

Estructuras de Datos

Clase 10 – Árboles Generales

(primera parte)

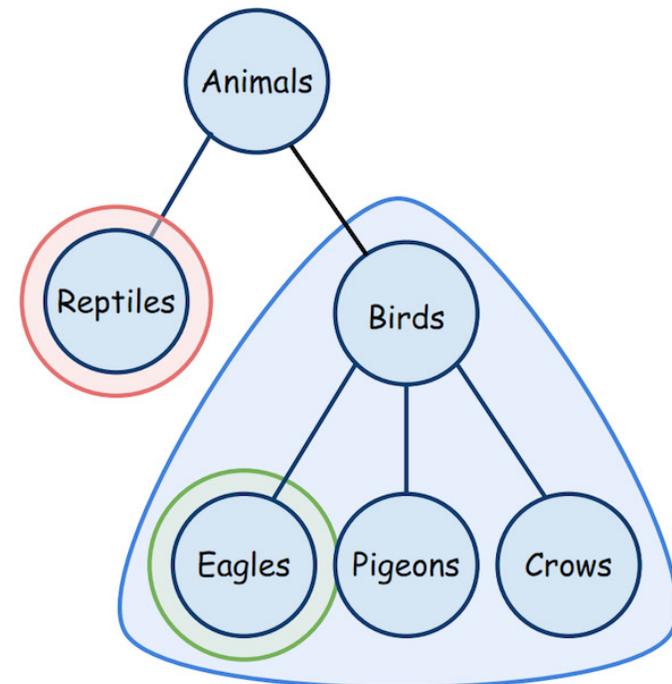
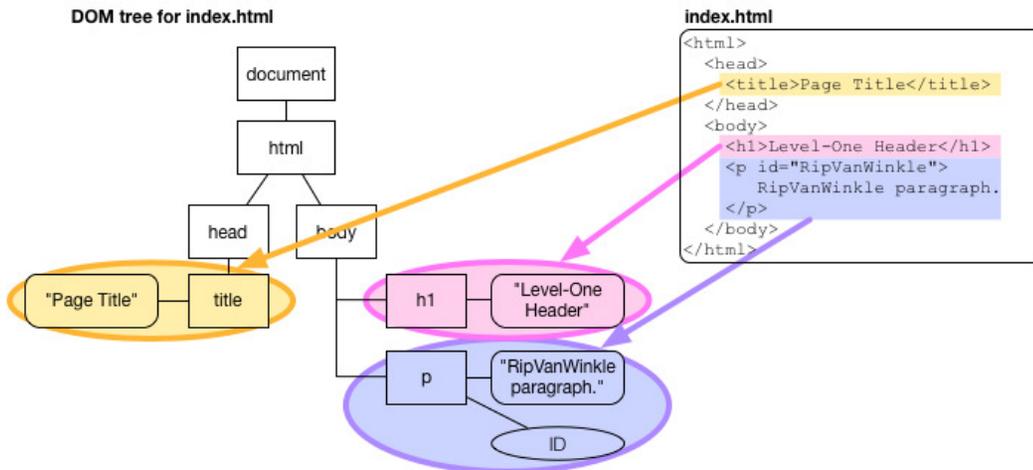
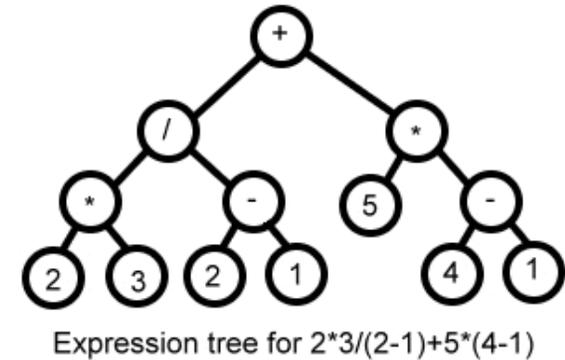
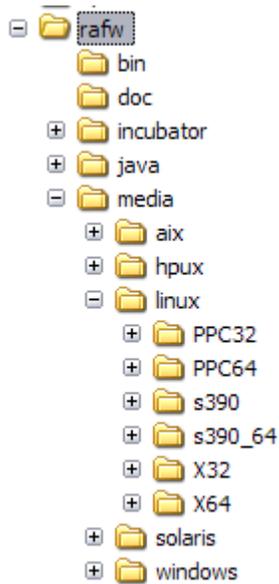


