Tema II.2 Árboles generales

jose.velez@urjc.es
angel.sanchez@urjc.es
mariateresa.gonzalezdelena@urjc.es

abraham.duarte@urjc.es
raul.cabido@urjc.es

Descripción de la asignatura

- Tema 1: Introducción — Bloque 1
- **Tema 2: Árboles generales** — Bloque 2
- Tema 3: Mapas y Diccionarios — Bloque 2
- Tema 4: Mapas y Diccionarios Ordenados — Bloque 2
- Tema 5: Grafos — Bloque 2
- Tema 6: EEDD en memoria secundaria — Bloque 3

Resumen

- Introducción a los árboles
  - Árboles n-arios
  - Árboles binarios

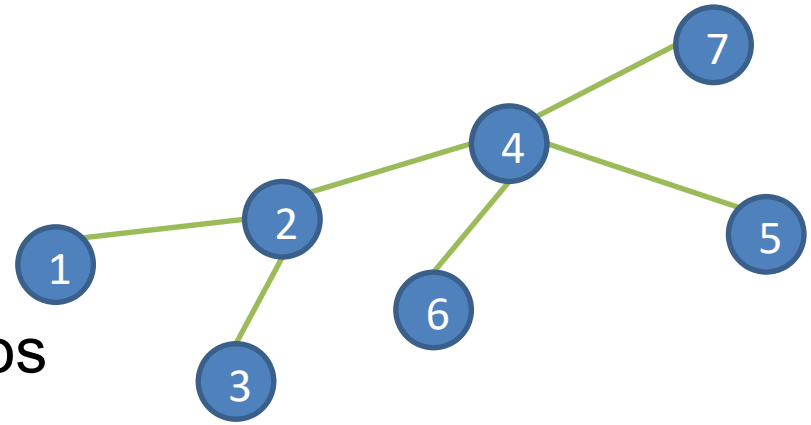
Objetivos

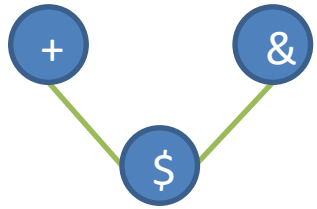
- Dado un problema, **identificar** la estructura de datos arborescente más adecuada
- **Similitudes y diferencias** entre los distintos árboles
- **Implementación** de las operaciones asociadas a cada tipo de árbol

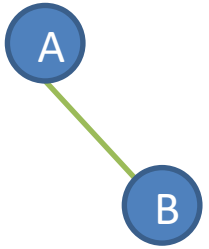
TAD Árbol: terminología

TAD Árbol:Definición de árbol

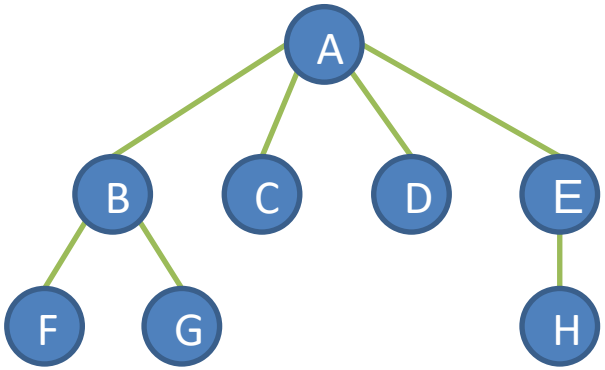
- Modelo abstracto de una estructura jerárquica
- Un árbol está formado por nodos con una relación padre-hijo
- Aplicaciones
  - Flujos en organizaciones
  - Sistemas de ficheros
  - Entornos de programación
  - etc.



