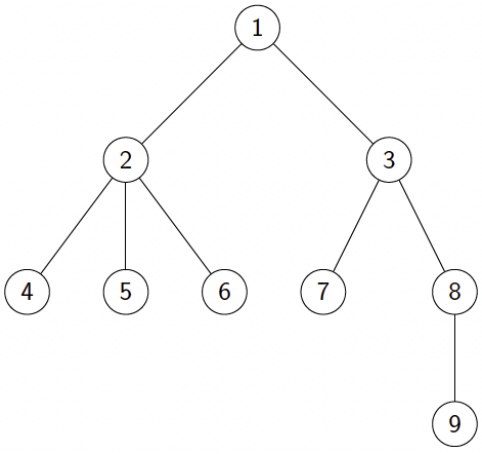
TEMA 7: ÁRBOLES GENERALES Y BINARIOS

Árboles Generales:
El objetivo de los árboles es organizar los datos de forma jerárquica utilizando otros TADs para ello como las colas con prioridad, maps ordenados, … Un árbol general es o bien vacío, o bien tiene dos componentes: (1) un nodo raíz que contiene un elemento y (2) un conjunto de cero o más (sub)árboles hijos.

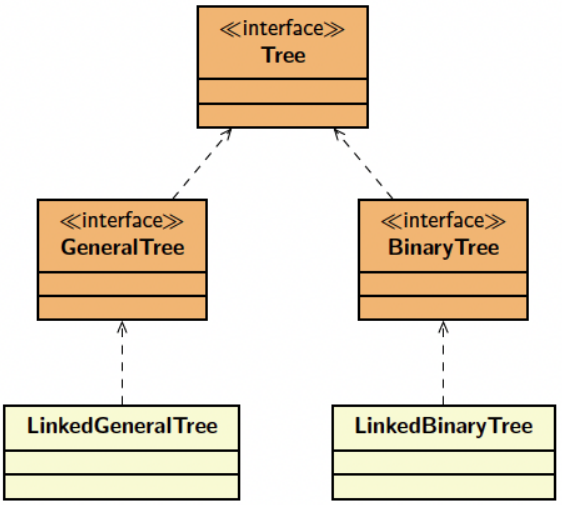
Terminología:
- Raíz (root): nodo sin padre
- Nodo interno (internal node): nodo con al menos un hijo
- Nodo externo/hoja (external/leaf node): nodo sin hijos
- Subárbol (subtree): árbol formado por el nodo considerado como raíz junto con todos sus descendientes
- Ancestro de un nodo: un nodo 'w' es ancestro de 'v' sii 'w' es 'v' o el padre de 'v' o ancestro del padre de 'v'
- Descendiente de un nodo (inverso a ancestro): 'v' es descendiente de 'w' sii 'w' es ancestro de 'v'
- Hermano (sibling) de un nodo: nodo con el mismo padre
- Arista (edge) de un árbol: par de nodos en relación padre-hijo o hija-padre
- Grado (degree) de un nodo: número de hijos del nodo
- Grado (degree) de un árbol: el máximo de los grados de todos los nodos
- Camino (path) de un árbol: secuencia de nodos tal que cada nodo consecutivo forma una arista. La longitud del camino es el número de aristas
- Árbol ordenado (ordered tree): existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc.
- Profundidad de un nodo (depth): longitud del camino desde la raíz hasta ese nodo (o viceversa)
- Altura de un nodo (height): longitud del mayor de todos los caminos que van desde el nodo hasta una hoja
- Altura de un árbol no vacío: altura desde la raíz
- Nivel (level): conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel h, donde h es la altura del árbol

Ejercicio: dado el siguiente árbol, encontrar la raíz, los nodos internos y externos/hoja, un posible subárbol, los ancestros de 5 y descendientes de 3, 2 nodos hermanos, una arista, el grado del nodo 3, el grado del árbol, un posible camino, la profundidad del nodo 7, la altura del nodo raíz e identificar los diferentes niveles que tiene.



Raíz → 1
Nodos internos → 2, 3, 8
Nodos externos → 4,5,6,7,9
Un posible sub-árbol → 1,2,4
Ancestros del 5 → 1,2
Descendientes de 3 → 7,8,9
2 posibles nodos hermanos → 2 y 3
Una posible **arista** sería 3-8
Grado(3) → 2
Grado(árbol) → 3
Camino → 1-3-8
Profundidad(7) → 2 (num aristas desde la raíz)
Altura → 3
Nivel 0 → 1 // Nivel 1 → 2,3 // Nivel 2 → 4,5,6,7,8 // Nivel 3 → 9

Jerarquía de clases e interfaces en la biblioteca `aedlib`




```
public interface Tree <E> extends Iterable <E> {
  public int size();
  public boolean isEmpty();
  public Position<E> addRoot(E e) throws NonEmptyTreeException;
  public Position<E> root();
  public Position<E> parent(Position<E> p) throws IllegalArgumentException;
  public boolean isInternal(Position<E> p);
  public boolean isExternal(Position<E> p); // nodo hoja
  public boolean isRoot(Position<E> p);
  public E set(Position<E> p, E e) throws IllegalArgumentException;
  public Iterable<Position<E>> children(Position<E> p);
}
```

