

**UNIVERSIDAD DE CORDOBA**

**FACULTAD DE INGENIERIAS**

**PROGRAMA INGENIERIA DE SISTEMAS**

**CURSO:** Programación II

**TEMA:** Arboles en Java.

**DESCRIPCION:** En este documento se desarrolla el tema de la estructura de datos de árboles, considerando su conceptualización, características, tipos, aplicaciones, ejemplos, ilustraciones, mecanismos de representación y recorridos. Igualmente se presentan ejemplos gráficos de los diferentes conceptos relacionados con los árboles, de modo que permitan un mayor entendimiento de los mismos, especialmente en lo relacionado con las formas de representar o implantar esta estructura de datos en un lenguaje de programación. Así mismo, se plantean y explican distintos diagramas de clases UML para modelar tanto arboles n-arios como binarios, considerando las características y operaciones principales de esta estructura de datos. Finalmente se desarrolla un ejemplo práctico, utilizando el lenguaje de programación Java con el IDE Netbeans, desarrollándose una de interfaz gráfica de usuario para la construcción, despliegue y realización de los recorridos de un árbol binario.

## **OBJETIVOS**

- ♦ Realizar un estudio de las características, aplicaciones, métodos de representación de la estructura de datos de árboles, especialmente de los arboles binarios, con las respectivas representaciones gráficas y diseño de clases UML de los conceptos tratados.
- ♦ Implementar en el lenguaje Java una aplicación de interfaz gráfica de usuario, en el IDE NetBeans que permita crear y mostrar arboles binarios, usando un control *JTree* además de realizar los tres recorridos en profundidad.

**PALABRASCLAVES:** Arboles, arboles binarios, recorridos en arboles binarios, mecanismos de representación de árboles, implementación de árboles binarios en Java, aplicaciones de interfaz gráfica de usuario en Java.

## 1. Árboles

Un árbol es una estructura de datos no lineal, compuesta por un conjunto de elementos llamados nodos, los cuales están organizados mediante una relación de contención de la forma padre e hijos; o sea una asociación en la que algunos nodos son padres o contenedores de otros nodos y consecuentemente otros serán hijos o contenidos en otro nodo. Nótese que esta relación que caracteriza la organización de los nodos dentro de un árbol no es una relación de herencia, aun cuando, a efectos de la implementación de la estructura de datos, pudieren existir relaciones de herencia entre clases; claro está, en el caso de que los nodos del árbol se diseñen con una jerarquía de clases; siendo entonces cada nodo un objeto o instancia de alguna de las clases padre o hijas de la jerarquía. Razón por la cual, a efecto del diseño de los diagramas de clases, para representar árboles binarios y en particular a sus nodos y las relaciones entre estos, es más apropiado modelar dichas relaciones mediante una composición o agregación por valor; aunque también podemos encontrar modelos con una relación de agregación por referencia.

Por otra parte, cabe resaltar que en un árbol un nodo puede tener como máximo un solo y único nodo padre; esto es, todo nodo tendrá un nodo padre exceptuando el nodo raíz que es el único nodo dentro del árbol que no tiene padre; por lo tanto, carece de ascendiente (contenedor) pero de él se desprenden (contiene) a los demás nodos; sin embargo, un nodo cualquiera puede tener de cero a "n" nodos hijos. Precisamente, esta característica es la que fundamenta la idea de considerar a un árbol como una estructura de datos no lineal, toda vez que eventualmente existe más de un camino entre dos elementos o nodos; es decir, que a un nodo dado puedo llegar usando diferentes rutas; o en otras palabras, dado un elemento o nodo dentro del árbol a partir de este podríamos seguir en varias direcciones hacia otros nodos, tantas como nodos hijos tenga el nodo en cuestión.

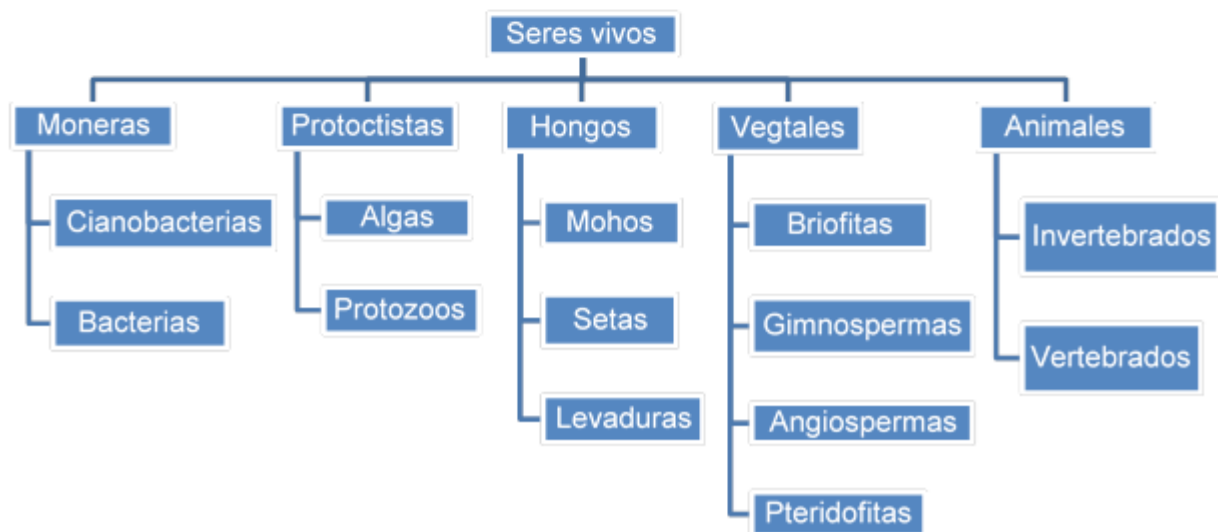
Es importante anotar que entre un nodo hijo y su padre existe siempre un único camino o dirección y que -al menos para el árbol- no se requiere -y en lo posible no debería haber- relaciones adicionales entre los nodos "hermanos", más allá de tener el mismo padre o contenedor común, ya que esto conllevaría a desperdiciar o usar memoria adicional en la implementación del árbol y sus nodos. Esta característica en consecuencia sugiere que no es posible ir directamente de nodo hijo a un nodo

hermano, sino a través o pasando primero por el nodo padre; lo cual se analizara más en detalle en el aparte de los recorridos en un árbol binario.

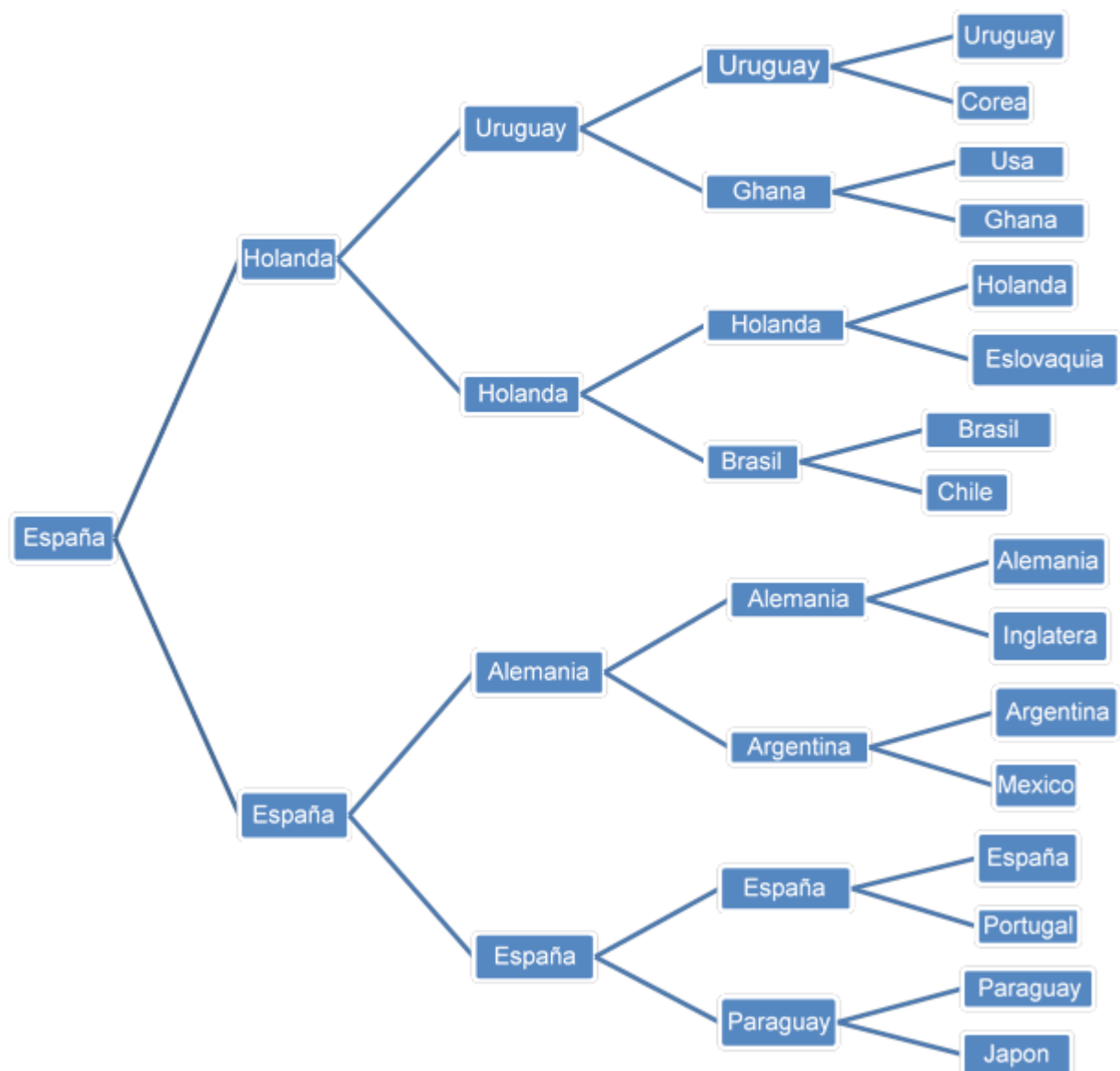
## 2. Aplicaciones o usos de los arboles

Son muchas las aplicaciones que encontramos para la estructura de datos de árboles, pero en general vale la pena señalar que son útiles o apropiadas en aquellos casos en donde es necesario representar información que se organiza de forma jerárquica, con cierta prioridad conocida, o que mantienen una relación de contención de la forma padre hijo. Así por ejemplo, el organigrama de una empresa debería ser representado mediante un árbol, igualmente el índice de un libro que pudiere estar compuesto por unidades, las cuales contienen capítulos y estos se dividen en temas que comprenden subtemas. Otra situación en la que podemos usar un árbol es para la estructura de los archivos y directorios de un unidad tal como un disco duro; en la cual, el nodo raíz es el disco duro y los hijos del nodo raíz son todas las carpetas de primer nivel del disco duro; las cuales a su vez pueden contener a otras carpetas o subdirectorios y cada directorio y subdirectorio eventualmente podrían tener dentro de ellos un conjunto de archivos.

