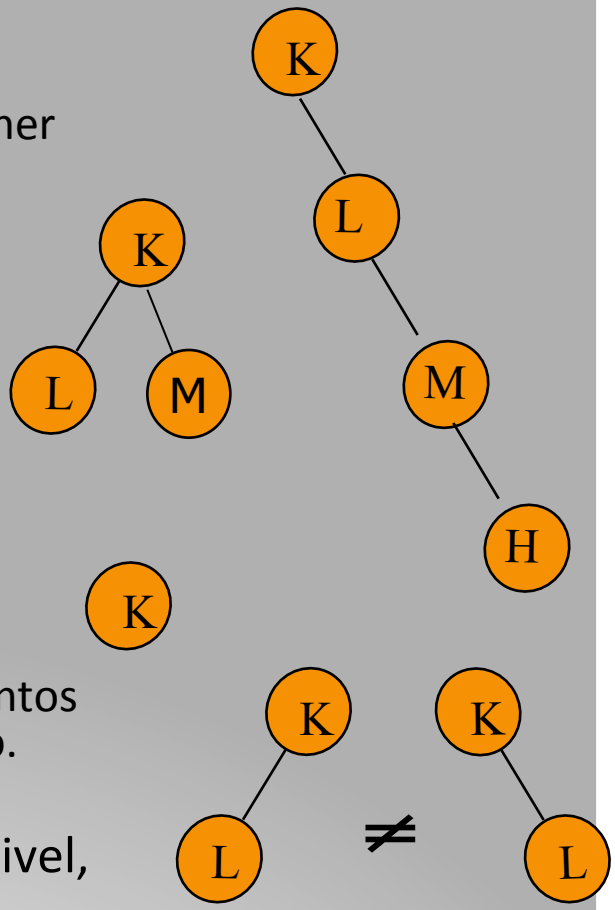
Árboles Binarios

Objetivos

- Definir las características de las estructuras árboles binarios
- Describir la terminología de las ED de tipo árbol binario
- Describir el diseño lógico del TDA árbol binario (AB) incluyendo las operaciones de búsqueda, inserción y eliminación de un elemento
- Diseñar diferentes modos de implementación de un árbol binario
  - razonar las ventajas y las desventajas de las diferentes implementaciones.

Terminología de los Árboles Binarios

- Árbol binario es un árbol tal que cada nodo puede tener hasta dos ramas
- Se admite el concepto de árbol binario vacío
  - Cuando el número de nodos es cero
- **Definición**:
  - Conjunto finito de nodos tal que:
    - está vacío, o
    - consiste de una raíz y dos árboles binarios disjuntos llamados subárbol izquierdo y subárbol derecho.
- Definir los conceptos: hoja, padre, hijo, grado, nivel, etc.



¿Cómo podemos estimar la cantidad de nodos en un AB?

Propiedades:

(1) ¿Podemos calcular el número máximo de nodos en el nivel *i de un AB?*

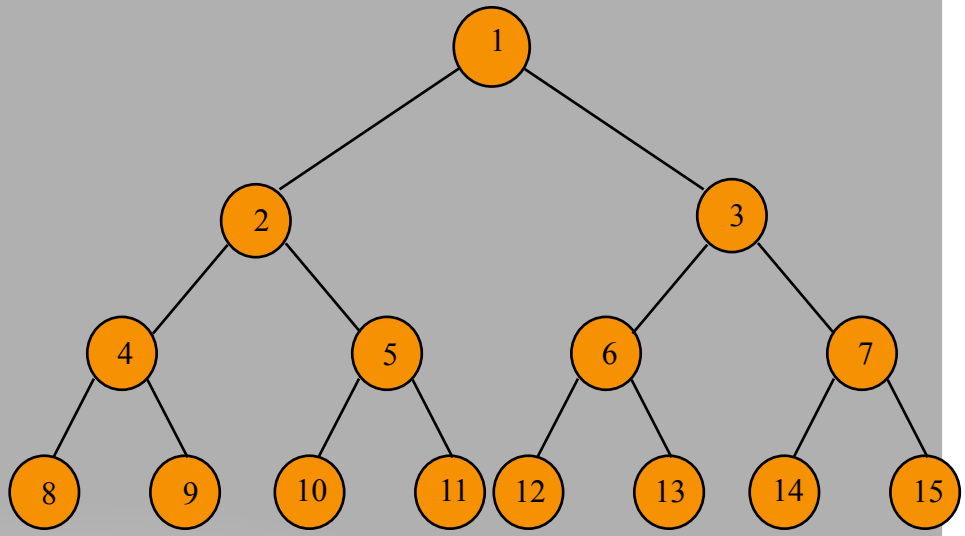
Respuesta: $2^{i-1,}$ i>0

Demostración por inducción:

si *i* == 1 => $2^{i-1}=1$

Sea *i* >1, suponemos que en el nivel *i*-1 el número de nodos es $2^{i-1-1}=2^{i-2}$.