Dr. Sergio A. Gómez
<http://cs.uns.edu.ar/~sag>



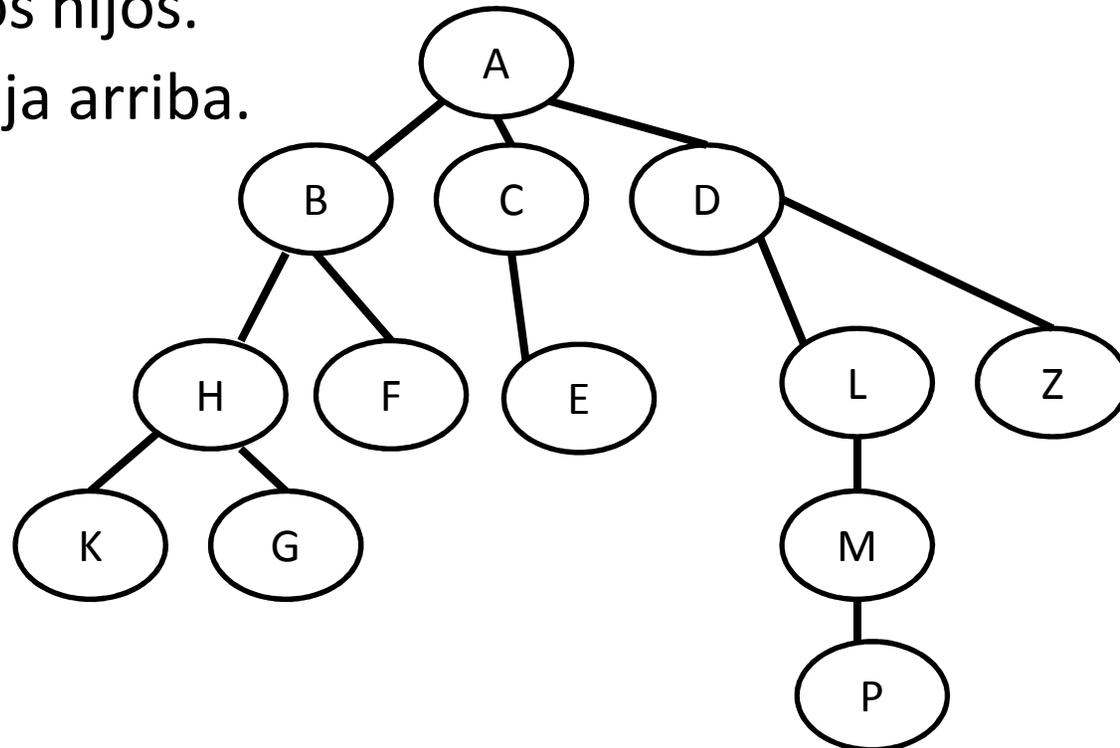
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Estructuras de datos jerárquicas: motivaciones



Preliminares

- Un árbol es un TDA que almacena los elementos jerárquicamente
- Con la excepción del elemento tope (raíz), cada elemento en un árbol tiene un elemento padre y cero o más elementos hijos.
- La raíz se dibuja arriba.



Definición formal de árbol

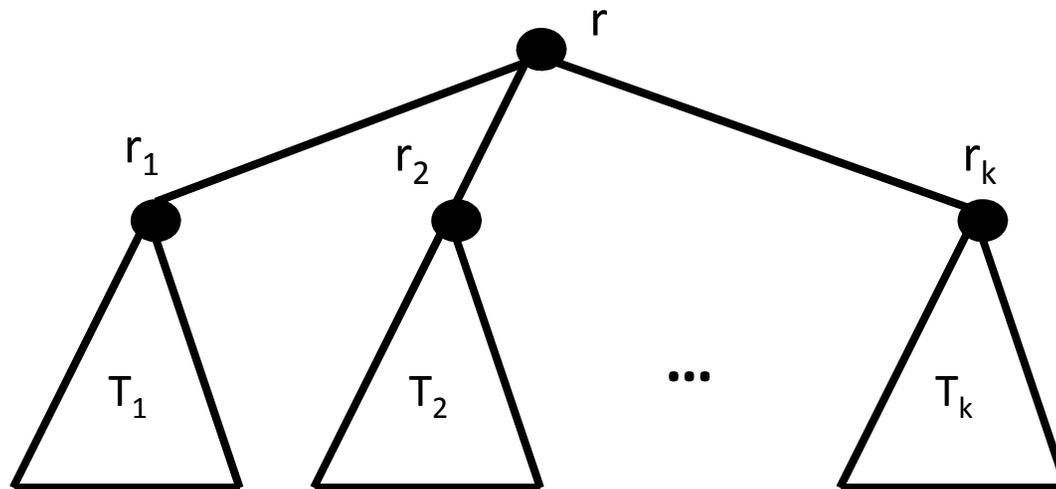
Un árbol T se define como un conjunto de nodos almacenando elementos tales que los nodos tienen una relación padre-hijo, que satisface:

- Si T es no vacío, tiene un nodo especial, llamado la raíz de T , que no tiene padre.
- Cada nodo v de T diferente de la raíz tiene un único nodo padre w
- Cada nodo v con padre w es un hijo de w .

Definición recursiva de árbol

Un árbol T es:

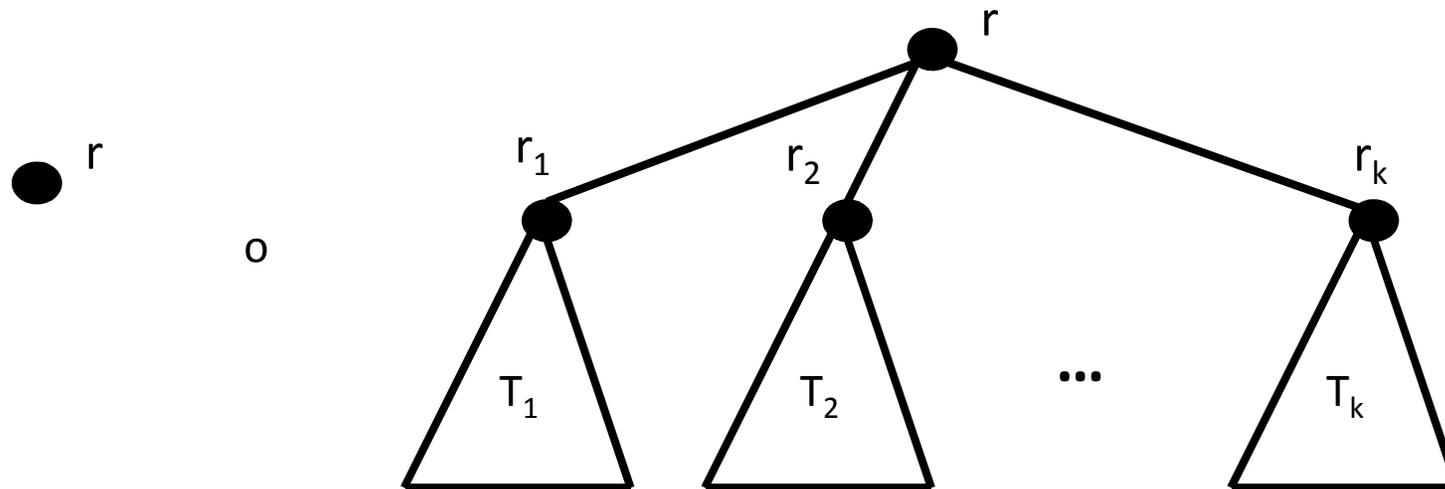
- Vacío (es decir, no tiene nodos)
- Es un nodo r (llamado la raíz de T) y un conjunto (posiblemente vacío) de árboles T_1, T_2, \dots, T_k cuyas raíces r_1, r_2, \dots, r_k son los hijos de r .