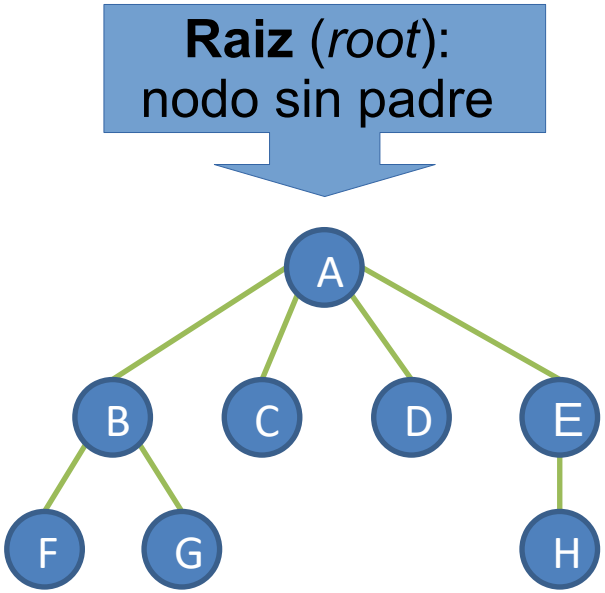




TAD Árbol:Terminología en árboles

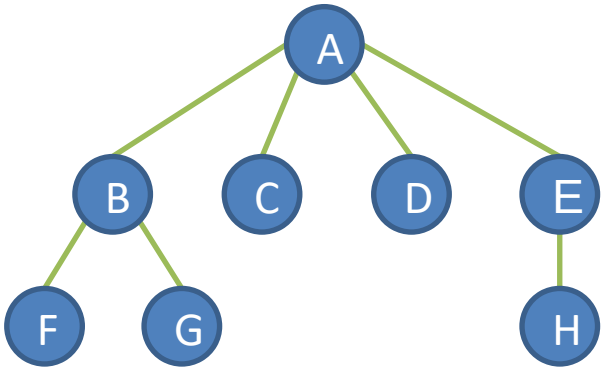


TAD Árbol:Terminología en árboles



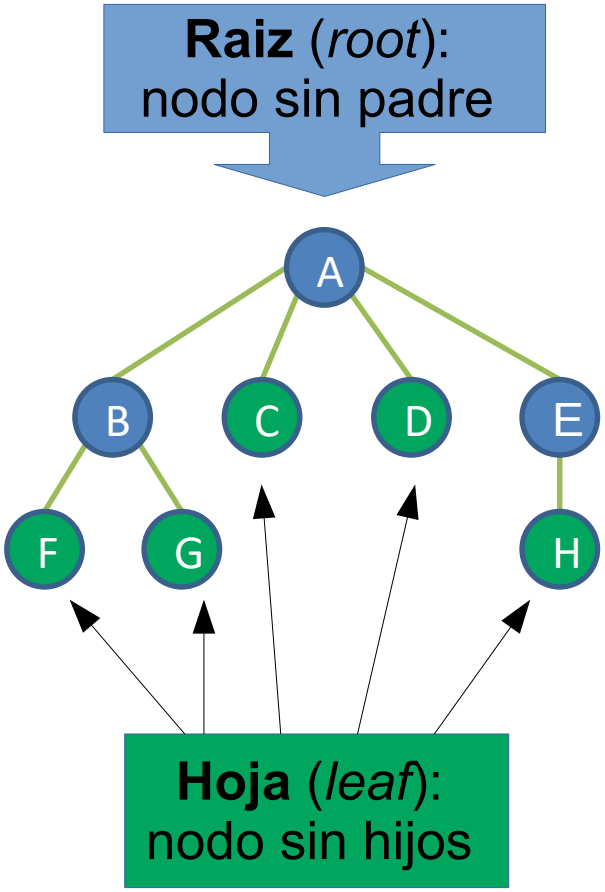
TAD Árbol:Terminología en árboles

Raiz (*root*):
nodo sin padre

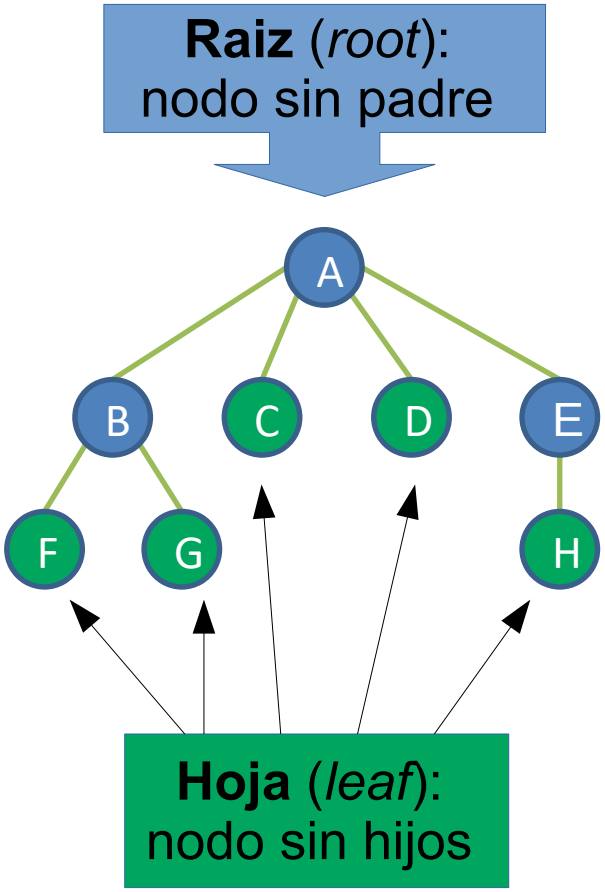


Hoja (*leaf*):
nodo sin hijos

TAD Árbol:Terminología en árboles

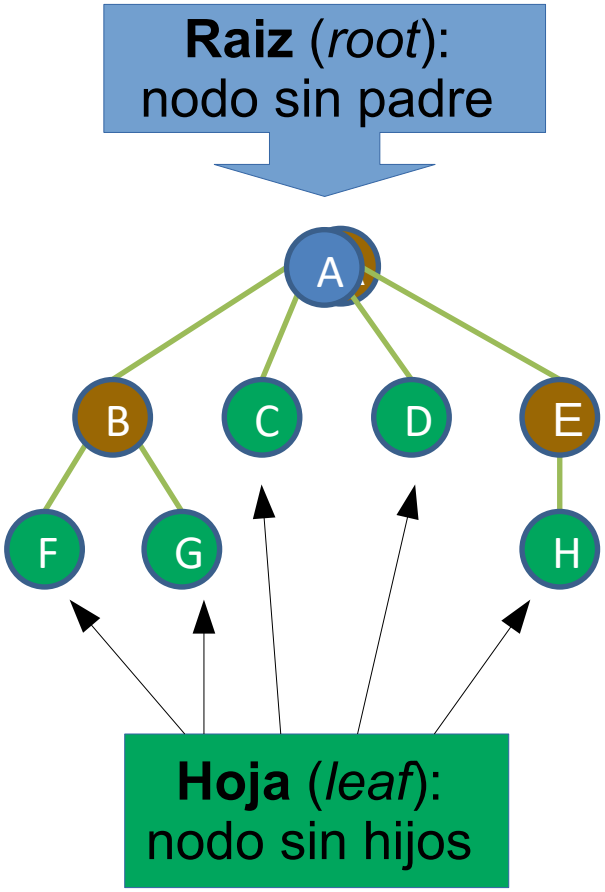


TAD Árbol:Terminología en árboles



Nodo interno:
tiene al menos un hijo

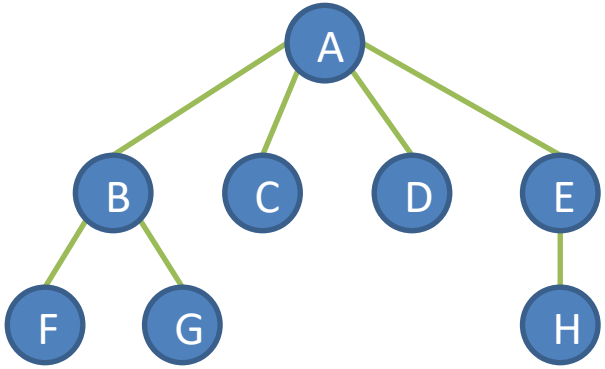
TAD Árbol:Terminología en árboles



Nodo interno:
tiene al menos un hijo

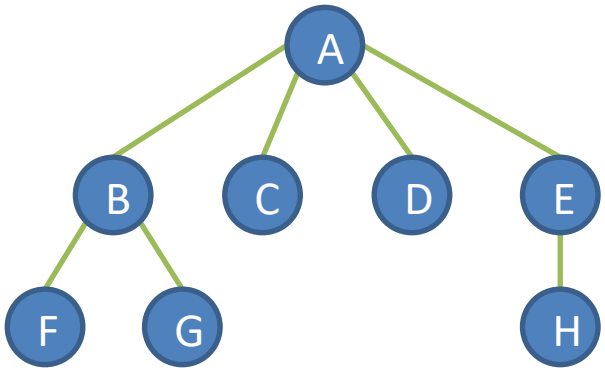
TAD Árbol:Terminología en árboles

- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …



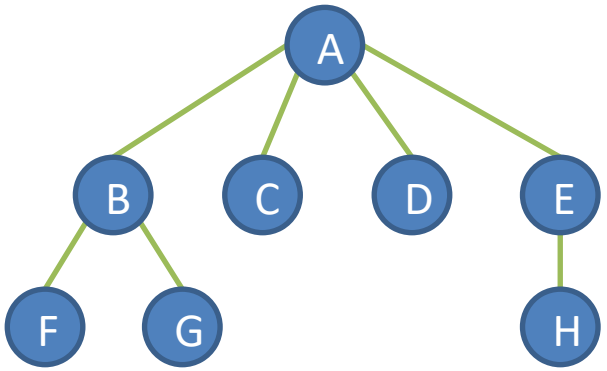
TAD Árbol:Terminología en árboles

- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …



TAD Árbol:Terminología en árboles

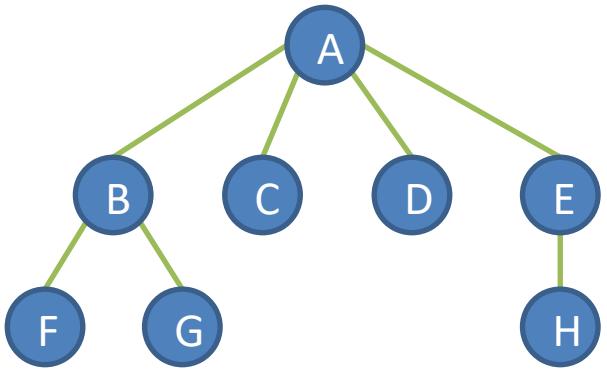
- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …



Ancestros de G: B y A

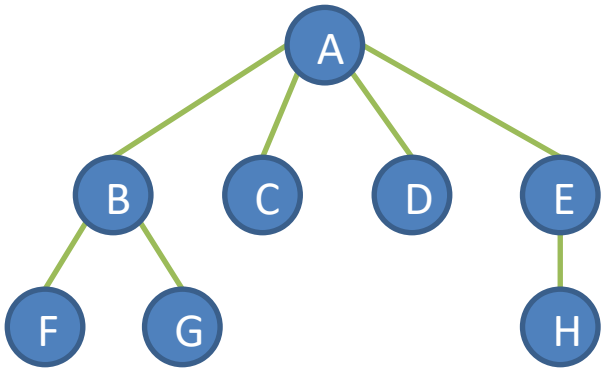
TAD Árbol:Terminología en árboles

- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Descendiente** de un nodo: hijo, nieto, bisnieto, …



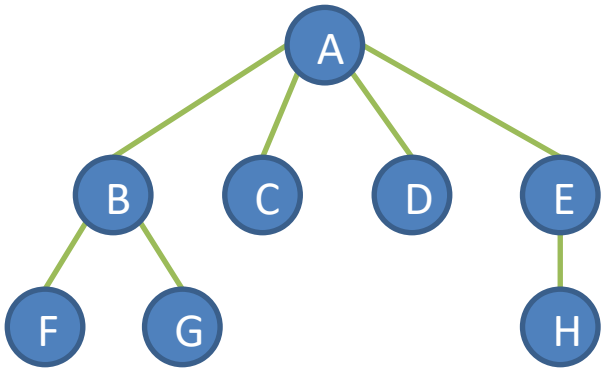
TAD Árbol:Terminología en árboles

- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Descendiente** de un nodo: hijo, nieto, bisnieto, …



TAD Árbol:Terminología en árboles

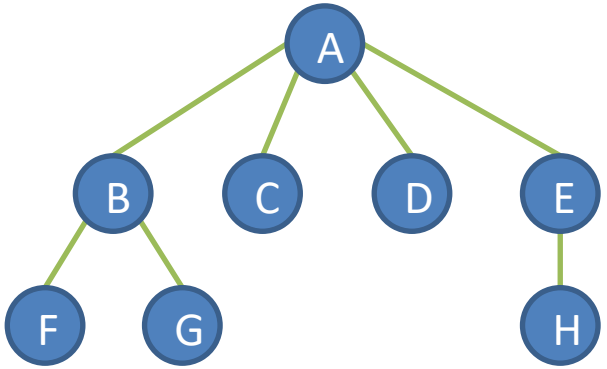
- **Raiz** (*root*): nodo sin padre (**A**)
- **Nodo interno**: tiene al menos un hijo (**A**,**B**,**E**)
- **Hoja** (*leaf*): nodo sin hijos (**F, G**, **C**, **D**, **H**)
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Descendiente** de un nodo: hijo, nieto, bisnieto, …



Descendientes de A: B, C, D, E, F, G y H

TAD Árbol:Terminología en árboles

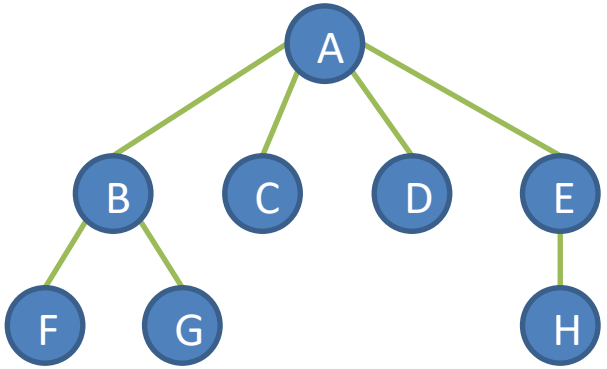
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Profundidad** de un nodo: número de ancestros



TAD Árbol:Terminología en árboles

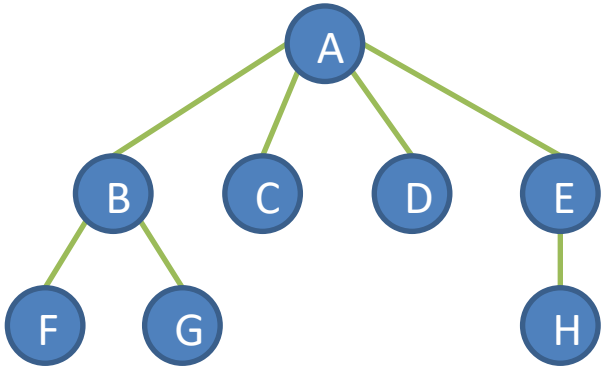
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Profundidad** de un nodo: número de ancestros

Profundidad de G: 2



TAD Árbol:Terminología en árboles

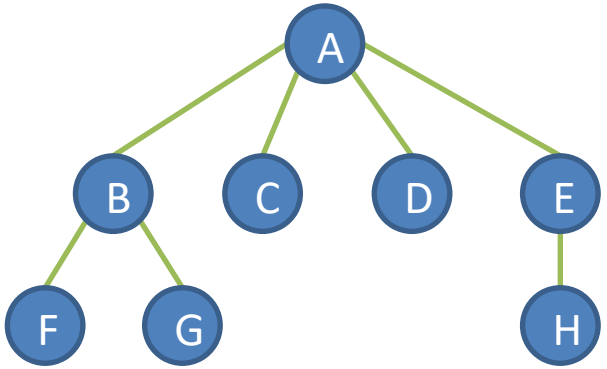
- **Ancestros**: padres, abuelos, bisabuelos, …
- **Profundidad** de un nodo: número de ancestros
- **Altura del árbol**: máxima profundidad (2)
  - Un árbol con un solo nodo tiene altura cero
  - Un árbol sin nodos es un árbol vacío
  - No tiene sentido hablar de la altura de un árbol vacío



Altura del árbol: 2

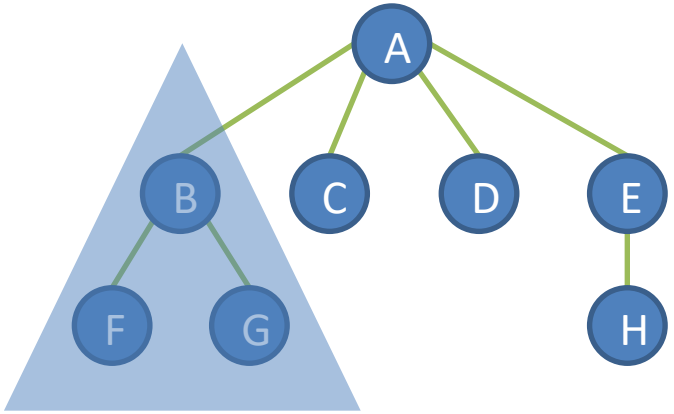
TAD Árbol:Terminología en árboles

- **Descendiente** de un nodo: hijo, nieto, bisnieto, …
- **Sub-árbol**: árbol que consiste en un nodo y sus descendientes



