


# Arboles binarios en Java y árboles de expresión

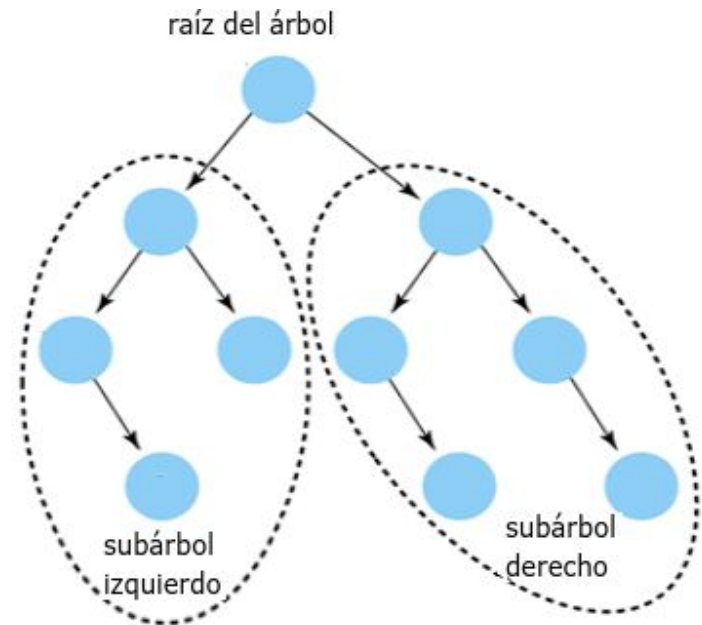
# Arboles Binarios

## Estructura

| <div>  <b>BinaryTree&lt;T&gt;</b> </div> <div>tp2</div>  |
|---|
| <div> <div>data: T</div> <div>leftChild: BinaryTree&lt;T&gt;</div> <div>rightChild: BinaryTree&lt;T&gt;</div> </div>  |
| <div> <div>● BinaryTree(): void</div> <div>● BinaryTree(data: T): void</div> <div>● getData(): T</div> <div>● setData(data: T): void</div> <div>● getLeftChild(): BinaryTree&lt;T&gt;</div> <div>● getRightChild(): BinaryTree&lt;T&gt;</div> <div>● addLeftChild(child: BinaryTree&lt;T&gt;): void</div> <div>● addRightChild(child: BinaryTree&lt;T&gt;): void</div> <div>● removeLeftChild(): void</div> <div>● removeRightChild(): void</div> <div>● isEmpty(): boolean</div> <div>● isLeaf(): boolean</div> <div>● hasLeftChild(): boolean</div> <div>● hasRightChild(): boolean</div> <div>● toString(): String</div> <div>● contarHojas(): int</div> <div>● espejo(): BinaryTree&lt;T&gt;</div> <div>● entreNiveles(n: int, m: int): void</div> </div> |

-leftChild

-rightChild



# Arboles Binarios

```
public class BinaryTree <T> {  
    private T data;  
    private BinaryTree<T> leftChild;  
    private BinaryTree<T> rightChild;
```

```
    public BinaryTree() {  
        super();  
    }  
    public BinaryTree(T data) {  
        this.data = data;  
    }
```

Constructores

```
    public T getData() {  
        return data;  
    }
```

```
    public void setData(T data) {  
        this.data = data;
```

Preguntar antes  
de invocar si  
hasLeftChild()/  
hasRightChild()

```
    }  
    public BinaryTree<T> getLeftChild() {  
        return leftChild;  
    }
```

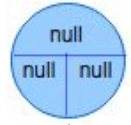
```
    public BinaryTree<T> getRightChild() {  
        return rightChild;  
    }
```

```
    public void addLeftChild(BinaryTree<T> child){  
        this.leftChild = child;  
    }
```

```
    public void addRightChild(BinaryTree<T> child) {  
        this.rightChild = child;  
    }
```

```
    public void removeLeftChild() {  
        this.leftChild = null;  
    }
```

```
    public void removeRightChild() {  
        this.rightChild = null;  
    }
```



Arbol vacío

```
    public boolean isEmpty(){  
        return (this.isLeaf() && this.getData() == null);  
    }
```

```
    public boolean isLeaf() {  
        return (!this.hasLeftChild() && !this.hasRightChild());  
    }
```

```
    public boolean hasLeftChild() {  
        return this.leftChild!=null;  
    }
```

```
    public boolean hasRightChild() {  
        return this.rightChild!=null;  
    }
```