En la siguiente imagen vemos una parte del esquema de clasificación de los seres vivos, el cual obedece a una organización jerárquica (en orden evolutivo) cuya estructura es no lineal, apropiada para ser representada mediante un árbol.



En la imagen de abajo, ilustramos el proceso de realización del mundial de fútbol de Sudáfrica desde la etapa de octavos de final hasta la gran final; leído de izquierda a derecha, el cual al ser una estructura no lineal, es un buen ejemplo para implementarse o representarse mediante un árbol.



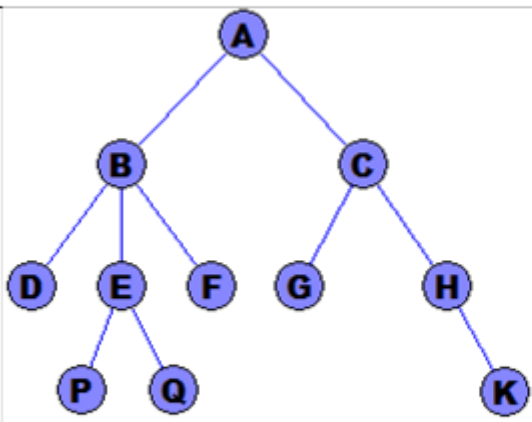


### 3. Características de un árbol

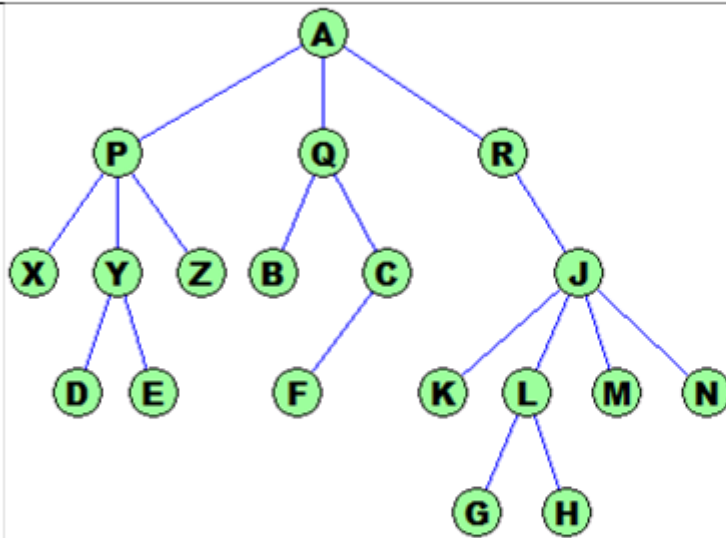
En un árbol podemos encontrar algunas propiedades importantes como las siguientes:

- ♦ **Nodo raíz.** Es el nodo donde comienza el árbol y se caracteriza por ser único nodo que no tiene antecedentes; es decir, es el nodo que tiene un padre. Cada árbol tiene solamente un nodo raíz, desde el cual “cuelgan” todos sus descendientes.
- ♦ **Nodo hoja.** Aquel que no tiene sucesores; es decir, aquellos nodos que no tienen hijos; y que por supuesto se encuentran en los extremos del árbol.
- ♦ **Nodo interno.** También llamados nodos ramas, y son aquellos nodos que no son ni raíz ni hoja; es decir, todo nodo que tenga hijos sin ser la raíz.
- ♦ **Grado de un nodo.** Es el número de hijos que le corresponden a un nodo.
- ♦ **Grado de un árbol.** Mayor de los grados de los nodos que componen al árbol; es decir, es el número máximo de nodos hijos que puede tener un nodo del árbol.
- ♦ **Camino.** Conjunto de nodos comprendidos entre dos nodos dados, y que se usa para llegar desde un nodo hasta el otro.
- ♦ **Longitud de un camino.** Representa la cantidad de nodos que comprende un camino dado.
- ♦ **Profundidad de un nodo.** También llamada **nivel** del nodo, se define como la longitud del camino único que va desde la raíz hasta ese nodo; es decir, será la cantidad de nodos que hay desde la raíz hasta el nodo en cuestión, considerando además que el nodo raíz tiene una profundidad o nivel igual a uno.
- ♦ **Altura de un nodo.** Es la longitud del camino más largo posible establecido entre un nodo dado y un nodo hoja; en otras palabras, la altura de un nodo está dada por el número de nodos comprendido entre el nodo y la hoja más lejana a él y con la cual el nodo en cuestión tiene un camino.
- ♦ **Altura de un árbol.** Es el número de nodos comprendidos entre el nodo raíz y el nodo hoja más distante de esta; esto es, el nivel del nodo que mayor nivel tiene dentro del árbol.

Para ilustrar más adecuadamente los conceptos anteriores, consideremos los siguientes arboles representados gráficamente.

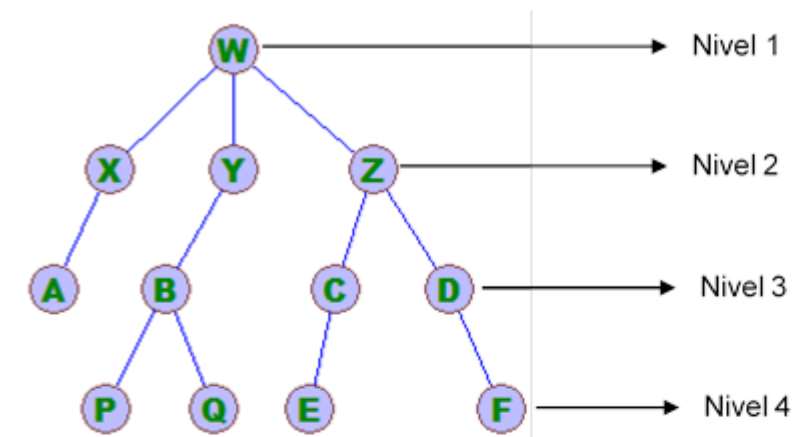
Árbol de ejemplo	
	
Características	Ejemplo
Nodo Raíz	Nodo <b>A</b>
Nodo Hojas	Nodos <b>D, P, Q, F, G, K</b>
Nodos Internos (ramas)	<b>B, C, E, H</b>
Grado de nodos	Grado de <b>A</b> =2, Grado de <b>B</b> =3, Grado de <b>H</b> =1, Grado de <b>E</b> =2, Grado de <b>P</b> =0, Grado de <b>K</b> =0.
Grado del árbol	3 pues es el mayor grado observado en sus nodos ( <b>B</b> ).
Caminos	Camino de <b>A</b> a <b>E</b> => <b>A, B, E</b> Camino de <b>A</b> a <b>K</b> => <b>A, C, H, K</b>
Longitud de caminos	Longitud Camino de <b>A</b> a <b>G</b> = 3 (tres nodos hay de <b>A</b> a <b>G</b> ) Longitud Camino de <b>A</b> a <b>K</b> = 4.
Profundidad o niveles	Profundidad de <b>B</b> =2 (Hay dos nodos entre la raíz <b>A</b> y <b>B</b> ) Profundidad de <b>G</b> =3, Profundidad de <b>P</b> =4.
Altura de nodos	Altura de <b>B</b> =3 (Hay tres nodos entre <b>B</b> y sus hojas más lejanas <b>P</b> o <b>Q</b> ). Altura de <b>H</b> =2.
Altura del árbol	Igual a la altura del raíz <b>A</b> =4 (Hay 4 nodos entre la raíz <b>A</b> y sus hojas más lejanas <b>P, Q</b> o <b>K</b> )

### Arbol de ejemplo

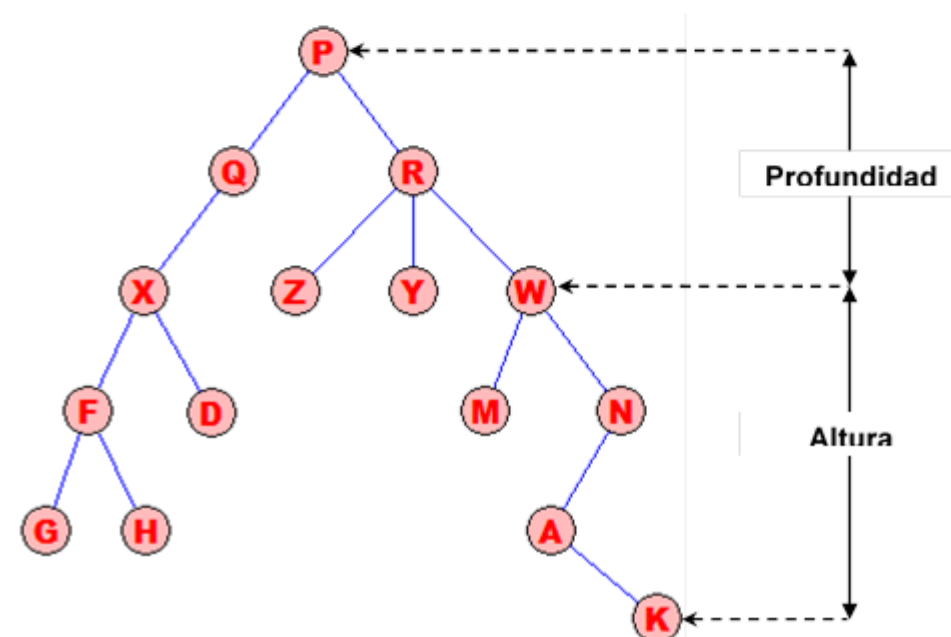


Características	Ejemplo
Nodo Raíz	Nodo <b>A</b>
Nodo Hojas	Nodos <b>X, Z, D, F, G, H</b> etc.
Nodos Internos (ramas)	<b>P, Q, R, C, Y</b> etc.
Grado de nodos	Grado de <b>A</b> =3, Grado de <b>Q</b> =2, Grado de <b>J</b> =4, Grado de <b>R</b> =1, Grado de <b>X</b> =0, Grado de <b>C</b> =1.
Grado del árbol	4 ya que es el mayor grado observado en sus nodos ( <b>J</b> ).
Caminos	Camino de <b>Q</b> a <b>F</b> => <b>Q, C, F</b> Camino de <b>A</b> a <b>G</b> => <b>A, R, J, L, G</b>
Longitud de caminos	Longitud Camino de <b>Q</b> a <b>F</b> = 3 (tres nodos hay de <b>Q</b> a <b>F</b> ) Longitud Camino de <b>A</b> a <b>G</b> = 5.
Profundidad o niveles	Profundidad de <b>B</b> =3 (Hay tres nodos entre la raíz <b>A</b> y <b>B</b> ) Profundidad de <b>D</b> =4, Profundidad de <b>H</b> =5, Profundidad de <b>R</b> =2.
Altura de nodos	Altura de <b>R</b> =4 (Hay cuatro nodos entre <b>R</b> y sus hojas más lejanas <b>G</b> o <b>H</b> ). Altura de <b>C</b> =2, Altura de <b>P</b> =3, Altura de <b>J</b> =3.
Altura del árbol	Igual a la altura del raíz <b>A</b> =5 (Hay 5 nodos entre la raíz <b>A</b> y sus hojas más lejanas <b>G</b> o <b>H</b> )