Definición recursiva de árbol no vacío

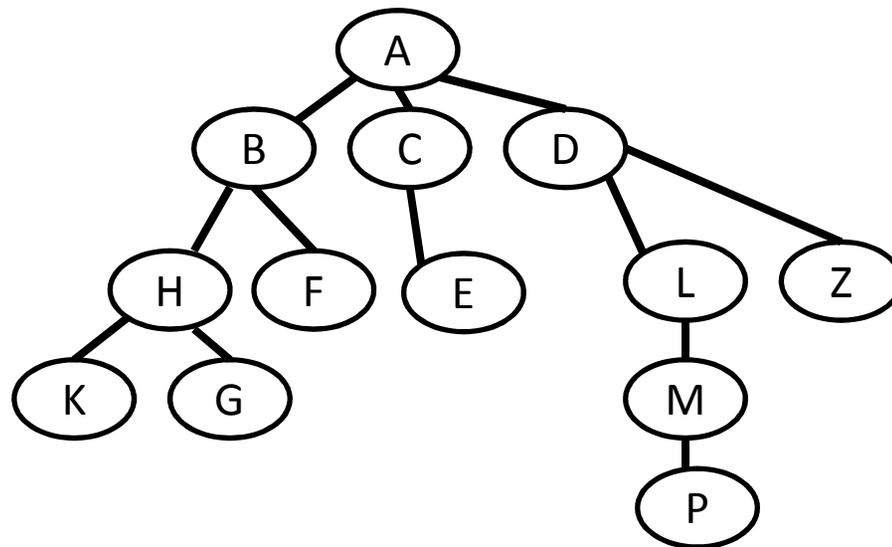
Un árbol no vacío T es:

- Hoja(r): Un nodo hoja r (llamado raíz de T)
- Nodo(r, T_1, T_2, \dots, T_k): Es un nodo r (llamado la raíz de T) y un conjunto de árboles no vacíos T_1, T_2, \dots, T_k cuyas raíces r_1, r_2, \dots, r_k son los hijos de r .



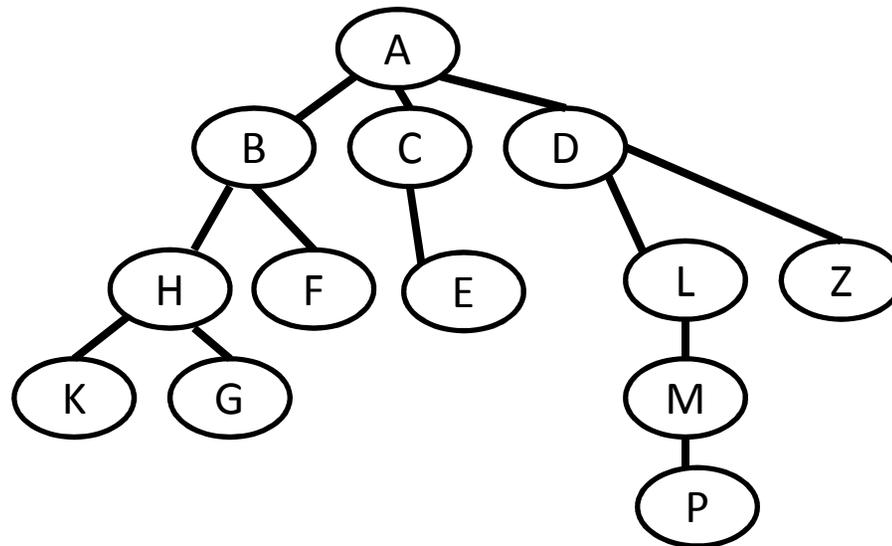
Relaciones entre nodos

- Nodos Hermanos: Dos nodos con el mismo padre se llaman hermanos. Ej. B y D son hermanos, L y Z lo son.
- Nodos externos u hojas: Un nodo v es externo si no tiene hijos. Ej: F, K, G, E, P y Z son hojas.
- Nodo interno: Un nodo v es interno si tiene uno o más hijos. Ej: A, B, C, D, H, L y M son nodos internos.



