

Taller 4

Fecha: Septiembre de 2024

**Indicador de logro a medir:** Aplicar los conceptos de las estructuras **Pila** y **Árbol Binario** en el desarrollo de una aplicación con interfaz gráfica de usuario basada en el lenguaje **Java** y un IDE adecuado.

NOTAS:

- Este taller (punto 4) se debe hacer con carácter evaluativo. Representa en total una calificación de 20%.
- Se entregan ejercicios resueltos como ejemplo para el desarrollo de los demás.

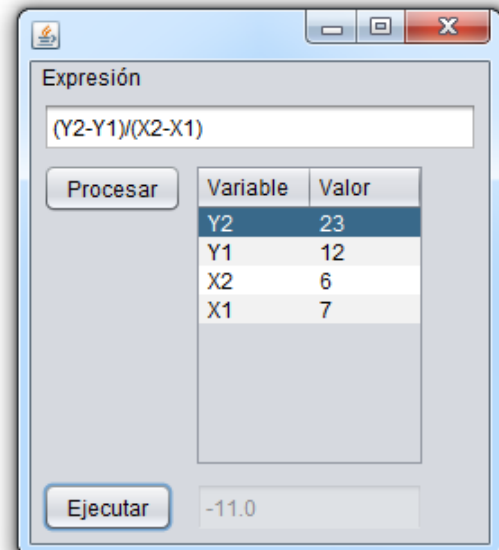
Elaborar el diagrama de clases básico (sin las clases correspondientes a la interface de usuario según el lenguaje de implementación) y la respectiva aplicación en un lenguaje orientado a objetos para los siguientes enunciados:

1. Implementar una calculadora de expresiones aritméticas, donde el parámetro de entrada es un texto conformado por variables, números y operadores aritméticos los cuales componen una expresión aritmética, la cual debe ejecutarse.

Por ejemplo, para calcular el valor de la pendiente de una recta se tiene la siguiente fórmula:

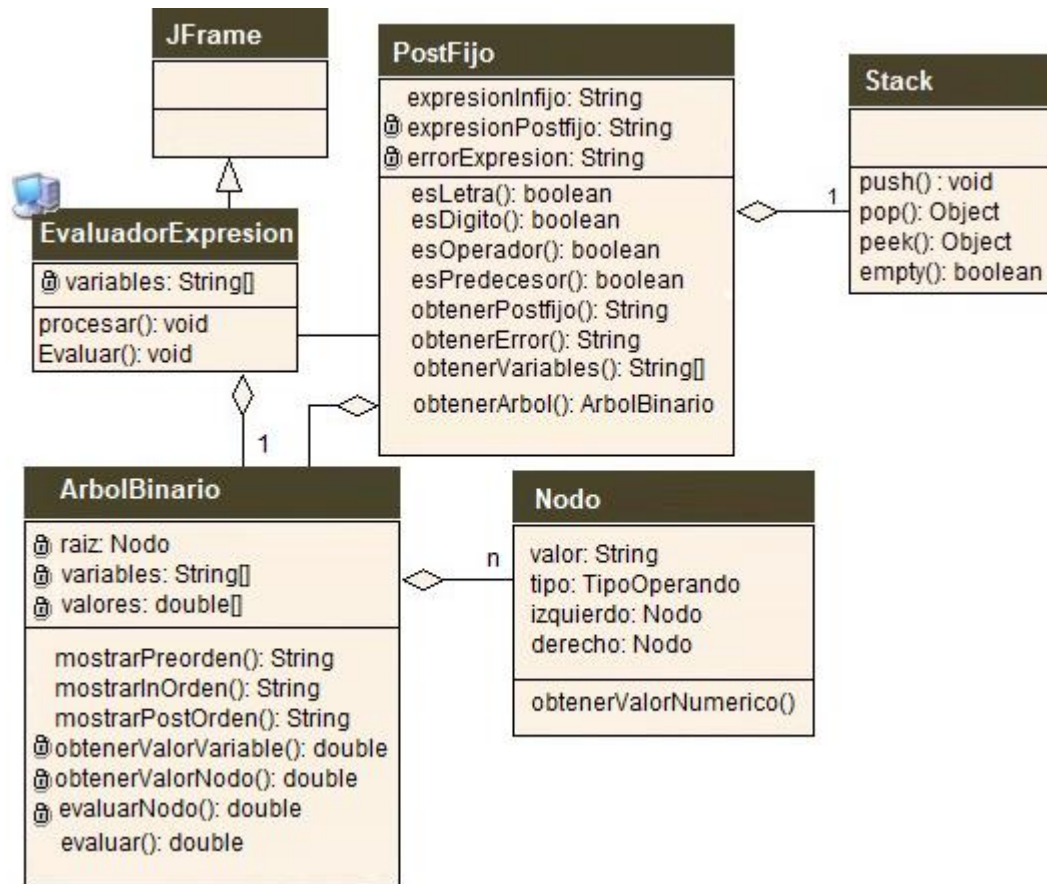
$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

La ejecución de dicho cálculo deberá lucir así:



R/

- El modelado bajo el paradigma Orientado a Objetos se ilustra en el siguiente diagrama de clases:



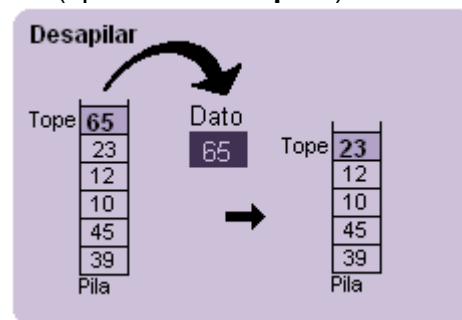
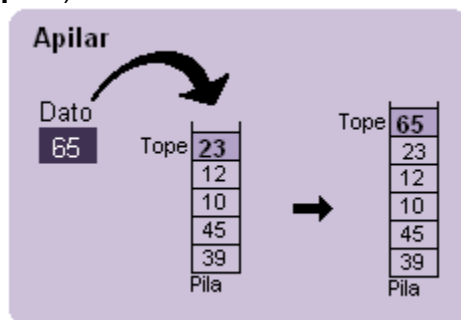
Para comprender este diagrama y el ejercicio, es importante tener en cuenta los siguientes fundamentos:

### Concepto de Pila

Una **Pila** es una lista de elementos en la cual siempre por un extremo, denominado **Tope**, se pueden insertar nuevos elementos o retirar otros. Es una lista en la que el último que entra es el primero que sale (estructura LIFO).

Son dos los cambios básicos que se pueden hacer en una pila:

- Agregar un nuevo elemento (operación **Apilar**)
- Quitar el último elemento agregado (operación **Desapilar**)




Para realizar estas operaciones existen algunas restricciones:

- No se puede *desapilar* si no hay elementos en la pila (condición **Pila Vacía**).
- En algunos casos, al apilar no se puede sobrepasar la capacidad máxima de la pila (condición **Pila Llena**)

#### Pila Vacía

Tope   
Pila

#### Pila Llena

Tope  Maximo  
58  
73  
65  
12  
10  
45  
39  
Pila

El siguiente sería el pseudocódigo orientado a objetos para la estructura Pila:

#### Clase Publico Pila

// Declaración de datos de la estructura

// Vector con los datos

**Privado p[] Es Objeto**

//Variable que indica el tope de la pila

**Privado tope Es Entero**

#### Metodo Constructor()

tope = -1

**FinMetodo**

//Método que determina si una pila está vacía

**Metodo pilaVacía() Es Booleano**

**Si tope = -1 Entonces**

**Retornar Verdadero**

**Sino**

**Retornar Falso**

**FinSi**

**FinMetodo**

//Método para apilar datos

**Metodo apilar ( valor Es Objeto ) Es RetornoVacio**

tope = tope + 1

p[tope] = valor

**FinMetodo**

//Método que permite desapilar datos

**Metodo desapilar() Es Objeto**

Valor **Es Objeto**

**Si No pilaVacía() Entonces**

valor = p[tope]

tope = tope - 1

**Retornar valor**

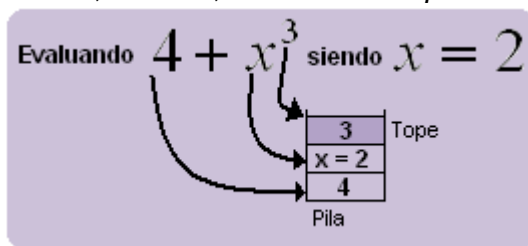
Sino  
Retornar RetornoVacio  
FinSi  
FinMetodo  
  
Fin Clase

### Notaciones Infixo y Postfijo

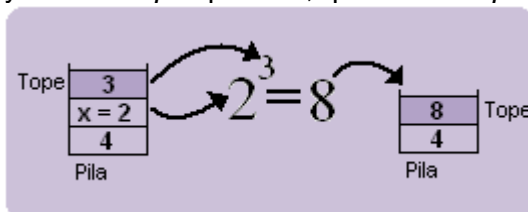
La notación **Postfijo** conocida también como notación **Polaca Inversa** es un método de introducción de datos en las expresiones alternativo al algebraico (conocido como notación **Infixo**). Fue creada por el matemático polaco *Jan Lukasiewicz* en 1920. Básicamente la diferencias entre la forma *infixo* y la *postfijo*, es que al evaluar los datos de una expresión no es necesario realizar un ordenamiento de los mismos, y que para ejecutar una operación, primero se deben introducir todos sus operandos. De esta manera, para ejecutar la suma " $a + b$ " en notación *infixo*, la notación *postfijo* la manejaría como " $a b +$ ", arrojando el resultado directamente. Veamos un ejemplo:

Para evaluar la expresión  $4 + x^3$  en notación *infixa* siendo  $x = 2$  primero se debe tomar el 2 (en vez de la  $x$ ), elevarlo al cubo y luego sumarle 4; se nota que esto no está en forma secuencial, es decir, se debe ir primero a la segunda parte de la expresión, evaluarla y luego sumarle al resultado a la primera.

Esto sería muy complejo si lo fuese a ejecutar un programa, pero si traducimos  $4 + x^3$  a la forma *postfija* " $4 x 3 ^ +$ ", entonces se convierte en una expresión más sencilla de evaluar ya que el algoritmo leería el texto de izquierda a derecha, *apilando* operandos cuando los encuentre y ejecutando las operaciones conforme aparezcan los operadores. De esta manera, se tomaría el 4 y se metería en una *pila* de números, luego se tomaría la  $x$  por lo que se *apilaría* el 2 (que es el valor que se está evaluando), luego se tomaría el 3 y se mete nuevamente en la *pila* de números, es decir, se tendría una *pila* de números como la siguiente:



El siguiente dato en la expresión sería el operador *potenciación*, por lo que se *desapilarían* los dos últimos números de la *pila* (2 y 3) y se realizaría la respectiva operación  $2^3 = 8$ , de esta manera, se sustituirían el 2 y el 3 en la *pila* por el 8, quedando la *pila* así:



Por último, sigue un signo de suma, el cual se realizaría con los dos números que están en la *pila*, por lo que se tomarían el 4 y el 8 arrojando como resultado  $4 + 8 = 12$  y al no haber más datos en la expresión, sería el resultado final.

### Convertir expresiones Infijo en expresiones Postfijo.

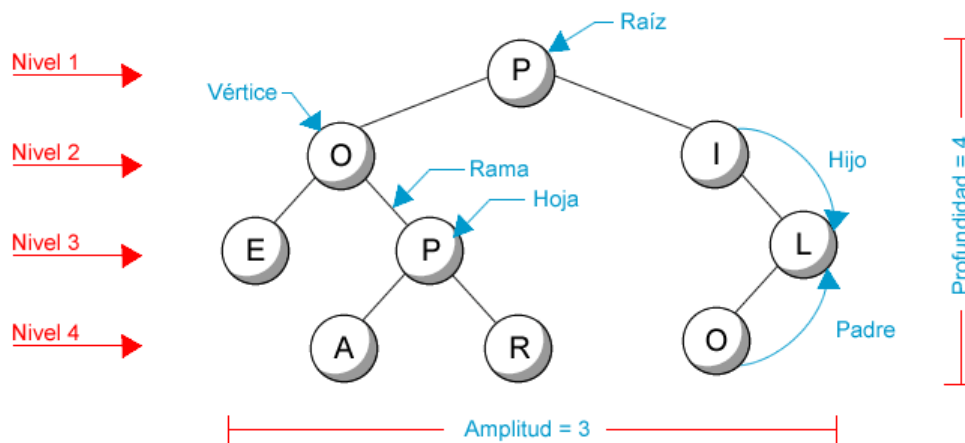
Lo que se describió anteriormente trata de como ejecutar una expresión *Postfijo* (en la que se hace uso de la estructura *pila*), pero aún falta saber cómo transformar la notación *infija* a *postfija*. El siguiente algoritmo, el cual va a necesitar también una *pila* (para almacenar los operadores y los paréntesis izquierdos), expone como hacerlo:

1. Leer la expresión *infija* carácter a carácter, de izquierda a derecha.
2. Si es un operando, éste pasa directamente a formar parte de la expresión *postfija*.
3. Si es un operador:
  - 3.1. Si la *pila* está vacía o el operador tiene mayor prioridad que el que está en el *tope* de la *pila* se mete en la *pila*.
  - 3.2. Si la prioridad es menor o igual, el operador se saca de la *pila*, se mete en la expresión *postfija* y se vuelve a hacer la comparación con el nuevo elemento del *tope* de la *pila*.
4. Si es un paréntesis izquierdo, se mete en la *pila* con la mínima prioridad.
5. Si es un paréntesis derecho, hay que sacar todos los operadores de la *pila*, y añadirlos a la expresión *postfija*, hasta llegar al paréntesis izquierdo, que se elimina del *tope* de la *pila*.
6. El algoritmo termina cuando no quedan más ítems en la expresión y la *pila* está vacía.

