

# Algoritmos y Estructuras de Datos II

Recorriendo grafos

6 de junio de 2016

# Clase de hoy

- 1 Repaso
- 2 Recorrida de grafos
  - Generalidades
  - Árboles binarios
  - Árboles finitarios
  - Grafos arbitrarios, DFS
  - Grafos arbitrarios, BFS

# Repaso

- cómo vs. qué
- 3 partes
  - 1 análisis de algoritmos
  - 2 tipos de datos
  - 3 técnicas de resolución de problemas
    - divide y vencerás
    - algoritmos voraces
    - backtracking
    - programación dinámica: problema de la moneda, problema de la mochila, algoritmo de Floyd
    - [recorrida de grafos](#)

## Recorrida de grafos

**Recorrer** un grafo, significa **procesar** los vértices del mismo, de forma organizada de modo de asegurarse:

- que todos los vértices sean **procesados**,
- que ninguno de ellos sea **procesado** más de una vez.

Se habla de **procesar** los vértices, pero también utilizaremos la palabra **visitar** los vértices. En este contexto, son sinónimos. Puede haber más de una forma natural de recorrer un cierto grafo.

## Recorrida de árboles binarios

Un caso de grafo sencillo que ya han visto es el de árbol binario. Se han visto 3 maneras de **recorrerlo**:

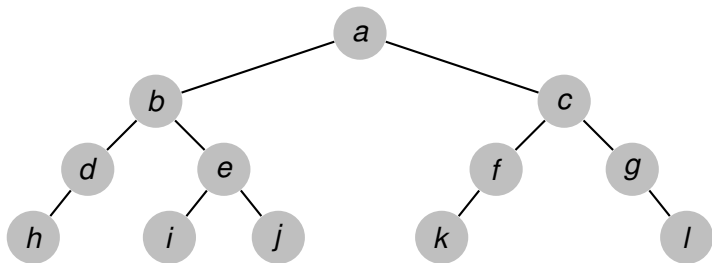
- pre-order** Se **visita** primero el elemento que se encuentra en la raíz, luego se **recorre** el subárbol izquierdo y finalmente se **recorre** el subárbol derecho.
- in-order** Se **recorre** el subárbol izquierdo, luego se **visita** el elemento que se encuentra en la raíz y finalmente se **recorre** el subárbol derecho.
- pos-order** Se **recorre** el subárbol izquierdo, luego el derecho y finalmente se **visita** el elemento que se encuentra en la raíz.

# Algoritmos

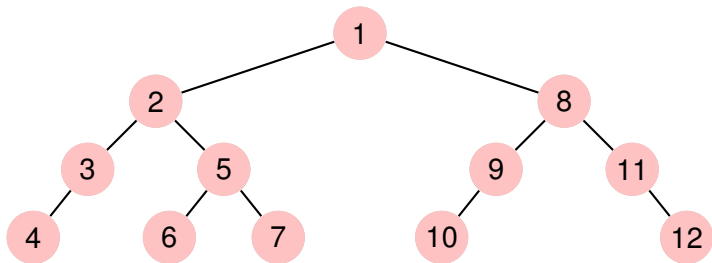
Los siguientes algoritmos reflejan estas distintas maneras de recorrer árboles binarios, generando listas con sus elementos. Los elementos aparecen en el orden en que se visitan.

$$\text{pre\_order}(\langle \rangle) = [ ]$$
$$\text{pre\_order}(\langle l, e, r \rangle) = e \triangleright \text{pre\_order}(l) ++ \text{pre\_order}(r)$$
$$\text{in\_order}(\langle \rangle) = [ ]$$
$$\text{in\_order}(\langle l, e, r \rangle) = \text{in\_order}(l) ++ (e \triangleright \text{in\_order}(r))$$
$$\text{pos\_order}(\langle \rangle) = [ ]$$
$$\text{pos\_order}(\langle l, e, r \rangle) = \text{pos\_order}(l) ++ \text{pos\_order}(r) \triangleleft e$$