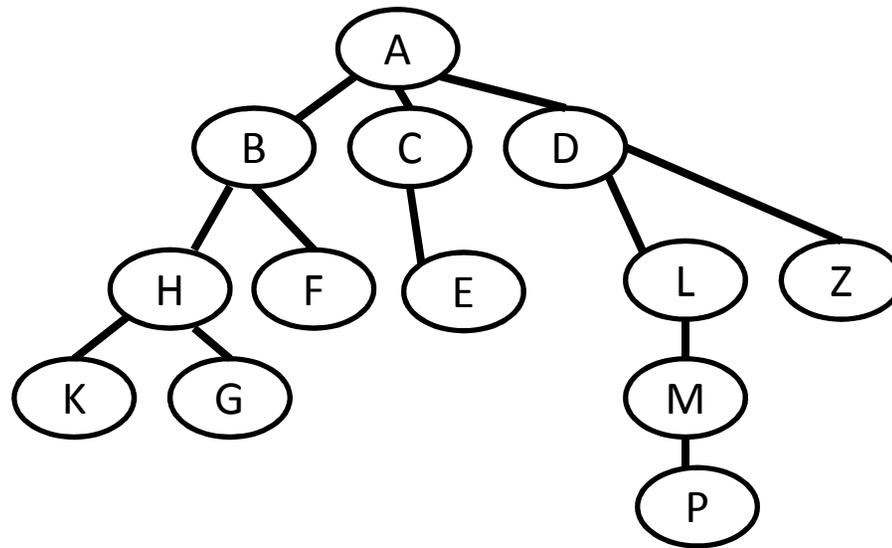
Más relaciones entre nodos

- Ancestro(u,v): Un nodo u es ancestro de un nodo v si $u=v$ o u es un ancestro del padre de v .
Ej: ancestro(C,C), ancestro(A,E), ancestro(D,P)
- Ancestro propio(u,v): u es ancestro propio de v si u es ancestro de v y $u \neq v$.
Ej: acentropropio(A,E).



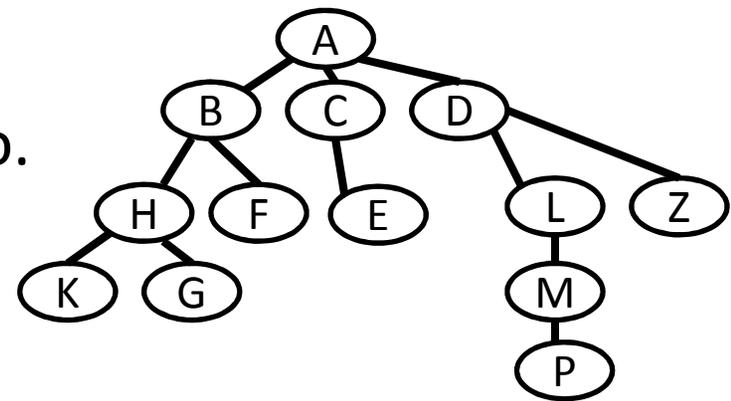
Y más relaciones entre nodos

- Descendiente(u,v): Un nodo u es descendiente de un nodo v si v es un ancestro de u. Ej: desc(C,C), desc(E,A), desc(P,D)
- Descendiente propio(u,v): u es descendiente propio de v si u es descendiente de v y $u \neq v$. Ej: descpropio(E,A).
- Subárbol: El subárbol de T con raíz en el nodo v es el árbol consistiendo de todos los descendientes de v.



Arcos y caminos en árboles

- Arco: Un arco de un árbol T es un par de nodos (u,v) tales que u es el padre de v , o viceversa.
- Ej: (D,L) es un arco y (Z,D) es otro arco.
- Camino: Un camino de T es una secuencia de nodos tales que cualquier par de nodos consecutivos en la secuencia forman un arco.
- Ej: A, B, F es un camino
- Ej: F, B, A, D, L, M es otro camino.



Arcos y caminos en árboles

- Ejemplo: La relación de herencia de clases en Java forman un árbol.
- La raíz, `java.lang.Object` es un ancestro de todas las clases.
- Cada clase `C` es un descendiente de `Object` y `C` es la raíz del subárbol formado por todas las clases descendientes de `C`.

Árboles ordenados

- Un árbol se dice ordenado si existe un orden lineal para los hijos de cada nodo,
- Es decir, se puede identificar el primer hijo, el segundo hijo y así sucesivamente
- Tal orden se visualiza de izquierda a derecha de acuerdo a tal ordenamiento.
- Ejemplo:
 - Los componentes de un libro:
 - El libro es la raíz,
 - parte, capítulo, sección, subsección, subsubsección, son los nodos internos
 - Los párrafos, figuras, tablas son las hojas.

ADT Arbol

- Position:
 - element(): retorna el objeto almacenado en esta posición
 - Tree: Métodos de acceso (reciben y retornan posiciones)
 - root(): Retorna la raíz del árbol, error si el árbol está vacío
 - parent(v): Retorna el padre de v, error si v es la raíz
 - children(v): Retorna una colección iterable conteniendo los hijos del nodo v
- Nota: Si el árbol es ordenado, children los mantiene en orden. Si v es una hoja children(v) es vacía.

ADT Arbol

- Tree: Métodos de consulta
 - isInternal(v): Testea si v es un nodo interno
 - isExternal(v): Testea si v es una hoja
 - isRoot(v): Testea si v es la raíz
- Tree: Métodos genéricos
 - size(): Retorna el número de nodos del árbol
 - isEmpty(): Testea si el árbol tiene o no nodos
 - iterator(): Retorna un iterador con los elementos ubicados en los nodos del árbol
 - positions(): Retorna una colección iterable de los nodos del árbol
 - replace(v,e): Reemplaza con e y retorna el elemento ubicado en v.

ADT Arbol

- Tree: Métodos de modificación (agregados por la cátedra a [GT])
 - createRoot(e): crea un nodo raíz con rótulo “e”
 - addFirstChild(p, e): agrega un primer hijo al nodo “p” con rótulo “e”
 - addLastChild(p, e): agrega un último hijo al nodo “p” con rótulo e
 - addBefore(p, rb, e): Agrega un nodo con rótulo “e” como hijo de un nodo padre “p” dado. El nuevo nodo se agregará delante de otro nodo hermano “rb” también dado.

