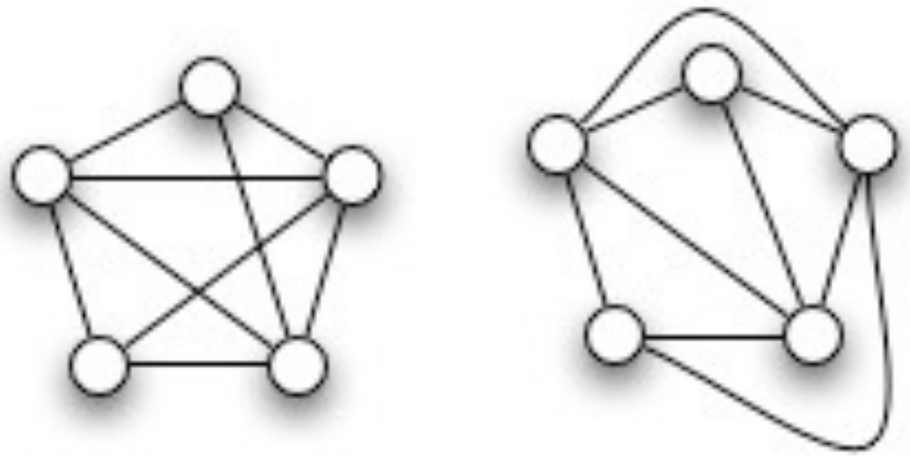
Árboles

Grafos

- Un ***grafo*** es un conjunto de nodos atados por un conjunto de ejes que conectan pares de nodos distintos (con un eje conectando un par de nodos.)



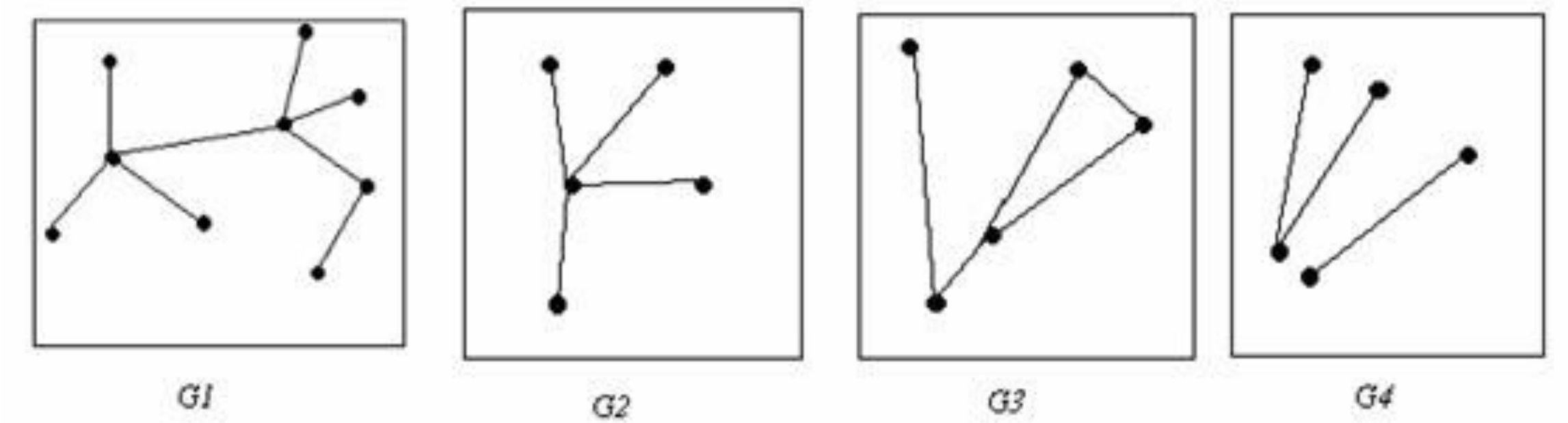
Árboles

- Un ***árbol*** es una colección no-vacía de vértices y ejes.
- Un ***vértice*** o ***nodo*** es un objeto simple que puede tener un ***nombre*** y información asociada.
- Un ***eje*** es una ***conexión*** entre dos vértices o nodos.
- Un ***camino*** en un árbol es una lista de vértices **distintos** sucesivos conectados por ejes.
- En un árbol hay ***solamente un camino*** conectando dos nodos.
- Si hay ***más de un camino*** conectando dos nodos, la estructura se llama ***grafo.***
- Un conjunto ***disjunto*** de árboles se llama ***bosque.***

Árboles como grafos

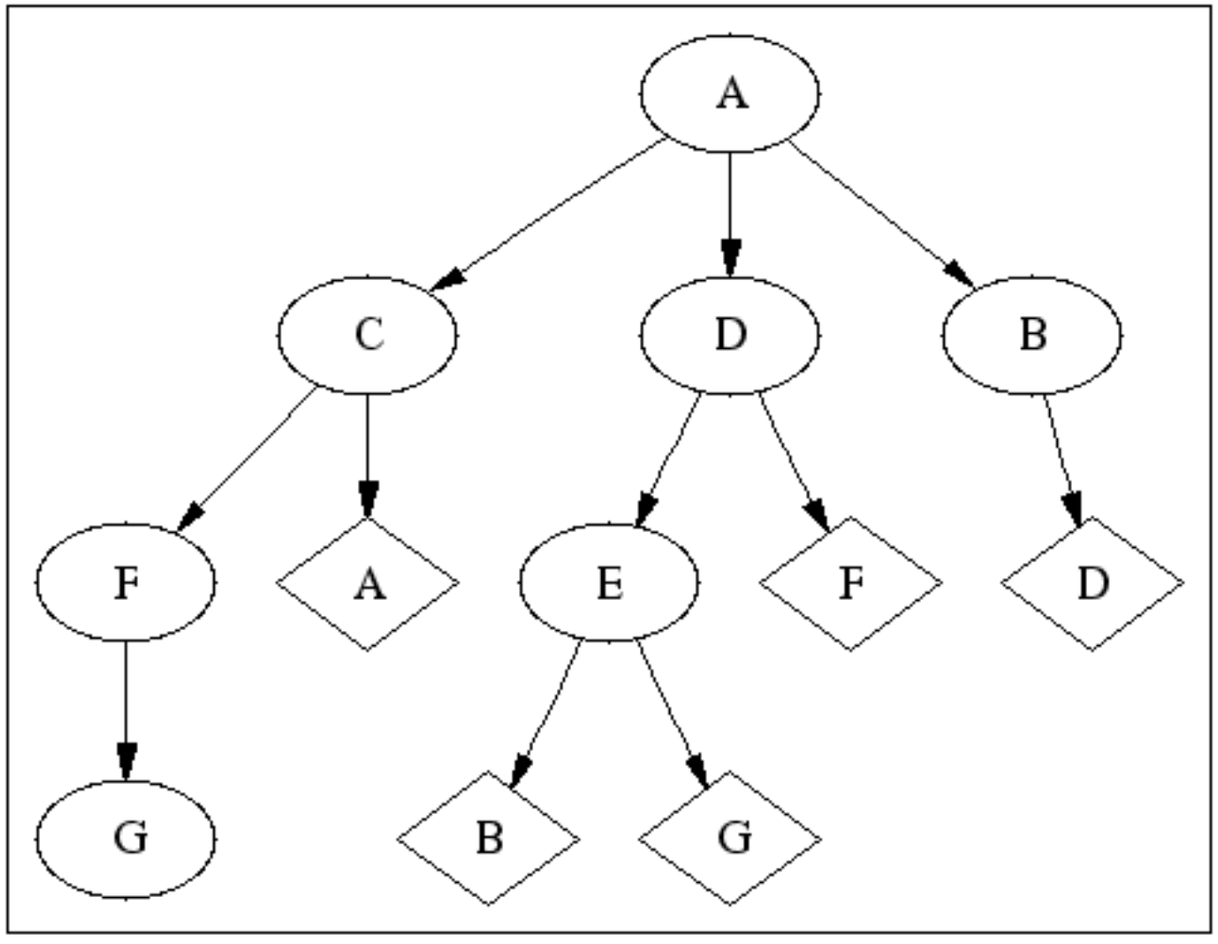
- Si empezamos en un nodo cualquiera, seguimos un eje hacia otro nodo, luego otro eje, etc, se dice que tenemos un ***camino simple.***
- Un camino en que el nodo inicial y el nodo final son el mismo, se llama ***ciclo.***
- Todo ***árbol*** es un ***grafo***, pero ¿cuáles grafos son árboles?
- Un grafo es un árbol ssi:
  - ***G*** tiene ***N-1*** ejes y no tiene ciclos.
  - ***G*** tiene ***N-1*** ejes y está conectado.
  - Exactamente un camino simple conecta cada par de vértices en ***G***.
  - ***G*** está conectado, pero no permanece conectado al borrar un eje.