Como el grado máximo de cada nodo es 2, el máximo número de nodos en el nivel *i* es dos veces el máximo número de nodos en el nivel *i*-1, e.d. $2*2^{i-2=}2^{i-1.}$



¿Cómo podemos estimar la cantidad de nodos en un AB?

Propiedades:

(1) ¿Podemos calcular el número máximo de nodos en un árbol binario de profundidad k?

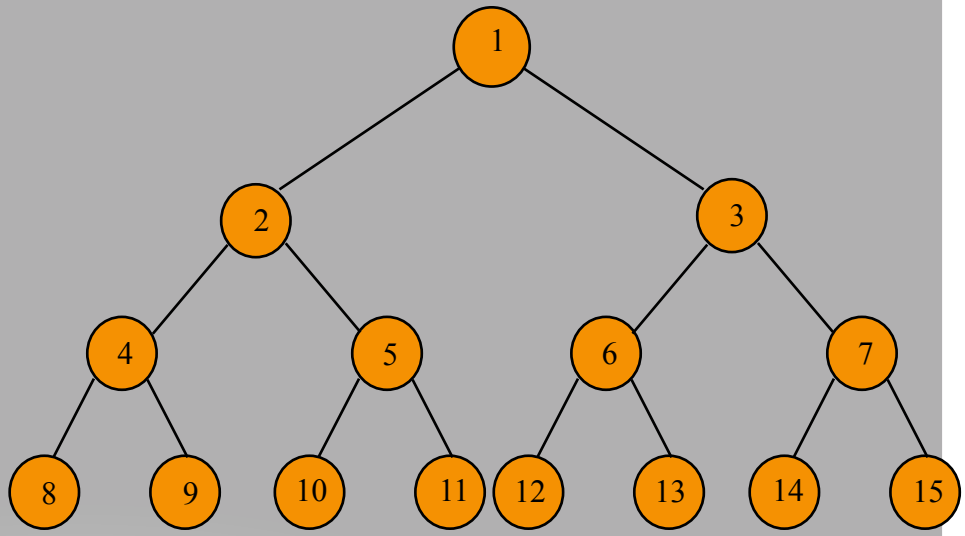
Respuesta: 2^k-1, k>0

Demostración por inducción:

Si k == 1 => $2^k-1=1$

Sea k >1. Suponemos que en un AB de nivel k-1 el número de nodos es $2^{k-1}-1$.

Como el número máximo del nivel k es 2^{k-1}, añadiendo el nivel k, obtendremos $2^{k-1}-1+ 2^{k-1}=2*2^{k-1}-1=2^k-1$.



¿Qué es un AB completo?

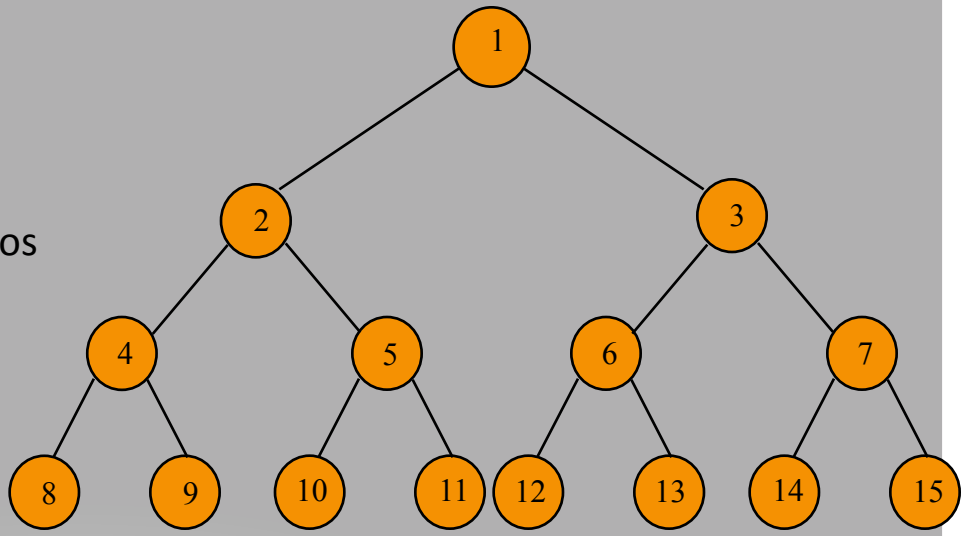
Árbol binario completo: un árbol binario de profundidad k y n nodos es completo si $n=2^k-1$.

Propiedad: La altura H de un árbol binario completo de n nodos es: $\log_2 (n+1)$.

Demostración: si el AB es completo, tenemos $n=2^k-1$.