Para aclarar un poco más algunos conceptos presentados anteriormente, considere que los nodos de un mismo nivel son nodos hermanos, primos etc., como puede deducirse de la siguiente ilustración.



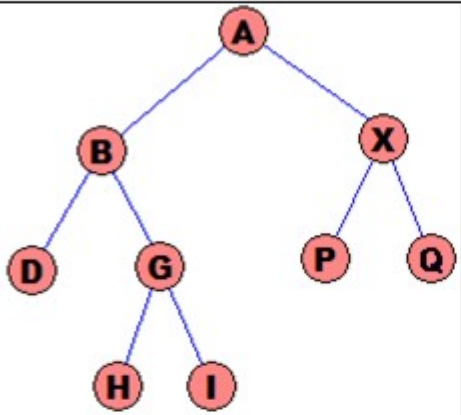
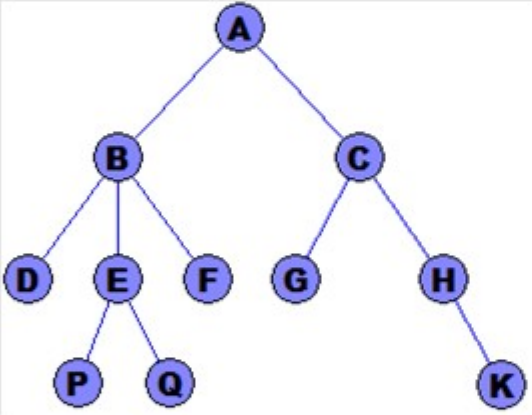
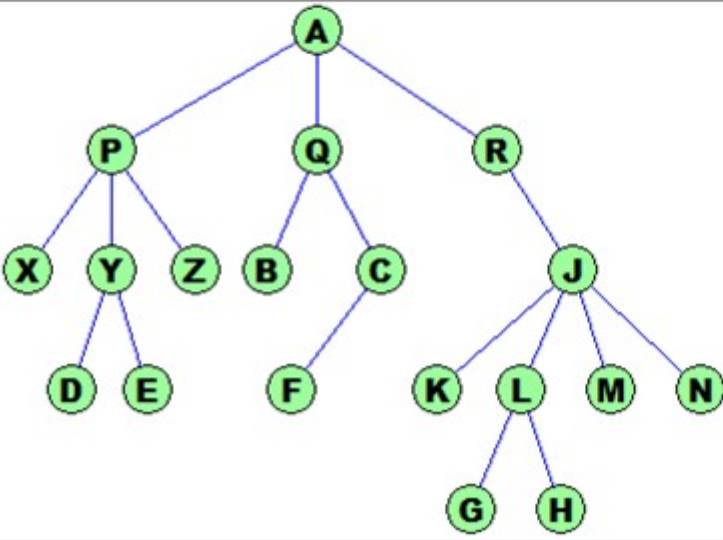
Ahora bien, para diferenciar los conceptos de profundidad y altura de un nodo, considere el nodo **W** del siguiente árbol, en el que se ilustra que la profundidad o nivel se mide de la raíz (**P**) al nodo; es decir, del nodo hacia la parte de arriba del gráfico, teniendo para el caso representado el valor de 3. Mientras tanto, la altura se mide desde el nodo hasta la hoja más lejana (**K**), es decir, del nodo hacia la parte de debajo de la imagen, que en el ejemplo toma el valor de 4.





#### 4. Tipos de arboles

Podemos encontrar varios criterios para clasificar o caracterizar a un árbol, así por ejemplo, según el grado los arboles más comunes suelen ser los arboles binarios (grado dos), ternarios (grado tres) o n-arios (grado mayor a tres) que también se les llama arboles generales. A continuación se ilustran algunos ejemplos:

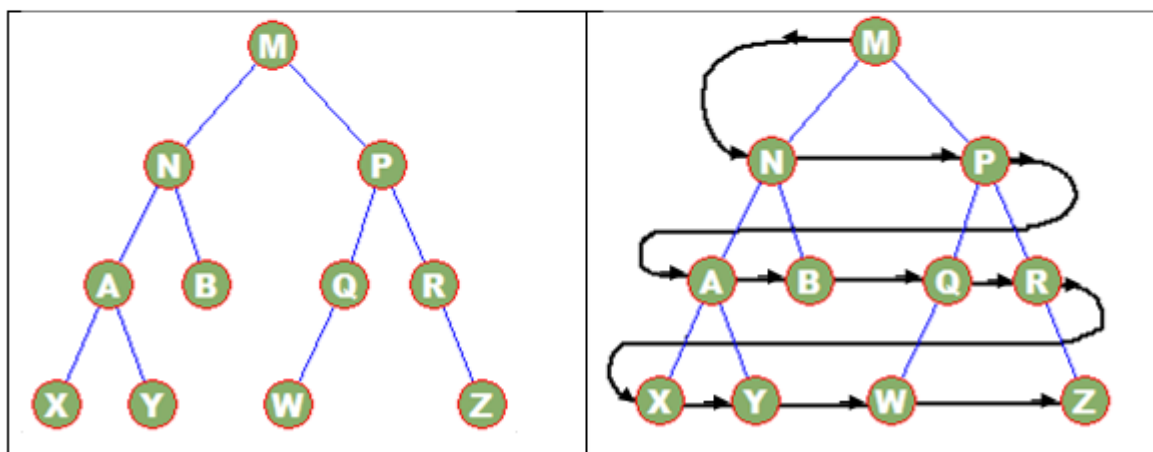
Ejemplo de árbol	Grado	Tipo
	Dos (2)	Árbol Binario
	Tres (3)	Árbol Ternario
	Cuatro (4)	Árbol n-ario o árbol general

## 5. Recorridos en un árbol binario

El recorrido de un árbol binario es una de las operaciones más comunes y fundamentales hecha sobre esta estructura de datos, considerando que recorrer un árbol es, en general y de forma intuitiva, el proceso mediante el cual "visitamos" todos los nodos del árbol; claro está, entendiendo por visitar la realización de alguna operación con la información contendida en cada nodo. Para tal efecto y tomando en cuenta que un árbol es una estructura de datos no lineal, un árbol lo podemos recorrer en anchura o en profundidad.

### 5.1 Recorrido en anchura.

Este recorrido consiste en visitar los nodos del árbol de izquierda a derecha y de arriba hacia abajo; es decir, en el mismo sentido en que leemos. En otras palabras, empezamos por el nodo raíz y luego bajamos al siguiente nivel (primera generación de nodos hijos) y visitamos los hijos de la raíz de izquierda a derecha; es decir primero el hijo izquierdo y luego el hijo derecho. Luego bajamos al siguiente nivel (segunda generación) cuyos nodos visitamos igualmente de izquierda a derecha; es decir empezamos por el hijo izquierdo del hijo izquierdo de la raíz, luego pasamos al hijo derecho del hijo izquierdo de la raíz y finalmente terminamos con este nivel visitando el hijo izquierdo y luego el derecho del hijo derecho de la raíz. El proceso continúa de esta manera pasando por todos los demás niveles del árbol. Como ejemplo, consideremos el siguiente árbol binario y a la izquierda el sentido o dirección de su recorrido en anchura.



Haciendo el recorrido en anchura descrito anteriormente sobre este árbol y tomando en cuenta que vamos a recorrer el árbol para mostrar sus nodos, el resultado es el siguiente: **M, N, P, A, B, Q, R, X, Y, W, Z.**

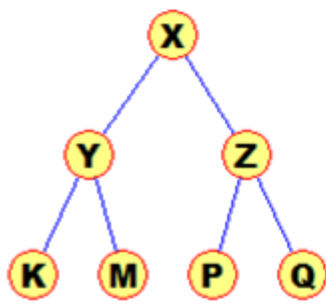
No obstante este recorrido es poco usual, toda vez que como se indicó anteriormente no debería existir “enlaces” o relaciones adicionales entre los nodos hermanos de un árbol más allá de la “conexión” con su nodo padre, con el fin de no desperdiciar memoria. Esto en consecuencia, implica que no habría forma de “saltar directamente” entre los nodos hermanos; por lo cual, para ir del hijo izquierdo al hijo derecho de un nodo, es necesario primero ir del padre al hijo izquierdo, luego “retroceder” al padre y entonces ir al hijo derecho; pero aun así existe otro inconveniente en el mismo sentido, pero de mayor complejidad, por ejemplo cuando debamos pasar entre nodos “primos” y que aumenta en complejidad cuando avanzamos a niveles mayores. Como es de esperar y fácil de deducir, esta forma de recorrido es ineficiente y por demás dura de implementar, porque adicionalmente su codificación requiere de otras estructuras de datos complementarias (especialmente pilas), que por supuesto consumen su propia cantidad de memoria, sin contar el código extra para su implementación.