ejemplos

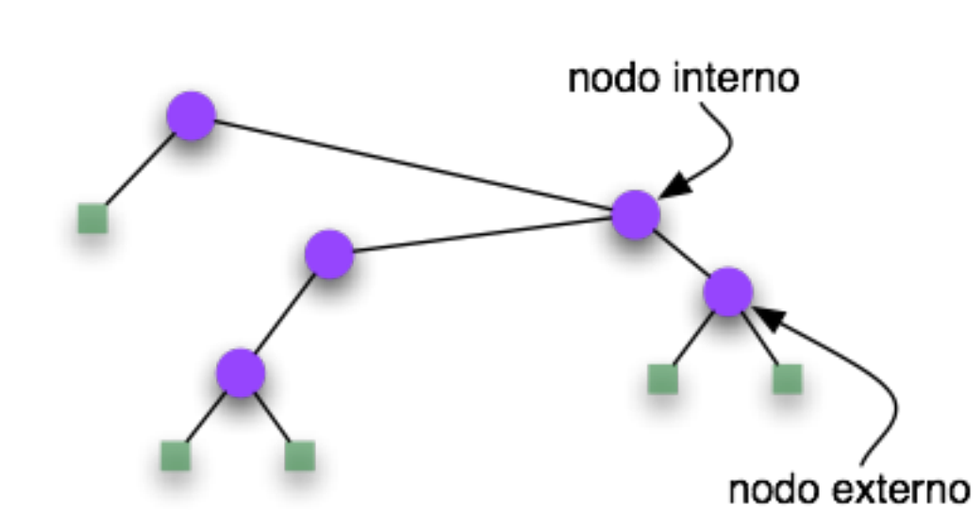


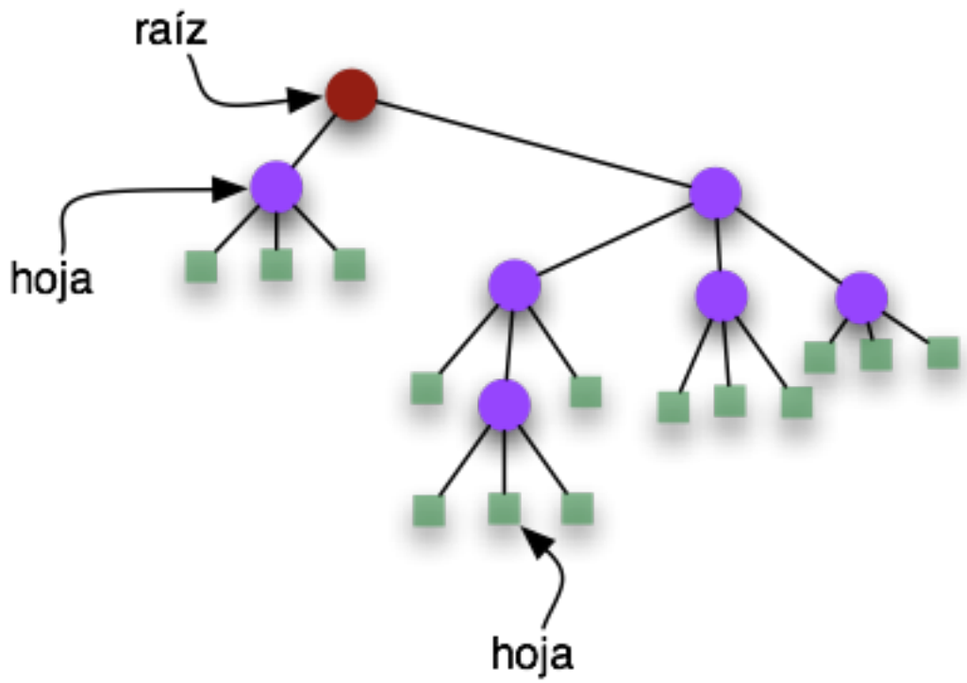
Árboles enraizados

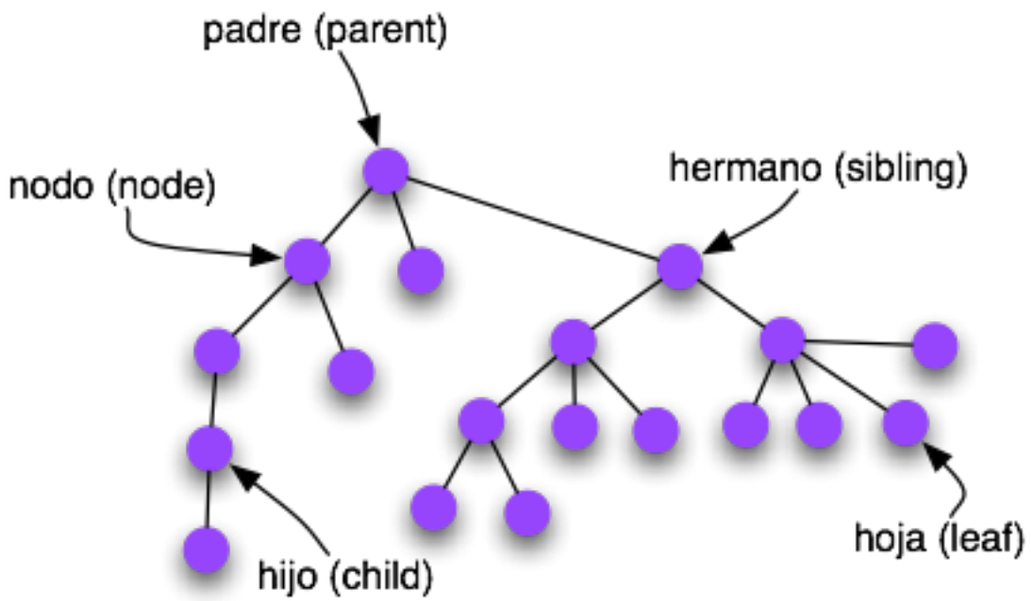
- Es un nodo (llamado raíz) conectado a un conjunto de árboles enraizados
- El tipo más general de árboles es aquel en que no se distingue nodo raíz.

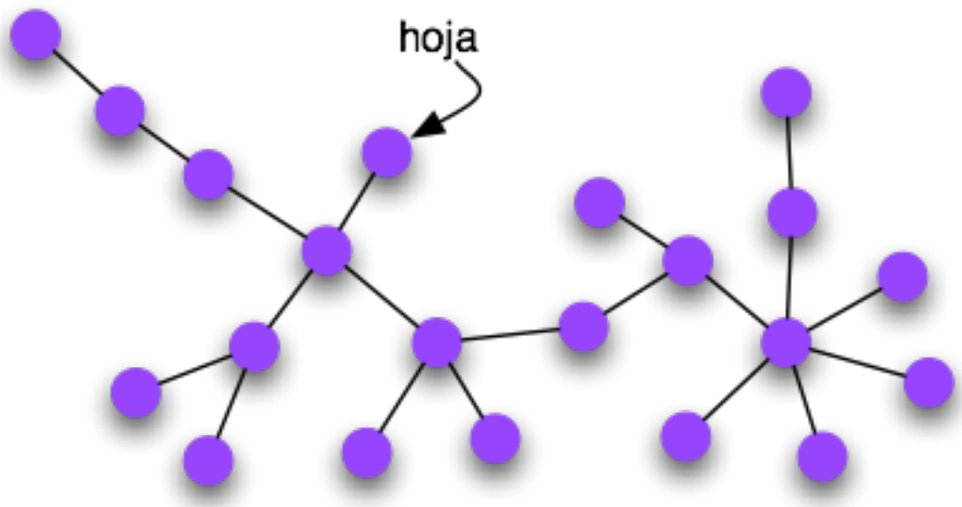


Árboles


nodo interno
nodo externo


raíz
hoja
hoja


padre (parent)
nodo (node)
hermano (sibling)
hijo (child)
hoja (leaf)