@Override

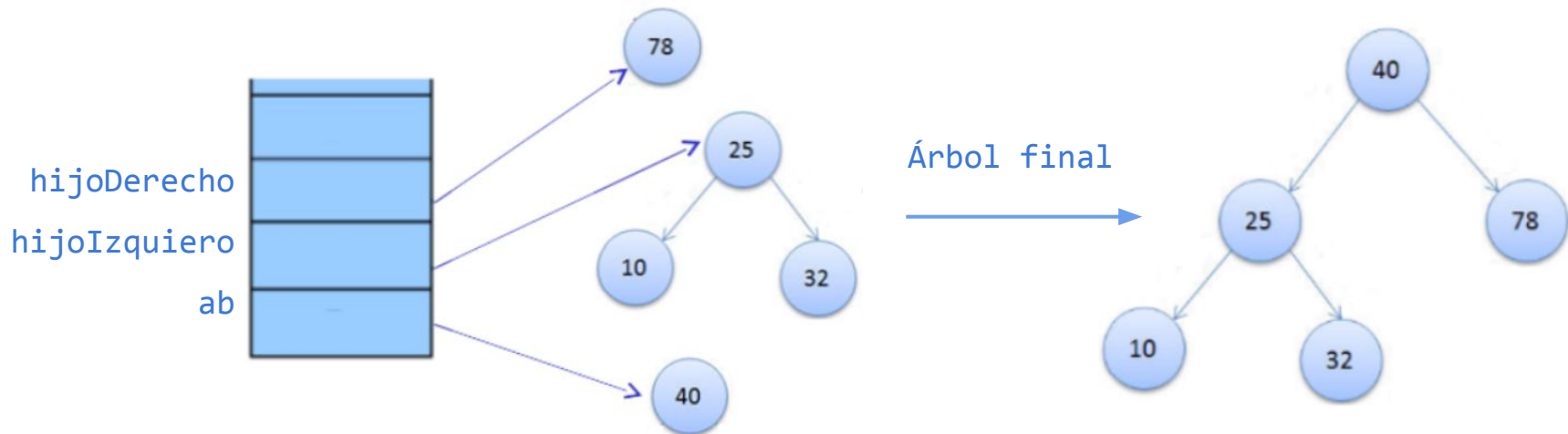
```
    public String toString() {  
        return this.getData().toString();  
    }
```

```
}
```

# Arboles Binarios

## Creación

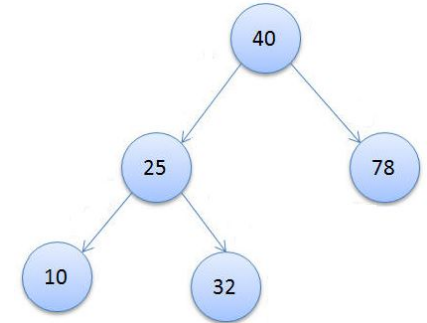
```
BinaryTree<Integer> ab = new BinaryTree<Integer>(40);  
BinaryTree<Integer> hijoIzquierdo = new BinaryTree<Integer>(25);  
hijoIzquierdo.addLeftChild(new BinaryTree<Integer>(10));  
hijoIzquierdo.addRightChild(new BinaryTree<Integer>(32));  
BinaryTree<Integer> hijoDerecho = new BinaryTree<Integer>(78);  
ab.addLeftChild(hijoIzquierdo);  
ab.addRightChild(hijoDerecho);
```



# Arboles Binarios

## Recorridos (1/2)

Los árboles binarios se pueden recorrer de diferentes maneras, de acuerdo al orden en el que se visitan sus nodos. En el cuadro se sintetiza cada uno de ellos.



### Preorden

Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.

40, 25, 10, 32, 78

```
public void preorden() {  
    imprimir (dato);  
    si (tiene hijo_izquierdo)  
        hijoIzquierdo.preorden();  
    si (tiene hijo_derecho)  
        hijoDerecho.preorden();  
}
```

### Inorden

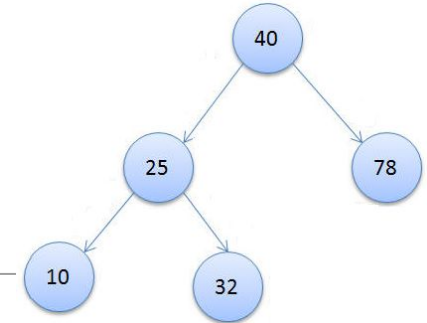
Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho.