## **5.2 Recorrido en profundidad.**

También conocido como recorrido por niveles y consiste en visitar los nodos del árbol desde arriba (iniciando en la raíz) hacia abajo (hacia niveles más altos); esto es, siguiendo una dirección vertical siempre en el sentido de los nodos padres hacia los nodos hijos, y siempre para estos primero el hijo izquierdo y luego el derecho. En este orden de ideas, realmente existen tres formas o tipos de recorridos en profundidad, dependiendo del orden o momento en que se visita al nodo padre en relación al momento en que se visitan a cada uno de sus nodos hijos. Estos tres tipos de recorridos en profundidad toman los nombres de recorridos preorden, inorden y postorden, teniendo estos tres recorridos en común que siempre inician en el nodo raíz y cada nodo hijo se visita recursivamente en la misma forma de recorrido con el que se visitó su nodo padre (aprovechando la definición recursiva de árbol donde cada nodo de este puede verse como un subárbol, que no es más que otro árbol).

### ♦ Recorrido PreOrden

Consiste en “visitar” primero al nodo padre empezando desde la raíz, luego se visita recursivamente en modo preorden al nodo hijo izquierdo, finalizando con la visita recursiva en modo preorden al nodo hijo derecho. De esta manera el orden de las visitas de los nodos del árbol es: padre  $\rightarrow$  hijo izquierdo  $\rightarrow$  hijo derecho; esto es, la operación a realizar con los nodos se hace primero en los nodos padres antes que en sus hijos, y para estos, siempre se realiza primero en el hijo izquierdo y sus hijos antes que en el hijo derecho y los hijos de este último. En este aparte, vale la pena señalar que por “visitar recursivamente” debemos sobreentender que también se visita en modo preorden a todos los hijos de todas las generaciones –niveles- del nodo hijo visitado tanto para el caso del hijo izquierdo como del derecho. Como ejemplo considere el siguiente árbol binario:



El recorrido en preorden es el siguiente: **X, Y, K, M, Z, P, Q**. note que en este modo de recorrido se procesa primero la raíz y de ultimo la hoja que está más a la derecha

### ♦ Recorrido InOrden

Este recorrido en profundidad consiste en “visitar” primero y de forma recursiva en modo inorden al nodo hijo izquierdo y a todos los hijos de este en todos los niveles subsiguientes; luego se visita al nodo padre y finalmente se visita recursivamente en modo inorden al nodo hijo derecho. Así en ese recorrido el orden de las visitas de los nodos del árbol es: hijo izquierdo  $\rightarrow$  padre  $\rightarrow$  hijo derecho; es decir, la operación para el nodo padre en la que consiste la visita, se hace después de haberla hecho en todos sus descendientes hijos izquierdos y en los hijos de estos por todas las generaciones. De manera análoga al caso anterior, este recorrido también inicia con el nodo raíz. Para el árbol anterior el recorrido inorden es el siguiente: **K, Y, M, X, P, Z, Q** donde de primero se procesa la hoja que está más a la izquierda,

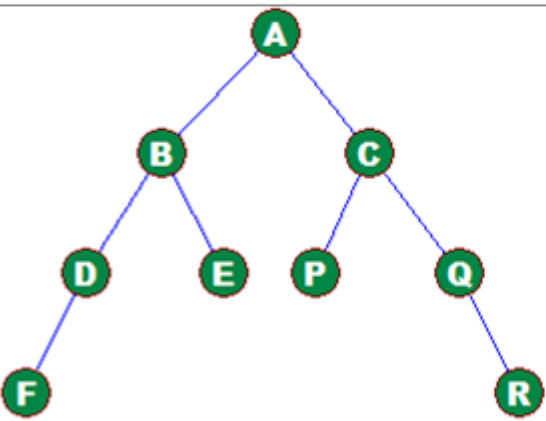


quedando más o menos en el centro el nodo raíz y de último la hoja que está más a la derecha.

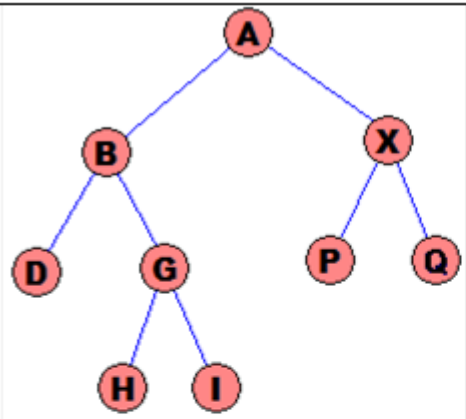
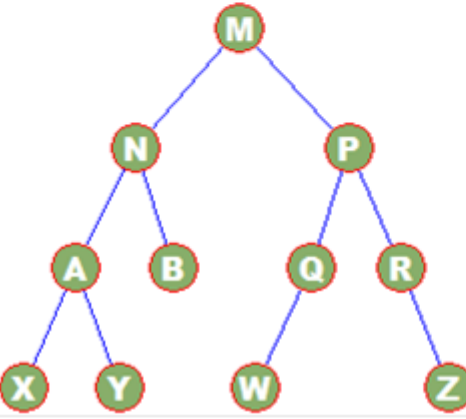
#### ♦ Recorrido PostOrden

En esta forma de recorrido en profundidad se “visita” primero recursivamente en modo postorden al nodo hijo izquierdo y a todos los hijos de este en todos los niveles. Se continúa visitando al nodo hijo derecho recursivamente en modo postorden y a todos los hijos de este también en modo postorden. Finalmente se visita al nodo padre que esta vez queda en tercer lugar en relación a sus dos posibles hijos; así el orden de las visitas de los nodos es: hijo izquierdo → hijo derecho → padre; es decir, la operación en la que consiste la vista se hace de último para el nodo padre y siempre después de realizarla en cada uno de sus hijos izquierdo y derecho y sobre los descendientes de estos. Continuando con la estructura del último árbol presentado anteriormente, el recorrido en modo postorden de este queda: **K, M, Y, P, Q, Z, X** que se caracteriza por que el nodo raíz se procesa de último, procesándose de primero el nodo hoja que se encuentra más a la izquierda.

Para complementar este aparte, analice y verifique los tres modos de recorridos en profundidad expuestos anteriormente en relación a la estructura de los arboles binarios ilustrados en las siguientes gráficas.

Arbol de ejemplo	Recorrido	Resultado
	PreOrden	A, B, D, F, E, C, P, Q, R
	InOrden	F, D, B, E, A, P, C, Q, R
	PostOrden	F, D, E, B, P, R, Q, C, A



Árbol de ejemplo	Recorrido	Resultado
	PreOrden	A, B, D, G, H, I, X, P, Q
	InOrden	D, B, H, G, I, A, P, X, Q
	PostOrden	D, H, I, G, B, P, Q, X, A
	PreOrden	M, N, A, X, Y, B, P, Q, W, R, Z
	InOrden	X, A, Y, N, B, M, W, Q, P, R, Z
	PostOrden	X, Y, A, B, N, W, Q, Z, R, P, M

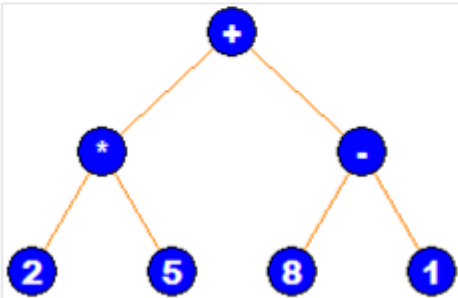
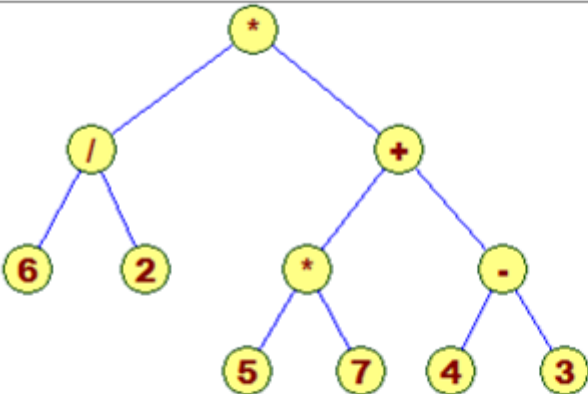
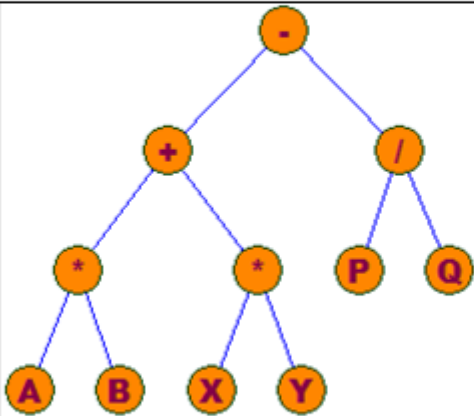
## 6. Árboles de expresión

Un árbol de expresión es un árbol binario que se emplea para representar diferentes tipos de expresión, especialmente expresiones aritméticas, lógicas y de operadores de bits. En este sentido, los nodos de un árbol de expresión contienen operándos que son los datos que intervienen en la operación, que se corresponderán con los nodos hojas; además de operadores, que son los signos o símbolos que procesan los datos de la expresión y en consecuencia serán los nodos ramas y la raíz. Por lo anterior, los árboles de expresión constituyen una de las más frecuentes e importante aplicaciones de los árboles binarios.

En cuanto al resultado de los recorridos en profundidad de un árbol de expresión, tenemos que al resultado del recorrido en preorden se le llama **notación prefija** de la expresión, pues primero se anotan los operadores (signos o símbolos) de la expresión y luego van los operándos o datos. La **notación postfija** por su parte, se origina

como resultado de recorrer un árbol de expresión en modo postorden, ya que en esta notación van primero los operándos y después los signos de la expresión.

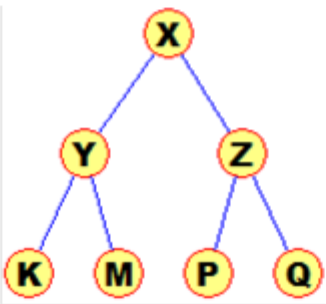
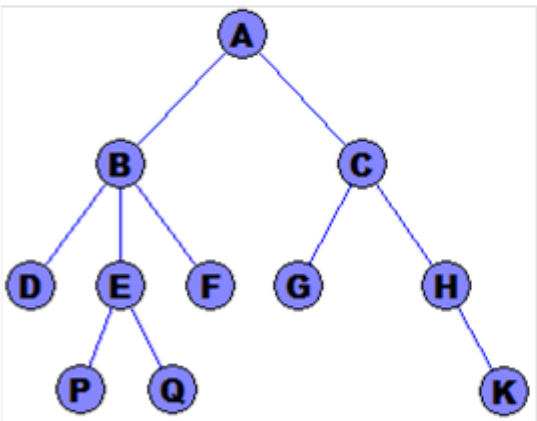
La **notación infija** de la expresión, surge al recorrer un árbol de expresión en modo infijo, caracterizándose por que se combinan operando y operador en ese orden. En la siguiente tabla se muestran algunos ejemplos de árboles de expresión, con las respectivas notaciones originadas por los diferentes tipos de recorridos.