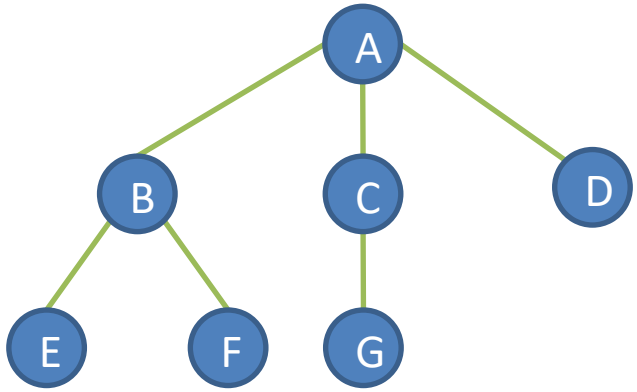
TAD Árbol:Terminología en árboles

- **Descendiente** de un nodo: hijo, nieto, bisnieto, …
- **Sub-árbol**: árbol que consiste en un nodo y sus descendientes



TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - Dos nodos con el mismo padre se llaman hermanos

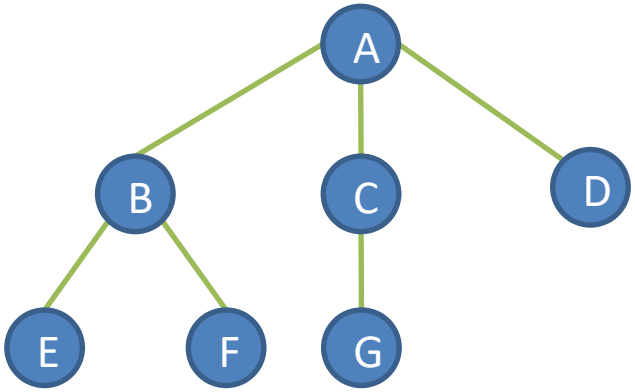


TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - Dos nodos con el mismo padre se llaman hermanos

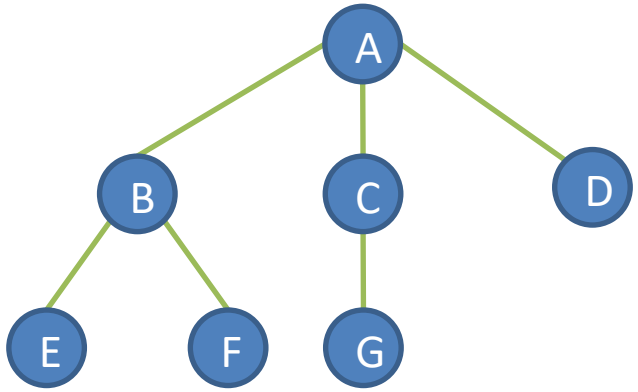
Los nodos E y F son hermanos

El nodo G no tiene hermanos



TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - El número de hijos de un nodo define su grado

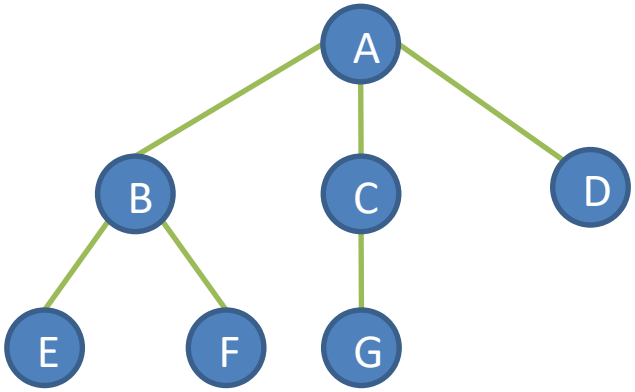


TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - El número de hijos de un nodo define su grado

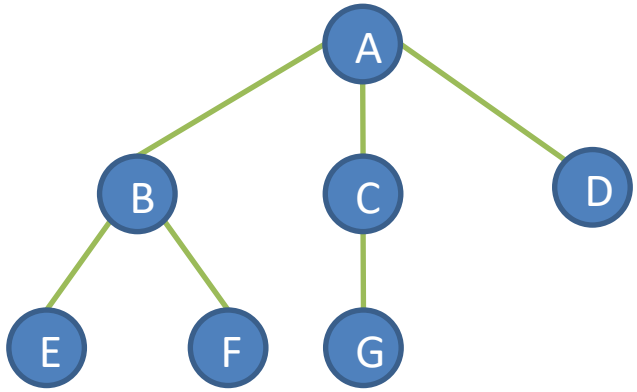
Grado de E = 0

Grado de A = 3



TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - Grado de un árbol: número de hijos que tiene el nodo con más hijos del árbol



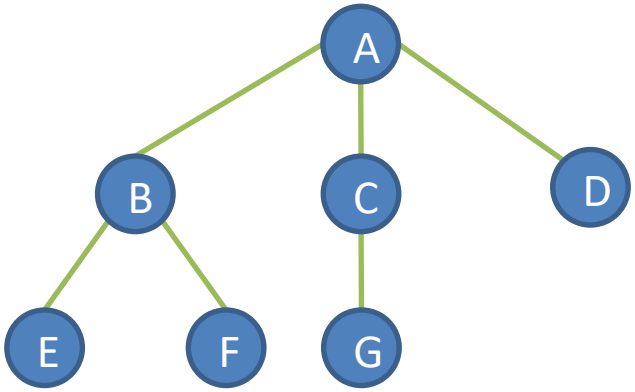
TAD Árbol:Terminología en árboles

- Definición de un árbol con raíz
  - Grado de un árbol: número de hijos que tiene el nodo con más hijos del árbol

Grado de A = 3
Grado de B = 2
Grado de C = 1
Grado de D = 0
Grado de E = 0
Grado de F = 0
Grado de G = 0

max.

Grado del árbol: 3



TAD Árbol: operaciones

TAD Árbol

- Métodos básicos
  - `boolean isEmpty()`
  - `Iterator iterator()`
- Métodos de consulta
  - `boolean isLeaf(Node)`
  - `boolean isInternal(Node)`
  - `boolean isRoot(Node)`
- Métodos de acceso
  - `Node root()`
  - `Node parent(Node)`
  - `Iterable children(Node)`
- Métodos de actualización
  - `addRoot(E)`
  - `add(Node,E)`
  - `E replace(Node, E)`
  - `Tree cut(Node)`
  - `attach(Node,Tree)`

Se podrían definir métodos privados adicionales dentro del TAD

TAD Árbol: interfaz Java

Localización por contenido

```
public interface Tree<E> {

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Replaces the element e by v. */
  public E replace(E e1, E e2);

  /** Returns the root of the tree. */
  public E root();

  /** Returns the parent of a given node. */
  public E parent(E v);
...
```

Admitimos elementos repetidos?
Comportamiento `replace`?
Comportamiento `parent`?

TAD Árbol: interfaz Java

Localización mediante referencias a nodo

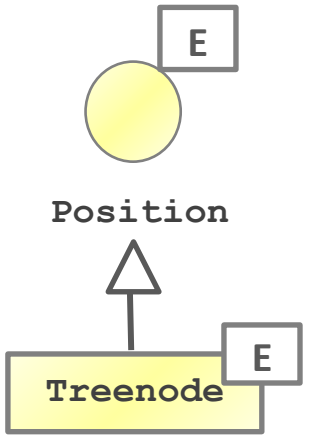
```
public interface Tree<E> {

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Replaces the TreeNode e by v. */
  public TreeNode<E> replace(TreeNode<E> e1, E e2);

  /** Returns the root of the tree. */
  public TreeNode<E> root();

  /** Returns the parent of a given node. */
  public TreeNode<E> parent(TreeNode<E> e);
   ...
```

Desencapulación!!!!
Es posible modificar las invariantes de la estructura de datos

TAD Árbol: interfaz Java

Localización mediante interfaz `Position`

```
public interface Tree<E> implements Iterable<Position<E>>{

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Returns an iterator of the elements stored in the tree. */
  public Iterator<Position<E>> iterator();

  /** Replaces the element stored at a given node.
      Problems: Invalid Position */
  public E replace(Position<E> v, E e);

  /** Add element E as child of node parent*/
  public Position<E> addRoot(E value);

  /** Returns the root of the tree.
      Problems: Boundary Violation */
  public Position<E> root();
...
```



TAD Árbol: interfaz Java

```
    /** Returns the parent of a given node.
        Problems: Invalid Position and Boundary Violation */
    public Position<E> parent(Position<E> v);

    /** Returns an iterable collection of the children of a given node.
        Problems: Invalid Position */
    public Iterable<Position<E>> children(Position<E> v);

    /** Returns whether a given node is a leaf or not.
        Problems: Invalid Position */
    public boolean isLeaf(Position<E> v) ;

    /** Returns whether a given node is the root of the tree.
        Problems: Invalid Position */
    public boolean isRoot(Position<E> v);
}
```