10, 25, 32, 40, 78

```
public void preorden() {  
    si (tiene hijo_izquierdo)  
        hijoIzquierdo.preorden();  
    imprimir (dato);  
    si (tiene hijo_derecho)  
        hijoDerecho.preorden();  
}
```

# Arboles Binarios

## Recorridos (2/2)



### Postorden

Se procesan primero los hijos, izquierdo y derecho, y luego la raíz

10, 32, 25, 78, 40

```
public void preorden() {  
    si (tiene hijo_izquierdo)  
        hijoIzquierdo.preorden();  
    si (tiene hijo_derecho)  
        hijoDerecho.preorden();  
    imprimir (dato);  
}
```

### Por niveles

Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

40, 25, 78, 10, 32

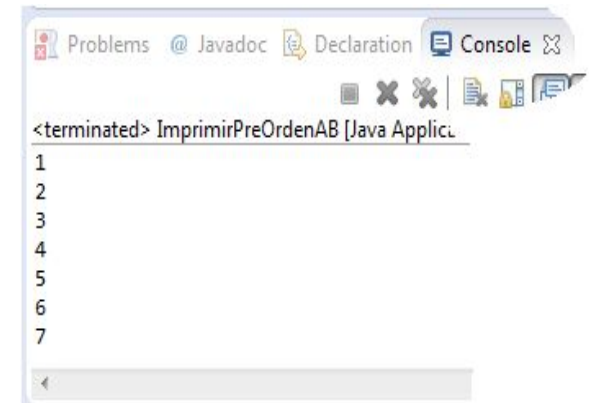
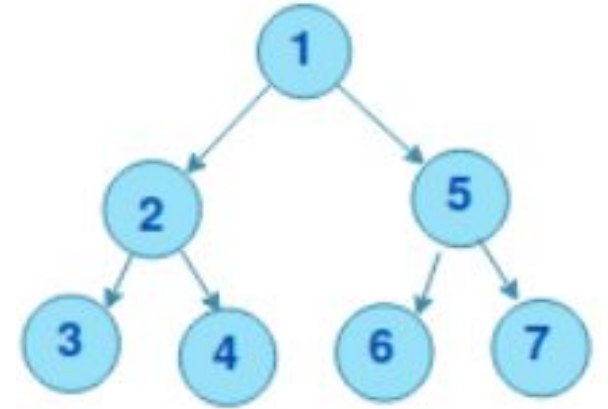
```
public void porNiveles() {  
    encolar(raíz);  
    mientras (cola no se vacíe) {  
        desencolar(v);  
        imprimir (dato de v);  
        si (tiene hijo_izquierdo)  
            encolar(hijo_izquierdo);  
        si (tiene hijo_derecho)  
            encolar(hijo_derecho);  
    }  
}
```

# Arboles Binarios

## Recorrido PreOrden

Se procesa primero la raíz y luego sus hijos, izquierdo y derecho

```
public class BinaryTree<T> {  
  
    private T data;  
    private BinaryTree<T> leftChild;  
    private BinaryTree<T> rightChild;  
  
    public void printPreorden() {  
        System.out.println(this.getData());  
        if (this.hasLeftChild()) {  
            this.getLeftChild().printPreorden();  
        }  
        if (this.hasRightChild()) {  
            this.getRightChild().printPreorden();  
        }  
    }  
}
```

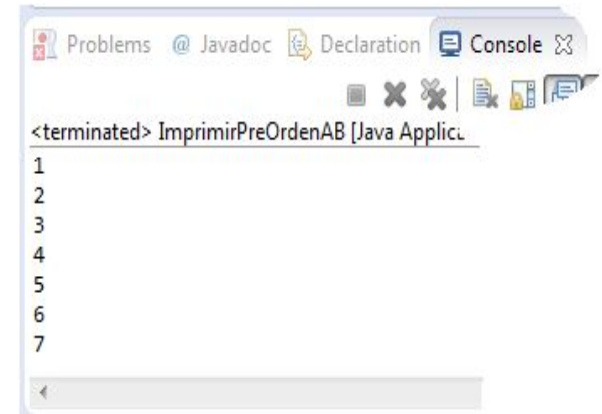
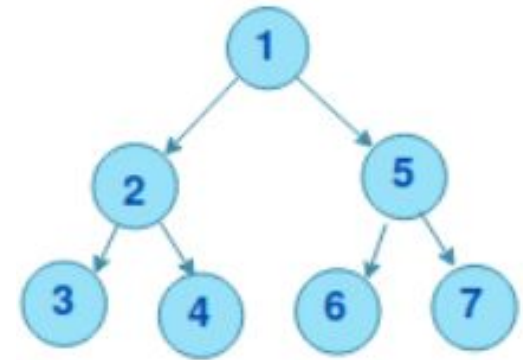