### Concepto de Árbol

En general, los árboles son muy utilizados en la ilustración de árboles genealógicos o estructuras jerárquicas de las organizaciones, pero en informática tienen gran aplicación para la creación de los directorios (carpetas) en los discos de las computadoras, para mostrar las relaciones lógicas entre los registros de una base de datos (colección de registros asociados a llaves foráneas) y las búsquedas de datos en información ordenada (véase árboles binarios).

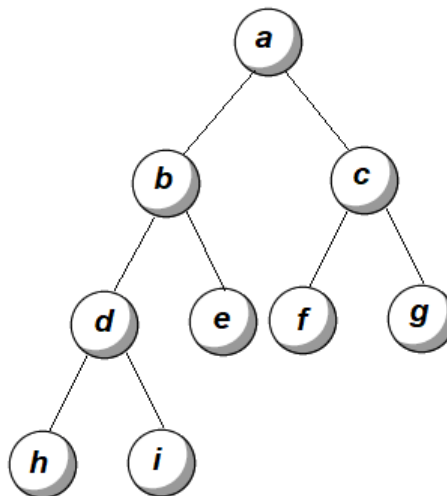
Un árbol es un grafo simple que cumple que entre dos vértices existe un único camino simple.



La representación y terminología de los árboles se realiza con las típicas notaciones de las relaciones familiares en los árboles genealógicos: padre, hijo, hermano, ascendente, descendiente, etc.

- **Raíz:** Todos los árboles que no están vacíos tienen un único nodo raíz. Todos los demás elementos o nodos derivan o descienden de él. El nodo Raíz no tiene Padre es decir no es hijo de ningún elemento.
- **Padre:** X es padre de Y sí y solo sí el nodo X apunta a Y. También se dice que X es antecesor de Y.
- **Hijo:** X es hijo de Y, sí y solo sí el nodo X es apuntado por Y. También se dice que X es descendiente directo de Y
- **Hermano:** Dos nodos serán hermanos si son descendientes directos de un mismo nodo
- **Hoja:** Se le llama hoja o Terminal a aquellos nodos que no tienen ramificaciones (hijos)
- **Nodo:** Son los Vértices o elementos del Árbol
- **Nodo interior:** Es un nodo que no es raíz ni Terminal
- **Grado:** Es el número de descendientes directos de un determinado nodo
- **Grado del árbol:** Es el máximo grado de todos los nodos del árbol
- **Nivel:** Es el número de arcos que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene nivel 1
- **Altura:** Es el máximo número de niveles de todos los nodos del árbol. Equivale al nivel más alto de los nodos más 1
- **Peso:** Es el número de nodos terminales del árbol
- **Longitud de camino:** Es el número de arcos que deben ser recorridos para llegar desde la raíz al nodo X. Por definición la raíz tiene longitud de camino 1, y sus descendientes directos longitud de camino 2 y así sucesivamente

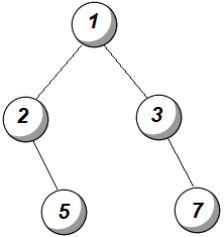
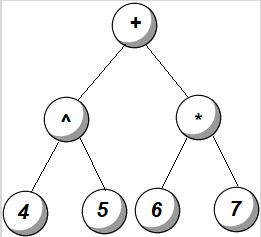
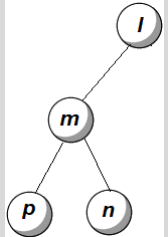
Un **árbol binario** es un tipo de árbol en que cada vértice máximo puede tener dos hijos; su nodo raíz está enlazado a dos subárboles binarios disjuntos denominados subárbol izquierdo y subárbol derecho. Los árboles binarios no son vacíos ya que como mínimo tienen el nodo raíz



Los árboles binarios pueden representarse en un vector o en una lista ligada.

### Recorridos en un árbol binario

Un **Recorrido** en un árbol binario es una operación que consiste en visitar todos sus vértices o nodos, de tal manera que cada vértice se visite una sola vez. En cada recorrido se tiene en cuenta la posición de la raíz (de ahí su nombre) y que siempre se debe ejecutar primero el hijo izquierdo y luego el derecho. Se distinguen tres tipos de recorrido:

| Recorrido | Descripción   |  |  |  |
|-----------|---|--|---|---|
| Inorden   | Primero recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho.<br>En síntesis: hijo izquierdo → raíz → hijo derecho        | 2-5-1-3-7  | $4^5+6*7$   | $p-m-n-l$   |
| Preorden  | Primero visita la raíz; segundo recorre el subárbol izquierdo y por último va a subárbol derecho.<br>En síntesis: raíz → hijo izquierdo → hijo derecho          | 1-2-5-3-7  | $+ ^ 4 5 * 6 7$   | $l-m-p-n$   |
| PostOrden | Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y, por último, visita la raíz.<br>En síntesis: hijo izquierdo → hijo derecho → raíz | 5-2-7-3-1  | $4 5 ^ 6 7 * +$   | $p-n-m-l$   |

### Árboles de expresión

Son árboles binarios que se utilizan para almacenar en la memoria de una computadora expresiones lógicas, aritméticas o algebraicas. Este proceso lo realizan los compiladores de los lenguajes de programación.

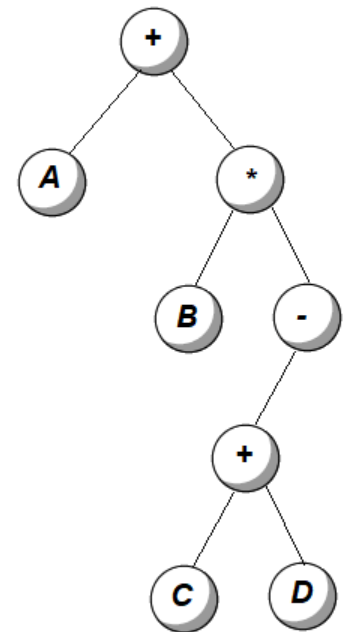
Las expresiones a pesar de estar escritas en *Inorden*, se evalúan en *PostOrden* usando la notación postfija (o notación polaca inversa) donde el operador aparece después de sus operandos; por ejemplo,  $AB/$  indica que A debe dividirse por B. Observe que la notación postfija tiene ventajas sobre la notación infija debido a que la notación postfija no necesita paréntesis ni tiene que predeterminedir un orden de prioridad para sus operadores (lógicos o aritméticos); es por tal razón que una expresión se evaluará sin ambigüedad.

El árbol de la derecha corresponde a la siguiente expresión en infijo:

$$A+B*(-(C+D))$$

Su respectivo recorrido en *PostOrden* sería:

$ABCD+-*+$



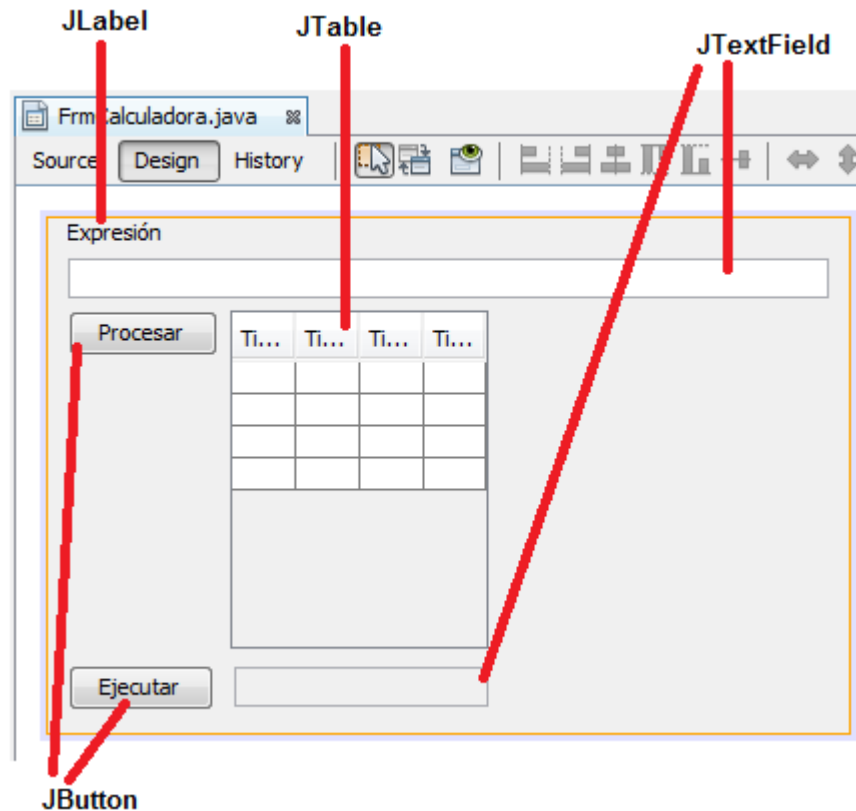
Para evaluar una expresión con recorrido *Inorden* pueden seguirse los siguientes pasos:

- 1) Transcribir la expresión de *Infijo* a *Postfijo*
- 2) Generar el árbol de expresión a partir de la expresión *Postfijo* obtenida. En efecto, el árbol se forma escribiendo como raíz el operador principal de la expresión y se escriben los operandos como subárboles izquierdo y derecho. Obsérvese que las hojas del árbol corresponderán a los operandos
- 3) Evaluar la expresión utilizando el árbol, dándole valores aritméticos o lógicos a los operandos

### Programa

Para la implementación del aplicativo en *Java* se debe comenzar con el diseño de un formulario como el siguiente (Utilizando el *IDE Netbeans*):





Este formulario corresponde a la clase *Calculadora* del anterior diagrama de clases. Al ser un formulario, hereda de la clase *JFrame*. La siguiente tabla relaciona los objetos a añadir con las propiedades cuyos valores deben ser cambiados:

| Tipo Control      | Nombre              | Otras Propiedades                   |
|-------------------|---------------------|-------------------------------------|
| <b>JLabel</b>     | <i>jLabel1</i>      | text = "Expresión"                  |
| <b>JButton</b>    | <i>btnProcesar</i>  | text = "Procesar"                   |
|                   | <i>btnEvaluar</i>   | text = "Ejecutar"<br>enabled= false |
| <b>JTextField</b> | <i>txtExpresion</i> | text = ""                           |
|                   | <i>txtResultado</i> | text = ""<br>enabled= false         |
| <b>JTable</b>     | <i>tbl</i>          |                                     |

De acuerdo al modelo de clases planteado, se deben programar las clases *Nodo*, *ArbolBinario* y *PostFijo* las cuales tendrá la funcionalidad para operar con esta forma de escribir las expresiones y hacer uso de la clase *Stack* (*Pila* en Inglés). A continuación se hará una breve descripción de esta última clase, la cual está en el paquete *java.util*:

| Stack   |
|---|
|   |
| push(): void<br>pop(): Object<br>peek(): Object<br>empty(): boolean |

- El método *push* permite apilar un dato en la pila.
- El método *pop()* permite desapilar un objeto de la pila
- El método *peek()* permite obtener el elemento que está en el tope de la pila (sin necesidad de desapilar)
- El método *empty()* devuelve un valor booleano que indica si la pila está vacía

Comenzando a editar la clase *Nodo*, estas es su estructura:

| Nodo  |
|---|
| valor: String<br>tipo: TipoOperando<br>izquierdo: Nodo<br>derecho: Nodo |
| obtenerValorNumerico()  |

Atributos:

- *valor* de tipo *String* y carácter público, almacena ya sea el nombre de una variable o el valor de una constante
- *tipo* es un enumerado que indica si el valor del nodo corresponde a una variable, constante numérica u otra cosa, como por ejemplo con los operadores.
- *Izquierdo* y *derecho* son instancias de esta misma clase y actuarán como apuntadores a los hijos

Métodos:

- *obtenerValorNumerico()* permite devolver en caso de que lo almacenado sea una constante numérica, su respectivo valor numérico.