`Tree<E>` es un interfaz pensada para trabajar directamente con `Position`. Los métodos que reciben un `Position<E>` podrán lanzar la excepción `IllegalArgumentException`. Tenemos el método `children` que devuelve un `Iterable` para recorrer los hijos de un nodo. `Tree<E>` fundamentalmente se dispone de métodos observadores, ya que está pensado para hacer recorridos de árboles ordinarios. Los métodos modificadores (`addRoot` y `set` principalmente) se definirán en las clases que extienden el interfaz `Tree<E>`

El interfaz `GeneralTree<E>` extiende el interfaz `Tree<E>` con los métodos modificadores necesarios para construir árboles:

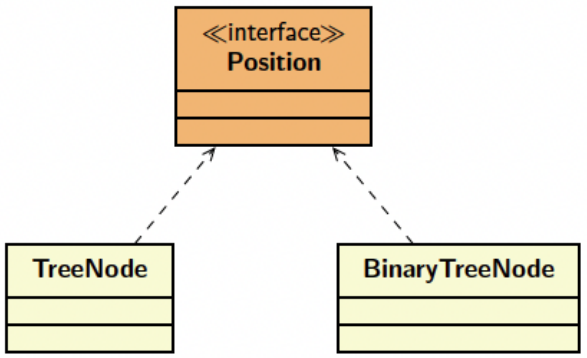
```
public interface GeneralTree<E> extends Tree<E> {
  Position<E> addChildFirst(Position<E> parentPos, E e);
  Position<E> addChildLast(Position<E> parentPos, E e);
  Position<E> insertSiblingBefore(Position<E> siblingPos, E e);
  Position<E> insertSiblingAfter(Position<E> siblingPos, E e);
  void removeSubtree(Position<E> p);
}
```

La clase `LinkedGeneralTree` implementa el interfaz `GeneralTree`

Ejemplo de uso:

```
GeneralTree<Integer> tree = new LinkedGeneralTree<Integer>();
tree.addRoot(1);
Position<Integer> n2 = tree.addChildLast(tree.root(), 2);
Position<Integer> n3 = tree.addChildLast(tree.root(), 3);
Position<Integer> n4 = tree.addChildLast(n2, 4);
Position<Integer> n6 = tree.addChildLast(n2, 6);
Position<Integer> n5 = tree.insertSiblingBefore(n6, 5);
Position<Integer> n7 = tree.addChildLast(n3, 7);
Position<Integer> n8 = tree.insertSiblingAfter(n7, 8);
Position<Integer> n9 = tree.addChildLast(n8, 9);
```

Posiciones en árboles:




```
class TreeNode <E> extends Node <E,Tree <E>> implements Position <E> {
  Position<E> parent; // parent
  PositionList<Position<E>> children; // children
  ...
}

class BinaryTreeNode <E> extends Node <E,Tree <E>> implements Position <E> {
  Position <E> parent; // parent
  Position <E> left, right; // children
  ...
}
```

Las estructuras de datos lineales (p.e. listas, arrays) presentan un único recorrido lógico de todos sus elementos. En el caso de árboles, es posible recorrer todos sus elementos en diferente orden:

- **Recorrido en profundidad**: se visitan todos los nodos de una rama (subárbol) antes de pasar a la siguiente rama
  - En **pre-orden**: se visita cada nodo antes de visitar los sub arboles hijos
    En el ejemplo anterior: 1, 2, 4, 5, 6, 3, 7, 8, 9
  - En **post-orden**: se visita cada nodo despu es de visitar los sub arboles hijos
    En el ejemplo anterior: 4, 5, 6, 2, 7, 9, 8, 3, 1
- **Recorrido en anchura** (por niveles): se visitan todos los nodos de un nivel antes de visitar los nodos del siguiente nivel. En el ejemplo anterior: 1, 2, 3, 4, 5, 6, 7, 8, 9

Árboles Binarios:

Es un tipo especial de árbol en el que todo nodo tiene como mucho 2 hijos, es decir, un árbol general de grado 2. Puede estar vacío o tener (1) un nodo raíz, (2) un (sub)árbol izquierdo y (3) un (sub)árbol derecho. Existen varios tipos:

(1) **Árbol binario propio (full binary tree)** → todo nodo interno tiene dos hijos

(2) **Árbol binario impropio (improper binary tree)** → árbol binario que no es propio

(3) **Árbol binario perfecto (perfect binary tree)** → árbol binario con el máximo número de nodos $(2^{h+1} - 1)$