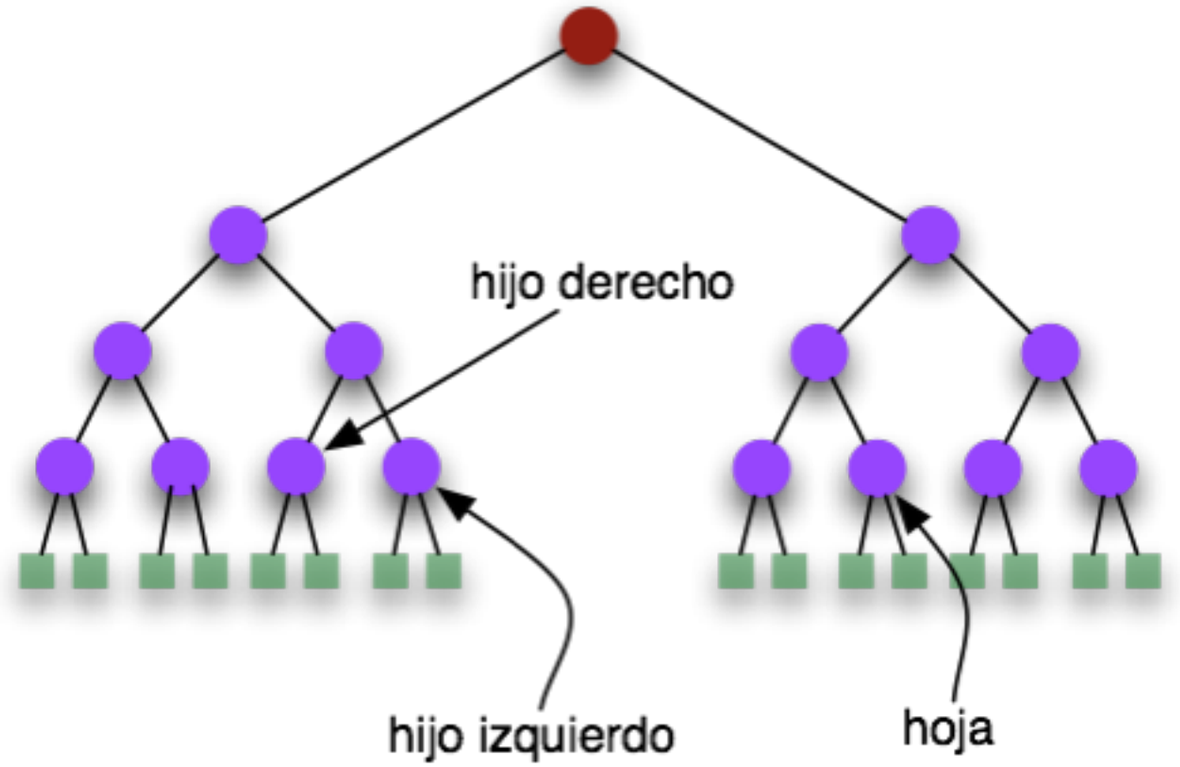

hoja

Árboles ordenados

- Un ***árbol ordenado*** es un árbol con raíz en el cual el orden de todos los nodos hijos es especificado.

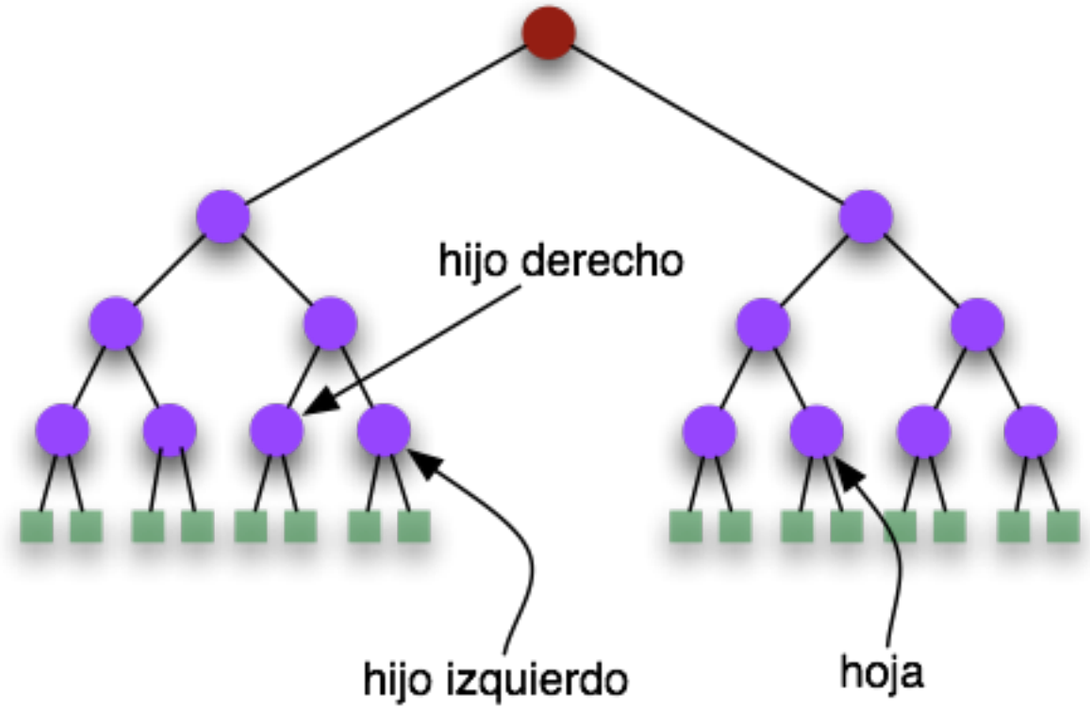
- Si cada nodo ***debe*** tener un número específico de hijos, se llama ***árbol M-nario.*** El más común entre estos es el ***árbol binario***.

- Un ***árbol binario*** es un árbol ordenado con ***nodos internos*** con ***dos*** hijos.



Árboles binarios

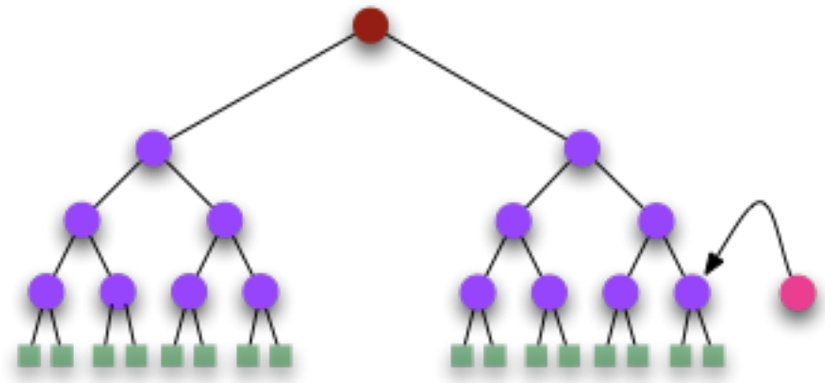
- Un ***árbol binario*** es un nodo externo, o uno interno conectado a un par de árboles binarios, llamados el ***sub-árbol derecho*** y el ***sub-árbol izquierdo***.
- Representación concreta para implementar árboles binarios: ***estructura con un link hacia el nodo derecho y un link hacia el nodo izquierdo.***
- Similar a las listas ligadas.
  - dos links en lugar de uno.
  - link a NULL en nodos externos.

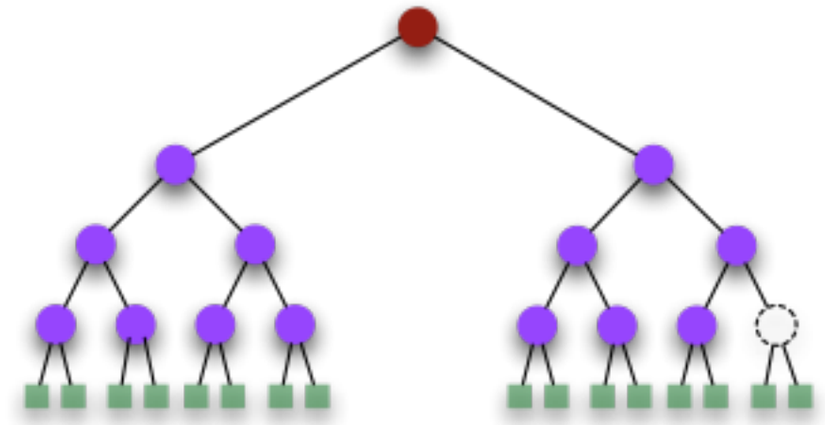


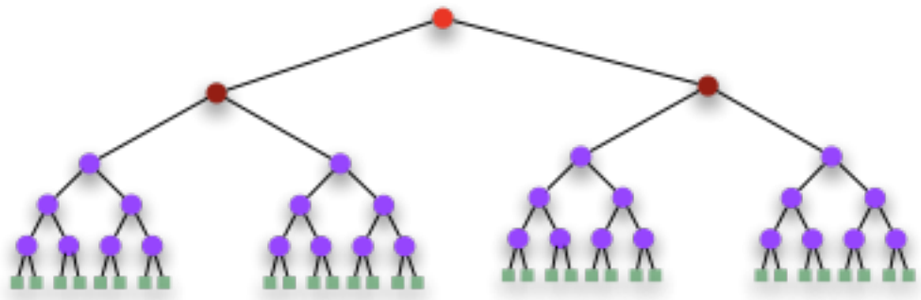

```
struct node
{
     Item item;
     node *l;
     node *r;
};

typedef node *link;
```

Árboles binarios

- Estructura parecida a una lista doblemente ligada.
- Operaciones simples con un árbol binario:
  - insertar un nodo al final del árbol
  - eliminar una hoja
  - combinar dos árboles creando una nueva raíz