=> $n+1=2^k$

$\log_2 (n+1)=\log_2 (2^k)$
$\log_2 (n+1)=k*\log_2(2)$
$k=\log_2 (n+1)$



Fijémonos: $\log_2 (1.000.000)=20$

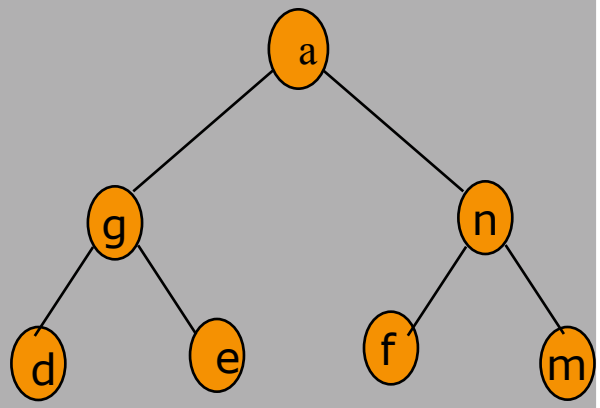
Nota: Podemos colocar hasta 1000000 nodos en un AB con una altura (profundidad) de sólo 20!!!

Árbol Binario como un TDA

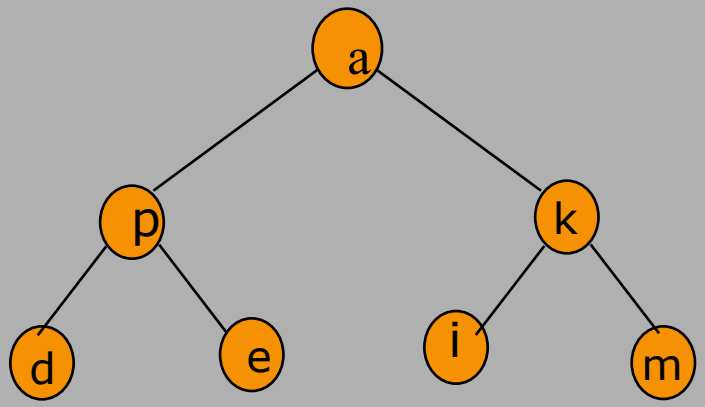
BinaryTree()	Crea una nueva instancia de un árbol binario.
getLeftChild()	Retorna el árbol binario izquierdo del nodo actual.
getRightChild()	Retorna el árbol binario derecho del nodo actual.
setRootVal(val)	Almacena el valor ***val*** en el nodo actual.
getRootVal()	Retorna el valor guardado en el nodo actual (la raíz).
insertLeft(val)	Crea un nuevo árbol binario y lo encadena como subarbol izquierdo del nodo actual.
insertRight(val)	Crea un nuevo árbol binario y lo encadena como subarbol derecho del nodo actual.

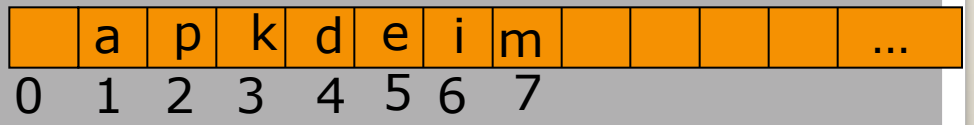
¿Cuál sería el diseño lógico del TDA árbol binario?

```
class BinaryTree(object):
                # conjunto finito de nodos:
                # __proot, a partir del cual se puede acceder
                # al subárbol izquierdo y el subárbol derecho

    def __init__(self,.....): #constructor:
                #Crea un árbol con __proot = ítem y subárboles t1 y t2

    def empty(self): #verdadero si el árbol esta vacío

    def getRootVal(self): #retorna los datos de la raíz

    def getLeft (self):
                #si no esta vacío retorna el subárbol izquierdo de la raíz

    def getRight(self):
                #si no está vacío retorna el subárbol derecho de la raíz

    def insertRight(self, val):

    def insertLeft(self,val):

    def setRootVal(self,val)
```



¿Cuál es la representación vectorial?

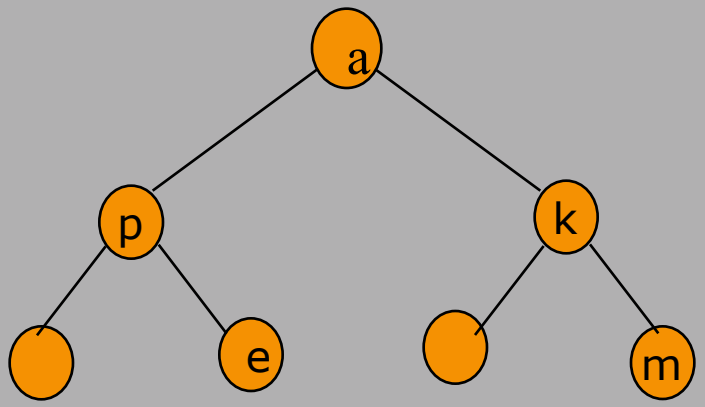




- Reglas para la representación secuencial:
  - Si la raíz está en la posición 1, los hijos izquierdo y derecho de cada nodo en posición i están en la posición 2*i y 2*i+1
  - El padre de cada nodo en posición i está en la posición i/2 (división entera)

....