Este es el código de la clase:

```
public class Nodo {

    String valor;
    Nodo izquierdo;
    Nodo derecho;
    TipoOperando tipo;

    public Nodo() {
        valor = "";
        izquierdo = null;
        derecho = null;
    }

    public Nodo(String valor, TipoOperando tipo) {
        this.valor = valor;
        this.tipo = tipo;
        izquierdo = null;
        derecho = null;
    }

    public double obtenerValorNumerico() {
        try {
            if (tipo == TipoOperando.CONSTANTE) {
                return Double.parseDouble(this.valor);
            }
        } catch (Exception ex) {
        }
        return 0;
    }
}
```

```
}
```

Y este es el enumerado:

```
public enum TipoOperando{  
    NINGUNO, VARIABLE, CONSTANTE  
}
```

Siguiendo con la clase *ArbolBinario*, la cual representará la expresión a ser evaluada, esta es su estructura:

| ArbolBinario  |
|---|
| <ul style="list-style-type: none"><li>raiz: Nodo</li><li>variables: String[]</li><li>valores: double[]</li></ul>  |
| <ul style="list-style-type: none"><li>mostrarPreorden(): String</li><li>mostrarInOrden(): String</li><li>mostrarPostOrden(): String</li><li>obtenerValorVariable(): double</li><li>obtenerValorNodo(): double</li><li>evaluarNodo(): double</li><li>evaluar(): double</li></ul> |

Atributos:

- *raiz* es un objeto de la clase *Nodo* y carácter privado, almacena el punto de entrada al árbol binario
- *variables* es una lista con los nombres únicos de las variables de la expresión
- *valores* es una lista con los valores numéricos que corresponden a cada nombre de variable único

Métodos:

- *mostrarPreorden()* permite de manera recursiva, obtener la expresión almacenada en el árbol binario en notación *Prefijo*
- *mostrarInOrden()* permite de manera recursiva, obtener la expresión almacenada en el árbol binario en notación *Infijo*
- *mostrarPostOrden()* permite de manera recursiva, obtener la expresión almacenada en el árbol binario en notación *Postfijo*
- *obtenerValorVariable()* permite buscar la variable en la lista y devolver el respectivo valor
- *obtenerValorNodo()* permite obtener el valor almacenado en el nodo si es un operando
- *evaluarNodo()* permite, de manera recursiva, calcular la operación definida por un nodo con operador y sus operandos hijos
- *evaluar()* calcula el valor total de la expresión almacenada en el árbol dándole unos valores a las variables

Este es el código de la clase:

```
import java.util.List;  
  
public class ArbolBinario {  
  
    private Nodo raiz;  
    private List<String> variables;  
    private List<Double> valores;  
  
    public ArbolBinario() {  
        this.raiz = null;  
    }  
  
    public ArbolBinario(Nodo raiz) {  
        this.raiz = raiz;  
    }  
}
```

```
private String mostrarPreOrden(Nodo n) {
    if (n != null) {
        return n.valor + " "
            + mostrarPreOrden(n.izquierdo) + " "
            + mostrarPreOrden(n.derecho);
    }
    return "";
}

private String mostrarInOrden(Nodo n) {
    if (n != null) {
        return mostrarInOrden(n.izquierdo) + " "
            + n.valor + " "
            + mostrarInOrden(n.derecho);
    }
    return "";
}

private String mostrarPostOrden(Nodo n) {
    if (n != null) {
        return mostrarPostOrden(n.izquierdo) + " "
            + mostrarPostOrden(n.derecho) + " "
            + n.valor;
    }
    return "";
}

public String mostrarPreOrden() {
    return this.mostrarPreOrden(this.raiz);
}

public String mostrarInOrden() {
    return this.mostrarInOrden(this.raiz);
}

public String mostrarPostOrden() {
    return this.mostrarPostOrden(this.raiz);
}

private double obtenerValorVariable(String variable) {
    double valor = 0;
    if (this.variables.size() > 0) {
        int p = this.variables.indexOf(variable);
        if (p >= 0) {
            valor = valores.get(p);
        }
    }
    return valor;
}

private double obtenerValorNodo(Nodo n) {
    return n.tipo == TipoOperando.CONSTANTE ? n.obtenerValorNumerico() :
    obtenerValorVariable(n.valor);
}

private double evaluarNodo(Nodo n) {
```

```

        if (n.izquierdo == null && n.derecho == null) {
            return obtenerValorNodo(n);
        } else {
            double operando1 = evaluarNodo(n.izquierdo);
            double operando2 = evaluarNodo(n.derecho);
            switch (n.valor) {
                case "+":
                    return operando1 + operando2;
                case "-":
                    return operando1 - operando2;
                case "*":
                    return operando1 * operando2;
                case "/":
                    return operando2 != 0 ? operando1 / operando2 : 0;
                case "%":
                    return operando2 != 0 ? operando1 % operando2 : 0;
                case "^":
                    return Math.pow(operando1, operando2);
            }
        }
        return 0;
    }

    public double evaluar(List<String> variables, List<Double> valores) {
        this.variables = variables;
        this.valores = valores;
        return this.evaluarNodo(this.raiz);
    }
}

```

En este código es importante observar estos detalles:

- La manera recursiva como se hacen los diferentes recorridos
- También la manera recursiva como se calcula la expresión completa haciendo uso del método recursivo *evaluarNodo()*
- La necesidad de pasarle al método evaluador de la expresión la lista de variables con sus respectivos valores

Ahora bien, la clase *PostFijo* se compondrá de las siguientes propiedades:

| PostFijo                     |
|------------------------------|
| expresionInfijo: String      |
| @expresionPostfijo: String   |
| @errorExpresion: String      |
| esLetra(): boolean           |
| esDigito(): boolean          |
| esOperador(): boolean        |
| esPredecesor(): boolean      |
| obtenerPostfijo(): String    |
| obtenerError(): String       |
| obtenerVariables(): String[] |
| obtenerArbol(): ArbolBinario |

- *expresionInfijo* de tipo *String* y carácter público, mediante la cual se define la expresión en notación *Infijo*.
- *expresionPostfijo()* de tipo *String* y carácter privado, que almacenará la expresión en notación *Postfija* resultante de convertir la anterior expresión en *Infijo*.
- *errorExpresion* de tipo *String* y carácter privado encargada de reportar errores en la conversión u obtención de variables de la expresión.

Y los siguientes métodos:

- El método *esLetra()* que permite evaluar si un carácter es letra o no. Utilizado para construir los nombres de variables.
- El método *esDigito()* que permite evaluar si un carácter es un dígito numérico o no. Utilizado para construir los valores constantes.
- El método *esOperador()* que permite evaluar si un carácter es un operador aritmético o paréntesis.
- El método *esPredecesor()* que permite verificar entre dos operadores si el primero precede al segundo en la jerarquía de operaciones aritméticas
- El método *obtenerPostfijo()* realiza la conversión a notación *Postfijo* de la expresión almacenada en la propiedad *expresionInfijo*. El resultado queda almacenado en la propiedad *expresionPostFijo* siempre y cuando no suceda un error en la conversión, en cuyo caso esta propiedad queda vacía y el mensaje de error queda en la propiedad *errorExpresion*.
- El método *obtenerError()* permite conocer el error resultante al realizar la conversión a notación *Postfija*. También se puede obtener el error al tratar de obtener las variables de la expresión.
- El método *obtenerVariables()* devuelve en un vector los nombres de las variables encontradas en la expresión
- El método *obtenerArbol()* devuelve el árbol binario de la expresión *Postfija* obtenida

El siguiente es el código completo de la clase en *Java*:

```
import java.util.*;

public class PostFijo {

    //Atributo que almacena la expresión en Infijo a ser convertida
    public static String expresionInfijo = "";
    //Almacena la expresion Postfijo obtenida
    private static String expresionPostfijo = "";
    //Almacena posible error al evaluar la expresion
    private static String errorExpresion = "";

    //Método que indica si un caracter es letra
    public static boolean esLetra(String dato) {
        if ((dato.compareTo("a") >= 0 && dato.compareTo("z") <= 0)
            || (dato.compareTo("A") >= 0 && dato.compareTo("Z") <= 0)) {
            return true;
        } else {
            return false;
        }
    }

    //Método que indica si un caracter es dígito
    public static boolean esDigito(String dato) {
        boolean d = false;
        if (dato.compareTo("0") >= 0 && dato.compareTo("9") <= 0) {
            d = true;
        }
        return d;
    }
}
```

```
//Método que indica si un caracter es operador
public static boolean esOperador(String dato) {
    boolean o = false;
    if (dato.equals("+") || dato.equals("-") || dato.equals("*") ||
dato.equals("/") || dato.equals("^")) {
        o = true;
    }
    return o;
}

//Método que indica si un operador es precedido por otro
public static boolean esPredecesor(String operador1, String operador2) {
    boolean p = false;
    if (operador1.equals("^")) {
        p = true;
    } else if (operador1.equals("/") || operador1.equals("*")) {
        if (!operador2.equals("^")) {
            p = true;
        }
    } else if (operador1.equals("-") || operador1.equals("+")) {
        if (operador2.equals("+") || operador2.equals("-")) {
            p = true;
        }
    }
    return p;
}

//Método que permite obtener una expresión Potfijo a partir de la
expresión Infijo
public static String obtenerPostfijo() {
    Stack p = new Stack();
    expresionPostfijo = "";
    boolean error = false;
    int parentesis = 0;
    int i = 0;
    int noOperador = 0;
    errorExpresion = "";
    //Recorrer cada uno de los caracteres
    while (i < expresionInfijo.length() && !error) {
        String caracter = expresionInfijo.substring(i, i + 1);
        if (caracter.equals("(")) {
            noOperador = 1;
            p.push(caracter);
            parentesis++;
        } else if (caracter.equals(")")) {
            noOperador = 2;
            if (parentesis == 0) {
                error = true;
                errorExpresion = "Hace falta parentesis izquierdo";
            } else {
                parentesis--;
                caracter = (String) p.peek();
                while (!p.empty() && !caracter.equals("(")) {
                    expresionPostfijo += " " + p.pop();
                    caracter = (String) p.peek();
                }
            }
        }
    }
}
```

```
        p.pop();
    }
    } else if (esOperador(caracter)) {
        if (noOperador < 2) {
            error = true;
            errorExpresion = "Hace falta operando antes de " +
caracter;
        } else {
            noOperador = 0;
            expresionPostfijo = expresionPostfijo + " ";
            while (!p.empty() && esPredecesor((String) p.peek(),
caracter)) {
                expresionPostfijo = expresionPostfijo + p.pop();
            }
            p.push(caracter);
        }
    } else if (esLetra(caracter) || esDigito(caracter)) {
        noOperador = 3;
        expresionPostfijo = expresionPostfijo + caracter;
    } else {
        error = true;
        errorExpresion = "Simbolo '" + caracter + "' indefinido ";
    }
    i++;
}
//Verificar errores
if (parentesis > 0) {
    errorExpresion = "Error convirtiendo: Hace falta parentesis
derecho";
    expresionPostfijo = "";
} else if (error || i == 0 || noOperador == 0) {
    errorExpresion = "Error convirtiendo: " + errorExpresion;
    expresionPostfijo = "";
} else {
    //Construir la expresión POSTFIJO
    expresionPostfijo = expresionPostfijo + " ";
    while (!p.empty()) {
        expresionPostfijo = expresionPostfijo + (String) p.pop();
    }
}
return expresionPostfijo;
}

//Metodo que devuelve el ultimo mensaje de error
public static String obtenerError() {
    return errorExpresion;
}

//Metodo para obtener los nombres de las variables que hay en la
expresion
public static List<String> obtenerVariables() {
    List<String> operandos = new ArrayList<>();
    int i = 0;
    int tipoOperando = 0;

    String texto = "";
```



```
boolean error = false;
errorExpresion = "";
while (i < expresionPostfijo.length() && !error) {
    String caracter = expresionPostfijo.substring(i, i + 1);
    if (PostFijo.esLetra(caracter) && tipoOperando == 2) {
        error = true;
    } else if ((PostFijo.esLetra(caracter) && tipoOperando < 2) ||
(PostFijo.esDigito(caracter) && tipoOperando == 1)) {
        tipoOperando = 1;
        texto = texto + caracter;
    } else if (PostFijo.esDigito(caracter) && tipoOperando != 1) {
        tipoOperando = 2;
        texto = texto + caracter;
    } else if (caracter.equals(" ") && tipoOperando == 1) {
        if (!operandos.contains(texto)) {
            operandos.add(texto);
        }
        texto = "";
        tipoOperando = 0;
    } else if (caracter.equals(" ") && tipoOperando == 2) {
        texto = "";
        tipoOperando = 0;
    }
    i++;
}
if (!error) {
    return operandos;
} else {
    errorExpresion = "Error obteniendo variables";
    return null;
}
}

public static ArbolBinario obtenerArbol() {
    Stack p = new Stack();

    int i = 0;
    TipoOperando tipo = TipoOperando.NINGUNO;

    String texto = "";
    boolean error = false;
    errorExpresion = "";
    while (i < expresionPostfijo.length() && errorExpresion.equals("")) {
        String caracter = expresionPostfijo.substring(i, i + 1);
        if (PostFijo.esLetra(caracter) && tipo == TipoOperando.CONSTANTE)
        {
            errorExpresion = "Una constante numérica no puede tener
letras";
        } else if ((PostFijo.esLetra(caracter) && tipo !=
TipoOperando.CONSTANTE)
|| (PostFijo.esDigito(caracter) && tipo ==
TipoOperando.VARIABLE)) {
            tipo = TipoOperando.VARIABLE;
            texto = texto + caracter;
        } else if (PostFijo.esDigito(caracter) && tipo !=
TipoOperando.VARIABLE) {
```