ADT Arbol

- Tree: Métodos de modificación (agregados por la cátedra a [GT])
 - addAfter (p, lb, e): Agrega un nodo con rótulo “e” como hijo de un nodo padre “p” dado. El nuevo nodo se agregará a continuación de otro nodo “lb”
 - removeExternalNode (p): Elimina la hoja “p”
 - removeInternalNode (p): Elimina el nodo interno “p”. Los hijos del nodo eliminado lo reemplazan en el mismo orden en el que aparecen. La raíz se puede eliminar si tiene un único hijo.
 - removeNode (p): Elimina el nodo “p”.
- Ver interface Tree<E>.

```

public interface Tree<E> extends Iterable<E>
{
    public int size();
    public boolean isEmpty();
    public Iterator<E> iterator();
    public Iterable<Position<E>> positions();
    public E replace(Position<E> v, E e) throws InvalidPositionException;
    public Position<E> root() throws EmptyTreeException;
    public Position<E> parent(Position<E> v)
        throws InvalidPositionException, BoundaryViolationException;
    public Iterable<Position<E>> children(Position<E> v) throws InvalidPositionException;
    public boolean isInternal(Position<E> v) throws InvalidPositionException;
    public boolean isExternal(Position<E> v) throws InvalidPositionException;
    public boolean isRoot(Position<E> v) throws InvalidPositionException;
    public void createRoot(E e) throws InvalidOperationException;
    public Position<E> addFirstChild(Position<E> p, E e) throws InvalidPositionException;
    public Position<E> addLastChild(Position<E> p, E e) throws InvalidPositionException;
    public Position<E> addBefore(Position<E> p, Position<E> rb, E e)
        throws InvalidPositionException;
    public Position<E> addAfter (Position<E> p, Position<E> lb, E e)
        throws InvalidPositionException;
    public void removeExternalNode (Position<E> p) throws InvalidPositionException;
    public void removeInternalNode (Position<E> p) throws InvalidPositionException;
    public void removeNode (Position<E> p) throws InvalidPositionException;
}

```

Ejemplo de carga de un árbol

```
Tree<Character> arbol = new Arbol<Character>(); // Creo un árbol de caracteres
arbol.createRoot( 'A' ); // Agrego la raíz
Position<Character> raiz = arbol.root();
```

// Agrego hijos de A:

```
Position<Character> pB = arbol.addLastChild( raiz, 'B' );
Position<Character> pC = arbol.addLastChild( raiz, 'C' );
Position<Character> pD = arbol.addLastChild( raiz, 'D' );
```

// Agrego hijos de B:

```
Position<Character> pH = arbol.addLastChild( pB, 'H' );
arbol.addLastChild( pB, 'F' ); // Ocurre un "voiding"
```

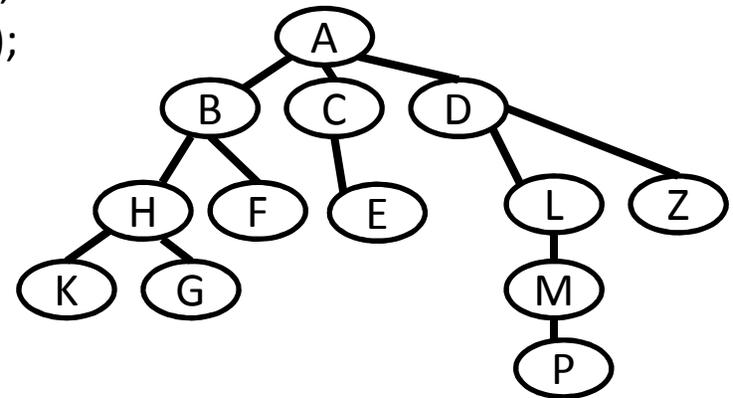
// Agrego los hijos de H:

```
arbol.addFirstChild( pH, 'G' );
arbol.addFirstChild( pH, 'K' );
```

// Agrego hijo de C:

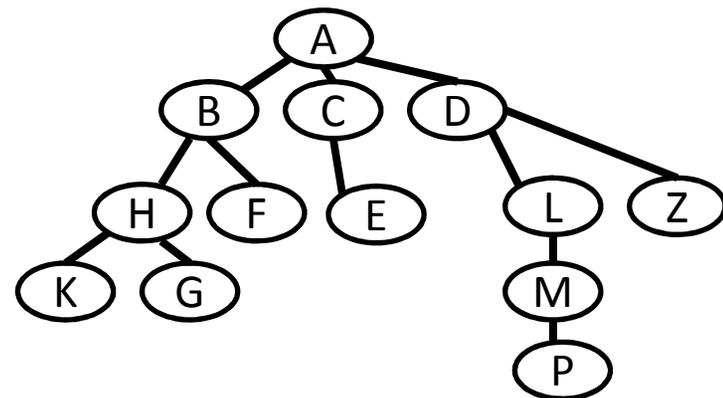
```
arbol.addLastChild( pC, 'E' ); // Completar con descendientes propios de D
```

NOTA: Veremos una implementación donde cada una de estas operaciones se realizan en tiempo constante en la cantidad de nodos del árbol.



Profundidad y altura

- Profundidad de un nodo v en un árbol T: Longitud del camino de la raíz de T al nodo v = cantidad de ancestros propios de v
- Longitud de un camino: Cantidad de arcos del camino
- Altura de un nodo v en un árbol T: Longitud del camino más largo a una hoja en el subárbol con raíz v.
- Altura de un árbol T: Altura del nodo raíz de T.
- Ej: profundidad de A = 0
- Ej: profundidad de E = 2
- Ej: profundidad de P = ?
- Ej: Altura de B = 2
- Ej: Altura de D = ?
- Ej: Altura de G = ?
- Ej: Profundidad de G = ?