¿Cuáles son las ventajas y las desventajas de la representación secuencial?



	a	p	k	d	e	i	m					...
0	1	2	3	4	5	6	7					

¿Se cumplen las fórmulas?

	a	p	k		e		m					...
0	1	2	3	4	5	6	7					

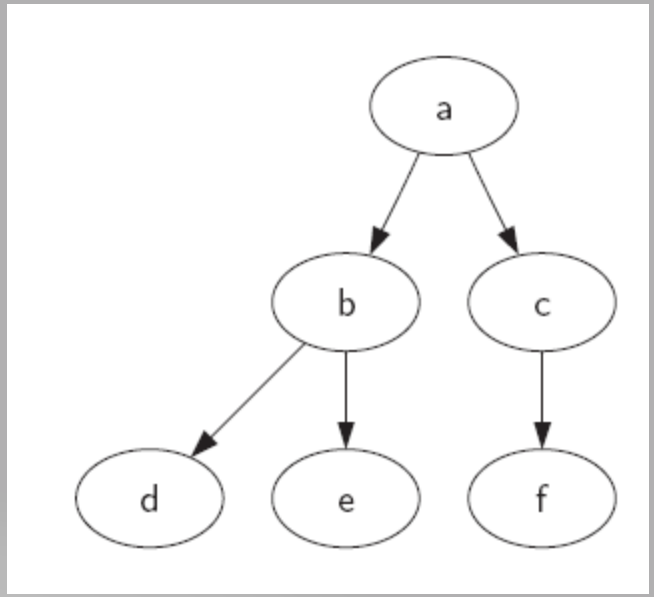
¿Se desperdicia memoria?

- Representación secuencial:
  - Fácil gestión
  - Fácil direccionamiento a los hijos y los padres (tiempo constante)
  - Eficiente solo para árboles binarios completos (o balanceados)
  - Si los árboles no son completos, se desperdicia memoria (huecos)

....

Representación del árbol como lista de listas

```
myTree = ['a', #root
            ['b', #left subtree
                ['d' [], []],
                ['e' [], []] ],
            ['c', #right subtree
                ['f' [], []],
                [] ]
          ]
```



(A(B(D,E),C(F)))

Representación del árbol como lista de listas

```
def BinaryTree(r): # crea un árbol binario
    return [r, [], []]

def insertLeft(root,newBranch):
                                # inserta un subárbol izquierdo
    t = root.pop(1)
    if len(t) > 0:
            root.insert(1,[newBranch,t,[]])
    else :
            root.insert(1,[newBranch, [], []])
    return root
```

Insertar un subárbol derecho

```
def insertRight(root,newBranch):
        t = root.pop(2)
        if len(t) > 0:
                root.insert(2,[newBranch,[],t])
        else :
                root.insert(2,[newBranch,[],[]])
        return root
```

Funciones de acceso

```
def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeft(root):
    return root[1]

def getRight(root):
    return root[2]
```

Ventajas:

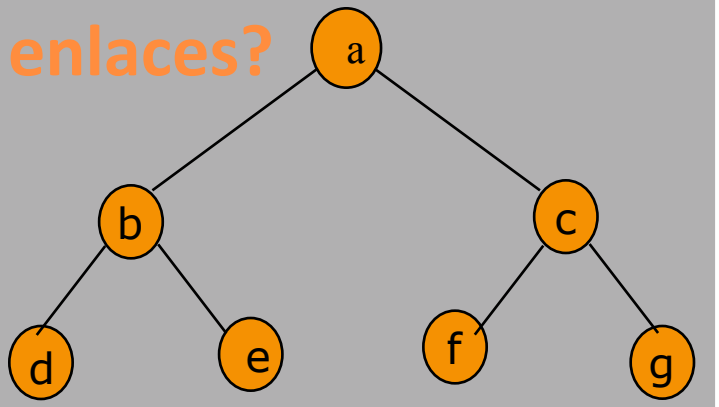
- Es fácil generalizar para mayor grado.
- La estructura recursiva se ve claramente.