## Ejemplo de árbol binario

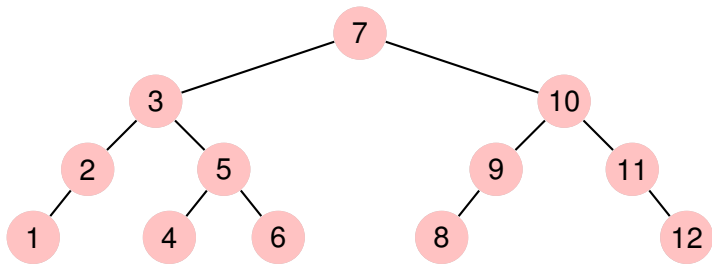


## Ejemplo, recorrida pre-order

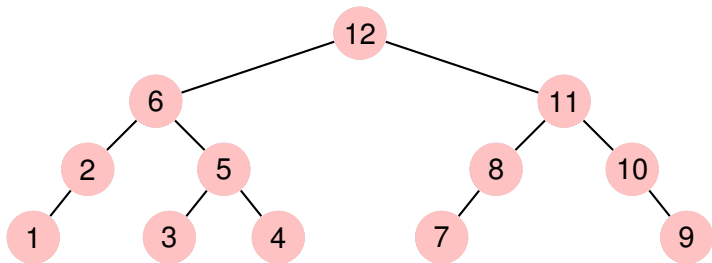




## Ejemplo, recorrida in-order



## Ejemplo, recorrida pos-order

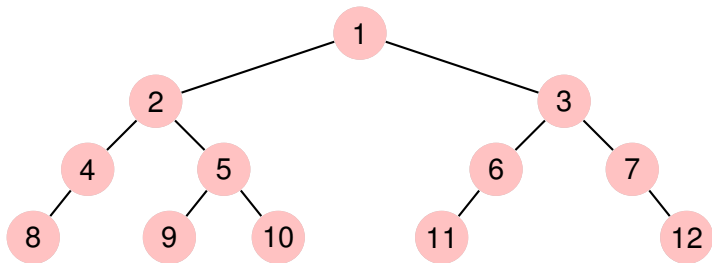


## Otras 3 maneras de recorrer árboles binarios

Hay otras tres maneras de recorrer: en cada una de las anteriores, intercambiar el orden entre las recorridas de los subárboles. Por ejemplo:

```
in_order_der_izq(<>) = []  
in_order_der_izq(< l, e, r >) =  
    in_order_der_izq(r) ++ (e ▷ in_order_der_izq(l))
```

## Otra manera más de recorrer árboles binarios



## Otra manera más de recorrer árboles binarios

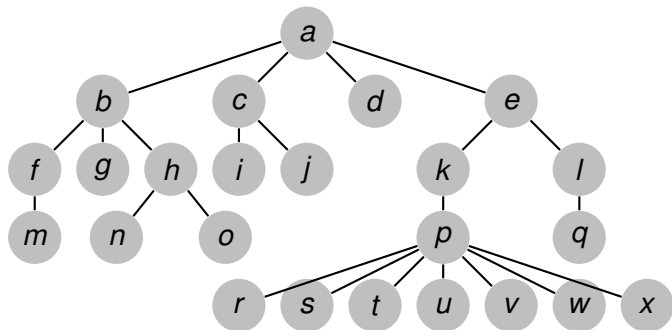
Algunas observaciones:

- salvo la última, todas las formas anteriores de recorrer, primero recorren **en profundidad**
- la última que presentamos, no,
- recorre **a lo ancho**.
- Todas las otras son ejemplo de DFS (Depth-first search).
- La última es ejemplo de BFS (Breadth-first search).
- Un programa que recorra en BFS es más difícil de escribir, se verá al final de la clase de hoy.