Árboles de expresión	Recorrido	Resultado
	PreOrden	<b>+ , * , 2 , 5 , - , 8 , 1</b> (Notación prefija)
	InOrden	<b>2*5+8-1</b> (Notación infija)
	PostOrden	<b>2,5,* , 8,1,- , +</b> (Notación postfija)
	PreOrden	<b>*, / , 6 , 2 , + , *, 5 , 7 , - , 4 , 3</b> (Notación prefija)
	InOrden	<b>6/2*5*7+4*3</b> (Notación infija)
	PostOrden	<b>6,2,/ , 5,7,* , 4,3,- , + , *</b> (Notación postfija)
	PreOrden	<b>- , + , * , A , B , * , X , Y , / , P , Q</b> (Notación prefija)
	InOrden	<b>A*B+X*Y-P/Q</b> (Notación infija)
	PostOrden	<b>AB*XY*+PQ/-</b> (Notación postfija)

En la notación infija se puede apreciar más claramente la expresión representada en el árbol; así por ejemplo, para el primer caso vemos que dicha expresión aritmética es **2\*5+8 – 1**, que al ser resulta obtenemos: 10 + 8 -1 = 17.

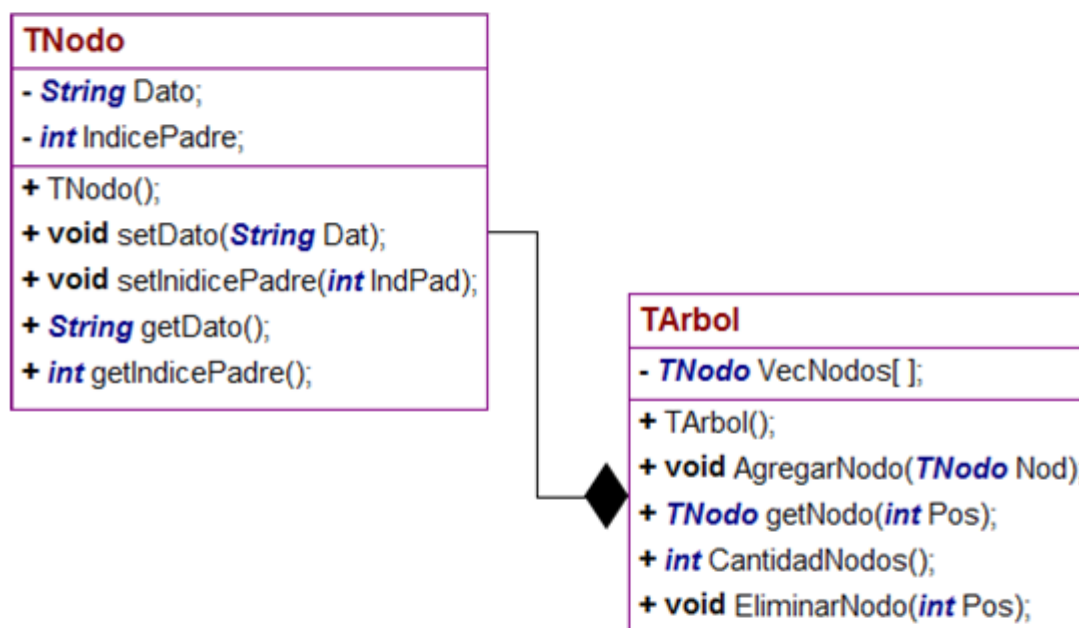
## 7. Mecanismos de representación de árboles (implantación)

Existen diferentes mecanismos o formas de representar arboles dentro de los lenguajes de programación; así por ejemplo, se suelen usar las llamadas listas o matrices de adyacencias, en las cuales cada fila de la matriz representa o describe a cada nodo; es decir, el número de filas será igual al número de nodos del árbol. Las columnas de cada fila describen los datos o propiedades de un nodo en particular, de modo que una de esas columnas contiene la posición o índice de fila de la matriz en la cual se encuentra el nodo padre de un nodo en particular; usándose comúnmente el valor de -1 para el nodo que no tiene padre; es decir, para el nodo raíz. Así por ejemplo, la siguiente tabla ilustra un par de árboles con su respectiva lista de adyacencia representativa.

Árbol	Posición	Nodo	Posición Padre
	0	X	-1
	1	Y	0 (X)
	2	Z	0
	3	K	1 (Y)
	4	M	1
	5	P	2 (Z)
	6	Q	2
Arbol	Posición	Nodo	Posición Padre
	0	A	-1
	1	B	0 (A)
	2	C	0
	3	D	1
	4	E	1 (B)
	5	F	1
	6	G	2 (C)
	7	H	2
	8	P	4 (E)
	9	Q	4
	10	K	7 (H)

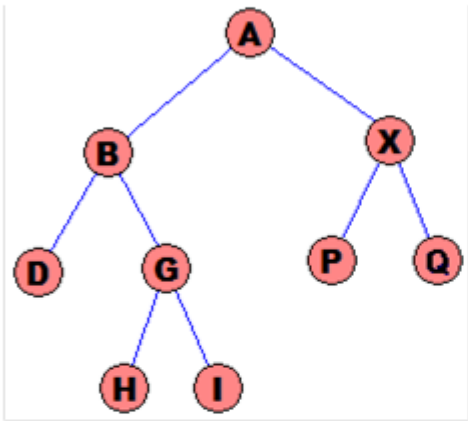
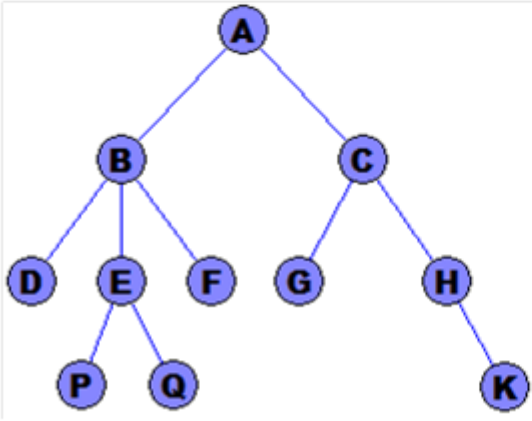
Una manera de implementar la anterior matriz de adyacencia para representar un árbol, es definiendo una clase para describir cada nodo; de modo que estos sean objetos o instancias de la clase y que se almacenen en un vector dinámico de objetos, en vez de usar una matriz propiamente dicha. La posición o índice del nodo padre bien puede estar declarada como atributo de la clase nodo y el árbol contiene como atributo al vector de objetos en cuestión.

Un diseño UML de clases tentativo para esta forma de representación del árbol, empleando vectores dinámicos de objetos, con un atributo numérico para cada nodo, que indique la posición del nodo padre en el vector es el siguiente:



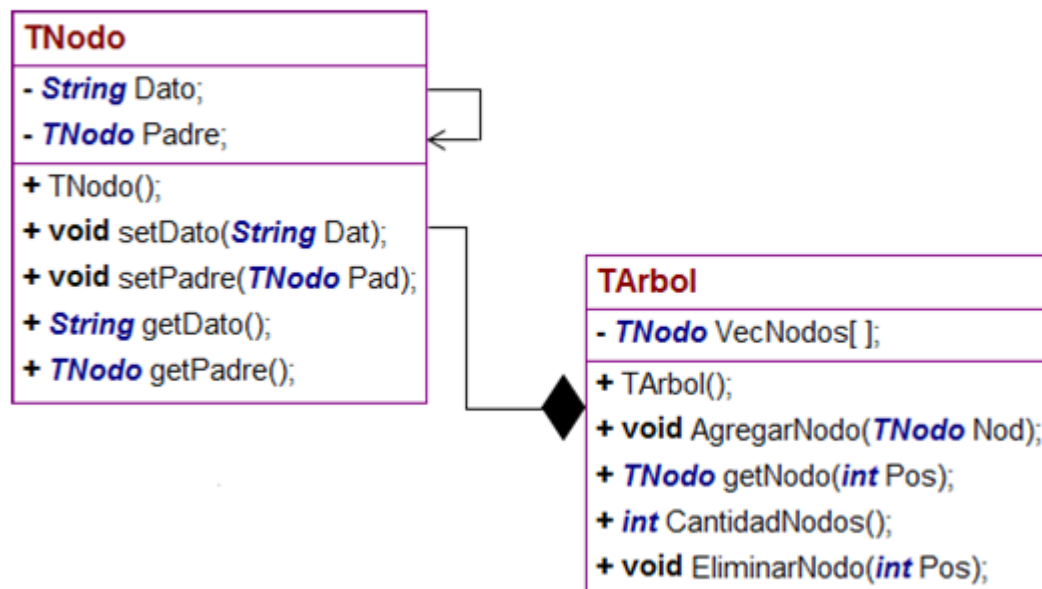
En este ejemplo, partimos del supuesto que la clase *TNode* solo contendrá como información el nombre o etiqueta del nodo, pero de igual forma la complejidad del nodo en cuanto al número de sus atributos es indiferente para el caso. De esta manera, el atributo *Dato* se definió del tipo *String*. Igualmente vemos que en la clase *TArbol* el atributo para el vector de nodos se define como un arreglo dinámico en tamaño de instancias de la clase *TNode*, empleando la notación con corchetes [ ] para el caso del arreglo, asociándose estas dos clases mediante una relación de composición.