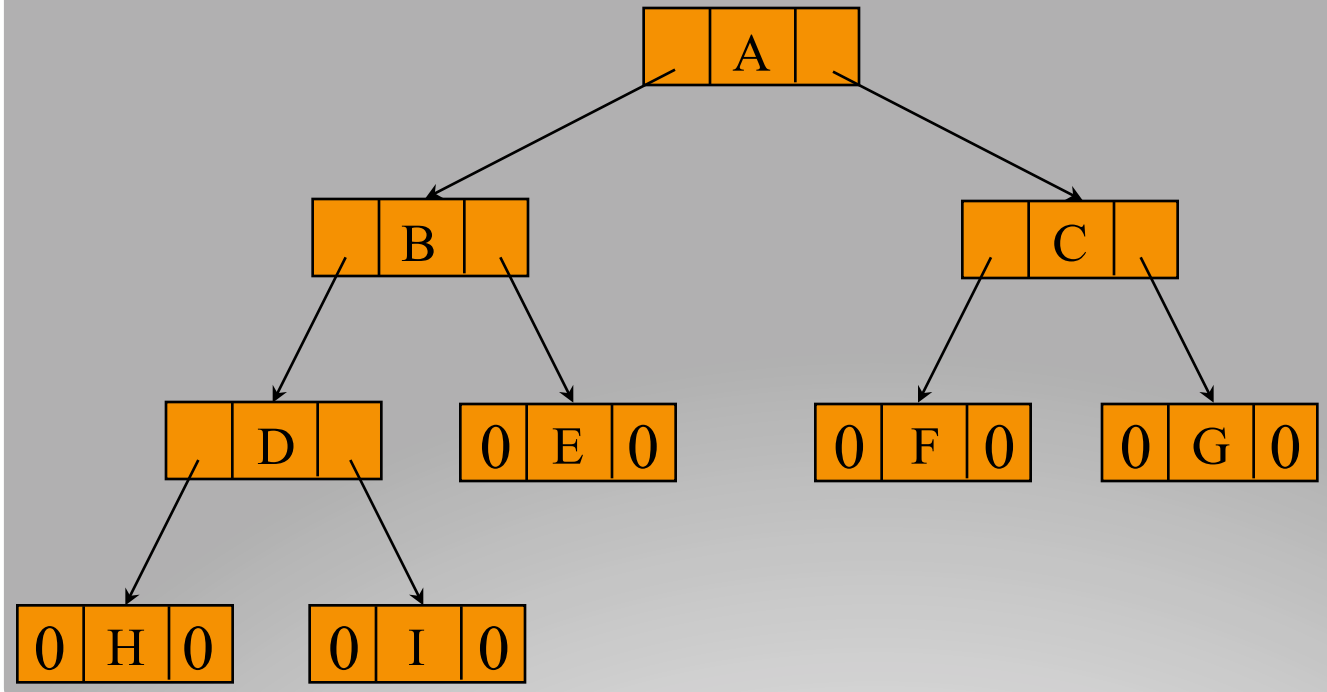
Illustración

```
>>>r=BinaryTree(3)
>>>insertLeft(r,4)

>>>insertLeft(r,5)

>>>insertRight(r,6)

>>>insertRight(r,7)

>>>l=getLeftChild(r)
>>>l

>>>setRootVal(1,9)
>>>r

>>>insertLeft(l,11)

>>>r
```

```
>>>r=BinaryTree(3)
>>>insertLeft(r,4)
[3, [4, [],[]], []]
>>>insertLeft(r,5)
[3, [5, [4, [],[]], []], []]
>>>insertRight(r,6)
[3, [5, [4, [],[]], []], [6, [],[] ]]
>>>insertRight(r,7)
[3, [5, [4, [],[]], []], [7, [],  [6, [],[] ]]]
>>>l=getLeftChild(r)
>>>l
[5,[4,[],[]], []]
>>>setRootVal(l,9)
>>>r
[3, [9,[4,[],[]],[]],[7,[],[6, [],[]]]]
>>>insertLeft(l,11)
[9,[11,[4],[]],[]]
>>>r
[3, [9, [11, [4, [], []], []], []], [7, [], [6, [], []]]]
```