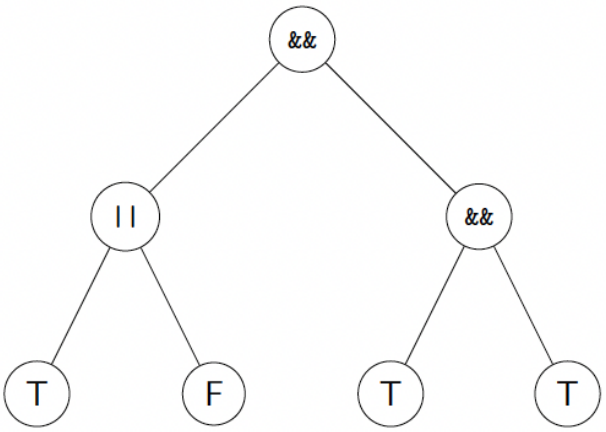
(4) **Árbol binario equilibrado (balanced binary tree)** → árbol en el que para todo nodo, el valor absoluto de la diferencia de altura entre los dos subárboles hijos es como máximo 1, es decir, si tenemos un árbol de altura h, o bien sus dos hijos tienen la misma altura, h-1, o un hijo tiene altura h-1 y el otro h-2.

```
public interface BinaryTree<E> extends Tree<E> {
  public boolean hasLeft(Position<E> p);
  public boolean hasRight(Position<E> p);
  public Position<E> left(Position<E> p);
  public Position<E> right(Position<E> p);
  public Position<E> insertLeft(Position<E> parentPos, E e)
     throws NodeAlreadyExistsException;
  public Position<E> insertRight(Position<E> parentPos, E e)
     throws NodeAlreadyExistsException;
  public void removeSubtree(Position<E> pos);
}
```

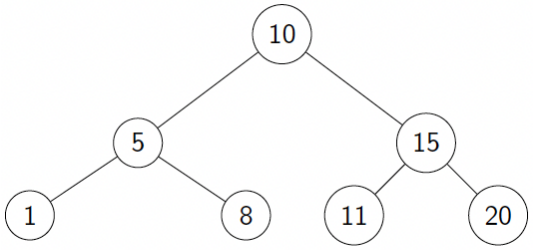
La clase `LinkedBinaryTree` implementa el interfaz `BinaryTree`

```
BinaryTree<Integer> tree = new LinkedBinaryTree<Integer>();
tree.addRoot(1);
Position<Integer> left = tree.insertLeft(tree.root(),2);
Position<Integer> right = tree.insertRight(tree.root(),3);
Position<Integer> n4 = tree.insertLeft(left,4);
Position<Integer> n5 = tree.insertRight(left,5);
Position<Integer> n6 = tree.insertLeft(right,6);
```

Mediante árboles binarios se pueden representar expresiones booleanas:



Árbol Binario de Búsqueda (Binary Search Trees): árbol binario con claves dentro, donde para todo nodo con clave k, su clave k' ≤ k ≤ k'' para todas las claves k' en su subárbol izquierdo y k'' en su subárbol derecho.




```
boolean member(BinaryTree<Integer> t,
    Position<Integer> p, Integer elem ) {
  if (p == null)
    return false;
  else if (elem == p.element())
    return true;
  else if (elem < p.element())
    return member(t, t.left(p), elem);
  else
    return member(t, t.right(p), elem);
}
```

Otros posibles métodos son:
La inserción (`insert`) de una clave nueva es fácil
El borrado (`remove`) de una clave nueva es más complejo.

(1) Borrar un nodo (1) sin hijos es trivial → se borra el nodo
(2) Borrar un nodo (5) con un hijo es fácil → se mueve el nodo hijo (1) a la posición de su padre (5)
(3) Borrar un nodo (10) con dos hijos es un poco más complicado → se puede mover la clave máxima desde el subárbol izquierdo (5) al nodo con la clave 10; y después borrar el antiguo nodo (5); o viceversa, se puede mover la clave mínimo desde el subárbol derecho (11) al nodo con la clave 10

NOTA: la complejidad algorítmica de una búsqueda en un árbol binario de búsqueda es O(n), aunque se puede mejorar, por ejemplo, con árboles AVL (autobalanceados) cuya complejidad es O(log n) para la búsqueda de un elemento.

EJERCICIOS

Ejercicios de árboles generales:

(1) Método que indica si un nodo w es ancestro de otro nodo v
(2) Método que indica si un nodo es hermano de otro
(3) Método que calcula la profundidad de un nodo de un árbol dado (recursivo)
(4) Método que recorre un árbol en pre-orden
(5) Método que recorre un árbol en post-orden
(6) Método que recorre un árbol en anchura

Ejercicios de árboles binarios:

(1) Método que devuelve la altura de un árbol binario
(2) Recorrido de un árbol binario en pre-orden y en post-orden
(3) Recorrido de un árbol binario en in-orden
(4) Método que imprime la expresión contenida en un árbol binario con sus correspondientes paréntesis
(5) Método que nos permite evaluar la expresión booleana contenida en un árbol
(6) Método que nos permite evaluar la expresión aritmética contenida en un árbol