Otra manera alternativa de emplear las listas de adyacencias, es considerando un arreglo o vector de objetos, en el que cada objeto de cada posición del arreglo representa a un nodo del árbol; con ello el tamaño del arreglo será igual al número de nodos del árbol representado. Pero en esta variante cada nodo (objeto) representa a su nodo padre mediante una referencia o apuntador a dicho nodo; diferente al caso anterior que se hizo a través del índice o posición del nodo padre en el arreglo. En la siguiente tabla se ilustra esta situación con dos árboles de ejemplo:

Árbol	Nodo	Padre
	A	Nulo
	B	A
	X	A
	D	B
	G	B
	P	X
	Q	X
	H	G
	I	G
Árbol	Nodo	Padre
	A	Nulo
	B	A
	C	A
	D	B
	E	B
	F	B
	G	C
	H	C
	P	E
	Q	E
	K	H

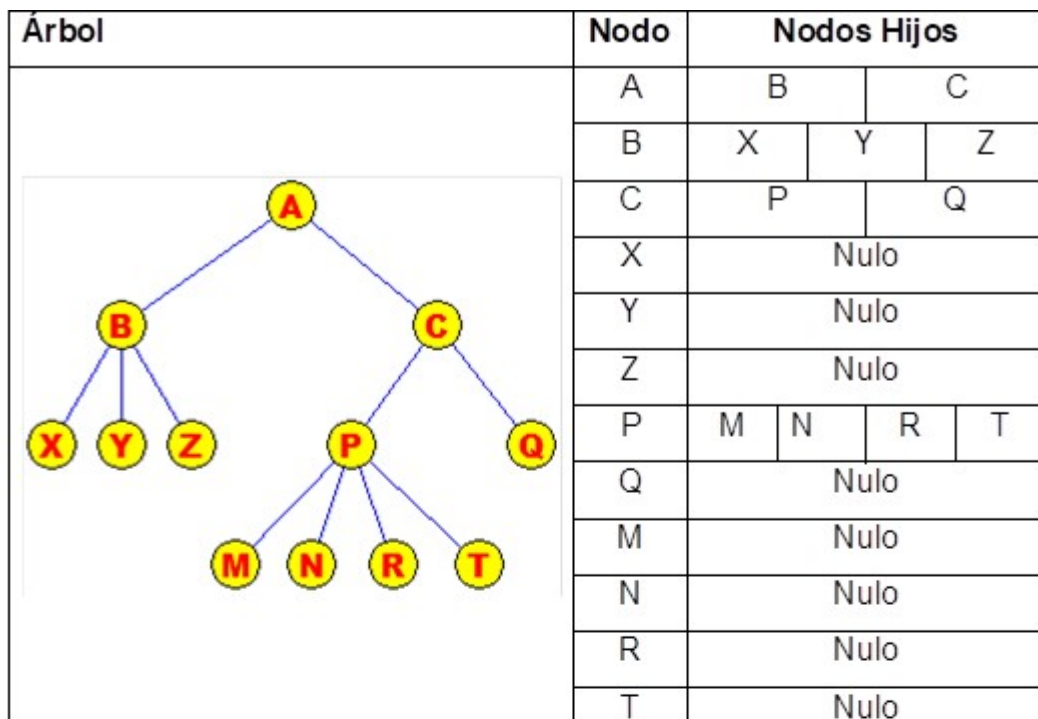


El diagrama de clases UML representativo para la estructura expuesta en la tabla anterior es el siguiente:

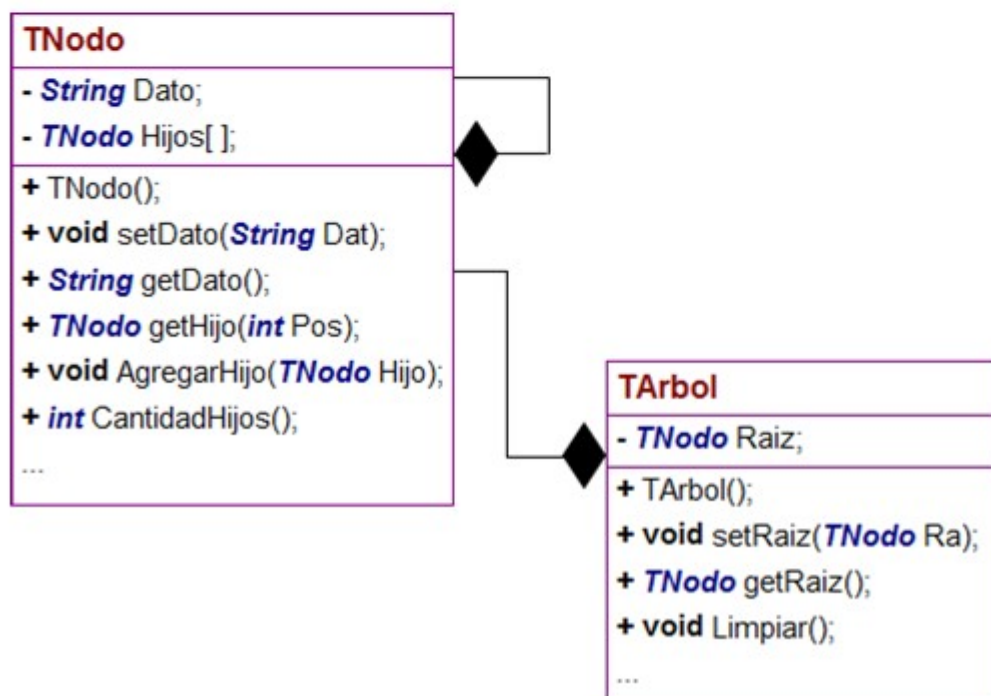


En esta propuesta de diseño, se sigue representando el conjunto de nodos del árbol como un vector de tamaño dinámico de objetos, como en el caso anterior; pero esta vez, como ya se indicaba, el nodo padre se asocia a través de una relación reflexiva (de la clase *TNode* consigo misma), indicada por la flecha trazada en la parte superior de la clase *TNode*. Por demás, la relación entre la clase *TNode* y la clase *TArbol* sigue siendo una composición o agregación por valor.

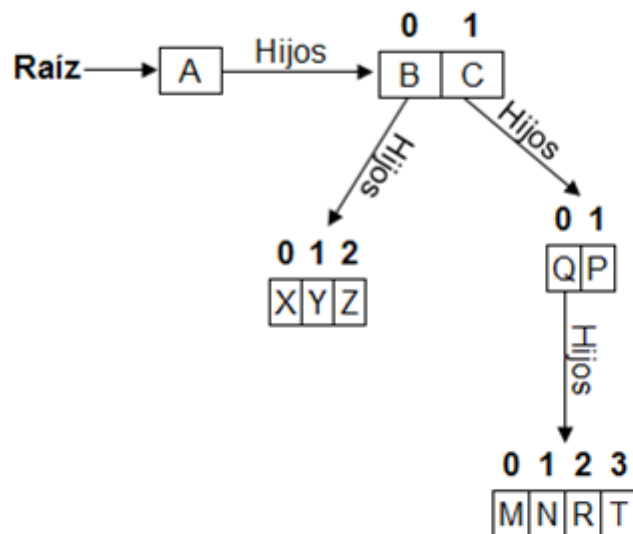
Existen otras formas de representar un árbol, especialmente los generales o n-arios, que consiste en definir como atributo de la clase del nodo un vector para almacenar los hijos del nodo; dicho vector se caracterizará por guardar instancias de la clase del nodo, además de ser dinámico en tamaño, toda vez que en ejecución el número de hijos de un nodo puede variar aumentando o disminuyendo en cualquier momento y sin límite alguno al respecto; pues es lo que se requiere para un árbol de grado general. Por su parte la clase diseñada para representar el árbol, define un atributo del mismo tipo de la clase del nodo para representar a la raíz. La siguiente tabla ilustra un caso de representación de un árbol de la forma descrita anteriormente.



El diagrama UML de clases para la representación de un árbol expuesta anteriormente es el siguiente, en el que se destaca esta vez, que la clase del árbol solo contiene el atributo **Raíz** como apuntador a un objeto de la clase *TNodo*, que basta para “iniciar” todas las relaciones entre los demás nodos del árbol:

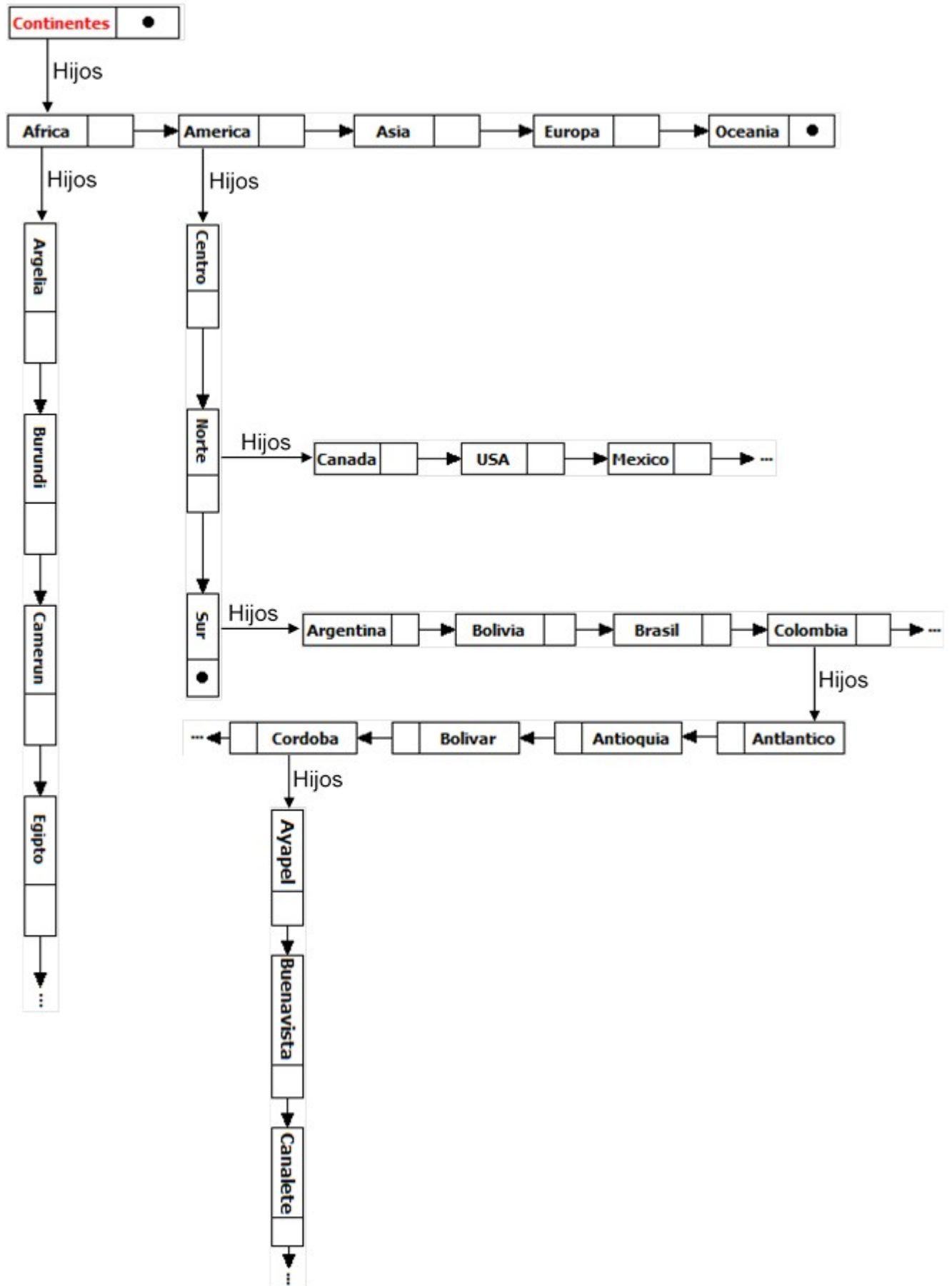


Gráficamente, la estructura de datos en memoria, representada por objetos de las clases del diagrama anterior, podría verse de la siguiente manera, donde los nodos hijos son vectores de instancias de la clase *TNode*:

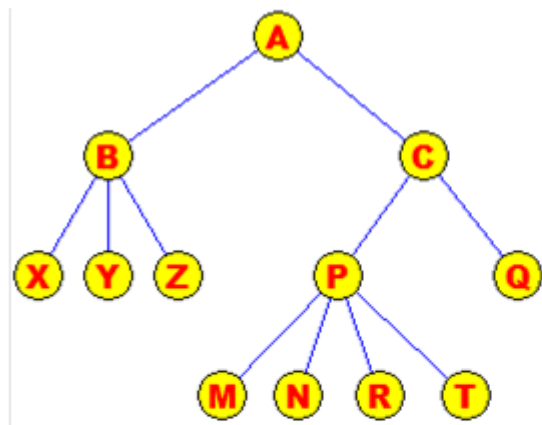


Alternativamente al caso anterior, cada nodo puede representar sus nodos hijos usando una lista enlazada (simple o doble) en vez de emplear un vector dinámico de objetos, lo cual conlleva a otra forma de representación del árbol consistente en emplear multilistas enlazadas. En este caso, lo más sencillo de hacer es considerar que la cabeza de la lista es un apuntador al primer hijo del nodo que la contenga; es decir, que cada nodo represente a sus hijos mediante un atributo del tipo lista enlazada.

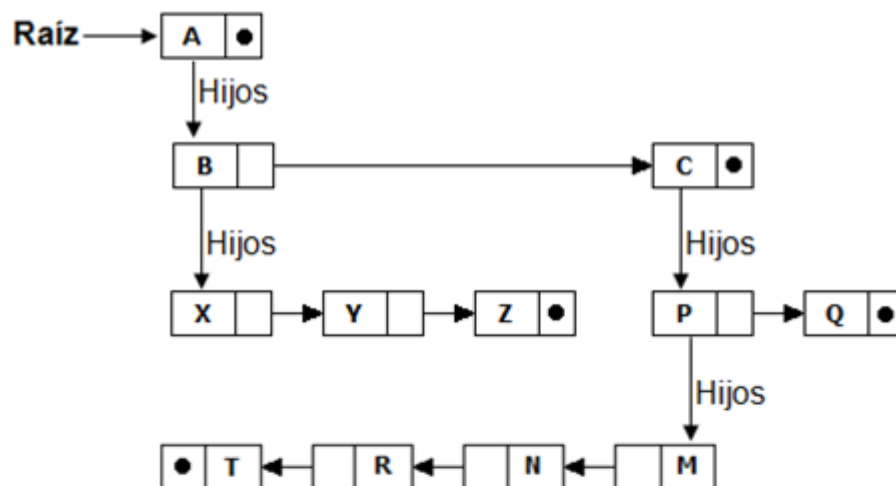
Por su parte, una multilista es una estructura de datos propia en si misma e independiente de los árboles, que además tiene sus propias aplicaciones y son tan comunes y antiguas como las listas enlazadas simples o dobles y se derivan de estas; así por ejemplo, para representar la distribución geográfica del mundo, considerando un orden en tamaño de los elementos de mayor a menor, desde los continentes pasando por los países y llegando, por ejemplo, hasta los pueblos municipios o veredas, dicha información la podemos modelar usando una multilista, dado su característica de contención (esencial en el concepto de árboles). Dicha multilista ineludiblemente representaría una estructura en forma de árbol, tal como se ilustra en la siguiente imagen, en la que el nodo continentes es el nodo raíz del árbol:



Considere ahora el siguiente árbol presentado anteriormente:

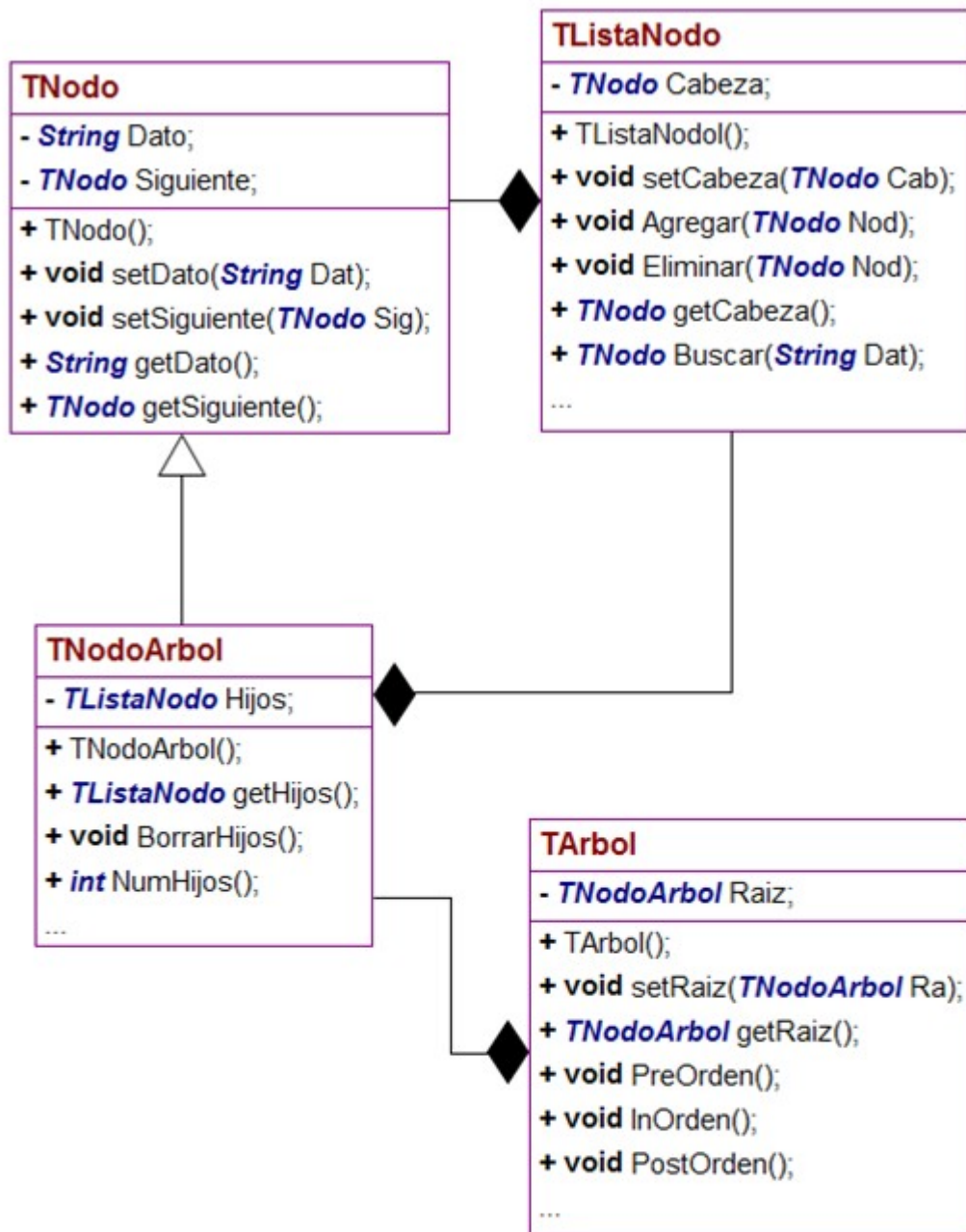


Su representación correspondiente, empleando multilistas de la manera descrita antes, es la siguiente:



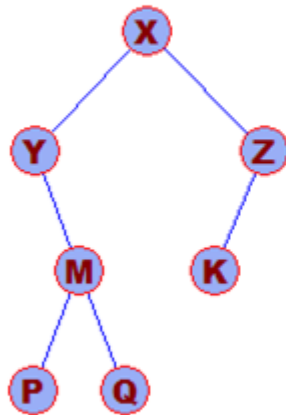
Para su representación a nivel del diseño de clases en UML, podemos construir varios modelos validos e implementables para una multilista, destinada a contener los nodos y operaciones de un árbol n-ario; aparte de los muchos modelos que existen al respecto, de distintos autores en sus libros y otras publicaciones. En cualquier caso, no sobra recordar que un mismo diagrama de clases puede significar una mayor o menor complejidad en su implementación de acuerdo al lenguaje. En este orden de ideas el diagrama UML de clases candidato al respecto es el siguiente:



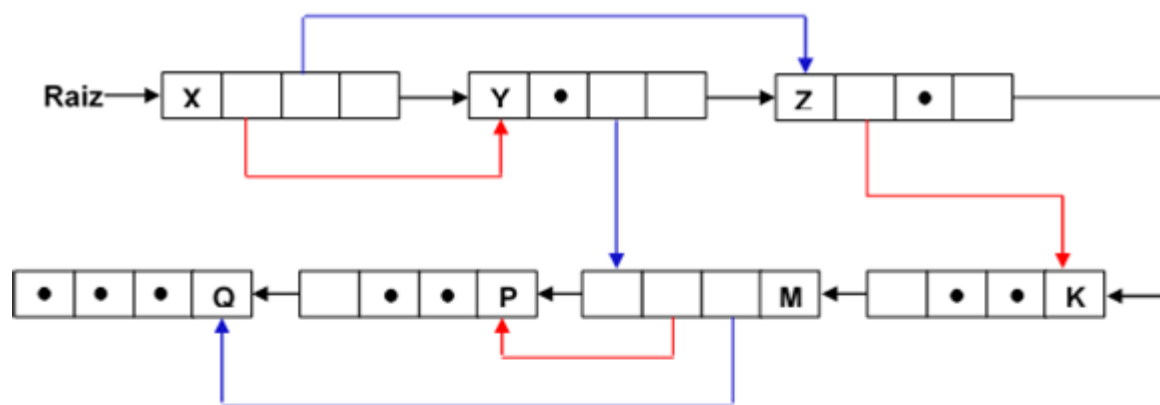


Observe como entre las clases *TNode* y *TLista* se modela lo que normalmente hacemos con una lista enlazada simple. La clase *TNodeArbol* para el árbol hereda de la clase *TNode*, de modo que para la clase *TNodeArbol* el atributo *Siguiente* debe ser interpretado como un enlace al siguiente hermano (si se quiere en orden cronológico de nacimiento). Esta última clase además, añade el atributo *Hijos* que es del tipo *TLista*; de tal suerte, que debemos entender que dicho atributo mantiene una lista enlazada que contiene todos los nodos hijos; específicamente diremos que la cabeza de dicha lista apunta al primer hijo del nodo.