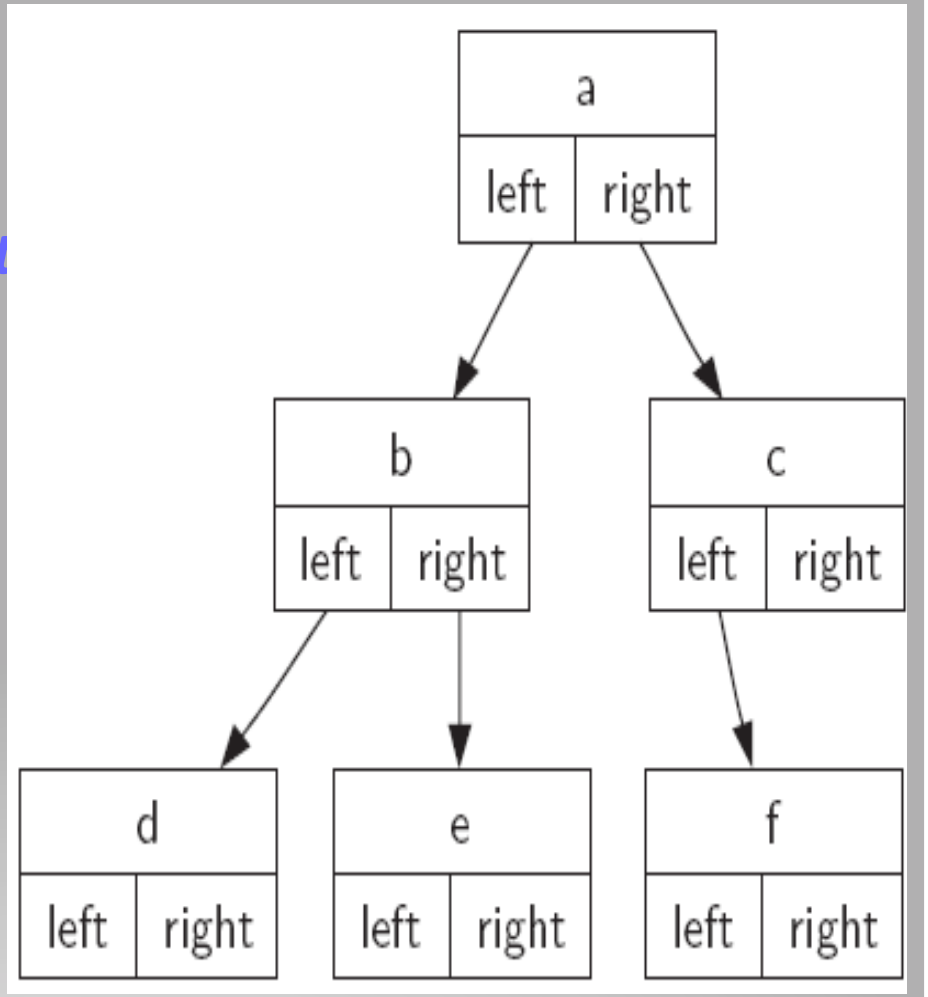
¿Cuál sería la representación con enlaces?





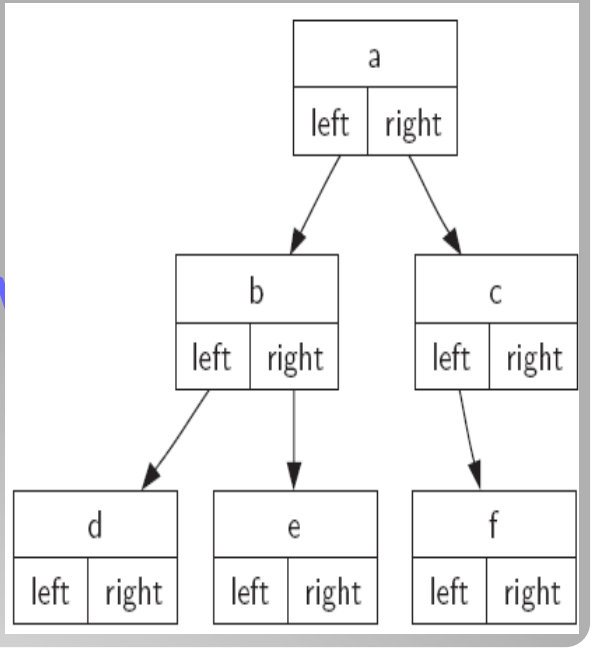
Implementación de un árbol con nodos y referencias

```
class Node(object):
    def __init__(self, val=0, left
        self._val=val
        self._left=left
        self._right=right

    def getVal(self):
        return self._val

    def setVal(self,item):
        self._val=item
```



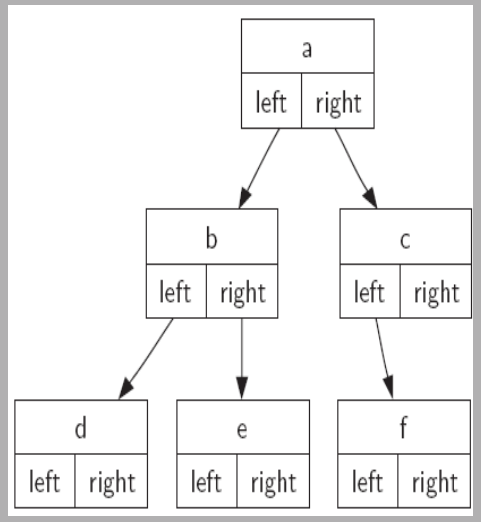
Implementación de un árbol con nodos y referencias

```
def getLeft(self):
        return self._left
def setLeft(self,item,left=None,right=None):
        self._left=Node(item,left,right)
def getRight(self):
        return self._right
def setRight(self,item,left=None,right=N
         self._right=Node(item,left,right)
def __str__(self):
    return str(self._val)
```