Profundidad (depth)

- Profundidad de un nodo v en un árbol T: Longitud del camino de la raíz de T al nodo v = cantidad de ancestros propios de v
- $\text{Profundidad}(T,v) = \text{Longitud del camino de } v \text{ a } T.\text{root}()$
- Algoritmo profundidad(T, v)
 - CB: Si v es la raíz de T, profundidad(T,v) = 0
 - CR: Si v no es la raíz de T,
profundidad(T,v) = 1 + profundidad(T, T.parent(v))
- Implementación Java:

```
public static <E> int profundidad( Tree<E> T, Position<E> v ) {  
    if (T.isRoot(v) )  
        return 0;  
    else  
        return 1 + profundidad( T, T.parent( v ) );  
}
```
- Tiempo de ejecución: es del orden de d_v donde d_v es la profundidad de v en T

Altura (Height) primera solución

- Altura de un nodo v en un árbol T: Longitud del camino más largo a una hoja en el subárbol con raíz v.
- Proposición: La altura de un árbol T no vacío es la máxima profundidad de las hojas de T
- Algoritmo Altura(T)

$h \leftarrow 0$

para cada vértice v en T hacer

si v es una hoja en T entonces

$h \leftarrow \max(h, \text{profundidad}(T, v))$

retornar h

- Implementación Java:

```
public static <E> int altura( Tree<E> T ) {  
    int h = 0;  
    for( Position<E> v : T.positions() )  
        if( T.isExternal(v) ) h = Math.max( h, profundidad( T, v ) );  
    return h; }
```

- Tiempo de ejecución: $T_{\text{altura}}(n) = O(n^2)$

Altura (Height) segunda solución

- Altura de un nodo v en un árbol T: Longitud del camino más largo a una hoja en el subárbol con raíz v.
- Definición recursiva de la altura de v en T (planteo):
 - CB: altura(hoja(n)) = 0
 - CR: altura(nodo(n, t₁, ..., t_k)) = 1 + max(altura(t₁), ..., altura(t_k))
- Algoritmo Altura(T,v)
 - si v es una hoja en T entonces
 - retornar 0
 - sino
 - h ← 0
 - para cada hijo w de v en T
 - h ← max(h, Altura(T, w))
 - retornar 1+h

Altura (Height) segunda solución

- Algoritmo Altura(T,v)

si v es una hoja en T entonces

retornar 0

sino

$h \leftarrow 0$

para cada hijo w de v en T

$h \leftarrow \max(h, \text{Altura}(T, w))$

retornar 1+h

- Implementación Java: Llamar con altura(T, T.root()) para altura de T

```
public static <E> int altura( Tree<E> T, Position<E> v ) {
```

```
    if( T.isExternal(v) ) return 0;
```

```
    else {
```

```
        int h = 0;
```

```
        for( Position<E> w : T.children(v) )
```

```
            h = Math.max( h, altura( T, w ) );
```

```
        return 1+h; } }
```

- Tiempo de ejecución: $T_{\text{altura}}(n) = O(n)$ (demostración en el pizarrón)

Recorridos de árboles

- Un recorrido de un árbol T es una manera sistemática de visitar todos los nodos de T .
- Los recorridos básicos son:
 - Preorden
 - Postorden
 - Inorden (o simétrico)
 - Por niveles

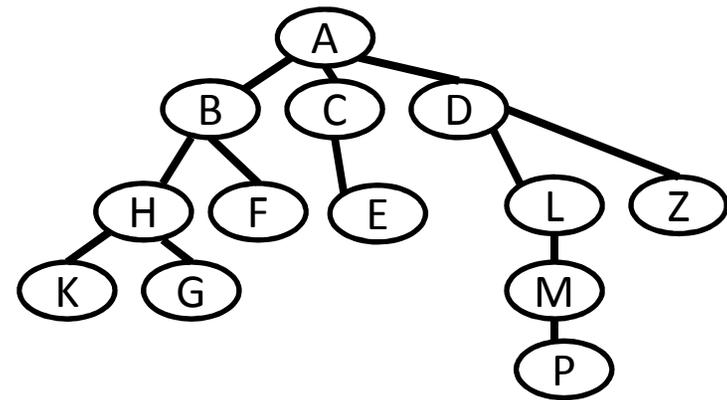
Recorridos de árboles: Preorden

- En el recorrido preorden de un árbol T , la raíz r de T se visita primero y luego se visitan recursivamente cada uno de los subárboles de r .
- Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
- El algoritmo se llama con $\text{preorden}(T, T.\text{root}())$
- Algoritmo $\text{preorden}(T, v)$
 - Visitar(T, v)
 - Para cada hijo w de v en T hacer $\text{preorden}(T, w)$
- Planteo Preorden:
 - CB: $\text{pre}(\text{hoja}(n)) = [n]$
 - CR: $\text{pre}(\text{nodo}(n, [t_1 t_2 \dots t_k])) = [n] + \text{pre}(t_1) + \text{pre}(t_2) + \dots + \text{pre}(t_k)$
- La acción “ $\text{visitar}(T,v)$ ” dependerá del problema.

Recorridos en árboles: preorden

Algoritmo PreordenShell(T)
preorden(T, T.root())

Algoritmo preorden(T, v)
Visitar(T, v)
Para cada hijo w de v en T
hacer
preorden(T, w)



• Planteo Preorden:

CB: $pre(\text{hoja}(n)) = [n]$

CR: $pre(\text{nodo}(n, [t_1 t_2 \dots t_k])) = [n] + pre(t_1) + pre(t_2) + \dots + pre(t_k)$

Ejemplo: El recorrido preorden es:

A - B - H - K - G - F - C - E - D - L - M - P - Z

Recorridos en árboles: preorden

- Aplicación: El recorrido preorden de un libro a partir de su tabla de contenidos examina el libro en forma secuencial.
- Aplicación: Retornar el listado preorden como un string.

```
public static <E> String toStringPreorder( Tree<E> T, Position<E> v ) {  
    String s = v.element().toString();  
    for( Position<E> w : T.children(v) )  
        s += " - " + toStringPreorder( T, w );  
    return s;  
}
```

Nota: Si n es la cantidad de nodos del árbol T , el tiempo de ejecución de preorden es $O(n)$ asumiendo una visita de $O(1)$ (demostración en el pizarrón). Es decir, preorden corre en tiempo lineal en el tamaño del árbol.