# Arboles Binarios

## Recorrido PreOrden

¿Qué cambio se debería hacer si el método `preorden()` debe definirse en otra clase diferente al **BinaryTree**?

```
public class BinaryTreePrinter<T> {  
  
    public void preorden(BinaryTree<T> ab) {  
        System.out.println(ab.getData());  
        if (ab.hasLeftChild()) {  
            this.preorden(ab.getLeftChild());  
        }  
        if (ab.hasRightChild()) {  
            this.preorden(ab.getRightChild());  
        }  
    }  
}
```





# Arboles Binarios

## Recorrido PreOrden

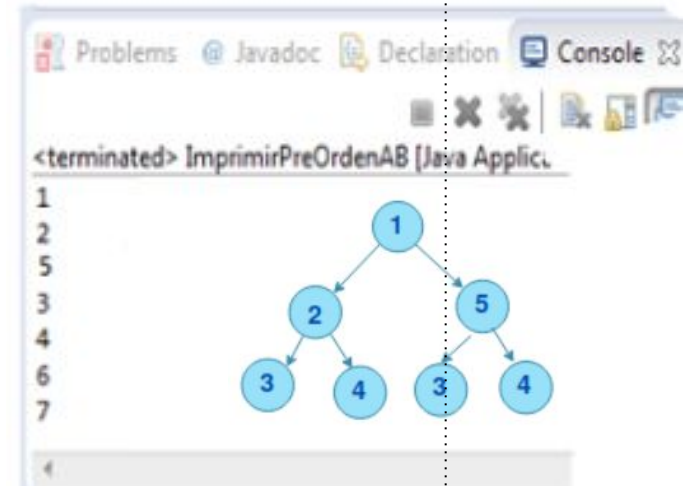
¿Qué cambio harías para **devolver una lista** con los elementos de un recorrido en preorden?

```
package tp2.ejercicio1;  
import java.util.List;  
import java.util.LinkedList;
```

```
public class BinaryTreePrinter<T> {
```

```
    public List<T> preorden(BinaryTree<T> ab) {  
        List<T> result = new LinkedList<T>();  
        this.preorden_private(ab, result);  
        return result;  
    }
```

```
    private void preorden_private(BinaryTree<T> ab, List<T> result) {  
        result.add(ab.getData());  
        if (ab.hasLeftChild()) {  
            preorden_private(ab.getLeftChild(), result);  
        }  
        if (ab.hasRightChild()) {  
            preorden_private(ab.getRigthChild(), result);  
        }  
    }  
}
```

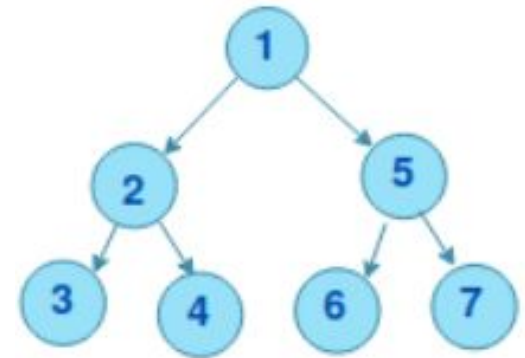


# Arboles Binarios

## Recorrido por Niveles

```
package tp1.ejercicio1;
import tp1.Queue;
public class BinaryTree<T> {
    . . .
    public void printLevelTraversal() {
        BinaryTree<T> ab = null;
        Queue<BinaryTree<T>> cola = new Queue<BinaryTree<T>>();
        cola.enqueue(this);
        cola.enqueue(null);
        while (!cola.isEmpty()) {
            ab = cola.dequeue();
            if (ab != null) {
                System.out.print(ab.getData());
                if (ab.hasLeftChild()) {
                    cola.enqueue(ab.getLeftChild());
                }
                if (ab.hasRightChild()) {
                    cola.enqueue(ab.getRightChild());
                }
            } else if (!cola.isEmpty()) {
                System.out.println();
                cola.enqueue(null);
            }
        }
    }
    . . .
}
```

Implementación del recorrido por niveles dentro de la clase BinaryTree.