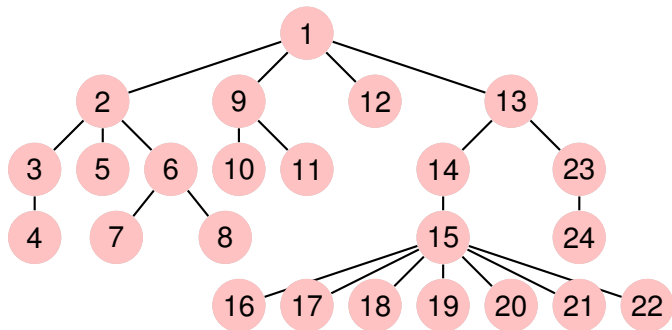
## Recorrida de árboles finitarios

- Son árboles en los que cada vértice tiene una cantidad finita (pero puede ser variable) de hijos.
- La recorrida in-order deja de tener sentido (habiendo más de dos hijos, ¿en qué momento habría que visitar el elemento que se encuentra en la raíz?).
- Las recorridas pre-order y pos-order (DFS) y BFS siguen teniendo sentido.

## Ejemplo de árbol finitario

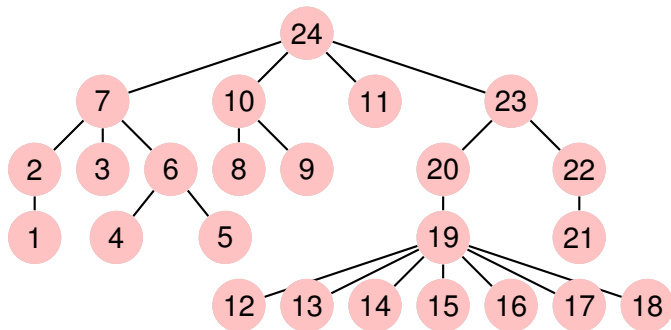


## Ejemplo, recorrida pre-order

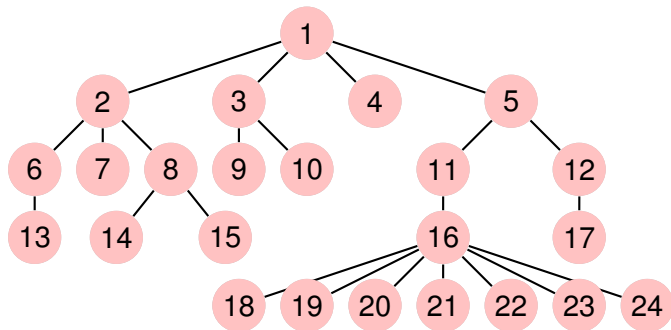




## Ejemplo, recorrida pos-order



## Ejemplo, recorrida BFS



# Algoritmos

## Marcas

Cuando se visita un vértice, se marca con un número positivo.

```
type tmark = tuple
    ord: array[V] of nat
    cont: nat
end
proc init(out mark: tmark)
    mark.cont:= 0
end
proc visit(in/out mark: tmark, in v: V)
    mark.cont:= mark.cont+1
    mark.ord[v]:= mark.cont
end
```

# Algoritmos

## pre-order

Asumimos que un árbol viene dado por su raíz (root) y una función (children) que devuelve (el conjunto o la lista de) los hijos de cada vértice.

```
fun pre_order(G=(V,root,children)) ret mark: tmark  
    init(mark)  
    pre_traverse(G, mark, root)  
end
```

```
proc pre_traverse(in G, in/out mark: tmark, in v: V)  
    visit(mark,v)  
    for w  $\in$  children(v) do pre_traverse(G, mark, w) od  
end
```