Árbol n-ario

Implementación de árboles n-arios

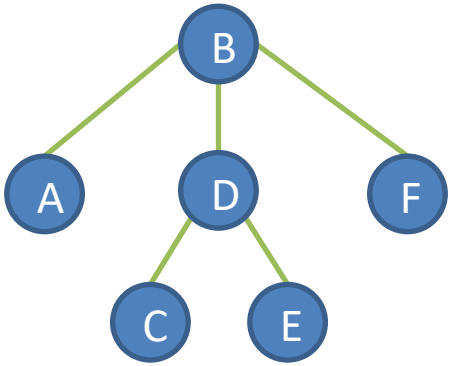
Interface

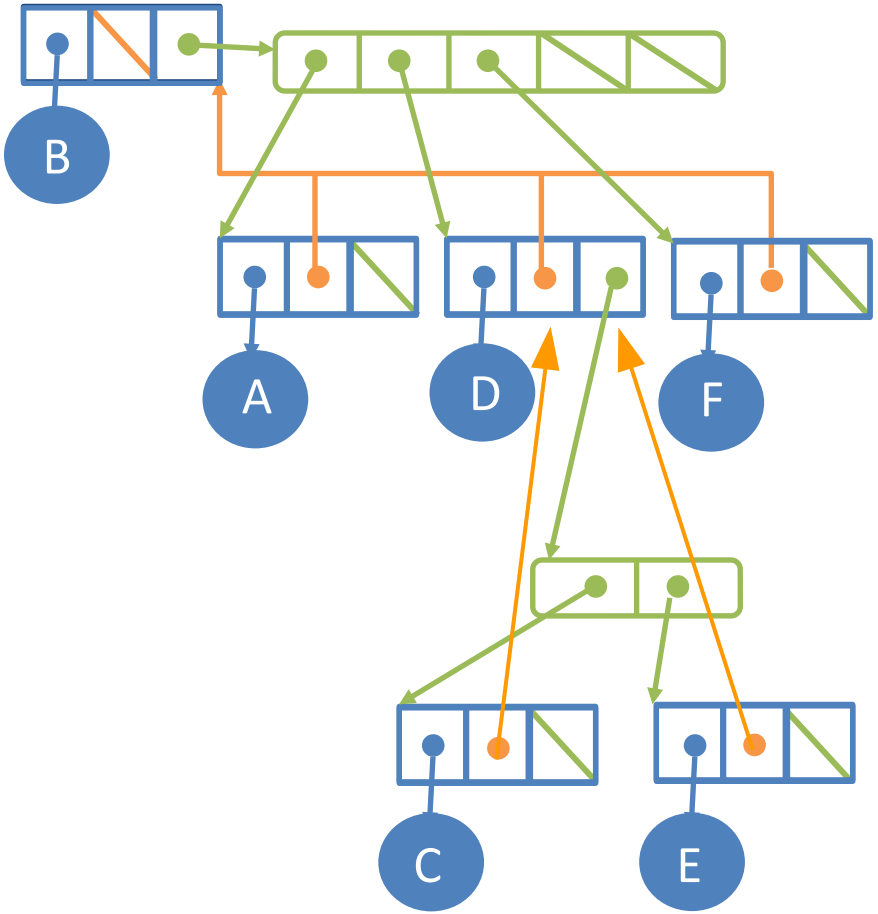
```
public interface NAryTree<E> implements Tree<E>{
  /** Adds element e as last child of node p. Returns the new node.
      Problems: Invalid Position */
  public Position<E> add(Position<E> p, E e);

  /** Adds element e as n child n of node p. Returns the new node.
      Problems: Invalid Position or invalid index */
  public Position<E> add(Position<E> p, final int n, E e);

  /** Removes the subree that begins in p and returns it as a new tree.
      Problems: Invalid Position */
  public NAryTree<E> subTree(Position<E> p);

  /** Attachs the tree t as child of p and clears t. Returns this.
      Problems: Invalid Position */
  public NAryTree<E> attach(Position<E> p, NAryTree<E> t);

  /** Attachs the tree t as n child of p and clears t. Returns this.
      Problems: Invalid Position or invalid index */
  public NAryTree<E> attach(Position<E> p, final int n, NAryTree<E> t);
}
```

Implementación de árboles n-arios

Linked Tree

- Cada nodo contiene:
  - Referencia al elemento
  - Referencia al padre
  - Secuencia de nodos hijo

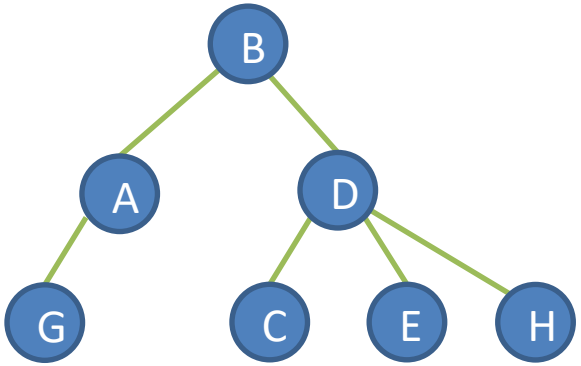


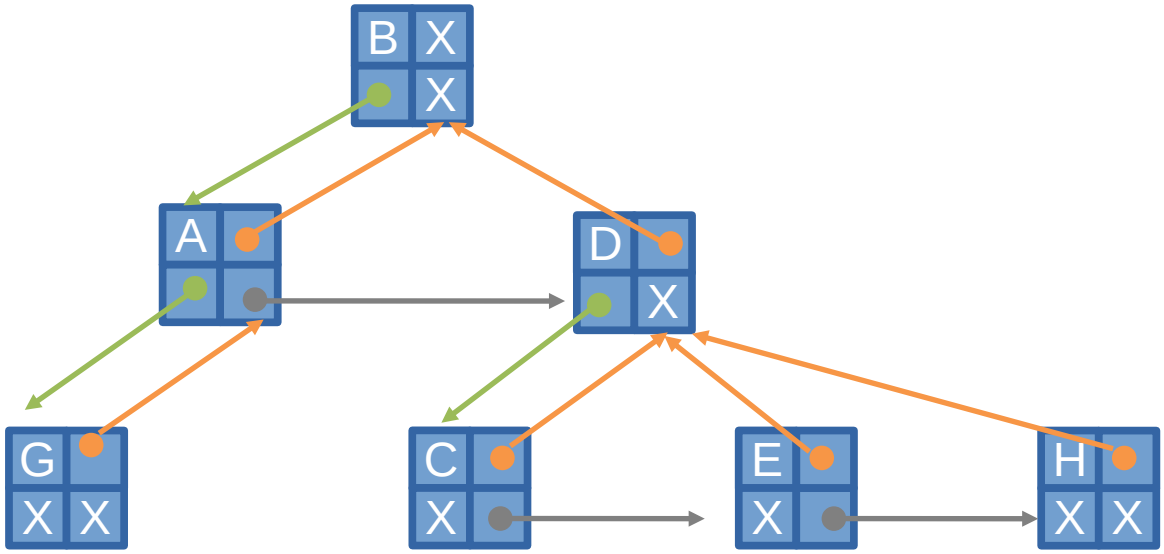


Implementación de árboles n-arios

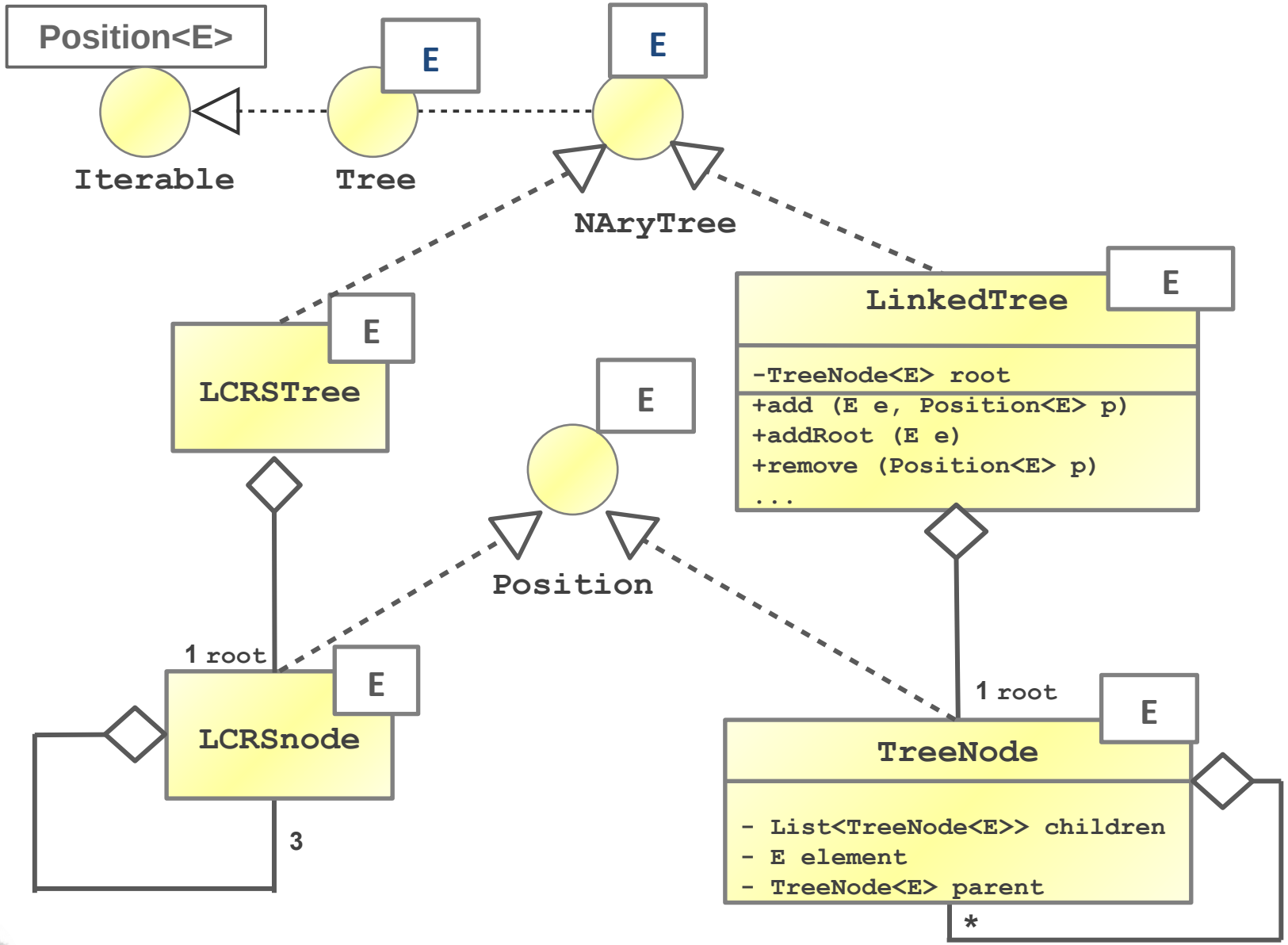
Left-Child, Right Sibling Tree (LCRSTree)

- Cada nodo contiene:
  - Referencia al elemento
  - Referencia al padre
  - Referencia primer hijo
  - Referencia siguiente hermano