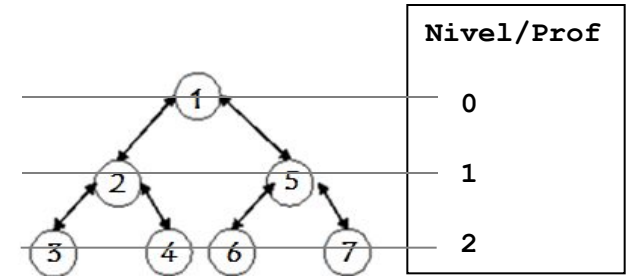


# Arboles Binarios

## ¿Es árbol lleno?

Dado un árbol binario de altura  $h$ , diremos que es un **árbol lleno** si cada nodo interno tiene grado 2 y todas las hojas están en el mismo nivel ( $h$ ). Implementar un método para determinar si un árbol binario es "lleno"

```
public boolean lleno() {
    BinaryTree<T> ab = null;
    Queue<BinaryTree<T>> cola = new Queue<BinaryTree<T>>();
    boolean lleno = true;
    int cant_nodos = 0;
    int nivel = 0;
    cola.enqueue(this);
    cola.enqueue(null);
    while (!cola.isEmpty() && lleno) {
        ab = cola.dequeue();
        if (ab != null) {
            if (ab.hasLeftChild()) {
                cola.enqueue(ab.getLeftChild());
                cant_nodos++;
            }
            if (ab.hasRightChild()) {
                cola.enqueue(ab.getRightChild());
                cant_nodos++;
            }
        } else if (!cola.isEmpty()) {
            if (cant_nodos == Math.pow(2, ++nivel)) {
                cola.enqueue(null);
                cant_nodos = 0;
            } else {
                lleno = false;
            }
        }
    }
    return lleno;
}
```



`arbol = cola.desencolar();`

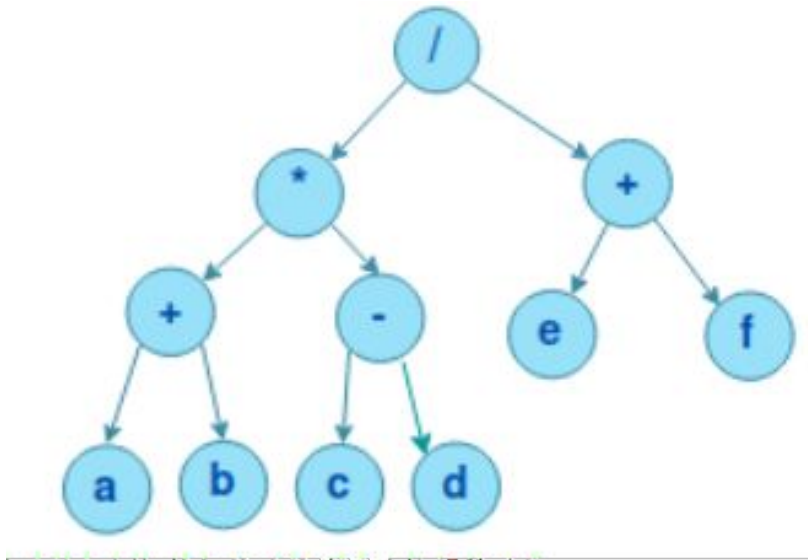
`arbol = null`  
`cant_nodos = 2`

# Árboles Binarios

## Árboles de Expresión

Un árbol de expresión es un árbol binario asociado a una expresión aritmética donde:

- Los nodos internos representan operadores
- Los nodos externos (hojas) representan operandos



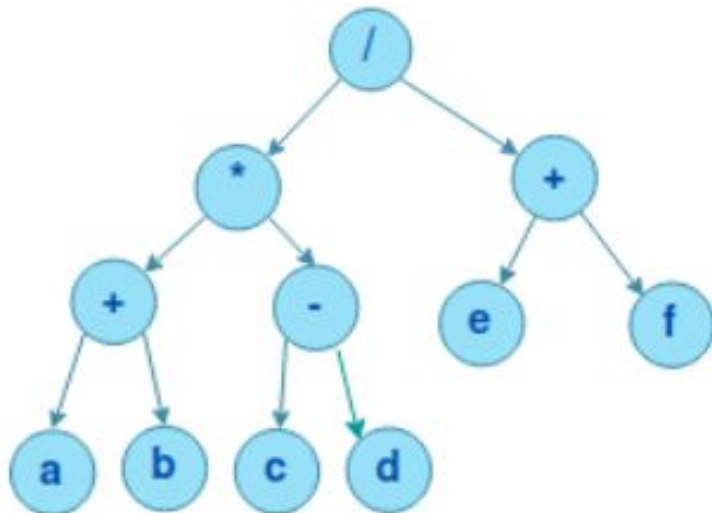
No necesitan el uso de  
paréntesis

# Árboles de Expresión

## Casos de uso

Algunas aplicaciones de los árboles de expresión son:

- En compiladores se usa para analizar, optimizar y traducir programas.
- Evaluar expresiones algebraicas o lógicas complejas de manera eficiente
- Los árboles pueden almacenar expresiones algebraicas y a partir de ellos se puede generar notaciones sufijas, prefijas e infijas.