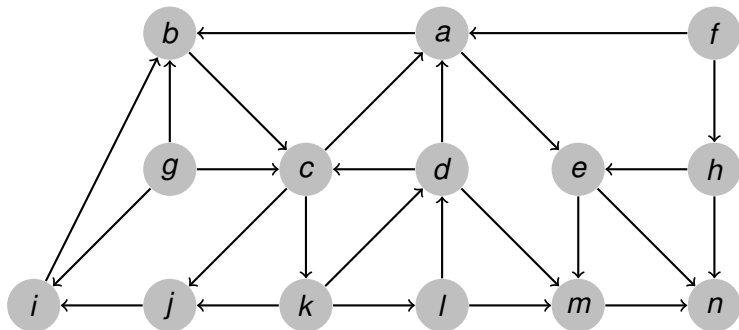
# Algoritmos

pos-order

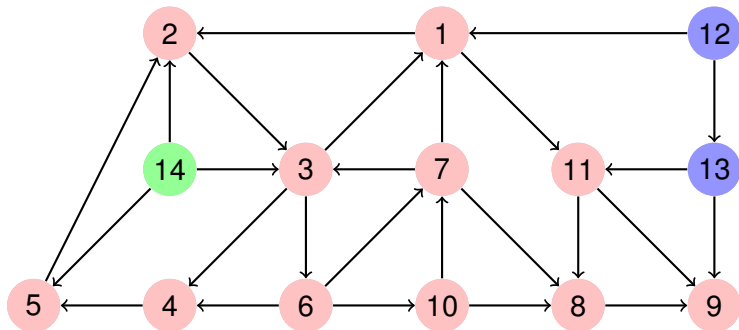
```
fun pos_order(G=(V,root,children)) ret mark: tmark  
    init(mark)  
    pos_traverse(G, mark, root)  
end
```

```
proc pos_traverse(in G, in/out mark: tmark, in v: V)  
    for w  $\in$  children(v) do pos_traverse(G, mark, w) od  
    visit(mark,v)  
end
```

## Ejemplo de grafo



## Ejemplo, DFS en pre-orden



# Algoritmos

## Marcas

Como ahora puede haber ciclos, es necesario poder averiguar si un vértice ya fue visitado.

```
proc init(out mark: tmark)
    mark.cont:= 0
    for v  $\in$  V do mark.ord[v]:= 0 od
end

fun visited(mark: tmark, v: V) ret b: bool
    b:= (mark.ord[v]  $\neq$  0)
end
```



## Algoritmo DFS

```
fun dfs(G=(V,neighbours)) ret mark: tmark
  init(mark)
  for v ∈ V do
    if ¬visited(mark,v) then dfsearch(G, mark, v) fi
  od
end

proc dfsearch(in G, in/out mark: tmark, in v: V)
  visit(mark,v)
  for w ∈ neighbours(v) do
    if ¬visited(mark,w) then dfsearch(G, mark, w) fi
  od
end
```

## DFS iterativo

Introducimos una pila para evitar recursión

```
proc dfsearch(in G, in/out mark: tmark, in v: V)
  var p: stack of V
  empty(p)
  visit(mark,v)
  push(v,p)
  while  $\neg$ is_empty(p) do
    if existe  $w \in$  neighbours(top(p)) tal que  $\neg$ visited(mark,w) then
      visit(mark,w)
      push(w,p)
    else pop(p)
    fi
  od
end
```

# BFS

Si cambiamos la pila por una cola obtenemos BFS

```
proc bfsearch(in G, in/out mark: tmark, in v: V)
  var q: queue of V
  empty(q)
  visit(mark,v)
  enqueue(q,v)
  while  $\neg$ is_empty(q) do
    if existe  $w \in$  neighbours(first(q)) tal que  $\neg$ visited(mark,w) then
      visit(mark,w)
      enqueue(q,w)
    else dequeue(q)
    fi
  od
end
```

## BFS, procedimiento principal

```
fun bfs(G=(V,neighbours)) ret mark: tmark  
  init(mark)  
  for v ∈ V do  
    if ¬visited(mark,v) then bfssearch(G, mark, v) fi  
  od  
end
```

## Ejemplo, BFS