Implementación de árboles n-arios

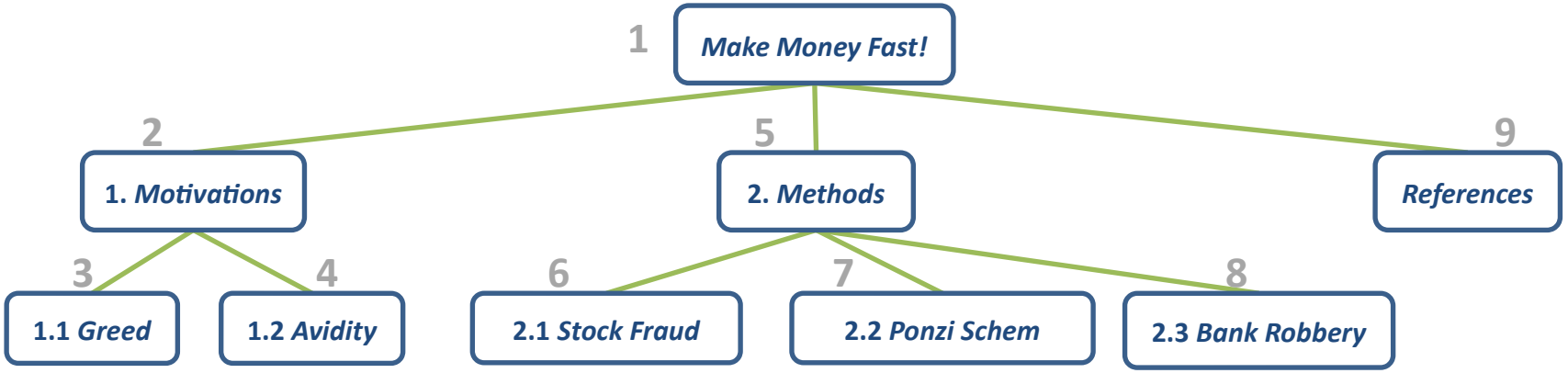


Recorridos

Recorridos de árboles. Preorden

- Recorre todos los nodos del árbol en un orden sistemático
- **Preorden**: un nodo es visitado antes que sus descendientes
- Aplicación. Imprimir un índice

```
Algorithm preOrder(v)
  visit(v)
  for each child w of v
    preOrder(w)
```

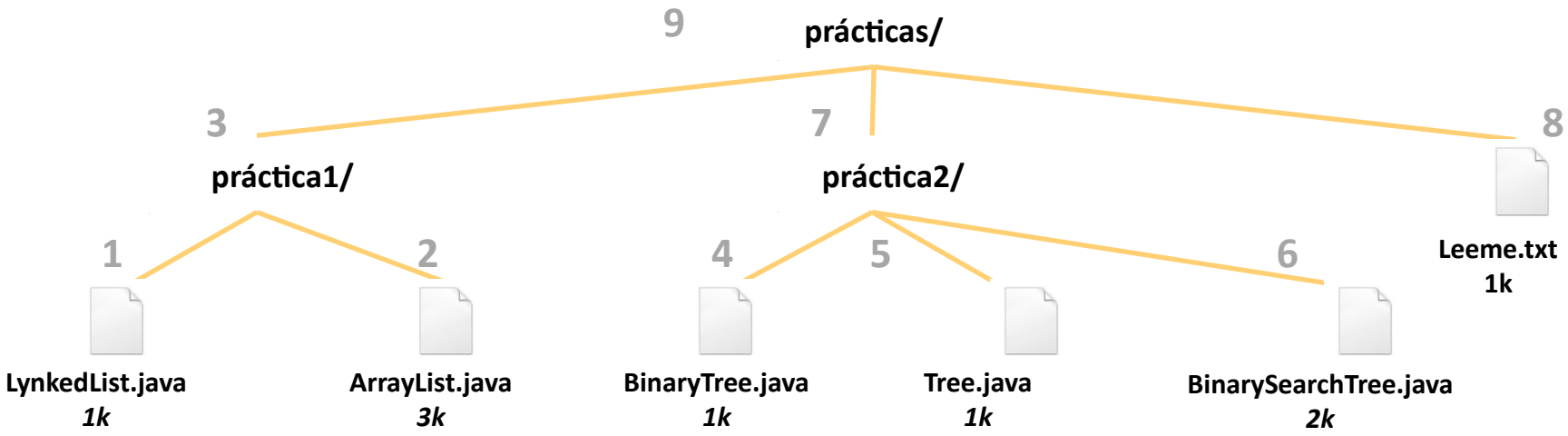


Recorridos de árboles. Postorden

- **Postorden**: un nodo es visitado después de sus descendientes

```
Algorithm postOrder(v)
 for each child w of v
    postOrder(w)
 visit(v)
```

- Aplicación. Calcular el espacio en memoria que ocupan una serie de ficheros dentro de un directorio con subdirectorios



Árboles binarios

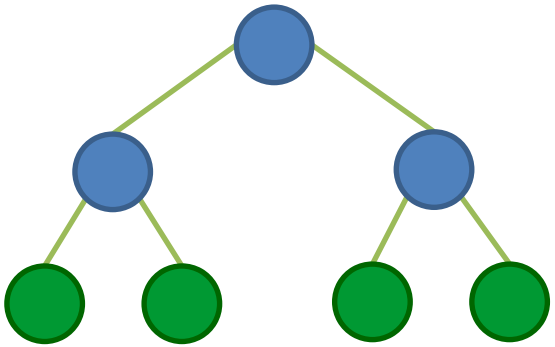
Árboles binarios

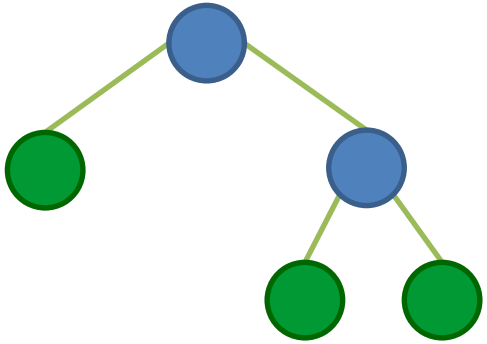
- Árbol *n*-ario tal que:
  - Todo nodo tiene máximo dos hijos
  - Cada nodo hijo se clasifica como h. izquierdo o h. derecho
    - Un h. izquierdo precede a un derecho en el orden de los hijos de un nodo

- $n < 2^{(h+1)} \rightarrow$ El número de nodos crece exponencialmente con la altura.

- Subárbol derecho/subárbol izquierdo

- Árbol binario completo
  - Todos los nodos internos tienen exactamente dos hijos



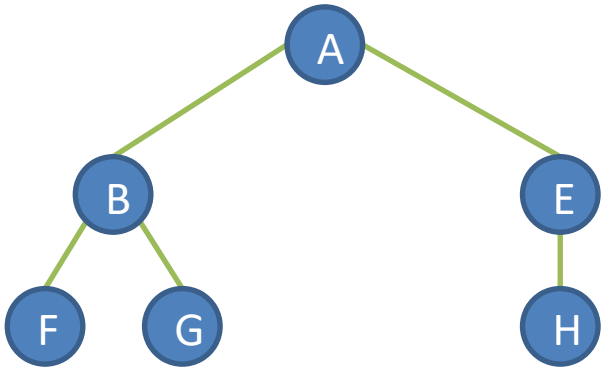


Árboles binarios

Un **árbol binario** se define recursivamente como una estructura definida sobre un conjunto finito de nodos que:

- No contiene nodos
- Está compuesto de 3 conjuntos de nodos disjuntos: un nodo raíz, un árbol binario llamado sub-árbol izquierdo, y otro árbol binario llamado sub-árbol derecho

Un **árbol binario no es sólo un árbol** con 2 hijos como máximo por nodo, ya que si solo tiene un nodo hijo importa saber si es izquierdo o derecho



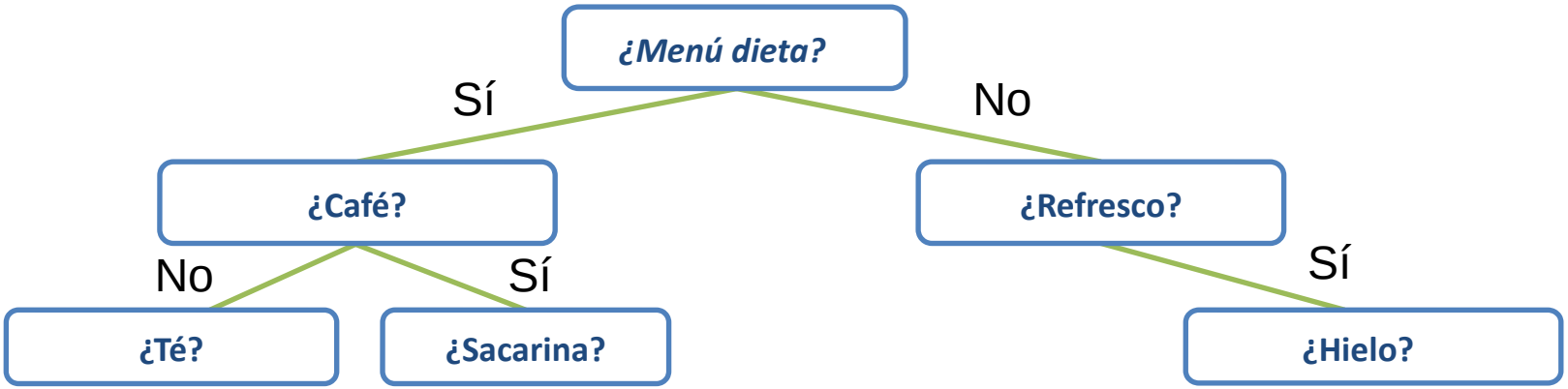
Ejemplos de árboles binarios

Árbol de decisión

- Expresan un resultado en base a diferentes preguntas-respuestas previas

- AB asociado con un proceso de decisión aritmética
  - Nodos internos: preguntas con respuestas si/no
  - Nodos hoja: decisiones

- Ejemplo: decisión para cenar



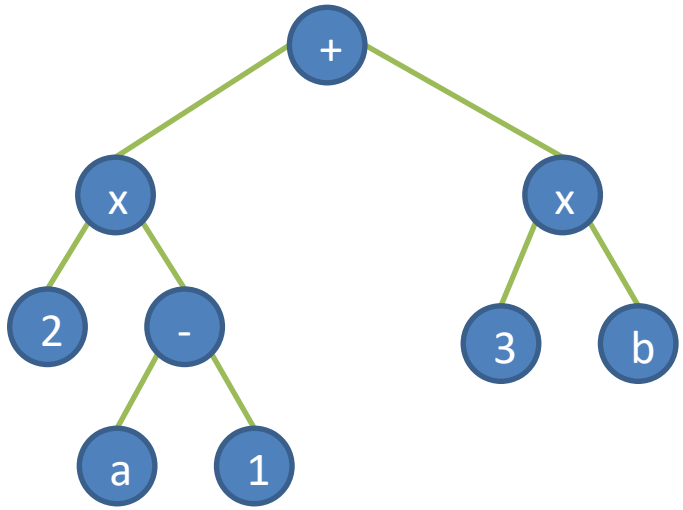
Ejemplos de árboles binarios

Árbol de expresiones aritméticas

- AB asociado con una expresión aritmética
  - Nodos internos: operadores
  - Nodos hoja: operandos