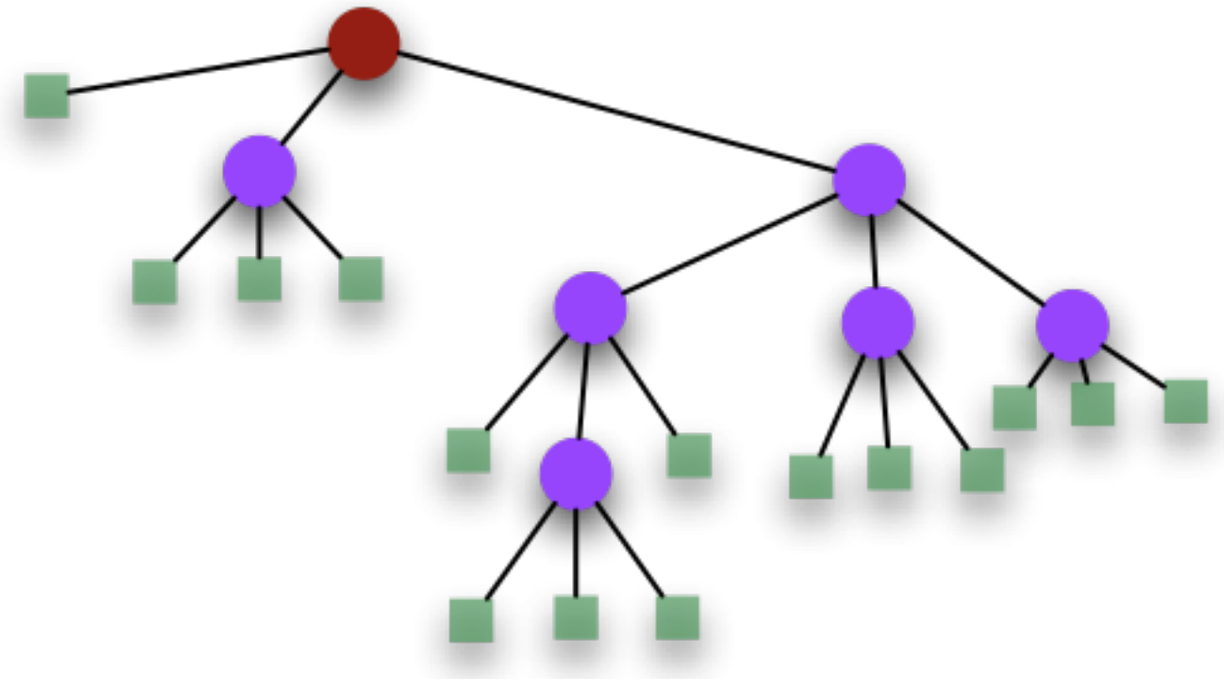






Árboles M-arios

- Un ***árbol M-ario*** es un nodo externo o interno conectado a una secuencia de árboles que también son M-arios.
- Estructura con ***M*** número de links a sus hijos y uno a su padre.
- Se puede usar un arreglo para almacenar los links ya que el valor de ***M*** es fijo.

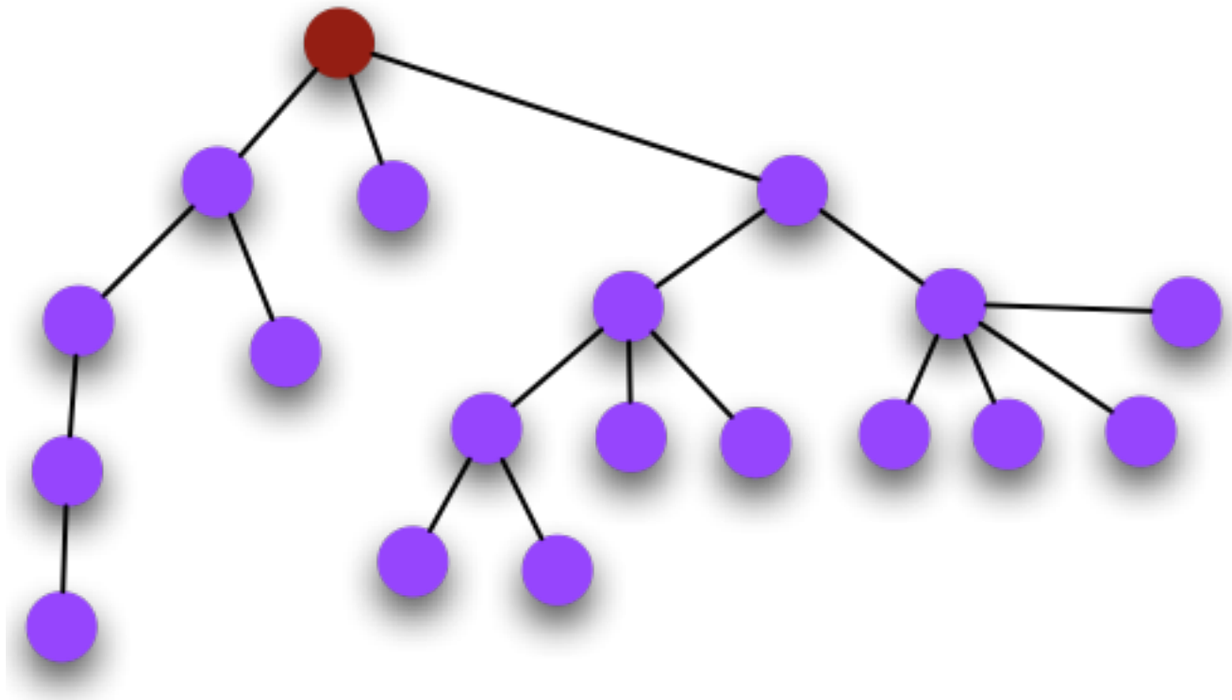


Otra nomenclatura

- Un **árbol binario lleno** es un árbol en el que cada nodo tiene cero o dos hijos.

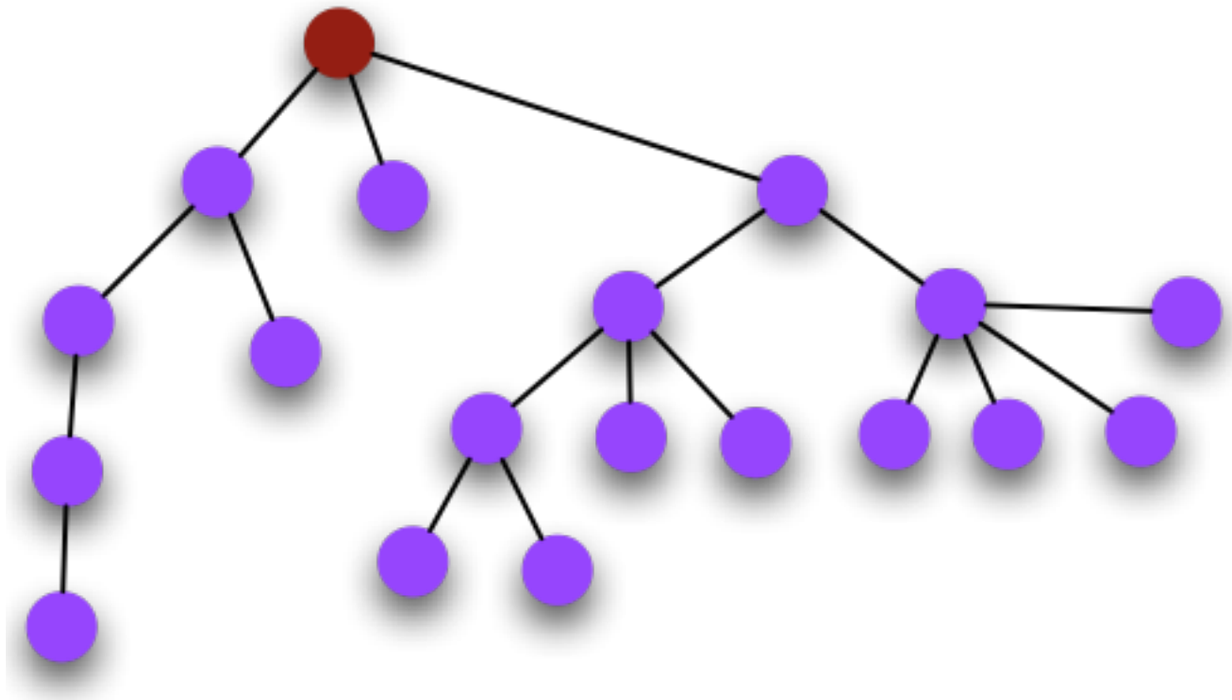
- Un **árbol binario perfecto** es un árbol binario lleno en el que todas las **hojas** (vértices con cero hijos) están a la misma profundidad (distancia desde la **raíz**, también llamada **altura**)

Árboles ordenados (árboles)