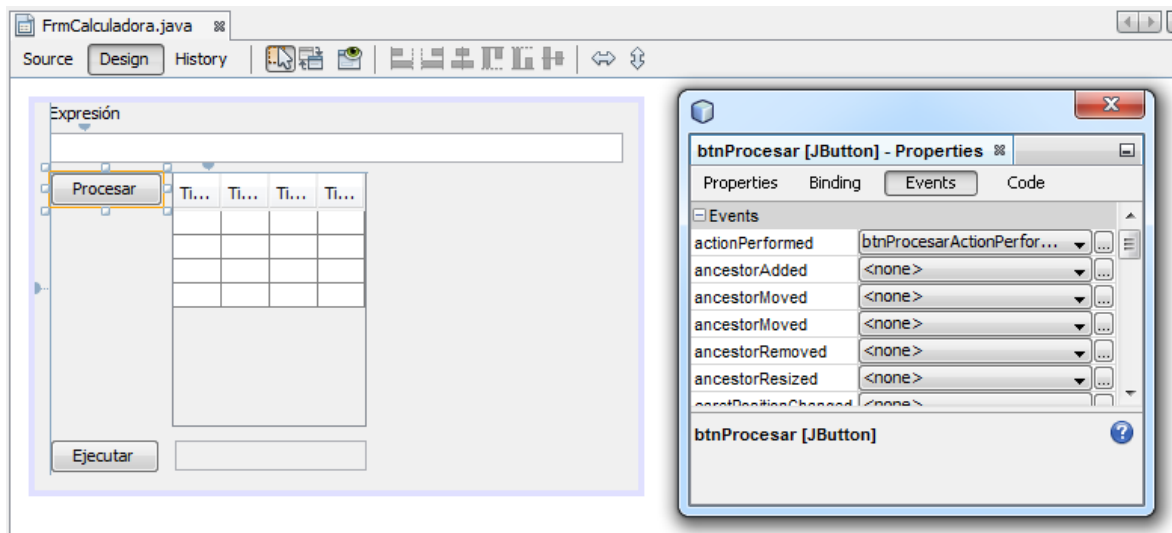
```
        tipo = TipoOperando.CONSTANTE;
        texto = texto + caracter;
    } else if (caracter.equals(" ") && tipo != TipoOperando.NINGUNO)
    {
        //Se acaba de obtener una constante o variable
        Nodo nOperando = new Nodo(texto, tipo);
        p.push(nOperando);
        texto = "";
        tipo = TipoOperando.NINGUNO;
    } else {
        caracter = expresionPostfijo.substring(i, i + 1);
        if (PostFijo.esOperador(caracter)) {
            Nodo nOperador = new Nodo(caracter,
TipoOperando.NINGUNO);
            Nodo nDerecho = (Nodo) p.pop();
            Nodo nIzquierdo= (Nodo) p.pop();
            nOperador.izquierdo = nIzquierdo;
            nOperador.derecho = nDerecho;
            p.push(nOperador);
        }
    }
    i++;
}
return errorExpresion.equals("") ? new ArbolBinario((Nodo) p.pop()) :
null;
}
}
```

En este código es importante observar estos detalles:

- Se tienen 3 métodos para identificar el tipo de dato que se evalúa en la expresión: *esCaracter()*, *esDigito()* y *esOperador()*.
- La variable *paréntesis* permite contar cuantas parejas de paréntesis se abren.
- La variable *noOperador* es un *interruptor* que se “enciende” cuando se encuentra un operando o un paréntesis en la expresión. Se “apaga” cuando se logra concretar una operación (cuando cada operador tiene sus dos operandos).
- La variable *caracter* permite almacenar uno a uno los caracteres de la cadena de texto con la expresión. Dependiendo del valor permite determinar qué hacer, incluso si debe apilarse.
- La variable *p* corresponde a la pila que permitirá apilar los operandos y los paréntesis izquierdos
- También otra pila permite almacenar los nodos que permiten construir el árbol binario de la expresión

Ahora bien, para continuar con la codificación, se deben programar los métodos *procesar()* e *evaluar()* de la clase *Juego* y que corresponderán a los eventos de los botones de comando agregados al formulario.

La conversión de la expresión *Infijo* a *Postfijo* para poder obtener las variables que componen la expresión se hará mediante un método que responde al evento ***actionPerformed*** del botón de comando ***btnProcesar***:



El código respectivo sería el siguiente:

```
private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    //Asignar la expresion INFIJO leida
    PostFijo.expresionInfiJo = txtExpresion.getText();
    //Obtener la expresion POSTFIJO respectiva
    PostFijo.obtenerPostfijo();
    //Verificar que no haya sucedido error
    if (PostFijo.obtenerError().equals("")) {
        //Obtener los nombres de variables que hay en la expresión
        variables = PostFijo.obtenerVariables();
        if (variables != null) {
            //Mostrar las variables en una tabla
            String[][] datos = new String[variables.size()][2];
            for (int i = 0; i < variables.size(); i++) {
                datos[i][0] = variables.get(i);
            }

            String[] columnas = new String[]{"Variable", "Valor"};
            DefaultTableModel dtm = new DefaultTableModel(datos,
columnas);

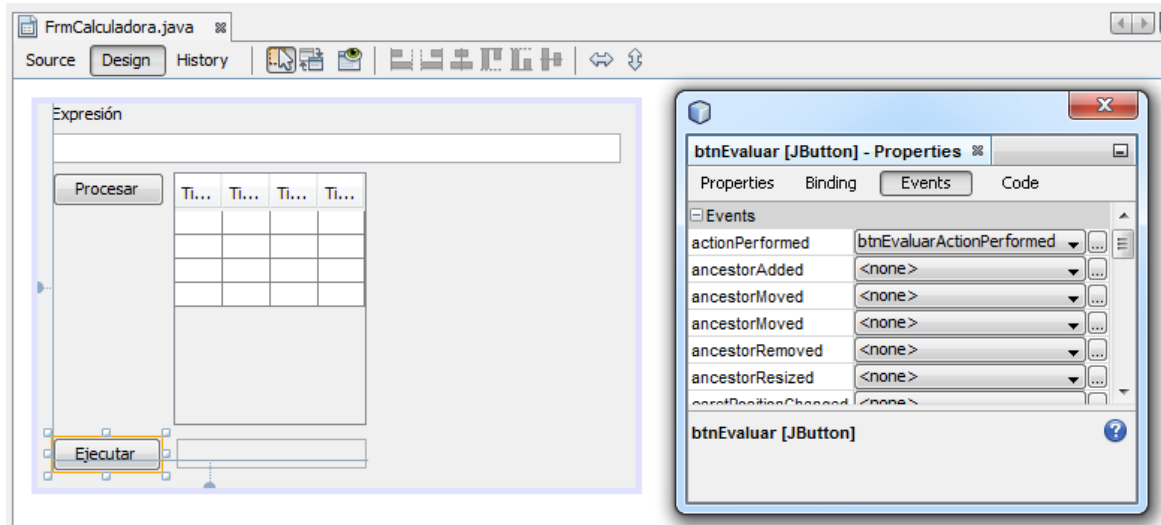
            tbl.setModel(dtm);
        }
    } else {
        JOptionPane.showMessageDialog(new JFrame(),
PostFijo.obtenerError());
    }
}
```

En este código, en primer lugar, se toma la expresión que el usuario digita en la caja de texto y se le asigna a la propiedad *expresionInfiJo* de la clase *PostFijo*. Luego se convierte a la respectiva expresión *PostFijo*. Una vez verificado que no haya error en la conversión, se procede a obtener las variables encontradas para poder definir el modelo de datos del objeto *JTable* mediante el cual el usuario digitará los respectivos valores.

Para almacenar los nombres de las variables, se debió declarar globalmente un vector así:

```
List<String> variables;
```

El cálculo del valor final de la expresión se hará mediante el método que responde al evento **actionPerformed** del botón de comando **btnEvaluar**:



El código respectivo sería el siguiente:

```
private void btnEvaluarActionPerformed(java.awt.event.ActionEvent evt) {
    ArbolBinario ab = PostFijo.obtenerArbol();

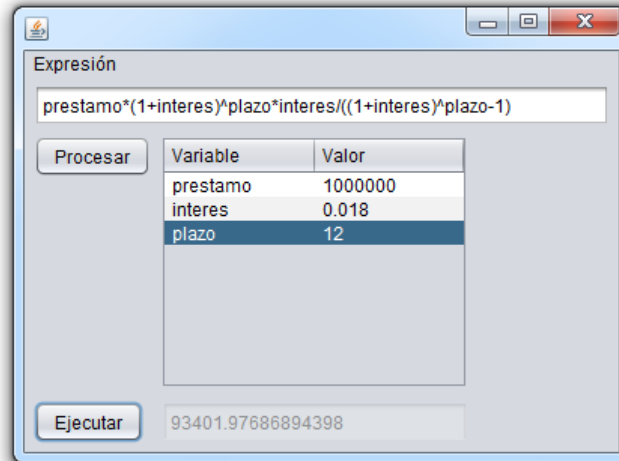
    //Declarar el vector que almacenará los valores de las variables
    List<Double> valores = new ArrayList<>();
    //Obtener los datos de la tabla
    DefaultTableModel dtm = (DefaultTableModel) tbl.getModel();
    //Obtener los valores numéricos de las variables
    for (int i = 0; i < variables.size(); i++) {
        try {
            valores.add(Double.parseDouble(dtm.getValueAt(i, 1).toString()));
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(new JFrame(),
                "Falta la variable " + dtm.getValueAt(i, 0).toString());
            return;
        }
    }
    txtResultado.setText(String.valueOf(ab.evaluar(variables, valores)));
}
```

Este código, en primer lugar, obtiene el árbol binario de la expresión que va a ser evaluada. Seguido, obtiene los valores para las variables que se encontraron en la expresión, los cuales el usuario digita en la segunda columna del objeto *JTable*. Por último y siempre y cuando se hayan obtenido todos los valores, se muestra el valor obtenido al evaluar la expresión en *Postfijo*.

