más 1 y el contenido del nodo hoja más 2 respectivamente.

```
void insertaEnHojas(Arbol a)
{
      if (a!=NULL){
            if (a->izquierdo==NULL && a->derecho==NULL){
                  a->izquierdo = malloc(sizeof(struct ArbolRep));
                  a->izquierdo->elemento = a->elemento+1;
                  a->izquierdo->izquierdo = NULL;
                  a->izquierdo->derecho = NULL;
                  a->derecho = malloc(sizeof(struct ArbolRep));
                  a->derecho->elemento = a->elemento+2;
                  a->derecho->izquierdo = NULL;
                  a->derecho->derecho = NULL;
            }
            else{
                  insertaEnHojas(a->izquierdo);
                  insertaEnHojas(a->derecho);
            }
      }
}
```

5.4 Ejercicios propuestos (1/4)

Dadas la siguientes definiciones de tipos:

```
struct Nodo
{
     Elemento elem;
     struct Nodo * hijoIzquierdo, * hermanoDerecho;
};
typedef struct Nodo * ArbolGeneral;
```

1) Implementar en *C* la función `recorrido_anchura`, que imprime en pantalla los nodos visitados de un árbol general al recorrerlo en anchura, para lo cual utiliza una cola interna.

   ```
   #include "Cola.h"
   void recorrido_anchura(ArbolGeneral a) { ... }
   ```

2) Implementar en *C* las funciones `suma`, `imprime_hojas`, `mayores`, `arbolTolista`, `modifica` y `nodos` definidas anteriormente para árboles binarios.

5.4 Ejercicios propuestos (2/4)

3) Implementar en C la función media, realizando para ello los siguientes pasos:

1. Implementar la función `suma` la cual devuelve la suma de los nodos del árbol `a`.

   ```
   int suma(ArbolGeneral a){...}
   ```

2. Implementar la función `numero` la cual devuelve el número de nodos del árbol `a`.

   ```
   int numero(ArbolGeneral a){...}
   ```

3. Implementar la función `media` usando las funciones `suma` y `numero` previamente definidas.

   ```
   double media(ArbolGeneral a){...}
   ```

Se asume que el árbol vacio se representa con `a=NULL`.

5.4 Ejercicios propuestos (3/4)

4) Implementar en *C* la siguiente función ordena la cual, dada una estructura enlazada `l` con nodo de encabezamiento de números enteros, devuelve una nueva estructura enlazada con nodo de encabezamiento de números enteros resultado de la ordenación de la estructura `l`, usando un árbol binario de búsqueda interno:

```
#include "ABB.h"

struct Nodo
{
    int elem;
    struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;

NodoPtr ordena(NodoPtr l){ ... }
```

5.4 Ejercicios propuestos (4/4)

Para ello realizar los siguientes pasos:

1. Modificar el árbol binario de búsqueda de la sección 5.2 para que permita representar elementos repetidos
2. Crear un árbol binario de búsqueda vacío `a`.
3. Insertar los elementos de la estructura enlazada `l` en un árbol binario de búsqueda `a`.
4. Recorrer el árbol binario de búsqueda `a` en orden interno para construir una estructura lineal enlazada ordenada con la siguiente función `inorden`:

   ```
   NodoPtr inorden(ABB a){...}
   ```

5. Devolver la nueva estructura enlazada ordenada.