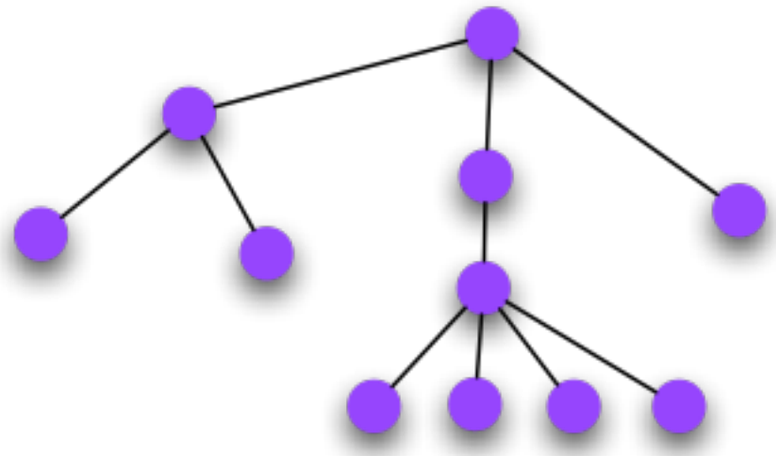


Árboles ordenados (árboles)

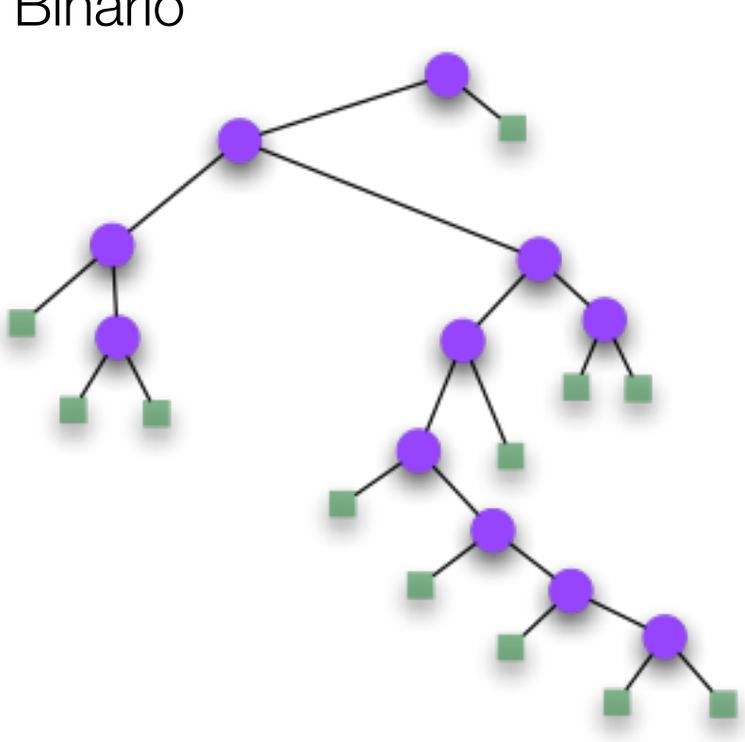
- Es un ***nodo*** (llamado ***raíz***) conectado a una secuencia de árboles disjuntos (bósque).
- La diferencia con un árbol M-ario es que no tiene que tener un número definido de hijos.
- Se conocen también como árboles generalizados.
- Para definirlos se usan listas de links: una lista que guarda a sus hijos, otra que guarda a sus hermanos.



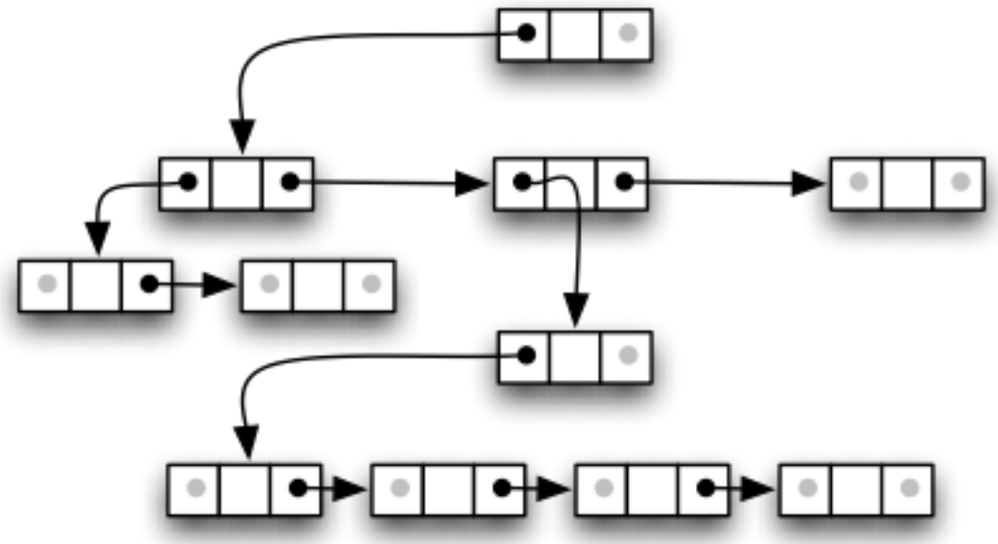
Arbol Ordenado

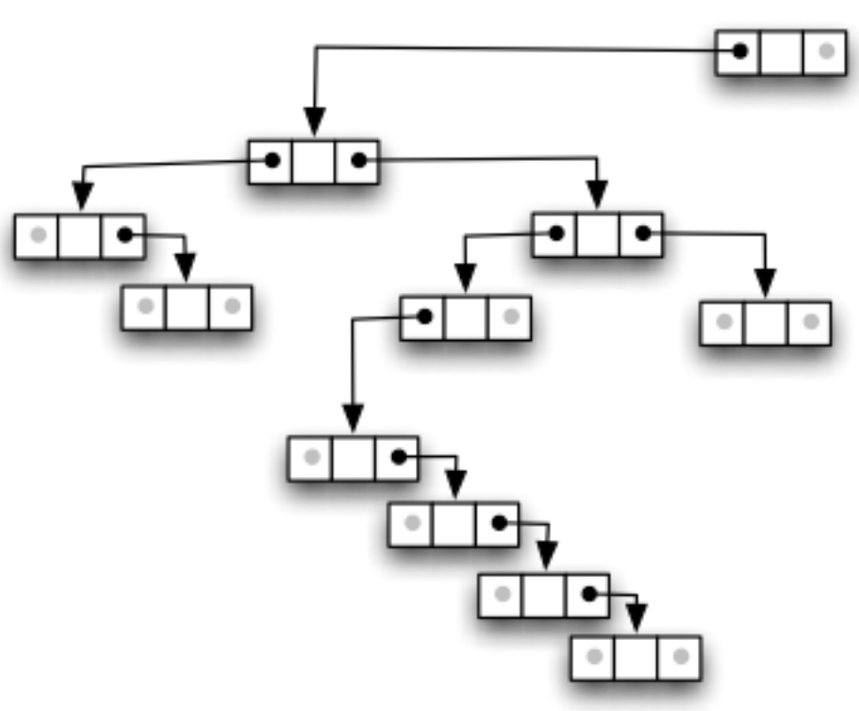


Arbol Binario



link izquierdo apunta al hijo de la izquierda.
link derecho apunta a su hermano derecho.





Propiedades de los árboles binarios llenos (1)

Propiedad 1:

*Un árbol binario con **N** nodos internos, tiene **N+1** nodos externos.*

Prueba por inducción.

para ***N=0*** : un árbol binario sin nodos internos tiene un nodo externo.

para ***N>0*** : un árbol binario con ***N*** nodos internos tiene ***k*** nodos internos en el sub-árbol izquierdo y ***N-1-k*** nodos internos en el sub-arbol derecho.

Entonces tiene ***k+1*** nodos externos a la izquierda y ***N-k*** nodos externos a la derecha, por un total de ***N+1*** nodos externos. ■

Propiedades de los árboles binarios llenos (2)

Propiedad 2:

*Un árbol binario con **N** nodos internos, tiene **2N** ligas o ejes: **N-1** links a nodos internos y **N+1** a nodos externos.*

En un árbol enraizado, todos los nodos, excepto la raíz, tienen un padre único y cada eje les conecta a su padre, entonces hay ***N-1*** links conectando a padres de nodos internos.

Cada uno de los ***N+1*** nodos externos (de la propiedad anterior) tiene un link, a su padre.