Ahora es posible ejecutar la aplicación y verificar su funcionamiento. Por ejemplo, podemos utilizar la fórmula para el cálculo del valor de una cuota de un préstamo dado el monto, la tasa de interés y número de períodos:

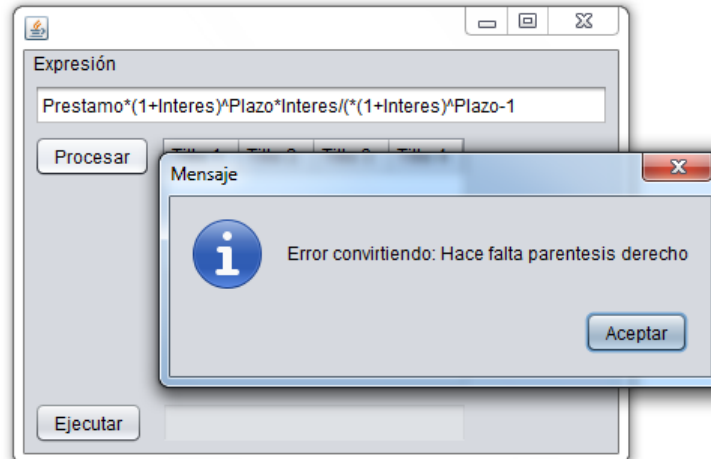
$$Cuota = Prestamo \frac{(1 + Interes)^{Plazo} Interes}{(1 + Interes)^{Plazo} - 1}$$

Cuya ejecución luciría así:



Ahora bien, si el usuario se equivoca:

- Faltando un paréntesis:



- Faltando un operador:



2. Implementar un graficador de funciones de una sola variable. El usuario debe digitar la función que se graficará.

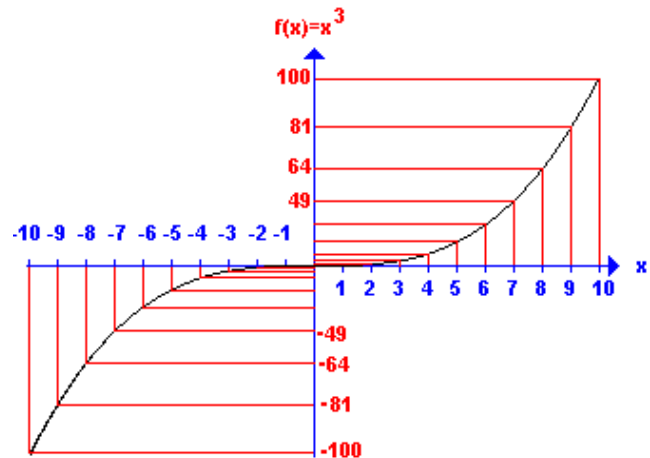
### Gráfica de una función

Una función matemática permite calcular un valor como resultado de aplicar una expresión a una variable. Por ejemplo, dada la función  $f(x)=x^3$ , para cada valor de  $x$ , donde  $x$  es un número real,  $f(x)$  corresponde al valor de  $x$  elevado al cubo. Con base en esto, se puede obtener la siguiente tabla:

| X    | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
|------|-----|----|----|----|----|----|----|----|----|----|---|---|---|---|----|----|----|----|----|----|-----|
| f(x) | 100 | 81 | 64 | 49 | 36 | 25 | 16 | 9  | 4  | 1  | 0 | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 | 81 | 100 |

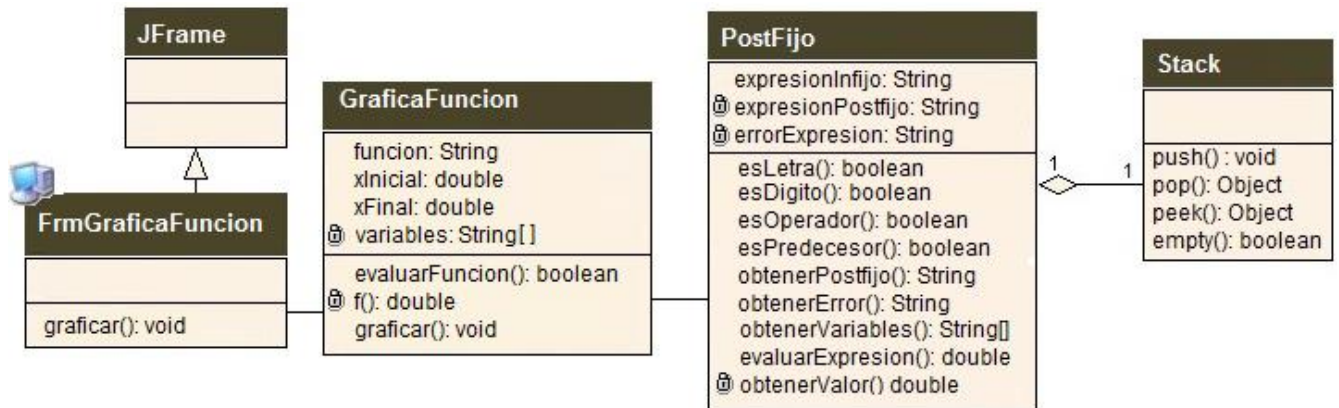
Esta tabla permite definir una gráfica en un plano cartesiano de dos ejes, uno para los valores de la variable  $x$  (horizontal) y otro para los valores de la función (vertical). La gráfica se realiza trazando líneas entre dos puntos cada uno correspondiente a la pareja compuesta por un valor de la variable y el respectivo valor de la función ( $x, f(x)$ ). Por ejemplo:

Punto 1= $(-10,-100)$  al Punto 2= $(-9,-81)$   
 Punto 2= $(-9,-81)$  al Punto 2= $(-8,-64)$



R/

- El modelado bajo el paradigma Orientado a Objetos se ilustra en el siguiente diagrama de clases:



## Programa

Para la implementación del aplicativo en *Java* se debe comenzar con el diseño de un formulario como el siguiente (Utilizando el *IDE Netbeans*):



Este formulario corresponde a la clase *FrmGraficaFuncion* del anterior diagrama de clases. Al ser un formulario, hereda de la clase *JFrame*. La siguiente tabla relaciona los objetos a añadir con las propiedades cuyos valores deben ser cambiados:

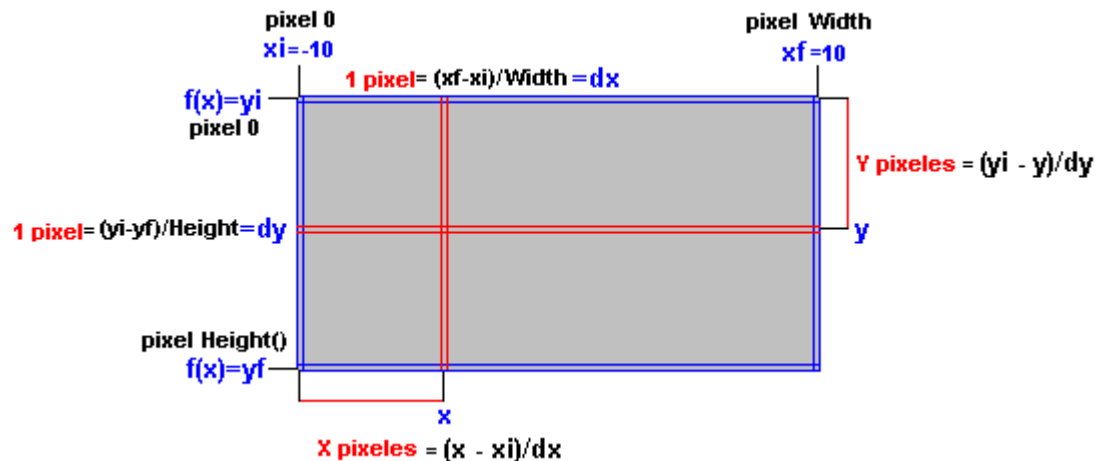
| Tipo Control  | Nombre         | Otras Propiedades  |
|---------------|----------------|--------------------|
| <b>JLabel</b> | <i>jLabel1</i> | text = "Función"   |
|               | <i>jLabel2</i> | text = "X inicial" |
|               | <i>jLabel3</i> | text = "X Final"   |

|                   |                    |                                 |
|-------------------|--------------------|---------------------------------|
| <b>JButton</b>    | <i>btnGraficar</i> | text = "Graficar"               |
| <b>JTextField</b> | <i>txtFuncion</i>  | text = ""                       |
|                   | <i>txtXInicial</i> | text = ""                       |
|                   | <i>txtXFinal</i>   | text = ""                       |
| <b>JPanel</b>     | <i>jPanel1</i>     |                                 |
|                   | <i>pnlGrafica</i>  | Background = [0, 0, 0]<br>Negro |

### Coordenadas Cartesianas vs Coordenadas de Píxeles

Para graficar en computadora se debe tener en cuenta lo siguiente:

- El uso de un objeto de la clase **Graphics**. Esta es una clase que permite trazar gráficas en algunos de los objetos contenedores tales como *JFrame* y *JPanel*. Dentro de los posibles trazos se tiene elipses, rectángulos, polígonos y líneas, así como también textos. En este ejercicio sólo se utilizará el método *drawLine()* para dibujar líneas. Para poder realizar el trazo, se requiere una instancia de la clase *Graphics*. El asunto es que no se puede instanciar directamente, sino a partir del método *getGraphics()* de objetos *JFrame* o *JPanel*.
- Las coordenadas de los trazos en el objeto de despliegue (Por ejemplo un *JPanel*), no corresponden a las coordenadas del plano cartesiano, por lo que hay que hacer los respectivos cálculos para hacer la adaptación.



Estos cálculos comprenden el manejo de las siguientes variables:

- xi**: Valor inicial de x. Corresponde al pixel 0
- xf**: Valor final de x. Corresponde al pixel del tamaño del objeto de despliegue, el cual se obtiene mediante la propiedad *getSize().width*.
- yi**: Valor máximo de la función. Corresponde al pixel 0
- yf**: Valor mínimo de la función. Corresponde al pixel del tamaño del alto del control de despliegue dado por la propiedad *getSize().height*.
- dx**: Valor de incremento de x para un pixel. Equivalente a  $(xf - xi) / Width()$ .
- dy**: Valor de incremento de la función para un pixel. Equivalente a  $(yi - yf) / Height()$

Para un valor de x determinado, el pixel correspondiente sería:  $(x - xi) / dx$

Para un valor de y determinado, el pixel correspondiente sería:  $(yi - y) / dy$



La gráfica de la función se hará trazando pequeñas líneas con una distancia horizontal de sólo 1 pixel lo cual dará el efecto de curva de la función. Para ello se calculan dos valores de la variable  $x$  contiguos (con una diferencia de  $dx$ ) y sus respectivos valores  $f(x)$  y se traza una línea. Seguido, se traza la línea siguiente y así sucesivamente de izquierda a derecha hasta llegar al último píxel que corresponde al valor  $xf$ .

Ahora bien, de acuerdo al modelo de clases planteado, se debe incluir la clase *PostFijo* la cual servirá tanto para verificarla expresión de la función como para ejecutarla en cada uno de los valores de la variable. Está clase ya fue descrita con detalle en el punto anterior. Se debe tener en cuenta que ella hace uso de la clase *Stack* (*Pila* en Inglés).

Queda entonces por describir la clase *GraficaFuncion* la cual tendrá la funcionalidad para realizar la gráfica de la función. Se compondrá de las siguientes propiedades:

| GraficaFuncion    |          |
|-------------------|----------|
| funcion:          | String   |
| xInicial:         | double   |
| xFinal:           | double   |
| @ variables:      | String[] |
| evaluarFuncion(): | boolean  |
| @ f():            | double   |
| graficar():       | void     |

- *funcion* de tipo *String* y carácter público, mediante la cual se define la función a graficar
- *xInicial* de tipo *double* que almacena el valor a partir del cual se graficará la función
- *xFinal* de tipo *double* que almacena el valor hasta donde se graficará la función
- *variables* un vector de *String* y carácter privado encargado de almacenar los nombres de las variables encontradas. Para poder graficar, sólo se debe encontrar una variable.

Y los siguientes métodos:

- El método *evaluarFuncion()* indica si la función ingresada está correctamente escrita y se compone de una sola variable.
- El método *f()* se encarga de evaluar la función para un valor dado de la variable. Es de carácter privado.
- El método *graficar()* realiza los trazos que permiten visualizar la gráfica de la función en un objeto *JPanel*. Utiliza la estrategia antes descrita.