Tiempo de ejecución de preorden

- Entrada: un árbol T
- Tamaño de la entrada: n = cantidad de nodos del árbol T
- Tiempo de ejecución: Vemos que el algoritmo pasa una vez por cada nodo i y en el nodo toma un tiempo constante c_2 y luego ejecuta un bucle que realiza h_i iteraciones, con h_i = la cantidad de hijos del nodo i .

$$\begin{aligned} T(n) &= c_1 + \sum_{i=1}^n (c_2 + c_3 h_i) = \\ &= c_1 + c_2 n + c_3 (n-1) = O(n) \end{aligned}$$

Notar que $\sum_{i=1}^n h_i = n - 1$ pues corresponde a la cantidad de arcos del árbol.

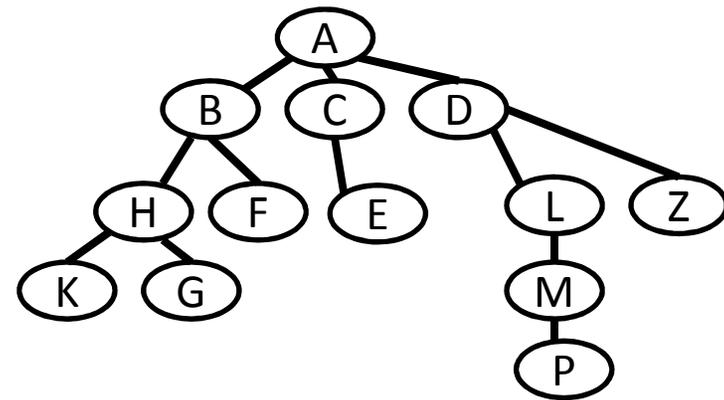
Recorridos de árboles: Postorden

- En el recorrido postorden de un árbol T , la raíz r de T se visita luego de visitar recursivamente cada uno de los subárboles de r .
 - Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
 - El algoritmo se llama con $\text{postorden}(T, T.\text{root}())$
 - Algoritmo $\text{postorden}(T, v)$
 - Para cada hijo w de v en T hacer
 - $\text{postorden}(T, w)$
 - Visitar(T, v)
- Planteo Postorden:
 - CB: $\text{post}(\text{hoja}(n)) = [n]$
 - CR: $\text{post}(\text{nodo}(n, [t_1 t_2 \dots t_k])) = \text{post}(t_1) + \text{post}(t_2) + \dots + \text{post}(t_k) + [n]$
- La acción “visitar(T,v)” dependerá del problema.

Recorridos en árboles: postorden

Algoritmo PostordenShell(T)
postorden(T, T.root())

Algoritmo postorden(T, v)
Para cada hijo w de v en T hacer
postorden(T, w)
Visitar(T, v)



Ejemplo: El recorrido postorden es:
K, G, H, F, B, E, C, P, M, L, Z, D, A

- Planteo Postorden:
CB: $\text{post}(\text{hoja}(n)) = [n]$
CR: $\text{post}(\text{nodo}(n, [t_1 t_2 \dots t_k])) =$
 $\text{post}(t_1) + \text{post}(t_2) + \dots + \text{post}(t_k) + [n]$

Tiempo de ejecución: Si el árbol T tiene n nodos entonces:

$T_{\text{postorden}}(n) = O(n)$ asumiendo visita de $O(1)$

Recorridos de árboles: Inorden

- En el recorrido inorden (o simétrico) de un árbol T con raíz r, primero se recorre recursivamente el primer hijo de la raíz r, luego se visita la raíz y luego se visita recursivamente al resto de los hijos de r.
- Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
- El algoritmo se llama con `inorden(T, T.root())`
- Algoritmo `inorden(T, v)`

Si v es hoja en T entonces

Visitar(T, v)

Sino

w ← primer hijo de v en T

`inorden(T, w)`

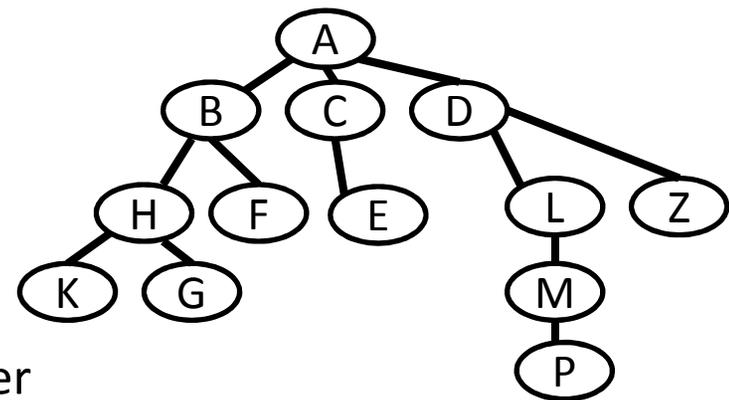
Visitar(T, v)

mientras w tiene hermano en T hacer

w ← hermano de w en T

`inorden(T, w)`

- El recorrido inorden para el ejemplo es: **K H G B F A E C P M L D Z**



- Planteo In order:

CB: $in(\text{hoja}(n)) = [n]$

CR: $in(\text{nodo}(n, [t_1 t_2 \dots t_k])) = in(t_1) + [n] + in(t_2) + \dots + in(t_k)$

Resumen de los recorridos

- Planteo Preorden:

CB: $\text{pre}(\text{hoja}(n)) = [n]$

CR: $\text{pre}(\text{nodo}(n, [t_1 \ t_2 \ \dots \ t_k])) =$
 $[n] + \text{pre}(t_1) + \text{pre}(t_2) + \dots + \text{pre}(t_k)$

- Planteo Postorden:

CB: $\text{post}(\text{hoja}(n)) = [n]$

CR: $\text{post}(\text{nodo}(n, [t_1 \ t_2 \ \dots \ t_k])) =$
 $\text{post}(t_1) + \text{post}(t_2) + \dots + \text{post}(t_k) + [n]$

- Planteo In order:

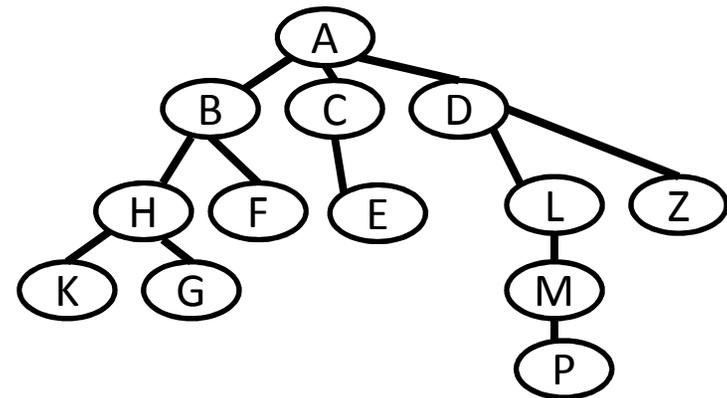
CB: $\text{in}(\text{hoja}(n)) = [n]$

CR: $\text{in}(\text{nodo}(n, [t_1 \ t_2 \ \dots \ t_k])) =$
 $\text{in}(t_1) + [n] + \text{in}(t_2) + \dots + \text{in}(t_k)$

Recorridos de árboles: por niveles

- Nivel: Subconjunto de nodos que tienen la misma profundidad
- Recorrido por niveles (level numbering): Visita todos los nodos con profundidad p antes de recorrer todos los nodos con profundidad $p+1$.
- Ej: Recorrido por niveles:

A
B C D
H F E L Z
K G M
P



Recorridos de árboles: por niveles (primera aproximación)

Algoritmo niveles(T)

Cola \leftarrow new Cola()

Cola.enqueue(T.root())

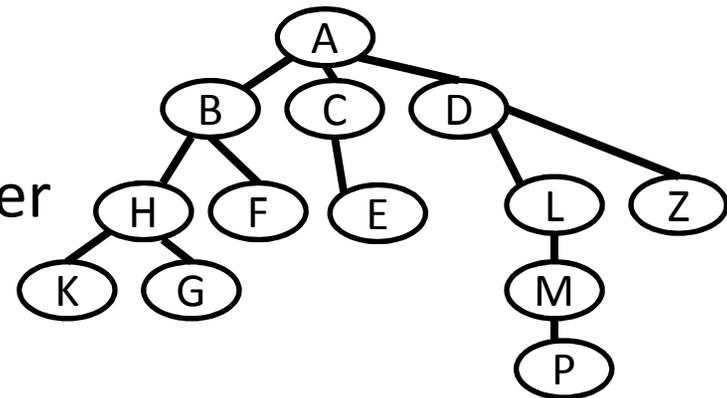
Mientras not cola.isEmpty()

 v \leftarrow cola.dequeue()

 mostrar v // visita de v

 para cada hijo w de v en T hacer

 cola.enqueue(w)



Genera el listado A B C D H F E L Z K G M P

$T_{\text{niveles}}(n) = O(n)$ (demo en siguiente diapositiva)

Tiempo de ejecución de listado por niveles

- Entrada: un árbol T
- Tamaño de la entrada: n = cantidad de nodos del árbol T
- Tiempo de ejecución:

Sea h_i la cantidad de hijos del nodo i

$$\begin{aligned} T(n) &= c_1 + \sum_{i=1}^n (c_2 + c_3 h_i) = \\ &= c_1 + c_2 n + c_3 (n-1) = O(n) \end{aligned}$$

Recordar que la suma de la cantidad de hijos de todos los nodos es $n-1$, porque es igual a la cantidad de padres en el árbol y todos los nodos tienen exactamente un padre menos la raíz.

Recorridos por niveles mostrando dónde termina cada nivel

Algoritmo niveles(T)

Cola \leftarrow new Cola()

Cola.enqueue(T.root())

Cola.enqueue(null) // Esto me sirve para detectar fin de línea

Mientras not cola.isEmpty()

 v \leftarrow cola.dequeue()

 si v \neq null entonces

 mostrar v // visita de v

 para cada hijo w de v en T hacer

 cola.enqueue(w)

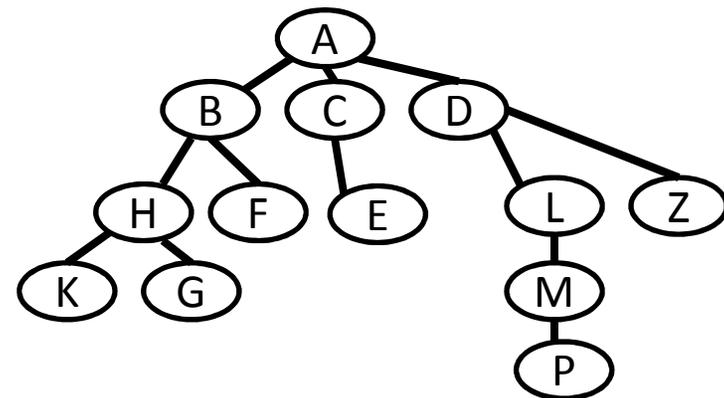
 sino

 imprimir fin de línea

 si not cola.isEmpty() entonces

 cola.enqueue(null)

Nota: $T_{\text{niveles}}(n) = O(n)$ Al tener en cuenta los null, se agregan a lo sumo n iteraciones, con lo que no cambia el orden.



A
B C D
H F E L Z
K G M
P

Bibliografía

- Goodrich & Tamassia, capítulo 7.