Por lo tanto hay **2N** ligas. ■

Árboles binarios

DEFINICIONES:

*El **nivel** de un nodo en un árbol es uno arriba del nivel de su padre (con la raíz en el nivel 0 por convención).*

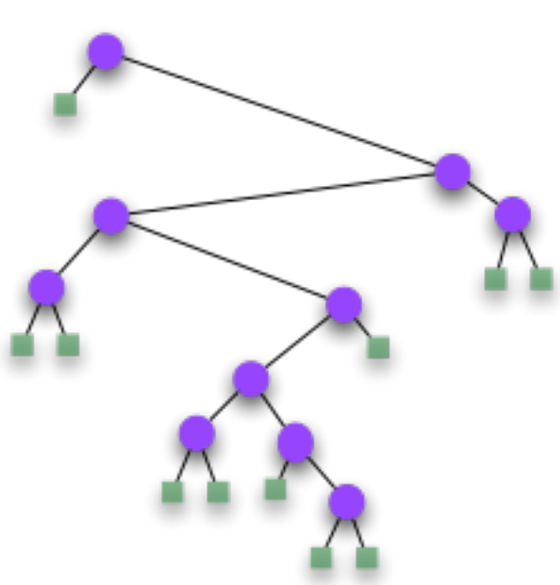
*La **altura** de un árbol es el máximo de niveles de los nodos del árbol.*

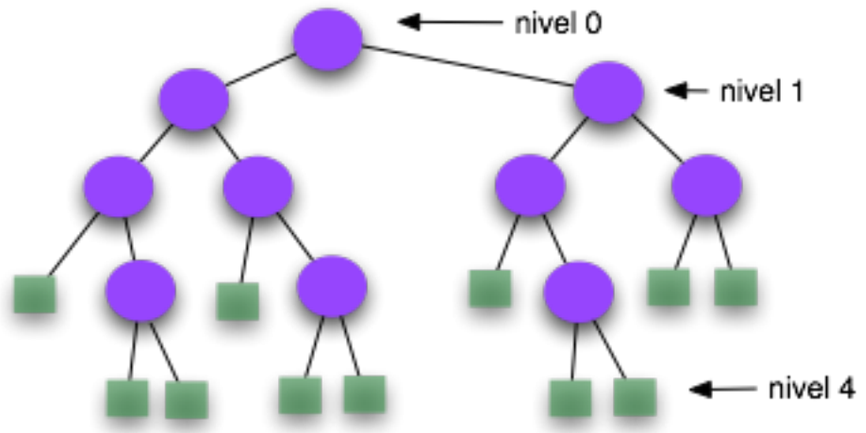
*La **longitud del camino** es la suma de los niveles de todos los nodos del árbol.*

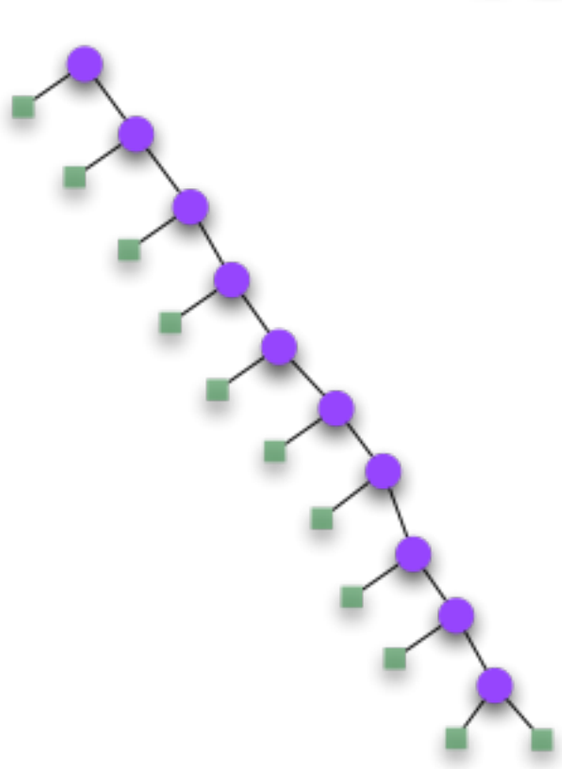
*La **longitud de camino interno** es la suma de todos los niveles de todos los nodos internos del árbol.*

*La **longitud del camino externo** es la suma de los niveles de todos los nodos externos.*

Longitud de camino como descriptor de estructura de los árboles binarios







Propiedades de los árboles binarios llenos (3)

Propiedad 3:

> *La longitud del camino externo de un árbol binario con **N** nodos internos es **2N** mayor que la longitud del camino interno.*

Prueba:

Empezar con el árbol que consta de ***un*** nodo externo (cero nodos internos), repetir el siguiente proceso **N** veces: Tomar un nodo externo y reemplazarlo por un nuevo nodo interno con dos nodos externos como hijos.

Si tomamos un nodo externo al nivel ***k***, el largo del camino interno aumenta por ***k*** pero el camino externo aumenta por ***k+2*** (es decir aumenta *2(k+1)-k*).

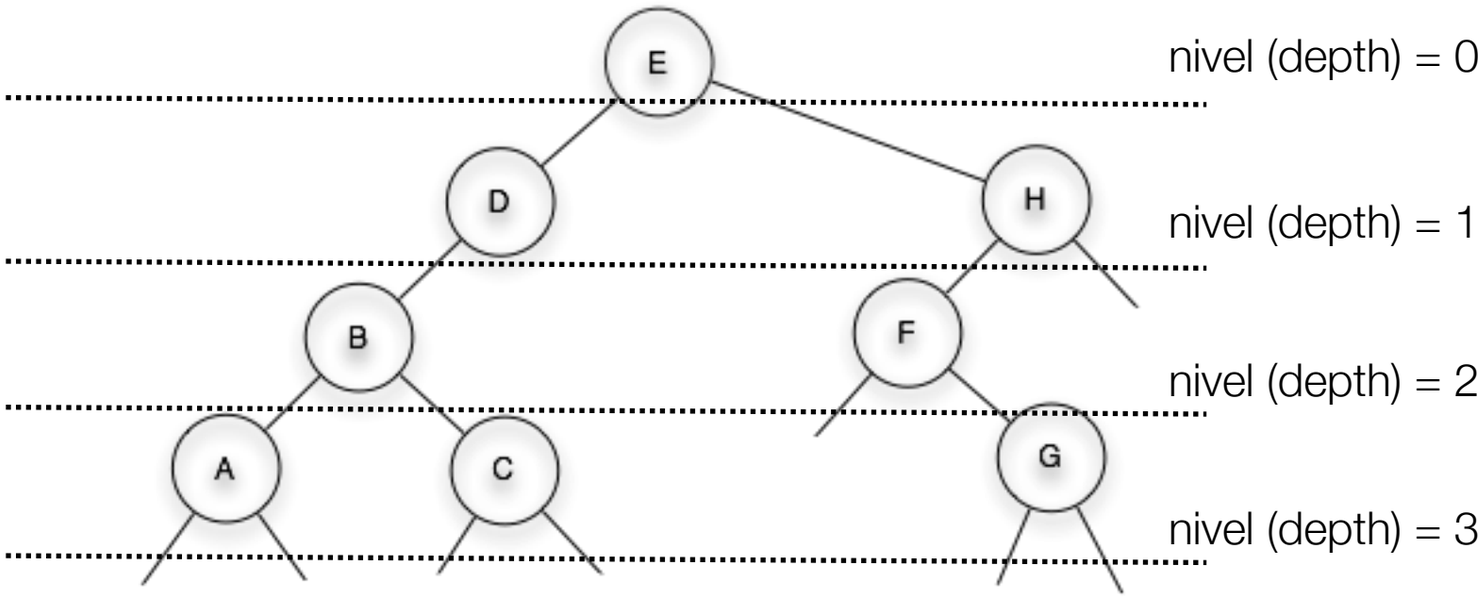
Como se hace ***N*** veces, el externo aumenta ***2N***.

■

Árboles binarios

- Aparecen en muchas aplicaciones dentro de las ciencias de la computación.
- Su desempeño es mejor cuando están balanceados.
- Ej. búsqueda binaria.
- Las propiedades nos dan la información necesaria para determinar algoritmos eficientes.