El código respectivo en *Java* sería:

```
import java.awt.*;
import javax.swing.*;

public class GraficaFuncion {

    public static String funcion;
    //Límites de X
    public static double xInicial = -100;
    public static double xFinal = 100;
    private static String[] variables;

    public static boolean evaluarFuncion() {
        //Tomar la función
        PostFijo.expresionInfijo = funcion;
        //Obtener la expresion postfijo
        PostFijo.obtenerPostfijo();
        //Verificar si no hubo errores de conversión
        if (PostFijo.obtenerError().equals("")) {
```

```
//Verificar que haya una sola variable
variables = PostFijo.obtenerVariables();
if (PostFijo.obtenerError().equals("") && variables.length == 1)
{
    return true;
} else {
    JOptionPane.showMessageDialog(new JFrame(),
        PostFijo.obtenerError());
    return false;
}
} else {
    JOptionPane.showMessageDialog(new JFrame(),
        PostFijo.obtenerError());
    return false;
}
}

private static double f(double x) {
    double[] valores = new double[1];
    valores[0] = x;
    return PostFijo.evaluarExpresion(variables, valores);
}

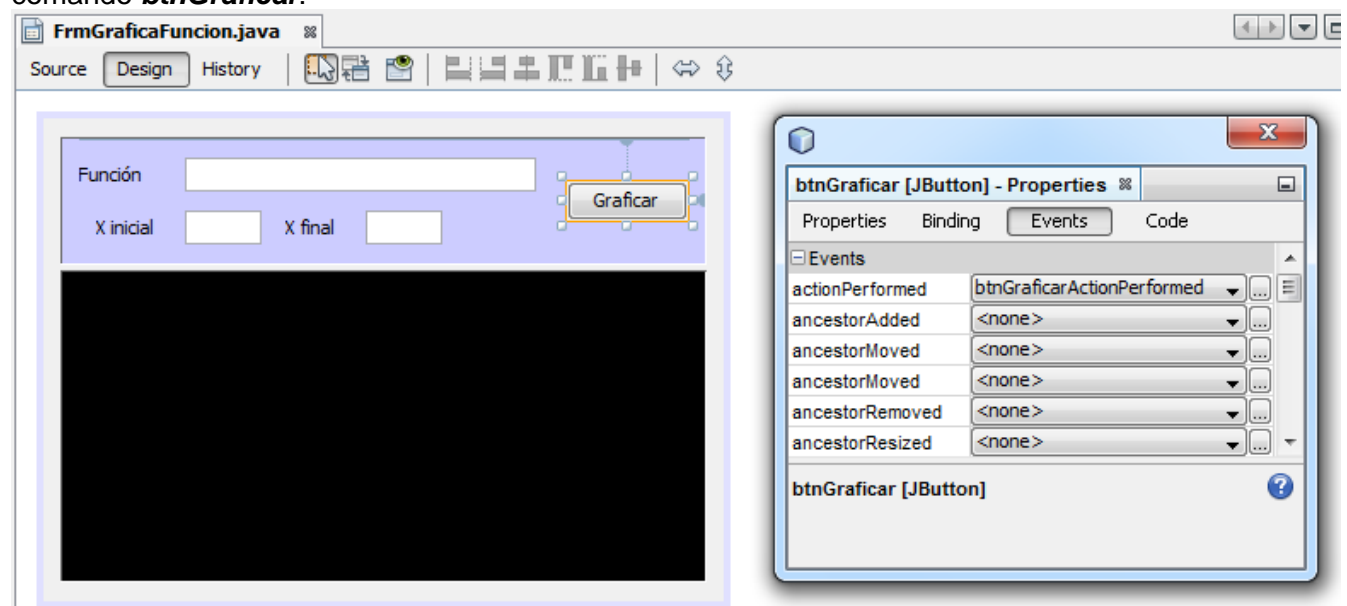
public static void graficar(JPanel pnl) {
    Graphics g = pnl.getGraphics();
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, pnl.getSize().width, pnl.getSize().height);
    if (evaluarFuncion()) {
        //Incremento de X por pixel
        double dx = (xFinal - xInicial) / pnl.getSize().width;
        //Máximo y mínimo de Y
        double yMaximo = f(xInicial);
        double yMinimo = f(xInicial);
        for (double x = xInicial; x <= xFinal; x += dx) {
            double fX = f(x);
            if (fX > yMaximo) {
                yMaximo = fX;
            }
            if (fX < yMinimo) {
                yMinimo = fX;
            }
        }
        //Incremento de Y por Pixel
        double dy = (yMaximo - yMinimo) / pnl.getSize().height;
        //Color para los ejes
        g.setColor(Color.WHITE);
        //Eje Vertical
        int px;
        if ((xInicial <= 0) && (0 <= xFinal)) {
            px = (int) (-xInicial / dx);
            g.drawLine(px, 0, px, pnl.getSize().height);
        }
        //Eje Horizontal
        int py;
        if ((yMinimo <= 0) && (0 <= yMaximo)) {
            py = (int) (-yMinimo / dy);
        }
    }
}
```

```

        g.drawLine(0, py, pnl.getSize().width, py);
    }
    g.setColor(Color.ORANGE);
    //Valores de x y f(x) iniciales
    double x1 = xInicial;
    double y1 = f(x1);
    //Ubicación de f(x) inicial en el sistema de pixeles
    int py1 = (int) ((yMaximo - y1) / dy);
    double x2, y2;
    int py2;
    //Recorrer todos los puntos del eje horizontal
    for (px = 0; px < pnl.getSize().width; px++) {
        //Valores de x y f(x) siguientes
        x2 = x1 + dx;
        y2 = f(x2);
        //Ubicación de f(x) siguiente en el sistema de pixeles
        py2 = (int) ((yMaximo - y2) / dy);
        //Dibujar la porción de la función
        g.drawLine(px, py1, px + 1, py2);
        //Preparar las coordenadas iniciales
        x1 = x2;
        y1 = y2;
        py1 = py2;
    }
}
}
}

```

Para terminar, se debe programar el método *graficar()* de la clase *FrmGraficaFuncion* y que corresponderá al evento del botón de comando agregado al formulario. La grafica de la función se hará mediante un método que responde al evento **actionPerformed** del botón de comando **btnGraficar**.

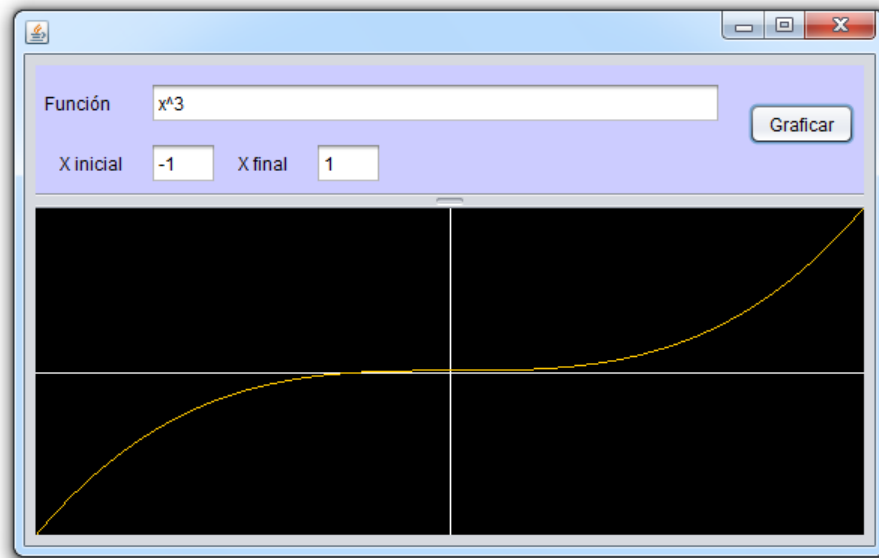


El código respectivo sería el siguiente:

```
private void btnGraficarActionPerformed(java.awt.event.ActionEvent evt) {
```

```
try {
    GraficaFuncion.funcion = txtFuncion.getText();
    GraficaFuncion.xInicial = Double.parseDouble(txtXInicial.getText());
    GraficaFuncion.xFinal = Double.parseDouble(txtXFinal.getText());
    GraficaFuncion.graficar(pnlGrafica);
} catch (Exception ex) {
    JOptionPane.showMessageDialog(new JFrame(),
        "No se pudo realizar la gráfica");
}
```

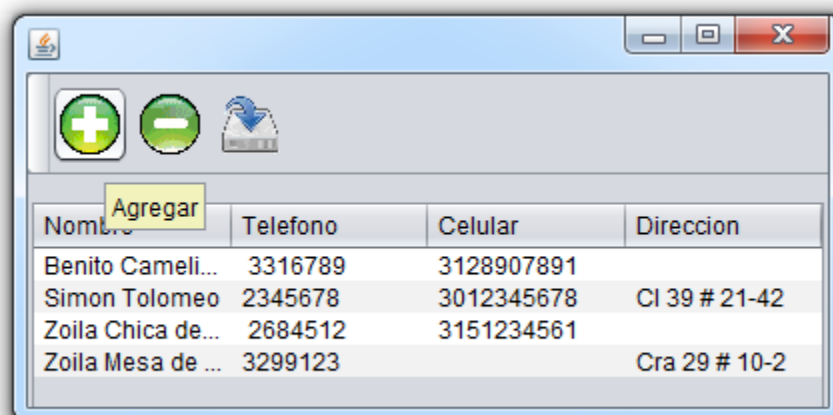
La ejecución luciría así para la función cúbica:



3. Elaborar una aplicación para gestionar una lista de contactos de manera ordenada por el nombre. Se deben permitir las siguientes funciones:
- Cargar ordenadamente desde archivo
  - Listar
  - Agregar
  - Modificar
  - Eliminar
  - Guardar en archivo

La información deberá almacenarse en un archivo plano y cargarse en memoria en un árbol binario de búsqueda

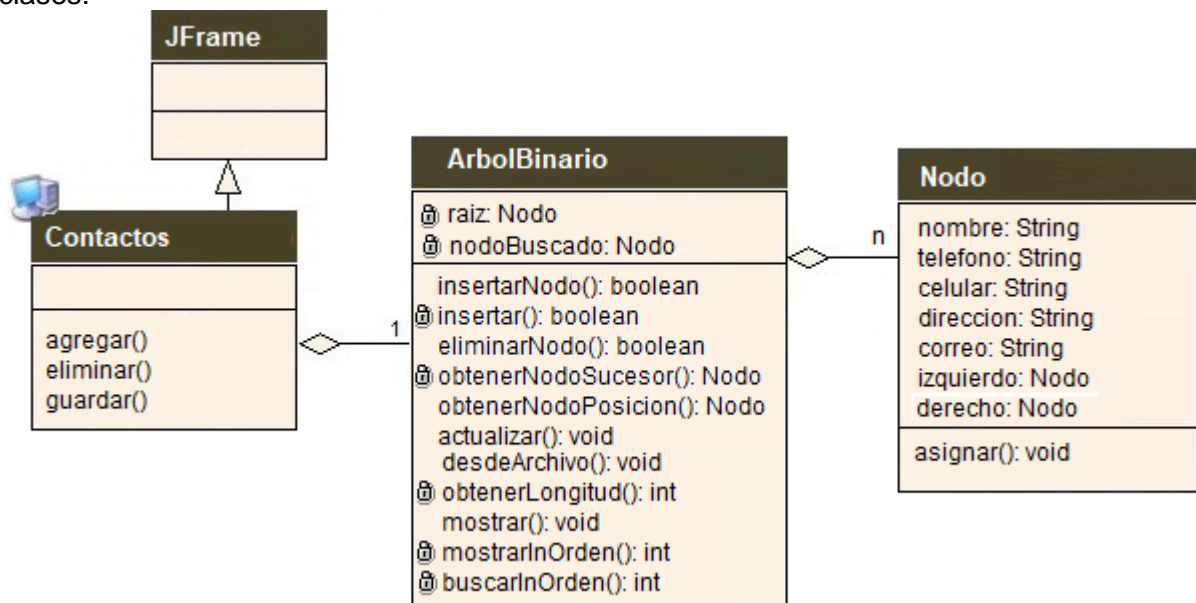
La ejecución podría lucir así:



| Nombre            | Telefono | Celular    | Direccion     |
|-------------------|----------|------------|---------------|
| Benito Cameli...  | 3316789  | 3128907891 |               |
| Simon Tolomeo     | 2345678  | 3012345678 | Cl 39 # 21-42 |
| Zoila Chica de... | 2684512  | 3151234561 |               |
| Zoila Mesa de ... | 3299123  |            | Cra 29 # 10-2 |

R/

- El modelado bajo el paradigma Orientado a Objetos se ilustra en el siguiente diagrama de clases:

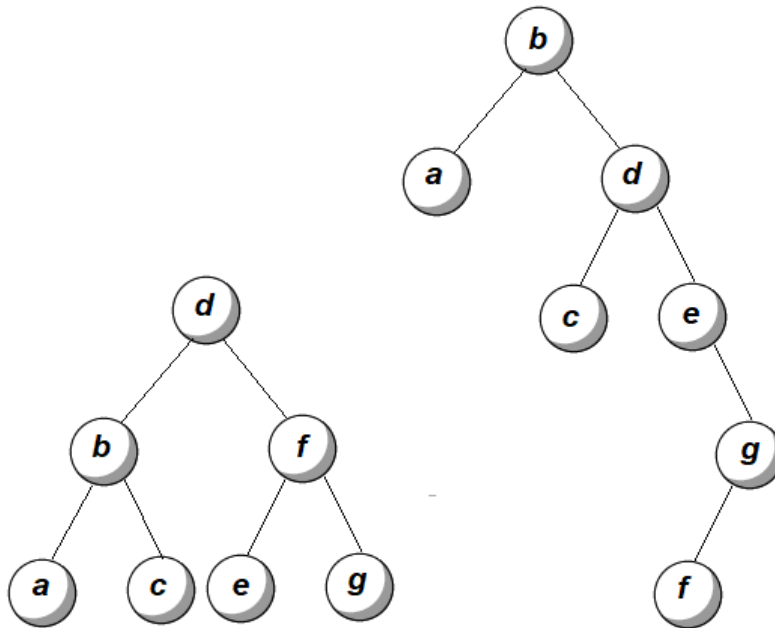


Para comprender este diagrama y el ejercicio, es importante tener en cuenta los siguientes fundamentos:

### Arboles Binarios de Búsqueda

Un árbol binario de búsqueda (ABB) es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo  $x$  son menores que el elemento almacenado en  $x$ , y todos los elementos almacenados en el subárbol derecho de  $x$  son mayores que el elemento almacenado en  $x$ .