### Recorridos

Inorden:  $((a + b) * (c - d)) / (e + f)$  → expresión infija

Preorden:  $/*+ab-cd+ef$  → expresión prefija

Postorden:  $ab+cd-*ef+ /$  → expresión posfija

# Árboles de expresión

## Construcción de un árbol de expresión a partir de una expresión posfija

### Estrategia I:

*convertir(expr\_posfija)*

*crear una Pila vacía*

mientras (existe un carácter) hacer

*tomo un carácter de la expresión*

si es un **operando**

□ *creo un nodo y lo apilo*

si es un **operador** (lo tomo como la raíz de los dos últimos nodos creados)

□ *creo un nodo R con ese operador*

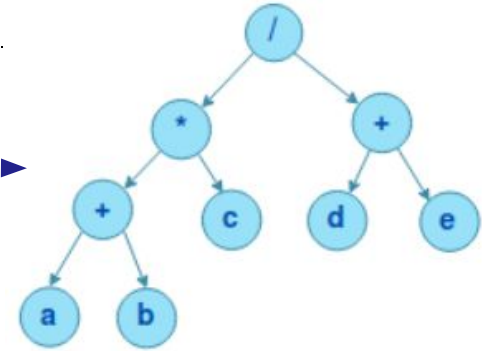
*desapilo y lo agrego como hijo derecho de R*

*desapilo y lo agrego como hijo izquierdo de R*

*apilo R.*

desapilar □ *árbol de expresión posfija final*

*ab+c\*de+/-* ---->



Este proceso es posible ya que la expresión posfija está organizada en una forma en la que los operandos aparecen antes de los operadores. Esto nos permite construir el árbol de expresión utilizando una pila, donde se apilan operandos hasta que se encuentre un nodo operador que tome los dos últimos nodos de la pila como hijos.

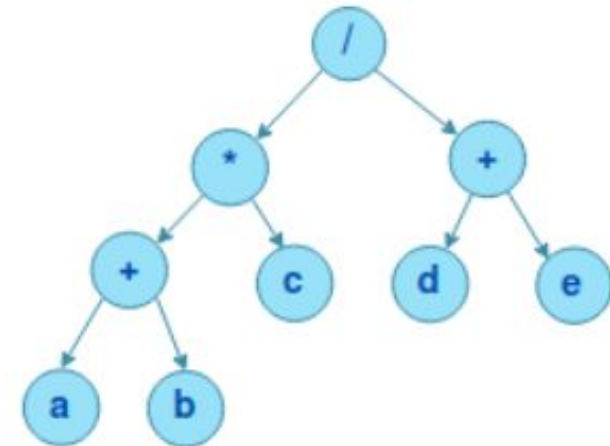
# Árboles de expresión

## Construcción de un árbol de expresión a partir de una expresión posfija

Este método convierte una expresión **postfija** en un **BinaryTree**. Puede estar implementado en cualquier clase.

```
public BinaryTree<Character> convertirPostfija(String exp) {  
  
    Character c = null;  
    BinaryTree<Character> result;  
    Stack<BinaryTree<Character>> p = new Stack<BinaryTree<Character>>();  
  
    for (int i = 0; i < exp.length(); i++) {  
        c = exp.charAt(i);  
        result = new BinaryTree<Character>(c);  
        if ((c == '+') || (c == '-') || (c == '/') || (c == '*')) {  
            // Es operador  
            result.addRightChild(p.pop());  
            result.addLeftChild(p.pop());  
        }  
        p.push(result);  
    }  
    return (p.pop());  
}
```

**ab+c\*de+/** ---->



# Árboles de expresión

## Construcción de un árbol de expresión a partir de una expresión prefija

### Estrategia II:

*convertir(expr\_prefija)*

*tomo primer carácter de la expresión*

*creo un nodo R con ese operador*

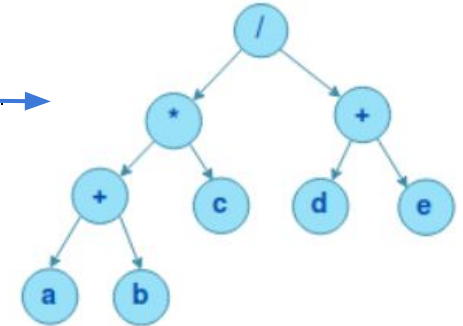
*si el carácter es un **operador***

- ☐ *agrego como hijo izquierdo de R(*convertir(expr\_prefija sin 1 carácter())*)*
- ☐ *agrego como hijo derecho de R(*convertir(expr\_prefija sin 1 carácter())*)*

*//es un operando*

*devuelvo el nodo R*

*/\*+abc+de*



Este proceso es posible ya que la expresión prefija está organizada en una forma en la que los operadores siempre aparecen antes de los operandos. Cuando se llega a las hojas, la recursión retorna y permite ir armando el árbol desde abajo hacia arriba.