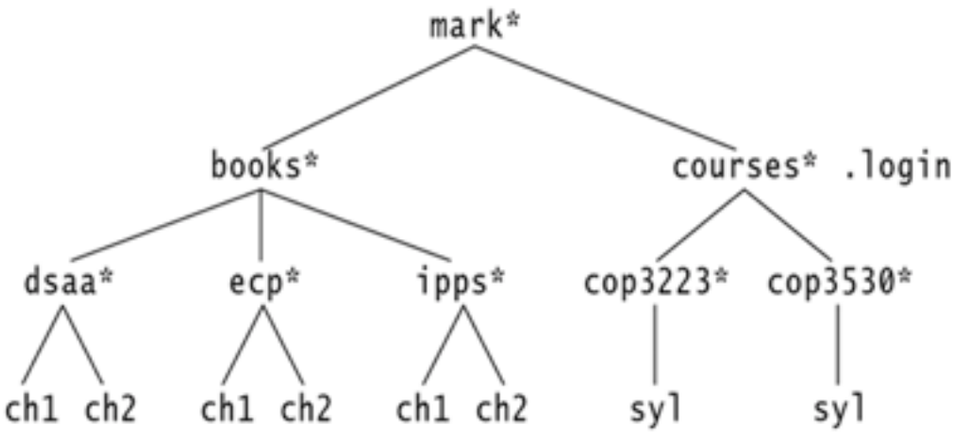
Jerarquía de un árbol



Ejemplos de aplicación

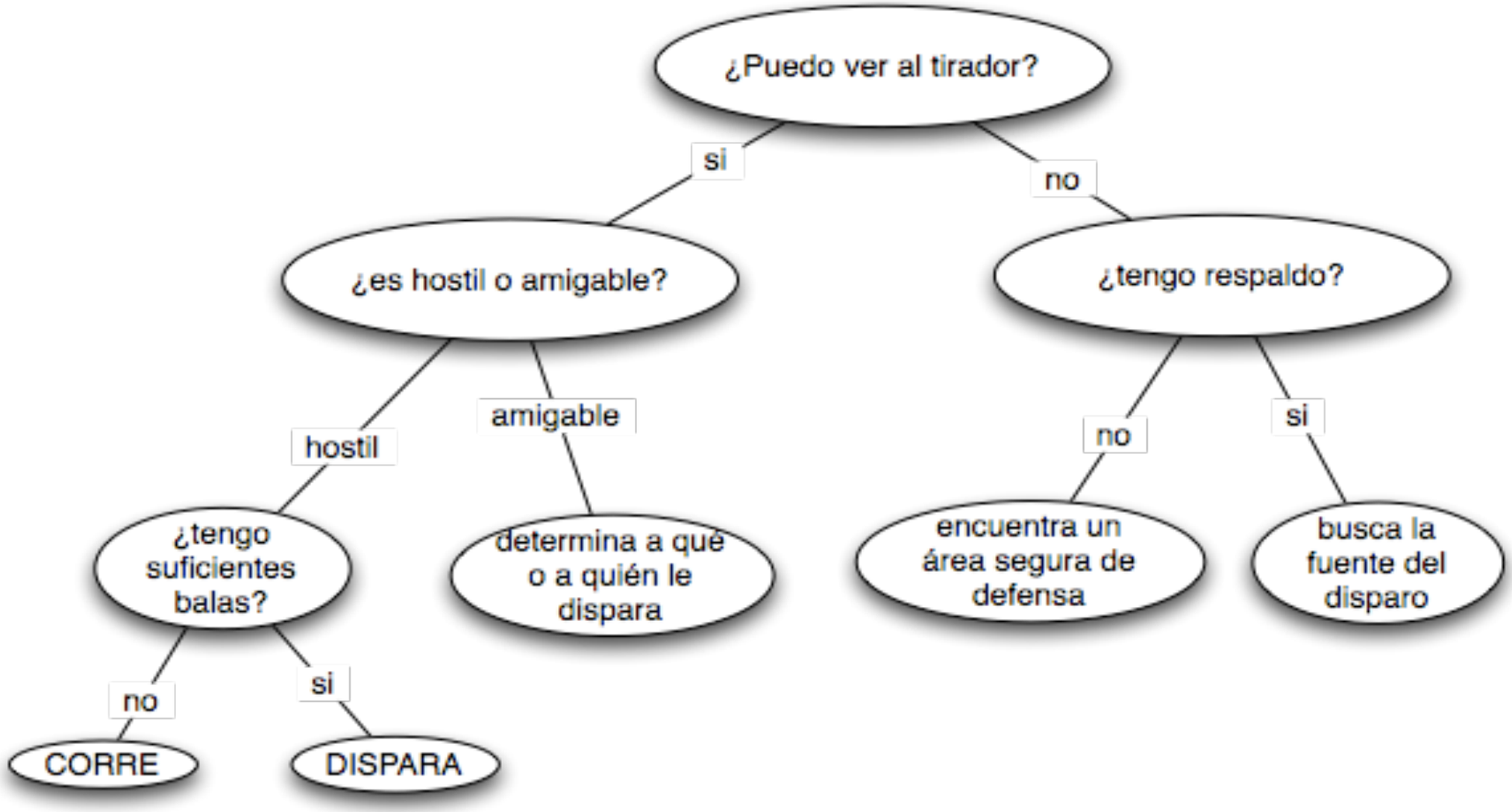
- Sistemas de archivos

```
1    void listAll( int depth = 0 ) // depth is initially 0
2    {
3        printName( depth );       // Print the name of the object
4        if( isDirectory( ) )
5            for each file c in this directory (for each child)
6                c.listAll( depth + 1 );
7    }
```




```
mark
        books
                dsaa
                        ch1
                        ch2
                ecp
                        ch1
                        ch2
                ipps
                        ch1
                        ch2
        courses
                cop3223
                        syl
                cop3530
                        syl
        .login
```

Ejemplos de aplicación



árboles de decisión S/N

Ejemplo de interfase ADT de un árbol

```
class Tree
{
// ---------------------------------------------------------------
// Name:              Tree::Tree
// Description:       Default constructor. Creates a tree node.
// Arguments:         data to intialize node with
// Return value:      none.
// ---------------------------------------------------------------

    Tree( Item data );

// ---------------------------------------------------------------
// Name:              Tree::~Tree
// Description:       Destructor. Deletes all child nodes
// Arguments:         none.
// Return value:      none.
// ---------------------------------------------------------------

    ~Tree();
```

Ejemplo de interfase ADT de un árbol

```
// ---------------------------------------------------------------
// Name:              Tree::addChild
// Description:       Adds a child node to the tree's child list
// Arguments:         tree : sub-tree to add
// Return value:      none.
// ---------------------------------------------------------------

void addChild( Tree* tree );

// ---------------------------------------------------------------
// Name:              Tree::start
// Description:       resets the child iterator to point to the
//                    first child in the current tree.
// Arguments:         none.
// Return value:      none.
// ---------------------------------------------------------------

void start();

// ---------------------------------------------------------------
// Name:              Tree::forth
// Description:       moves the child iterator forward to point to
//                    the next child.
// Arguments:         none.
// Return value:      none.
// ---------------------------------------------------------------

void forth();
```

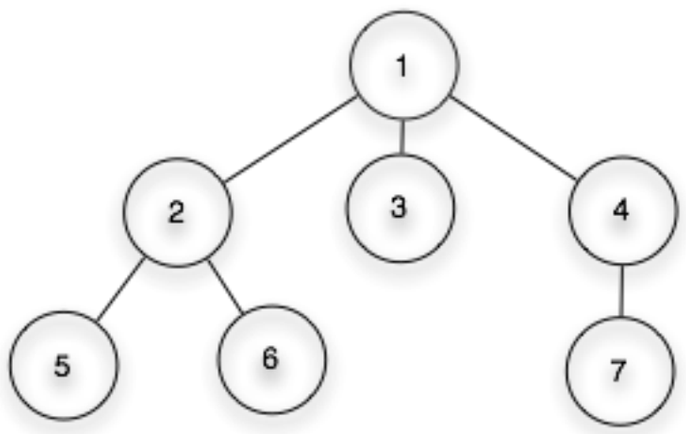
Ejemplo de interfase ADT de un árbol

```
// ------------------------------------------------------------
// Name:              Tree::isValid
// Description:       determines if the child iterator points to a
//                    valid child.
// Arguments:         none.
// Return value:      TRUE if the iterator points to a valid child
//                    FALSE otherwise
// ------------------------------------------------------------

bool isValid();

// ------------------------------------------------------------
// Name:              Tree::currentChild
// Description:       returns a pointer to the current child tree
//                    that the child iterator points to.
// Arguments:         none.
// Return value:      pointer to current child tree
// ------------------------------------------------------------

Tree* currentChild();

// ------------------------------------------------------------
// Name:              Tree::removeChild
// Description:       removes teh child that the child iterator
//                    is pointing to and moves the child iterator
//                    to the next child.
// Arguments:         none.
// Return value:      none.
// ------------------------------------------------------------

void removeChild();
```

Agregar nodos a un árbol

- Encontrar al padre deseado del nodo que queremos insertar
- Agregar el nodo a la lista de hijos de este padre.
- Se puede agregar un nodo a la vez o hacer un nuevo árbol con el número de nodos a insertar y agregar el sub-árbol.