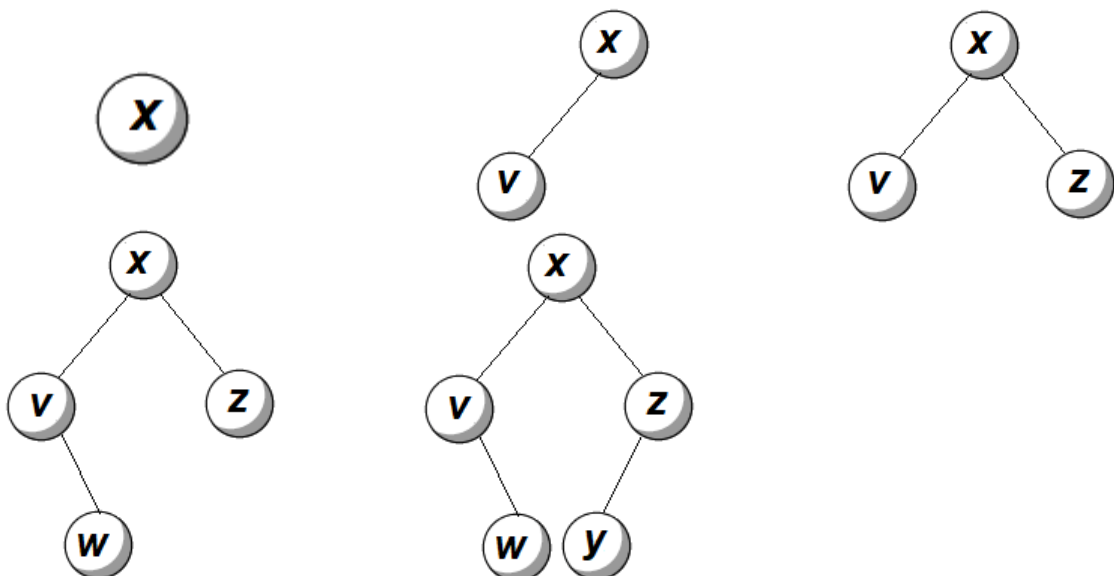
A continuación, se muestran dos ABB construidos con base en el mismo conjunto de datos:



Se puede observar la interesante propiedad de que si se listan los nodos del *ABB* en *inOrden* se da la lista de nodos ordenada.

El procedimiento de construcción de un *ABB* se basa en un procedimiento de inserción que va añadiendo elementos al árbol. Tal procedimiento comenzaría mirando si el árbol es vacío y de ser así se crearía un nuevo nodo para el elemento insertado devolviendo como resultado un puntero a ese nodo. Si el árbol no está vacío, se busca el elemento a insertar (como se haría en otro procedimiento que verifica la pertenencia) y al encontrar un puntero nulo se reemplaza por el puntero al nodo nuevo que contenga el elemento a insertar.

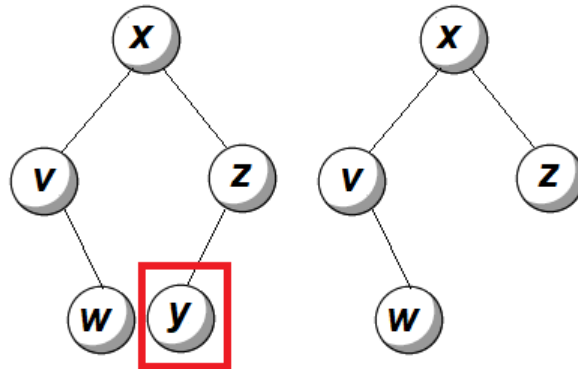
Por ejemplo supóngase que se desea construir un *ABB* a partir del conjunto de datos  $\{ x, v, z, w, y \}$  aplicando reiteradamente el proceso de inserción. El resultado es el que se muestra en las siguientes figuras:



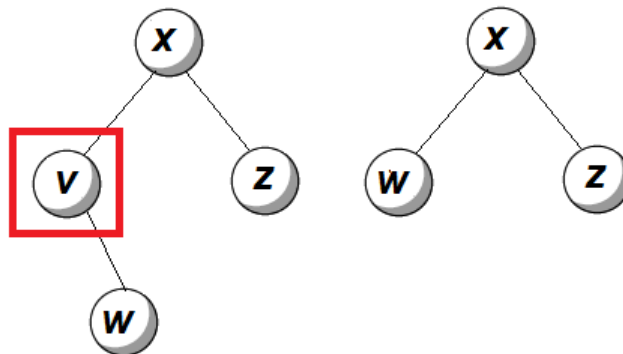
La operación de eliminación es la más complicada para los *ABB*. En primer lugar, para eliminar un nodo, hay que localizarlo en la estructura del árbol. Una vez hecho esto, se tendrá que actuar de distinta manera para eliminarlo dependiendo del número de hijos que tenga.

Básicamente se encuentran estas tres situaciones:

- Que el nodo no tenga hijos, es una hoja: Sencillamente se elimina el nodo y se hace nula la referencia que tenía el padre apuntando a dicho nodo

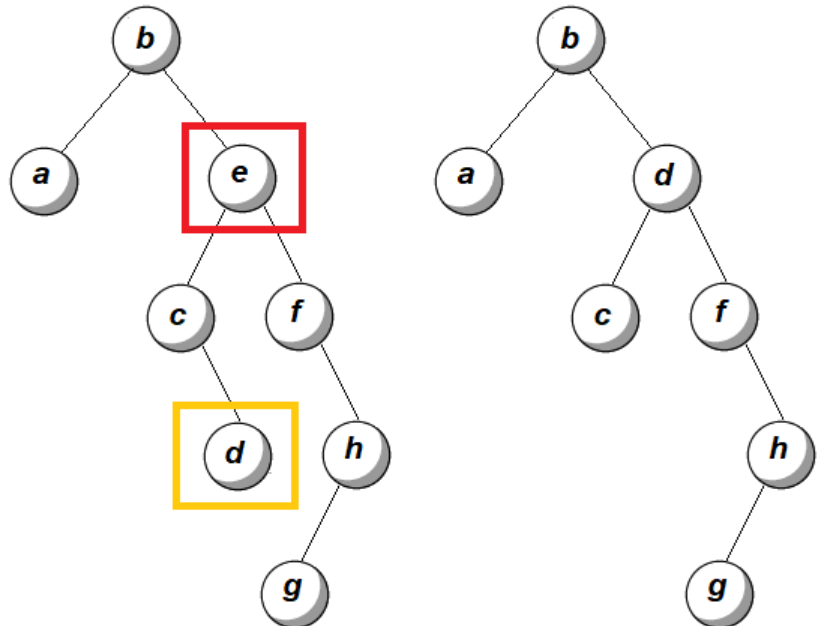


- Que tenga 1 hijo: Se debe hacer que el nodo padre del nodo a eliminar, apunte al único hijo que tiene el nodo a eliminar, y luego se elimina el nodo

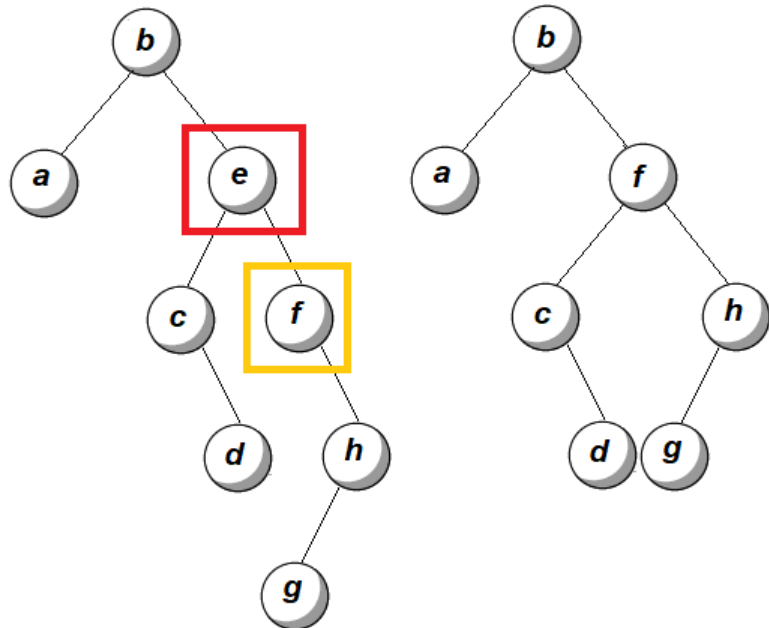


- Que tenga 2 hijos: Este es el caso más complejo. Al eliminar el nodo, ¿Qué nodo de sus dos subárboles se debe promocionar al hueco que ha dejado el nodo eliminado? Se tienen dos opciones:

1. Seleccionar del subárbol izquierdo el nodo que ocupara el sitio del nodo eliminado. Se debe buscar el nodo de mayor valor de todo el subárbol izquierdo, que debe ser el que se encuentre más a la derecha



2. Seleccionar del subárbol derecho el nodo que ocupara el sitio del nodo eliminado. Se debe buscar el nodo de menor valor de todo el subárbol derecho, que debe ser el que se encuentre más a la izquierda



Se comienza a desarrollar el aplicativo editando la clase *Nodo*, cuya estructura es la siguiente:

| Nodo   |
|--|
| nombre: String<br>telefono: String<br>celular: String<br>direccion: String<br>correo: String<br>izquierdo: Nodo<br>derecho: Nodo |
| asignar(): void  |

Atributos:

- *nombre* de tipo *String* y carácter público, almacena el nombre del contacto y es el dato clave para el ordenamiento y búsqueda
- *telefono* de tipo *String* almacena el número de teléfono fijo del contacto
- *celular* de tipo *String* almacena el número de teléfono móvil del contacto
- *direccion* de tipo *String* almacena la dirección física del contacto



- *correo* de tipo *String* almacena el correo electrónico del contacto
- *Izquierdo* y *derecho* son instancias de esta misma clase y actuarán como apuntadores a los hijos

Métodos:

- *asignar()* permite cambiar los valores de los datos almacenados.

Este es el código de la clase:








```
public class Nodo {

    //datos a almacenar
    String nombre;
    String telefono;
    String celular;
    String direccion;
    String correo;
    //enlaces
    Nodo izquierdo;
    Nodo derecho;

    //guardar la informacion y hacer del nodo una hoja
    public Nodo(String nombre,
                String telefono,
                String celular,
                String direccion,
                String correo) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.celular = celular;
        this.direccion = direccion;
        this.correo = correo;
        //el nodo no tiene hijos
        izquierdo = derecho = null;
    }

    //cambiar los valores almacenados
    public void asignar(String nombre,
                        String telefono,
                        String celular,
                        String direccion,
                        String correo)
    {
        this.nombre = nombre;
        this.telefono = telefono;
        this.celular = celular;
        this.direccion = direccion;
        this.correo = correo;
    } //asignar
}
```

Se continua el desarrollo el aplicativo editando la clase *ArbolBinario* que almacenará el *ABB* con los datos de la lista de contactos, y cuya estructura es la siguiente:

| ArbolBinario  |                             |
|---|-----------------------------|
|  | raíz: Nodo                  |
|  | nodoBuscado: Nodo           |
|   | insertarNodo(): boolean     |
|  | insertar(): boolean         |
|   | eliminarNodo(): boolean     |
|  | obtenerNodoSucesor(): Nodo  |
|   | obtenerNodoPosicion(): Nodo |
|   | actualizar(): void          |
|   | desdeArchivo(): void        |
|  | obtenerLongitud(): int      |
|   | mostrar(): void             |
|  | mostrarInOrden(): int       |
|  | buscarInOrden(): int        |

#### Atributos:

- *raíz* de tipo *Nodo* y carácter privado, almacena nodo de entrada al *ABB*
- *nodoBuscado* de tipo *Nodo* y carácter privado, permite almacenar el nodo que se está buscando