# Árboles de expresión

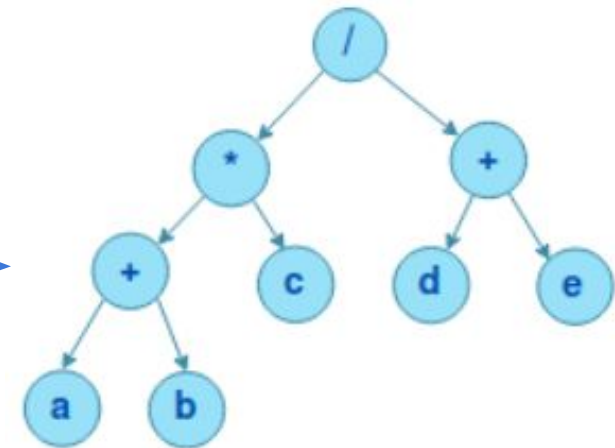
## Construcción de un árbol de expresión a partir de una expresión prefija

Este método convierte una expresión **prefija** en un árbol de expresión. Puede estar implementado en cualquier clase.

```
public BinaryTree<Character> convertirPrefija(StringBuffer exp) {  
  
    Character c = exp.charAt(0);  
    BinaryTree<Character> result = new BinaryTree<Character>(c);  
    if ((c == '+') || (c == '-') || (c == '/') || c == '*') {  
        // es operador  
        result.addLeftChild(this.convertirPrefija(exp.delete(0, 1)));  
        result.addRigthChild(this.convertirPrefija(exp.delete(0, 1)));  
    }  
    // es operando  
    return result;  
}
```

**/\*+abc+de**

---



# Árboles de expresión

## Construcción de un árbol de expresión a partir de una expresión infija

La estrategia para crear un árbol de expresión a partir de una expresión **infija** es un poco más compleja. Primero se debe convertir a una expresión **posfija**.

### Estrategia III:

#### Expresión infija

- (i) Se usa una pila y se tiene en cuenta la precedencia de los operadores



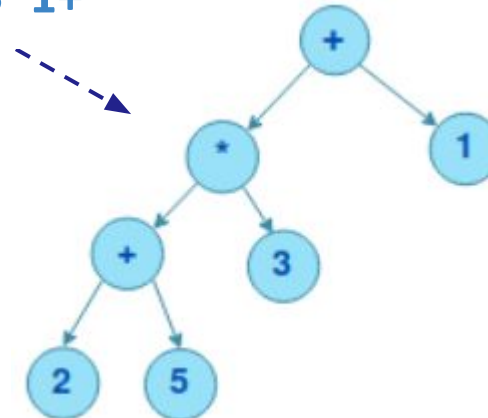
#### Expresión posfija

- (ii) Se usa la estrategia I

$(2+5)*3+1$



$25+3*1+$



# Árboles de expresión

## Construcción de un árbol de expresión a partir de una expresión infija

Este método convierte una expresión **infija** en una expresión **posfija**. Luego se aplica el algoritmo iterativo visto anteriormente.

*crear una Pila vacía*

*mientras ( existe un carácter ) hacer*

*tomo un carácter de la expresión*

*si es un **operando** □ coloca en la salida*

*si es un **operador** □ se analiza su prioridad respecto del tope de la pila:*

*si es un “(“ , “)” □*

*“(“ se apila*

*”)” se desapila todo hasta el “(“, incluido éste*

*sino*

*pila vacía u operador con > prioridad que el tope → se apila*

*operador con <= prioridad que el tope → se desapila, se manda a la salida*

*y se vuelve a comparar el operador con el tope de la pila*

*//se terminó de procesar la expresión infija*

*Se desapilan todos los elementos llevándolos a la salida, hasta que la pila quede vacía.*