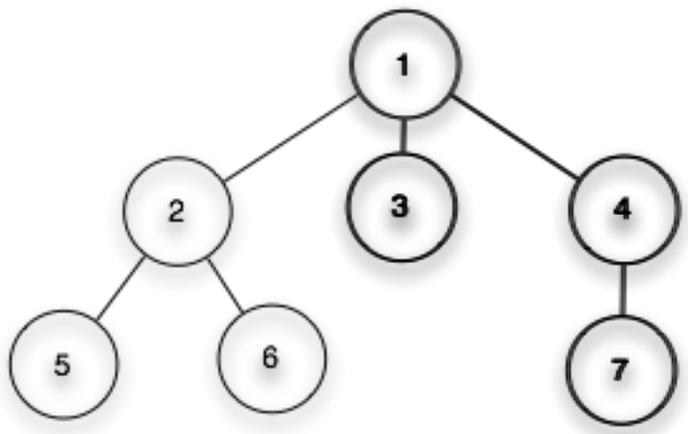
```
void main()
{
    Tree* tree = new Tree(1);
    Tree* temp = NULL;

    tree->addChild( new Tree(2) );
    tree->addChild( new Tree(3) );
    tree->addChild( new Tree(4) );

    tree->start();
    temp = tree->child;
    temp->addChild( new Tree(5) );
    temp->addChild( new Tree(6) );

    tree->forth();
    tree->forth();
    temp = tree->child();
    temp->addChild( new Tree(7) );
}
```



Quitar nodos de un árbol

- Encontrar el nodo a remover y borrarlo de la lista.
- Esto borra el sub-árbol
- En caso de no querer borrar los hijos hay que re-agregarlos

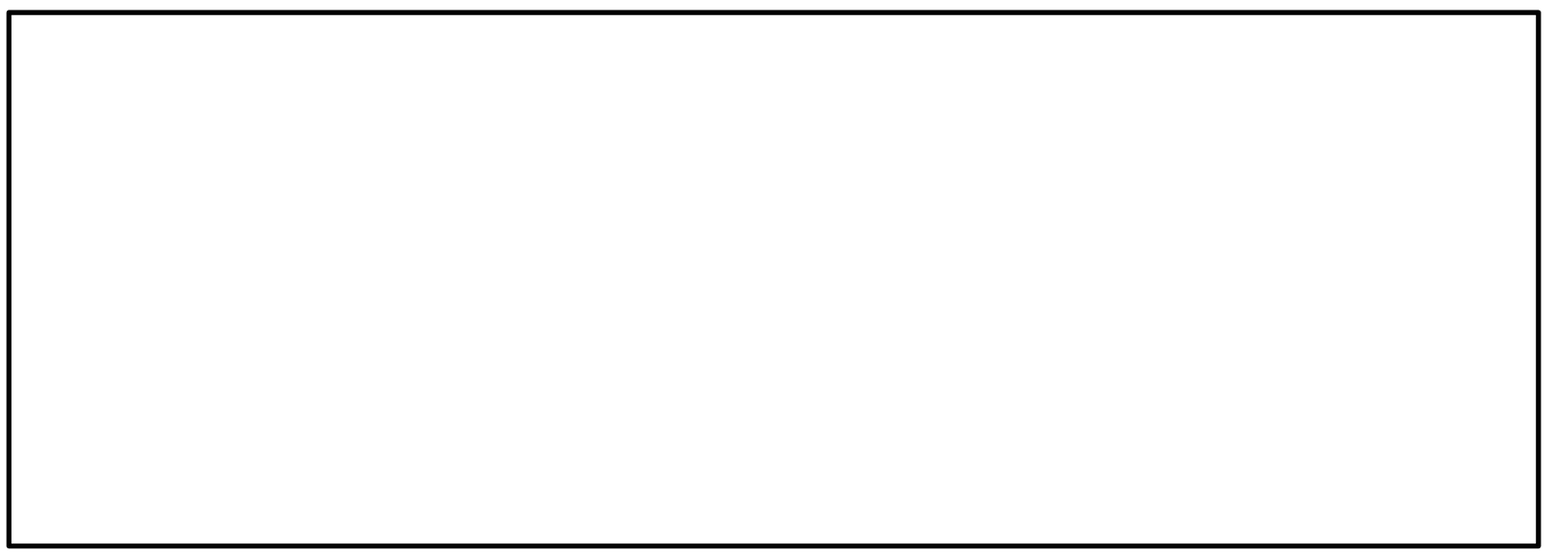
```
void main()
{
    Tree* tree = new Tree(1);

    // crear un arbol

    tree->start();
    tree->removeChild();
}
```



Templates

- Simplificacion en código

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}

float max(float x, float y)
{
    return (x < y) ? y : x;
}
```



Templates

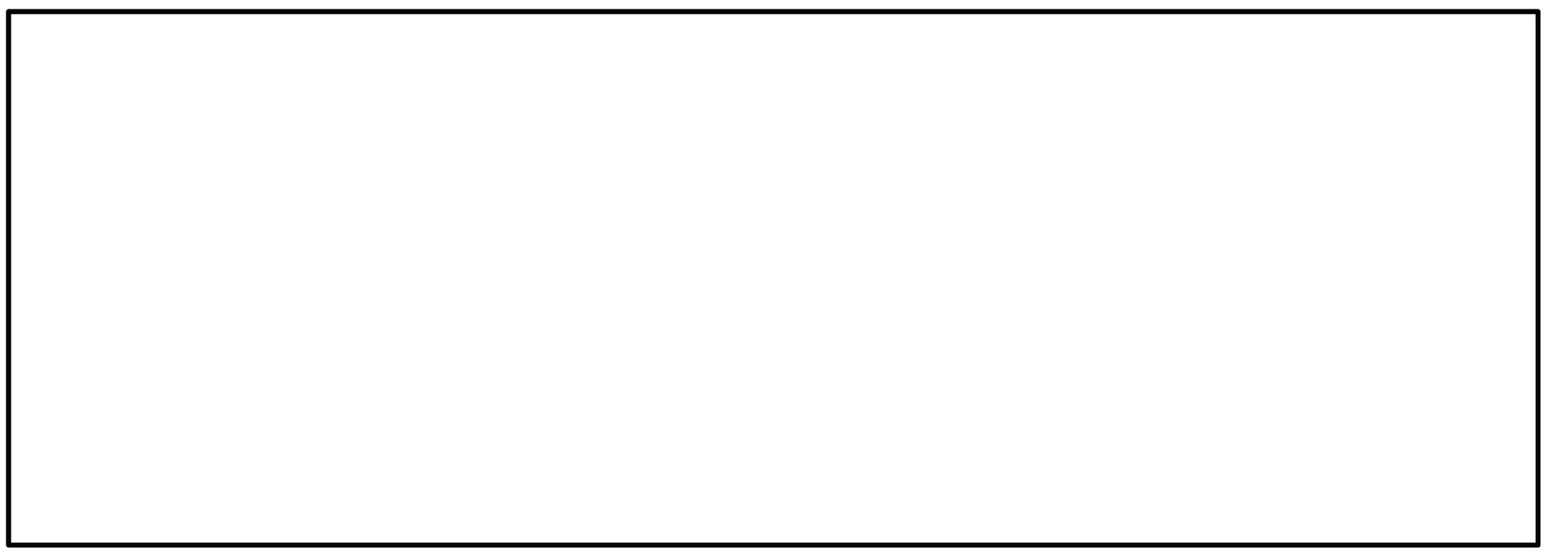
- Simplificacion en código

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}

float max(float x, float y)
{
    return (x < y) ? y : x;
}

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

Templates

- como lo usamos

```
template <typename T>
T max(T x, T y)
{
  T resultado = (x < y) ? y : x;
     return resultado;
}
```



Templates

- como lo usamos

```
template <typename T>
T max(T x, T y)
{
  T resultado = (x < y) ? y : x;
     return resultado;
}


 ...
int a,b,c;
c = max <int> (a,b);

float d,e,f;
f = max <float> (d,e);
...
```

Templates

- Mas ejemplos

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first, T second)
    {
     values[0]=first; values[1]=second;
    }

    T returnFirst(void){
        return values[0];
    }
};
```

Templates

- Mas ejemplos

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first, T second)
    {
     values[0]=first; values[1]=second;
    }

    T returnFirst(void){
        return values[0];
    }
};

...

mypair<int> myobject (115, 36);

mypair<double> myfloats (3.0, 2.18);
```

Templates

- Funciones fuera de la clase

```
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
```

```
template <class T>
T mypair<T>::getmax ()
{
   T retval;
   retval = a>b? a : b;
   return retval;
}
```

Templates

- Funciones fuera de la clase

```
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
```

```
template <class T>
T mypair<T>::getmax ()
{
   T retval;
   retval = a>b? a : b;
   return retval;
}

int main () {
   mypair <int> myobject (100, 75);
   cout << myobject.getmax();
   return 0;
}
```

Nodo y lista template (1)

```
template <class tipo>
class node{

public:
   typedef node<tipo>* link;
private:
   tipo* items;
   link next;
public:

   node(tipo itemArg1,tipo itemArg2,link nextArg){
        items = new tipo[2];
        items[0] = itemArg1;
        items[1] = itemArg2;
        next = nextArg;
        cout << "constructor" << items[0]<< items[1] << endl;
   };
   ~node(){
        cout << "destructor" << items[0]<< items[1] <<endl;
        delete items;
   };


};
```

Nodo y lista template (2)

```
template <class tipo>
class lista{

    typedef typename node<tipo>::link link;

    link nodePtr;

public:
    void insert(link arg){

        link unoMas;
        unoMas = arg;
        nodePtr = arg;
    }


};
```

Nodo y lista template (2)

```
int main(void){
  node<int> a(1,2,NULL);
  node<int> a2(3,4,&a);
  node<int> a3(5,6,&a2);
  node<int> a4(6,7,&a3);

  lista<int> lint;
  lint.insert(&a);



  node<char> b('A','B',NULL);
  node<char> b2('C','D',&b);

  lista<char> lchar;
  lchar.insert(&b);


  node<int>* unL = &a;
}
```