#### Métodos:

- *insertarNodo()* permite agregar un nuevo nodo al árbol utilizando el siguiente método recursivo
- *insertar()* es el método recursivo que permite agregar un nuevo nodo al árbol obteniendo la ubicación que le corresponde de acuerdo al nodo recibido
- *eliminarNodo()* permite retirar un nodo existente del árbol teniendo en cuenta el respectivo nodo padre
- *obtenerNodoSucesor()* permite obtener el nodo que continua de acuerdo al orden ascendente en que están los datos. Es utilizado por el método *eliminarNodo()* para poder realizar la sustitución de nodo en caso de ser necesario
- *obtenerNodoPosicion()* permite obtener el nodo que está en la posición determinada por la fila en que sale al desplegarse la lista completa
- *actualizar()* permite cambiar los valores de los datos almacenados en el nodo ubicado en una posición determinada, reordenándolo si se cambiar el valor del dato clave (el *Nombre*)
- *desdeArchivo()* permite poblar el *ABB* con datos provenientes de un archivo plano, utilizando el método *insertarNodo()*
- *obtenerLongitud()* permite calcular el total de nodos que tiene el árbol. Incluye una versión recursiva
- *mostrar()* permite desplegar el contenido del *ABB* en un objeto *JTable*, recorriéndolo en *InOrden*. Utiliza el siguiente método recursivo para poder llenar el modelo de datos del objeto *JTable*
- *mostrarInOrden()* permite de manera recursiva, llenar una matriz con los datos almacenados en todos los nodos del *ABB*, recorriéndolo en *InOrden*
- *buscarInOrden()* permite de manera recursiva, hallar la fila en que está ubicado un nodo

Este es el código de la clase:

```
import java.io.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;

public class Arbol {

    private Nodo raiz;

    //contruir un arbol vacio
    public Arbol() {
        raiz = null;
    }
}
```

```
//insertar un nuevo nodo en el arbol de busqueda binaria
public boolean insertarNodo(Nodo n) {
    //El arbol esta vacio?
    if (raiz == null) {
        raiz = n;
        return true;
    } else {
        //insertar el nodo donde corresponda
        return insertar(raiz, n);
    }
}

//buscar punto de insercion y agregar un nuevo nodo
private boolean insertar(Nodo n, Nodo nNuevo) {
    //insertar en subarbol izquierdo
    if (nNuevo.nombre.toUpperCase().compareTo(n.nombre.toUpperCase()) <
0) {

        //Esta disponible el nodo izquierdo?
        if (n.izquierdo == null) {
            n.izquierdo = nNuevo;
            return true;
        } else {
            //continuar recorriendo el subarbol izquierdo
            return insertar(n.izquierdo, nNuevo);
        }
    } //insertar en subarbol derecho
    else if
(nNuevo.nombre.toUpperCase().compareTo(n.nombre.toUpperCase()) > 0) {

        //Esta disponible el nodo derecho?
        if (n.derecho == null) {
            n.derecho = nNuevo;
            return true;
        } else //continua recorriendo el subarbol derecho
        {
            return insertar(n.derecho, nNuevo);
        }
    } else {
        return false;
    }
} //insertar

//retira un nodo del arbol
public boolean eliminarNodo(Nodo n) {
    boolean eliminado = false;

    // p apunta al elemento a borrar
    // padre apunta al padre del elemento a borrar
    Nodo p = raiz, padre = null;
    //Buscar el nodo para conocer su padre
    boolean encontrado = false;
    while (p != null && !encontrado) {
        if (p == n) {
            encontrado = true;

```

```
        } else {
            padre = p;
            if (n.nombre.compareTo(p.nombre) < 0) {
                p = p.izquierdo;
            } else {
                p = p.derecho;
            }
        }
    }
}
if (encontrado) {
    //Es un nodo hoja
    if (p.izquierdo == null && p.derecho == null) {
        //Es la raiz?
        if (padre == null && p == raiz) {
            raiz = null;
            eliminado = true;
        } else {
            if (padre.izquierdo == p) {
                padre.izquierdo = null;
            } else if (padre.derecho == p) {
                padre.derecho = null;
            }
            eliminado = true;
        }
    }
    //Es un nodo con hijo izquierdo
    else if (p.izquierdo != null && p.derecho == null) {
        //Es la raiz?
        if (padre == null && p == raiz) {
            raiz = p.izquierdo;
            eliminado = true;
        } else {
            if (padre.izquierdo == p) {
                padre.izquierdo = p.izquierdo;
            } else if (padre.derecho == p) {
                padre.derecho = p.izquierdo;
            }
            eliminado = true;
        }
    }
    //Es un nodo con hijo derecho
    else if (p.izquierdo == null && p.derecho != null) {
        //Es la raiz?
        if (padre == null && p == raiz) {
            raiz = p.derecho;
            eliminado = true;
        } else {
            if (padre.izquierdo == p) {
                padre.izquierdo = p.derecho;
            } else if (padre.derecho == p) {
                padre.derecho = p.derecho;
            }
            eliminado = true;
        }
    }
    //Es un nodo con los 2 hijos
    else if (p.izquierdo != null && p.derecho != null) {
        //Buscar el nodo sucesor
        n = obtenerNodoSucesor(p);
    }
}
```

```
        eliminarNodo(n);
        p.asignar(n);
        eliminado = true;
    }
}
if (eliminado) {
    n = null;
}
return eliminado;
}

//Obtener el nodo más a la izquierda del subárbol derecho
private Nodo obtenerNodoSucesor(Nodo n) {
    n = n.derecho;
    while (n.izquierdo != null) {
        n = n.izquierdo;
    }
    return n;
}

//actualizar valores en el arbol
public void actualizar(int fila,
    String nombre,
    String telefono,
    String celular,
    String direccion,
    String correo) {
    //Buscar el nodo que corresponda a la posicion
    buscarInOrden(raiz, -1, fila);
    Nodo n = nodoBuscado;
    //No hubo cambios en el nombre
    if (n.nombre.equals(nombre)) {
        n.asignar(nombre, telefono, celular, direccion, correo);
    } else {
        eliminarNodo(n);
        insertarNodo(new Nodo(nombre, telefono, celular, direccion,
correo));
    }
}

//Llena el arbol desde un archivo plano
public void desdeArchivo(String nombreArchivo) {
    //inicialmente el arbol esta vacio
    raiz = null;
    //leer el archivo
    BufferedReader br = Archivo.abrirArchivo(nombreArchivo);
    try {
        //leer la primer linea
        String linea = br.readLine();
        while (linea != null) {
            //partir la linea
            String[] textos = linea.split("\t");
            //Hay al menos 5 textos?
            if (textos.length >= 5) {
                //Insertar el nodo donde corresponda
                insertarNodo(new Nodo(textos[0],
```

```
        textos[1],
        textos[2],
        textos[3],
        textos[4]));
    }
    //leer siguiente linea
    linea = br.readLine();
}
} catch (IOException ex) {
}
}

//devuelve el total de nodos del arbol
public int obtenerLongitud() {
    return obtenerLongitud(raiz);
}

//obtiene recursivamente la longitud del arbol
private int obtenerLongitud(Nodo n) {
    if (n == null) {
        return 0;
    } else {
        //suma la longitud del subarbol izquierdo,
        //la del subarbol derecho y el nodo actual
        return obtenerLongitud(n.izquierdo) + 1
            + obtenerLongitud(n.derecho);
    }
}

//muestra en una tabla los contenidos
//El recorrido se hace INORDEN
public void mostrar(final JTable tbl) {
    //Definir la estructura para los datos de la tabla
    int filas = obtenerLongitud();
    String[][] datos = new String[filas][5];
    //recorrer el arbol y llenar la estructura
    mostrarInOrden(raiz, datos, -1);

    //Definir el modelo de datos
    DefaultTableModel dtm = new DefaultTableModel(datos,
        new String[]{"Nombre", "Telefono", "Celular", "Direccion",
"Correo"});
    //Incluir evento cuando haya cambios
    dtm.addTableModelListener(
        new TableModelListener() {

            public void tableChanged(TableModelEvent evt) {
                //obtener fila seleccionada en la tabla
                int f = evt.getFirstRow();
                //obtener modelo de datos de la tabla
                DefaultTableModel dtm = (DefaultTableModel)
evt.getSource();

                //actualizar el arbol
                actualizar(f,
                    (String) dtm.getValueAt(f, 0),
                    (String) dtm.getValueAt(f, 1),
```

```
(String) dtm.getValueAt(f, 2),
(String) dtm.getValueAt(f, 3),
(String) dtm.getValueAt(f, 4));
        mostrar(tbl);
    }
});

//asignar el modelo a la tabla
tbl.setModel(dtm);
}

//recorrido INORDEN del arbol
private int mostrarInOrden(Nodo n, String[][] datos, int fila) {
    //Si el nodo es nulo, no se muestra nada
    if (n != null) {
        //Invocacion recursiva para recorrer el nodo izquierdo
        fila = mostrarInOrden(n.izquierdo, datos, fila);
        //Mostrar los datos del nodo
        fila++;
        datos[fila][0] = n.nombre;
        datos[fila][1] = n.telefono;
        datos[fila][2] = n.celular;
        datos[fila][3] = n.direccion;
        datos[fila][4] = n.correo;
        //Invocacion recursiva para recorrer el nodo derecho
        fila = mostrarInOrden(n.derecho, datos, fila);
    }
    return fila;
}

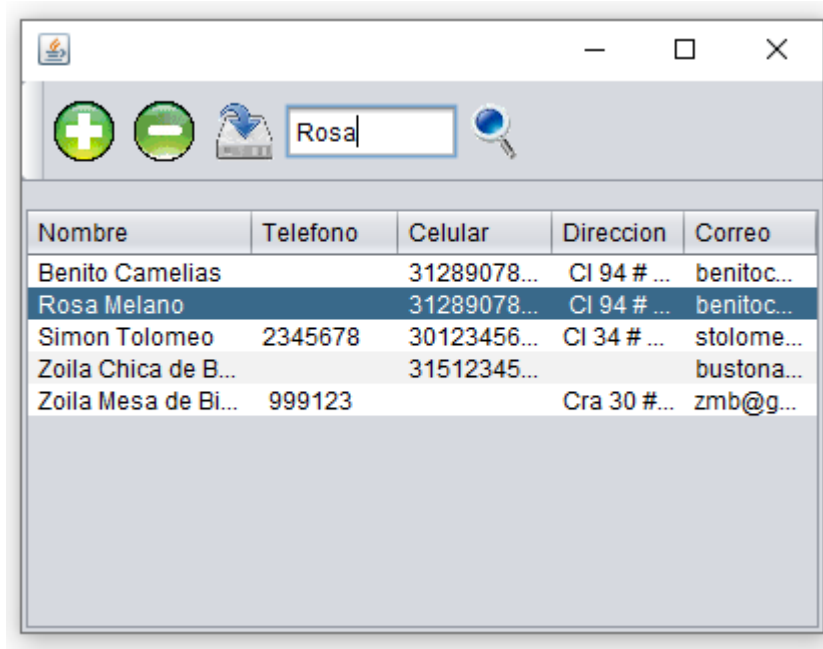
/** Busqueda de Nodos **
private Nodo nodoBuscado;

//Obtener el nodo ubicado en una posición en INORDEN
public Nodo obtenerNodoPosicion(int posicion) {
    nodoBuscado = null;
    buscarInOrden(raiz, -1, posicion);
    return nodoBuscado;
}

//recorrido INORDEN del arbol para buscar un nodo en una posición dada
private int buscarInOrden(Nodo n, int fila, int posicion) {
    if (n != null) {
        //Invocacion recursiva para recorrer el nodo izquierdo
        fila = buscarInOrden(n.izquierdo, fila, posicion);
        //Mostrar los datos del nodo
        fila++;
        if (fila == posicion) {
            nodoBuscado = n;
            return fila;
        }
        //Invocacion recursiva para recorrer el nodo derecho
        fila = buscarInOrden(n.derecho, fila, posicion);
    }
    return fila;
}
```

}

- Implementar la búsqueda de un contacto en el *ABB* del anterior ejercicio. Para ello se debe agregar a la interfaz una caja de texto donde el usuario pueda digitar parte del nombre a buscar y un botón que active la acción. Si lo halla, debe posicionarse en la respectiva fila, como se puede observar en la siguiente gráfica:



Para comprender este ejercicio, es importante tener en cuenta los siguientes fundamentos:

### Búsqueda en los ABB

Dada la forma como están contruidos los *ABB*, las búsquedas comienzan por el nodo raíz, que es con el primer nodo con el que se compara el dato buscado.

En el siguiente *ABB*, para hallar el dato “i”, se seguirían los siguientes pasos:

- Se compara el valor buscado (“i”) con el del nodo raíz (“f”), como el valor buscado es mayor, de existir en el árbol, estaría a la derecha.
- Se pasa al hijo derecho del nodo raíz. Se compara el valor buscado (“i”) con el valor del nodo (“j”), como el primero es menor, de existir en el árbol, estaría en el subárbol izquierdo.
- Se pasa al hijo izquierdo de “j”, que es “h”.
- Se compra el valor buscado (“i”) con el valor del nodo (“h”). Como el primero es mayor, de existir, estaría a la derecha.
- Se pasa al hijo derecho de “h”, que es “i”. Se ha encontrado al nodo buscado.