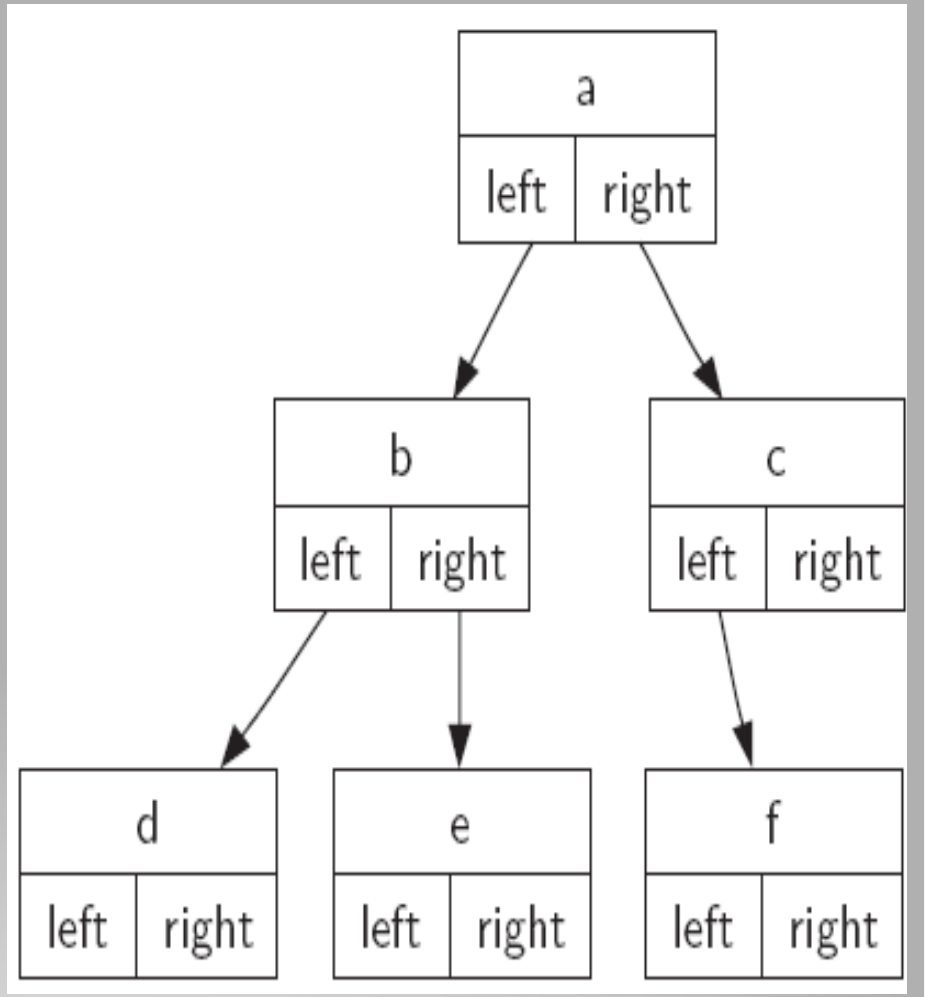
Implementación de un árbol con nodos y referencias

```
from Node import *

class Arbol(object):
    def __init__(self,val=None, left=None, right=None):
        if val==None:
            self._root=None
        elif left==None and right==None:
            self._root=Node(val)
        elif left==None and right!=None:
            self._root=Node(item,None,right._getRoot())
        elif left!=None and right==None:
            self._root=Node(item,left._getRoot())

        else: self._root=Node(val,left._getRoot(),right._getRoot())
```



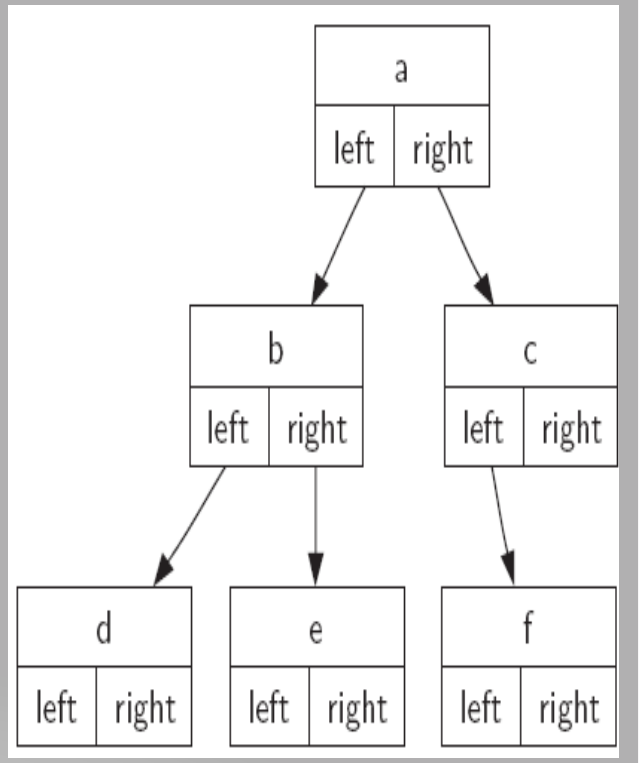
Implementación de un árbol con nodos y referencias

```
def empty(self):
    return self._root==None

def getRootVal(self):
    if self._root!=None:
        return self._root.getVal()
    else: return None

def setRootVal(self,item):
    if self._root!=None:
        self._root.setVal(item)
    else: self._root=Node(item)

def _getRoot(self):
    return self._root
```