| Prioridades |
|-------------|
| $\wedge$    |
| $*, /$      |
| $+, -$      |

$(2+5)*3+(10/5)$



$2\ 5+3*10\ 5\ /\+$

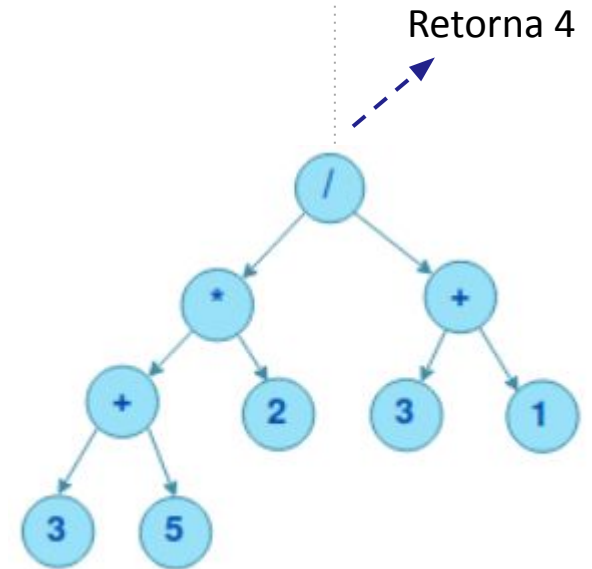
Nota: Los ' ( { [ ' se apilan siempre (como si tuvieran mayor prioridad) y a partir de ahí se considera como si la pila estuviera vacía.

# Árboles de expresión

## Evaluación

Este método evalúa y retorna un número de acuerdo a la expresión aritmética representada por el **ArbolBinario** que es enviado como parámetro.

```
public Integer evaluar(BinaryTree<Character> arbol) {  
    Character c = arbol.getData();  
    if ((c == '+') || (c == '-') || (c == '/') || c == '*') {  
        // es operador  
        int operador_1 = evaluar(arbol.getLeftChild());  
        int operador_2 = evaluar(arbol.getRightChild());  
        switch (c) {  
            case '+':  
                return operador_1 + operador_2;  
            case '-':  
                return operador_1 - operador_2;  
            case '*':  
                return operador_1 * operador_2;  
            case '/':  
                return operador_1 / operador_2;  
        }  
    }  
    // es operando  
    return Integer.parseInt(c.toString());  
}
```



# Árboles Binarios

## Problema: Encontrar la Valencia Total

El Sr. White ha encontrado una manera de maximizar la pureza de los cristales basados en ciertos compuestos químicos. Ha observado que cada compuesto está hecho de moléculas que están unidas entre sí siguiendo la estructura de un árbol binario completo.

Cada nodo del árbol almacena la valencia de una molécula y se representa como un número entero. El Sr. White utiliza un microscopio electrónico que descarga la estructura de la molécula como un stream de números enteros y le gustaría tener su ayuda para obtener automáticamente la valencia total de las hojas del árbol dado.

Cada línea de entrada comienza con un entero  $N$  ( $1 \leq N \leq 1000000$ ), seguido de  $N$  números enteros  $V_i$  que representan las valencias de cada molécula separadas por espacios en blanco ( $0 \leq V_i \leq 100$ ).

El final de la entrada se indica mediante un caso de prueba con  $N = 0$ .

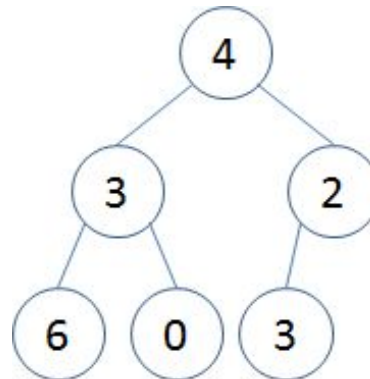
Ejemplo

**Input:**

```
6 4 3 2 6 0 3
7 1 1 1 2 1 2 1
0
```

**Output:**

```
9
6
```



# Árboles Binarios

## Problema: Encontrar la Valencia Total

crearBinaryTree(stream,1)

```
public class ValenciaTotal {  
  
    private static BinaryTree<Integer> crearBinaryTree(int[] stream, int i) {  
        int dato = stream[i];  
        BinaryTree<Integer> ab = new BinaryTree<Integer>(dato);  
        if (2*i < stream.length)  
            ab.addLeftChild(crearBinaryTree(stream, 2*i));  
        if (2*i+1 < stream.length)  
            ab.addRightChild(crearBinaryTree(stream, 2*i+1));  
        return ab;  
    }  
  
    public static int calcularValencia(BinaryTree<Integer> arbol) {  
        if (arbol.isLeaf())  
            return arbol.getData();  
        int suma = 0;  
        if (arbol.hasLeftChild()) {  
            suma = suma + calcularValencia(arbol.getLeftChild());  
        }  
        if (arbol.hasRightChild()) {  
            suma = suma + calcularValencia(arbol.getRightChild());  
        }  
        return suma;  
    }  
}
```

| stream |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|
| 6      | 4 | 3 | 2 | 6 | 0 | 3 |
| 0      | 1 | 2 | 3 | 4 | 5 | 6 |

Como es un árbol completo, dado un nodo i:

- hijo izquierdo  $2*i$
- hijo derecho  $2*i+1$