- Ejemplo: AB para la expresión

$$(2 \times (a - 1) + (3 \times b))$$



Arboles binarios completos (propiedades)

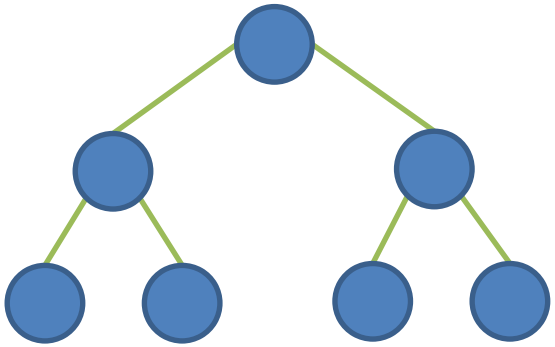
Notación

n : número de nodos

e : número de nodos hoja

i : número de nodos internos

h : altura



Propiedades

e = i + 1 → Internos y hojas solo se diferencian en uno

n = 2e – 1 → Hojas y total de nodos se diferencian en uno

h < n → La altura es menor que el numero de nodos

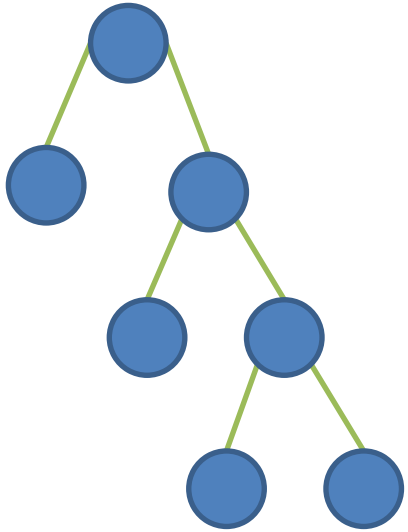
h ≤ i-1 → Altura menor o igual al número de nodos internos menos uno

i = (n-1)/2 → Numero de internos es la mitad del número de nodos menos 1

$h \geq \log_2 e$ → La altura crece con el logaritmo de los nodos hoja

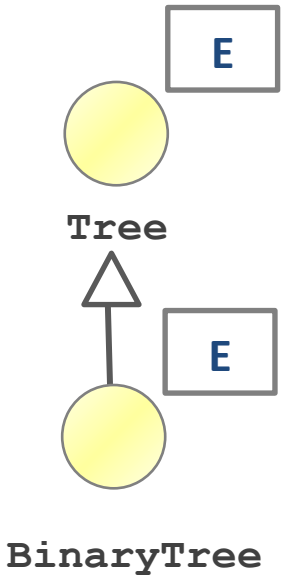
$h \geq \log_2 n$ → La altura crece con el logaritmo del número de nodos

En un árbol no completo la altura puede crecer más deprisa que el número de nodos, pero en una árbol binario completo esto no es posible.



TAD Árbol Binario

- La interfaz `BinaryTree` extiende la interfaz `Tree`
  - Hereda las declaraciones de los métodos definidos en la interfaz `Tree`
- Métodos adicionales
  - `Position<E> left(Position<E>)`
  - `Position<E> right(Position<E>)`
  - `boolean hasLeft(Position<E>)`
  - `boolean hasRight(Position<E>)`



TAD Árbol Binario: interfaz Java

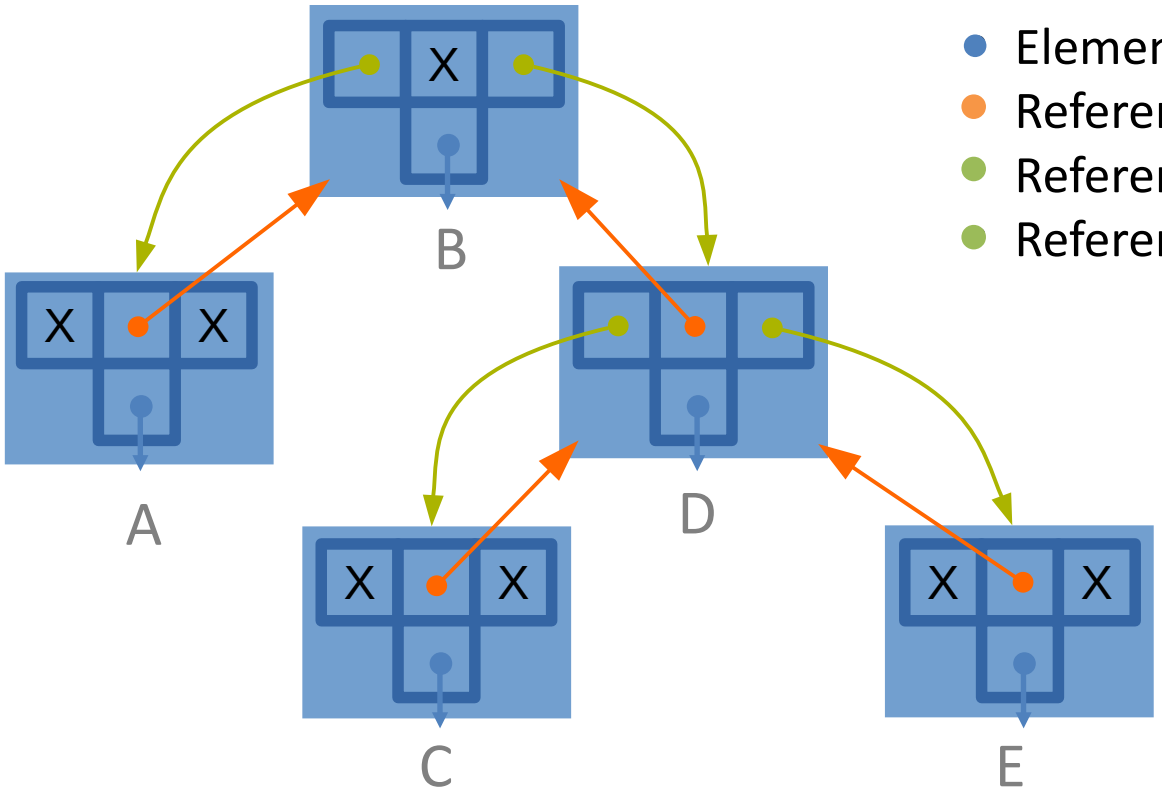
```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the left child of a node.
      Problems: Invalid position, Boundary violation */
  public Position<E> left(Position<E> v)

  /** Returns the right child of a node.
      Problems: Invalid position, Boundary violation */
  public Position<E> right(Position<E> v)

  /** Returns whether a node has a left child.
      Problems: Invalid position */
  public boolean hasLeft(Position<E> v)

  /** Returns whether a node has a right child. */
  public boolean hasRight(Position<E> v)
```

TAD Árbol Binario: interfaz Java

```
    /** Insert value as right child of a given parent node.
        Problems: Invalid Position */
    public void insertRight(Position<E> parent, E value);

    /** Insert value as left child of a given parent node.
        Problems: Invalid Position */
    public void insertLeft(Position<E> parent, E value);

    /** Attach the tree t in position p as right child. Clear t
        Problems: Invalid Position */
    public BinaryTree <E> attachRight(Position<E> p, BinaryTree <E> t);

    /** Attach the tree t in position p as left child. Clear t
        Problems: Invalid Position */
    public BinaryTree <E> attachLeft(Position<E> p, BinaryTree <E> t);

    /** Cuts the subtree begining in p and returns it.
        Problems: Invalid Position */
    public BinaryTree <E> subTree(Position<E> p);
}
```

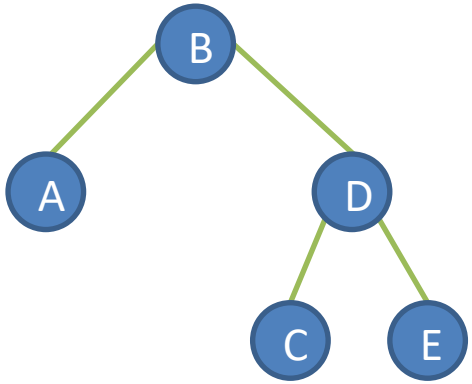
Implementación de árboles binarios (dinámico)

LinkedBinaryTree

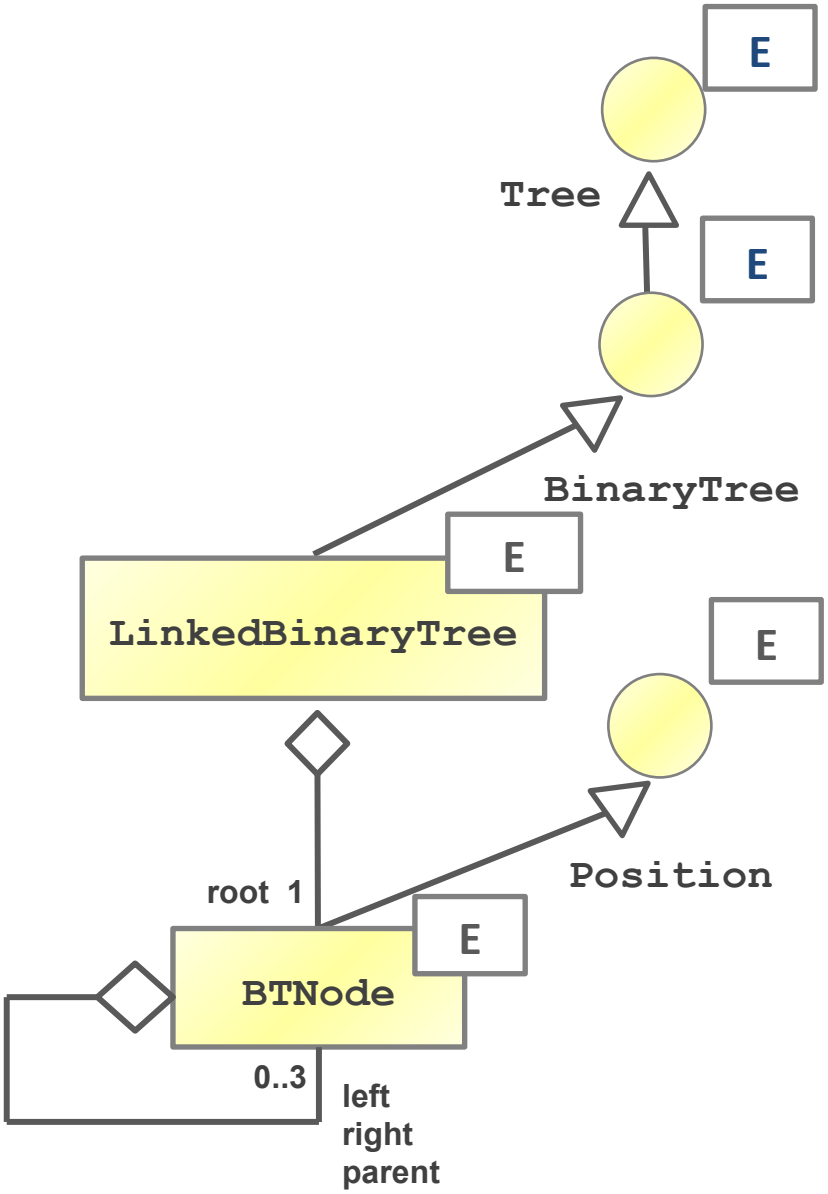


Cada nodo contiene:

- Elemento
- Referencia al padre
- Referencias al hijo izquierdo
- Referencias al hijo derecho



Implementación de árboles binarios (dinámico)

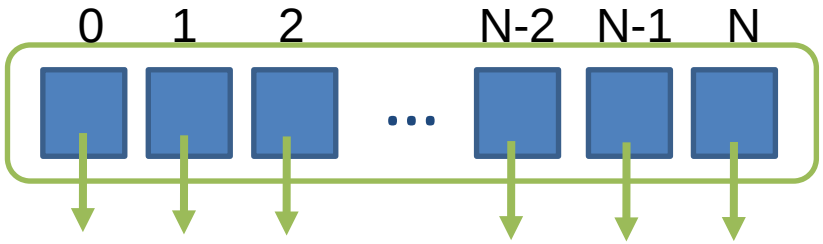


¿Complejidad operaciones? (`LinkedBinaryTree`)

Operación	Complejidad
`isEmpty`	O(1)
`iterator, positions`	O(1)
`replace`	O(1)
`root, parent, children, left, right, sibbling`	O(1)
`insertLeft, insertRight, hasLeft, hasRight, isInternal, isExternal, isRoot`	O(1)
`attach, subTree`	O(1)

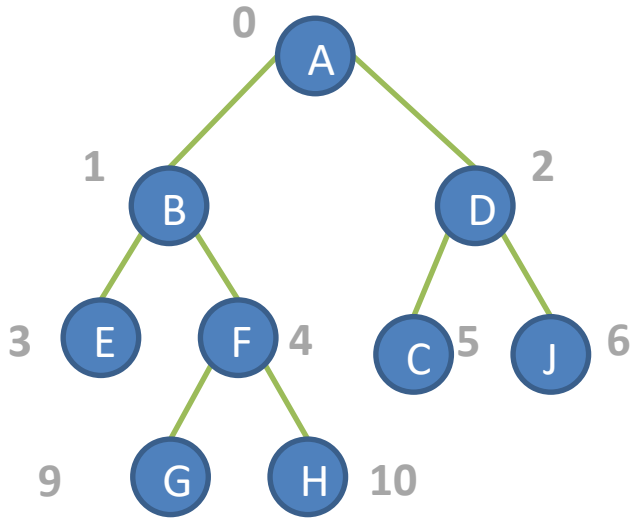
Implementación de árboles binarios (estático)

ArrayBinaryTree



La posición del nodo (*rank*) se define como:

- *rank(root)* = 0
- si *node* es hijo izquierdo de *parent(node)*
  rank(node) = 2 x rank(parent(node))+1
- si *node* es hijo derecho de parent(node)
  rank(node) = 2 rank(parent(node))+2*



¿Complejidad operaciones? (`ArrayBinaryTree`)

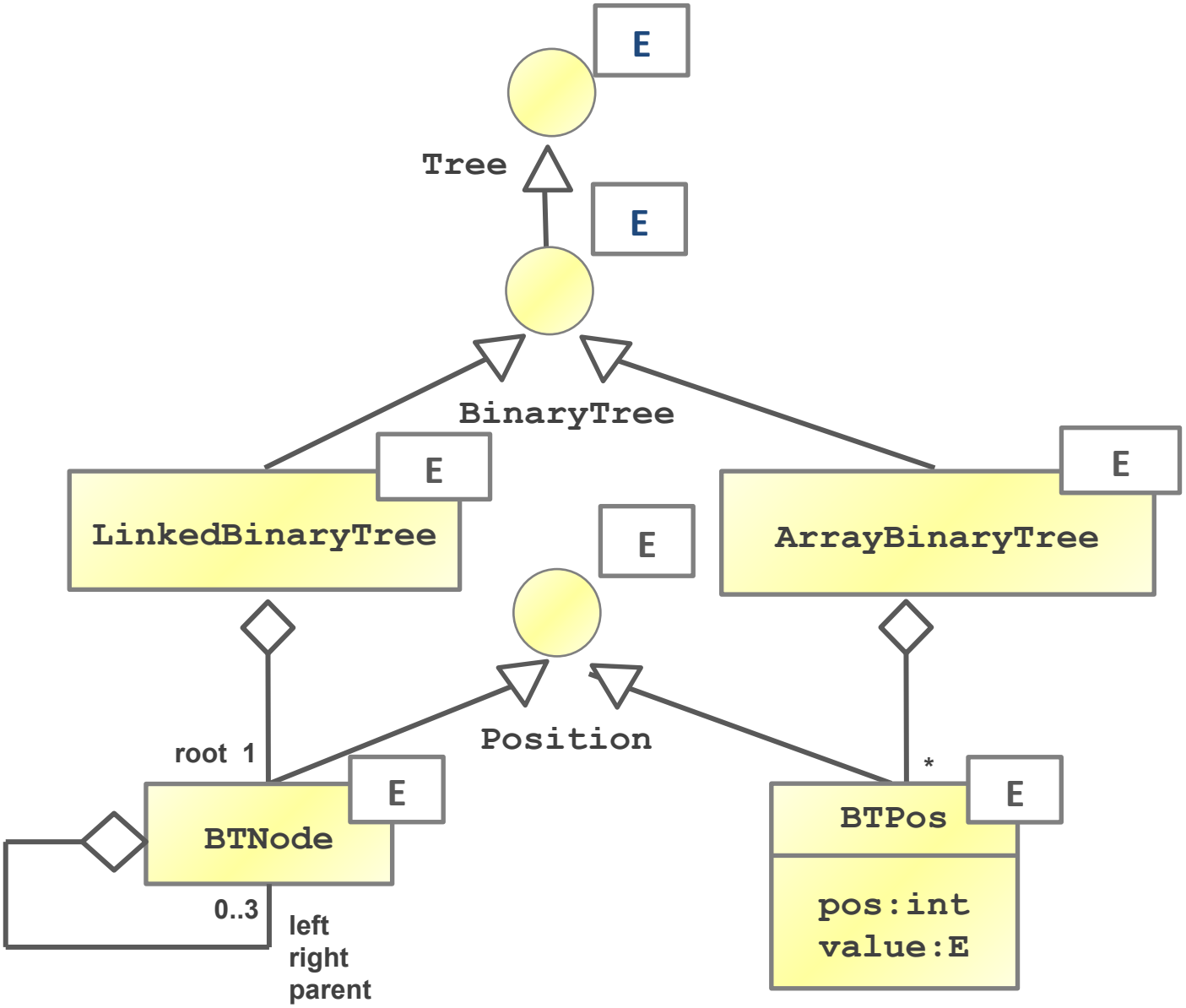
Operación	Complejidad
`isEmpty`	O(1)
`iterator, positions`	O(1)
`replace`	O(1)
`root, parent, children, left, right, sibbling`	O(1)
`insertLeft, insertRight, hasLeft, hasRight, isInternal, isExternal, isRoot`	O(1)
`attach, subTree`	O(n)

Implementación de árboles binarios

ArrayBinaryTree vs LinkedBinaryTree

- ArrayBinaryTree es más eficiente en ocupación de memoria cuando el árbol está cerca de ser completo.
- Las operaciones de subTree y attach tienen mayor complejidad en ArrayBinaryTree que en LInkedBinaryTree.

Diagrama de clases (árboles binarios)



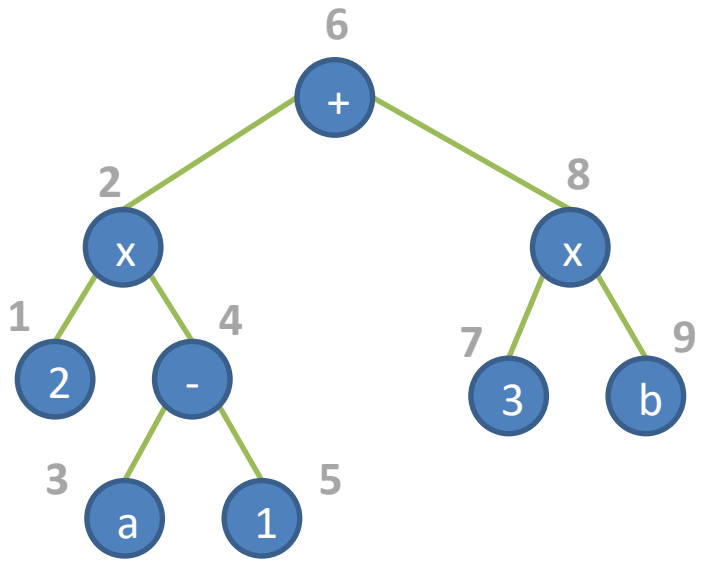
Recorridos de árboles binarios. Inorden

- **Inorden**: un nodo es visitado después de su hijo izquierdo pero antes que su hijo derecho

- Aplicación. Dibuja el contenido de un árbol binario

$$(2 \times (a - 1) + (3 \times b))$$

```
Algorithm inOrder(v)
  if hasLeft(v) then
    inOrder(left(v))
  visit(v)
  if hasRight(v) then
    inOrder(right(v))
```



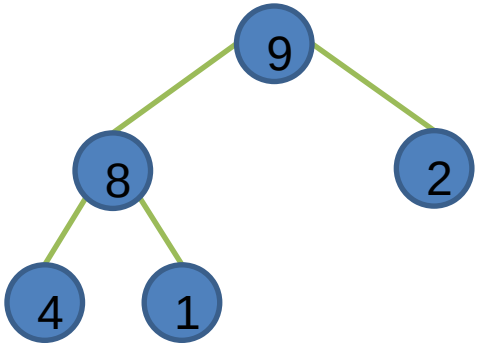
Montículo

Montículo

Un montículo (heap) es un árbol binario casi completo cuyos nodos cumplen una relación de orden según la cual el padre siempre es mayor que los hijos.