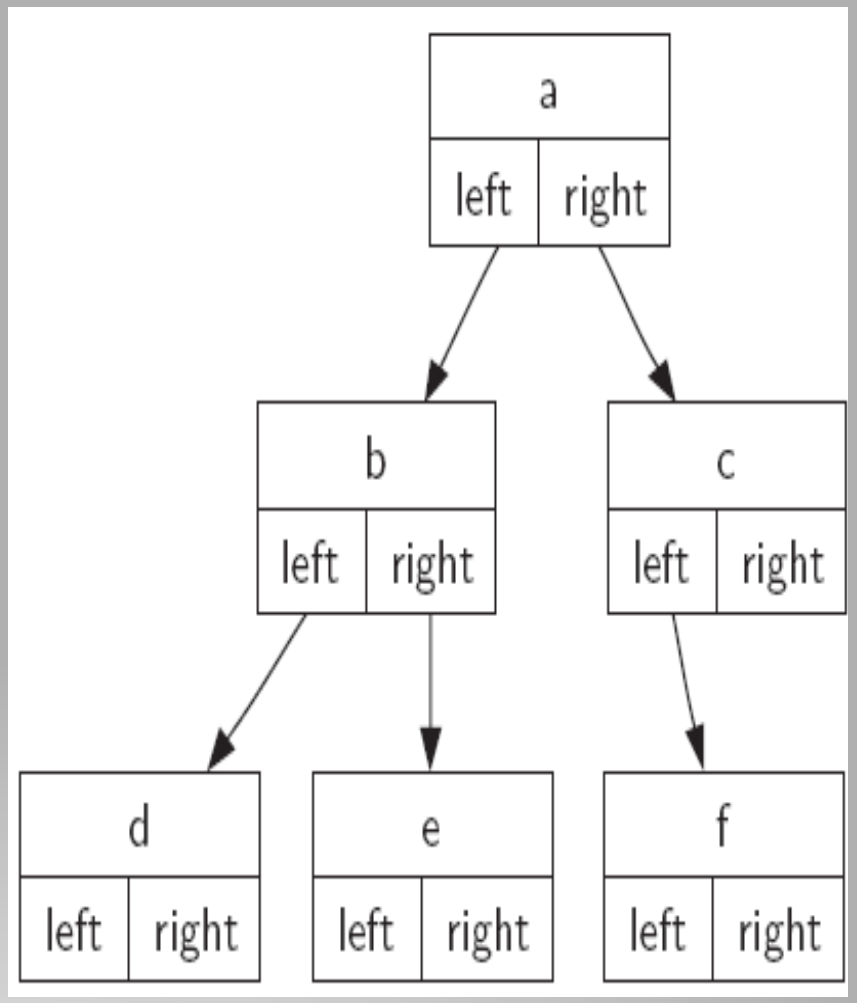
Implementación de un árbol con nodos y referencias

```
def getLeft(self):
    if self._root!=None:
        if self._root.getLeft()!=None:
            return self._root.getLeft().getVal()
    else: return None


  def insertLeft(self,item):
    if self._root!=None:
        if self._root.getLeft()!=None:
            self._root.setLeft(item,self._root.getLeft().getLeft(),
                               self._root.getLeft().getRight())
        else:  self._root.setLeft(item)
```



Funciones de acceso

```
def getRootVal(self,):
    return self.key

def setRootVal(self,obj):
    self.key = obj

def getLeftChild(self):
    return self.left

def getRightChild(self):
    return self.right
```



Conclusiones

- Los árboles binarios utilizan la misma terminología que los árboles.
- Los árboles completos son los árboles que optimizan el uso de memoria.
- Hay dos formas para representar un árbol binario:
  - Representación secuencial – óptima para árboles binarios completos.
  - Representación con enlaces – óptima respecto a la inserción y eliminación de nodos y no desperdicia memoria.
    - Pero necesita gestionar los enlaces de los nodos.