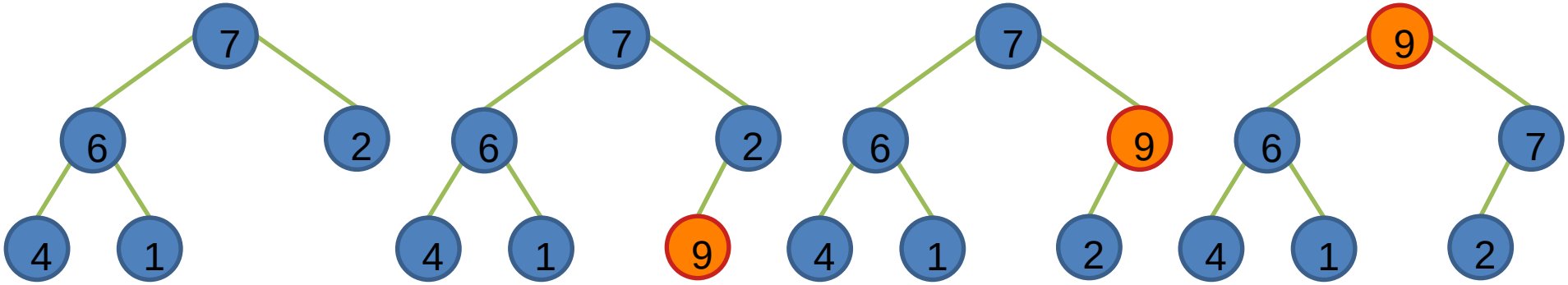
Usualmente se implementa con un array.

Los montículos se utilizan para implementar las colas de prioridad.



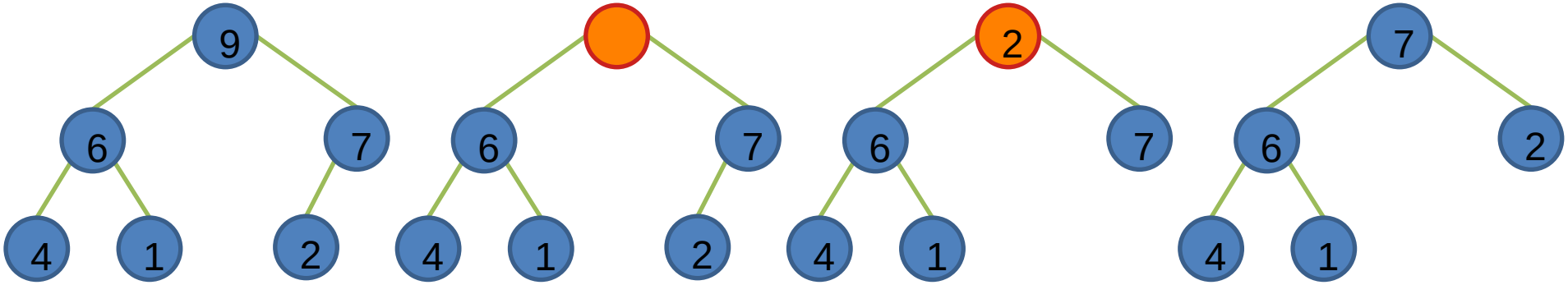
Montículo - Inserción

Siempre se inserta en la última posición libre del array y se intercambia el valor de nodo con el padre si es mayor hasta que deja de serlo o alcanza la raíz.



Montículo - Borrado

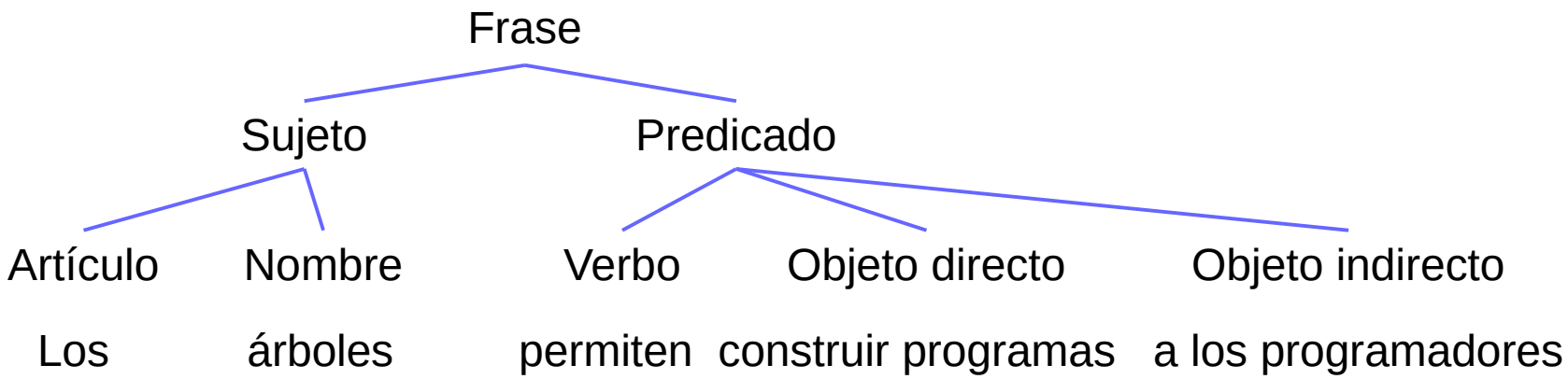
Al eliminar la raíz, su hueco se rellena con el último valor insertado. Luego este valor desciende en intercambios sucesivos con el mayor de sus hijos.



Aplicaciones que usan el TAD Árbol

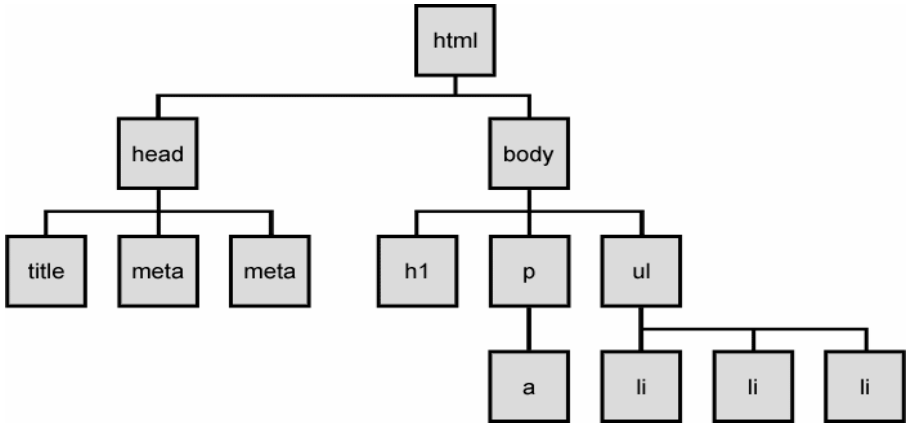
Análisis de expresiones

Para el análisis o la evaluación de expresiones se puede utilizar un árbol.



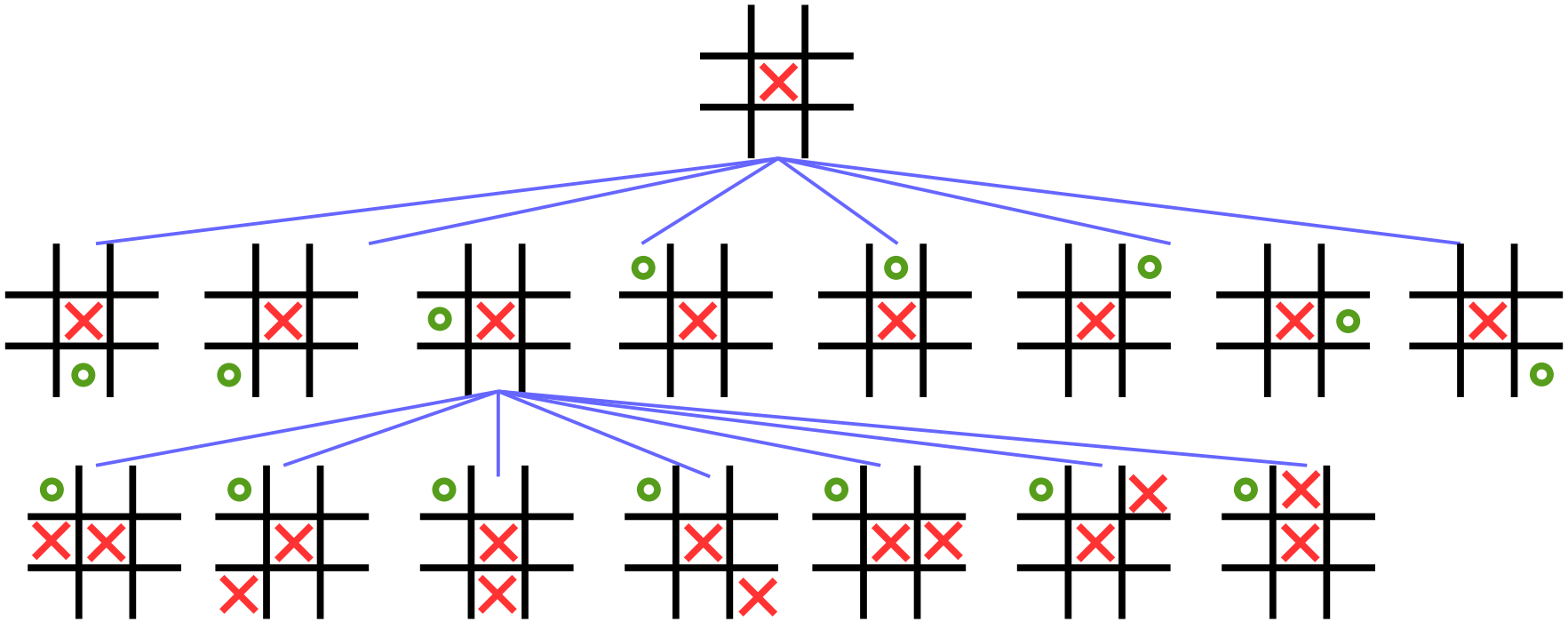
HTML

Para el análisis o la evaluación de páginas HTML o de documentos XML.



Exploración de soluciones

Para explorar las soluciones de un problema y elegir entre múltiples posibilidades siempre que se desee guardar el árbol para explorar posteriormente diferentes alternativas.



Interfaz gráfica de usuario

Para representar GUIs.