En cuanto a la representación del árbol binario, queda claro que para ellos podemos emplear cualquiera de los métodos presentados anteriormente, toda vez que un árbol binario es por lógica un caso particular de un árbol n-ario o general. Así ejemplo, dado el siguiente árbol binario:



Una representación de este mediante listas enlazadas simples es la siguiente:

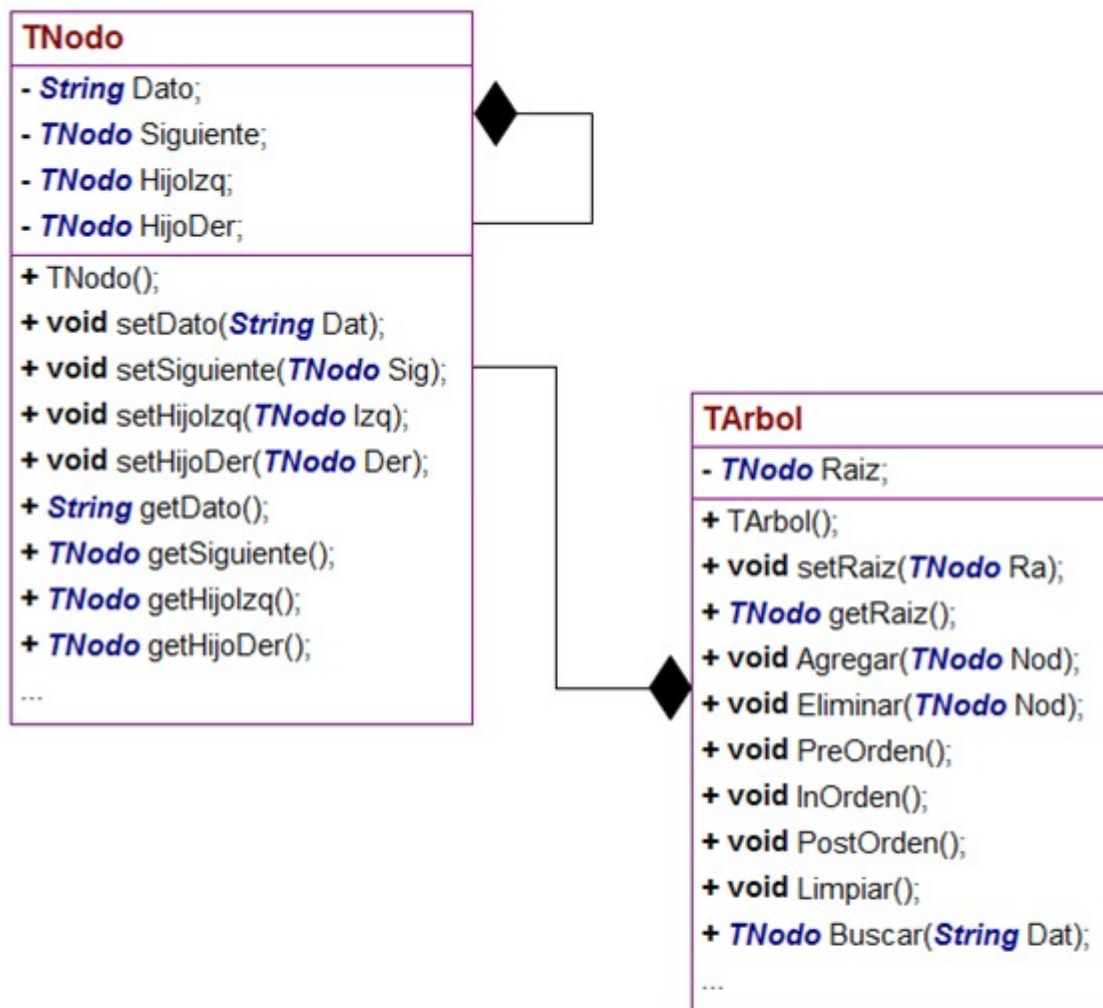


En esta representación cada nodo del árbol binario se guarda en una lista enlazada simple, y la clase para el nodo, además del atributo *Siguiente*, define otros dos adicionales, que son apuntadores a los nodos hijos izquierdo y derecho respectivamente. En el gráfico las flechas rojas muestran el enlace al hijo izquierdo, la azul el enlace al hijo derecho y la negra el enlace al nodo siguiente; el punto (•) en negrilla es el valor de **null**. Gráficamente cada nodo tiene esta estructura:

Dato	Izquierdo	Derecho	Siguiente
------	-----------	---------	-----------

Por su parte, la cabeza de la lista será la raíz del árbol; con lo cual concluimos, que en este mecanismo de representación del árbol binario, la clase para el árbol es una

combinación entre una lista enlazada con un árbol propiamente dicho; todo en una misma clase que incluye operaciones y atributos de estas dos estructuras de datos. Un diagrama de clase en UML para esta forma de representación del árbol es este:

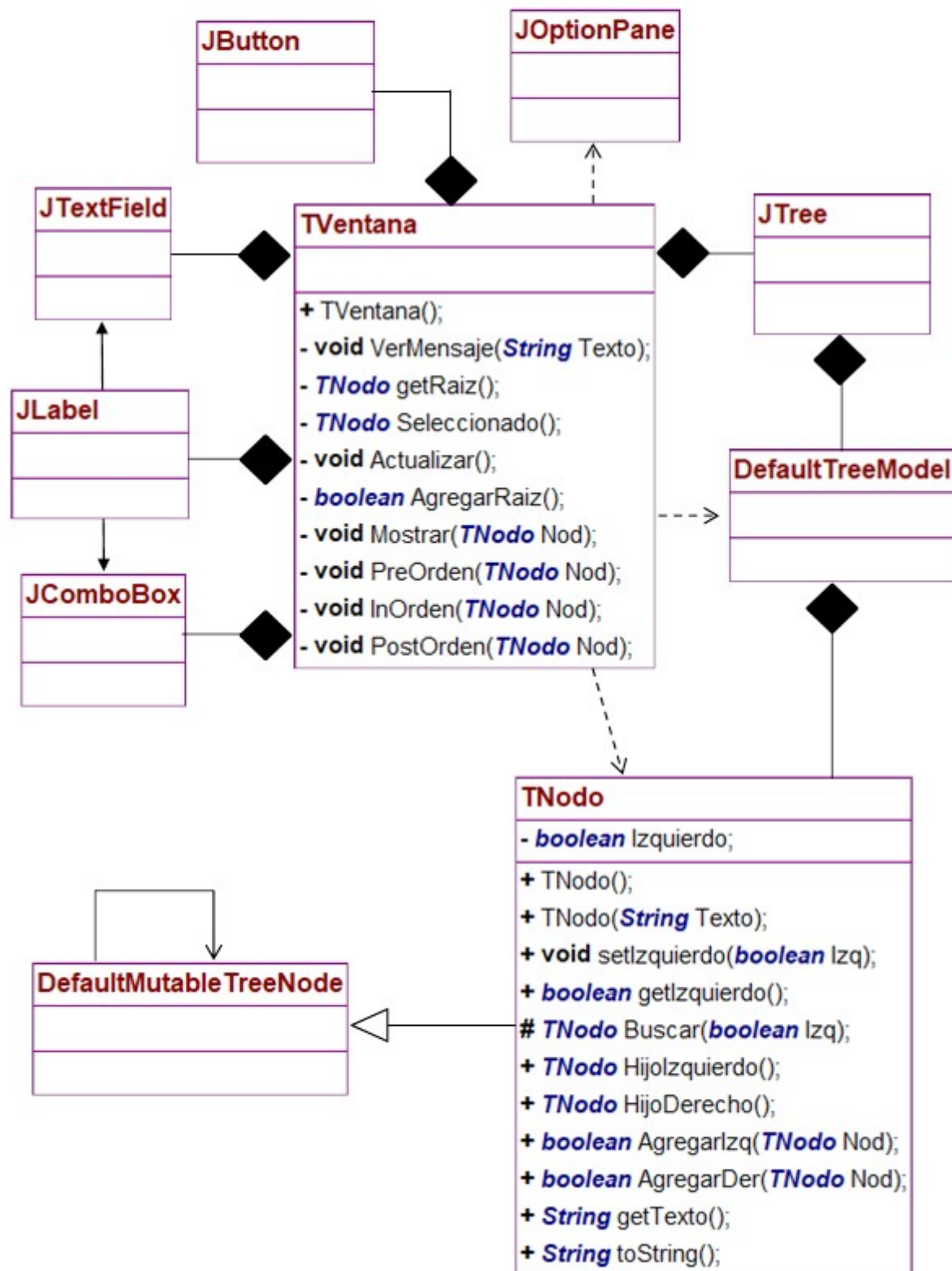


## 8. Ejemplo propuesto de árboles binarios

Como ejemplo práctico, para aplicar arboles binarios, consideremos una aplicación en Java con interfaz gráfica de usuario usando un *JFrame Form*, que le permita al usuario crear gráficamente un árbol binario; de modo que se habiliten las operaciones de eliminación de un nodo, borrado de todos los nodos del árbol, realización y despliegue de los tres tipos de recorridos en profundidad, así como efectuar algunas validaciones básicas, como por ejemplo que un nodo no tenga más de dos hijos.

## 8.1 Diagrama UML de clases

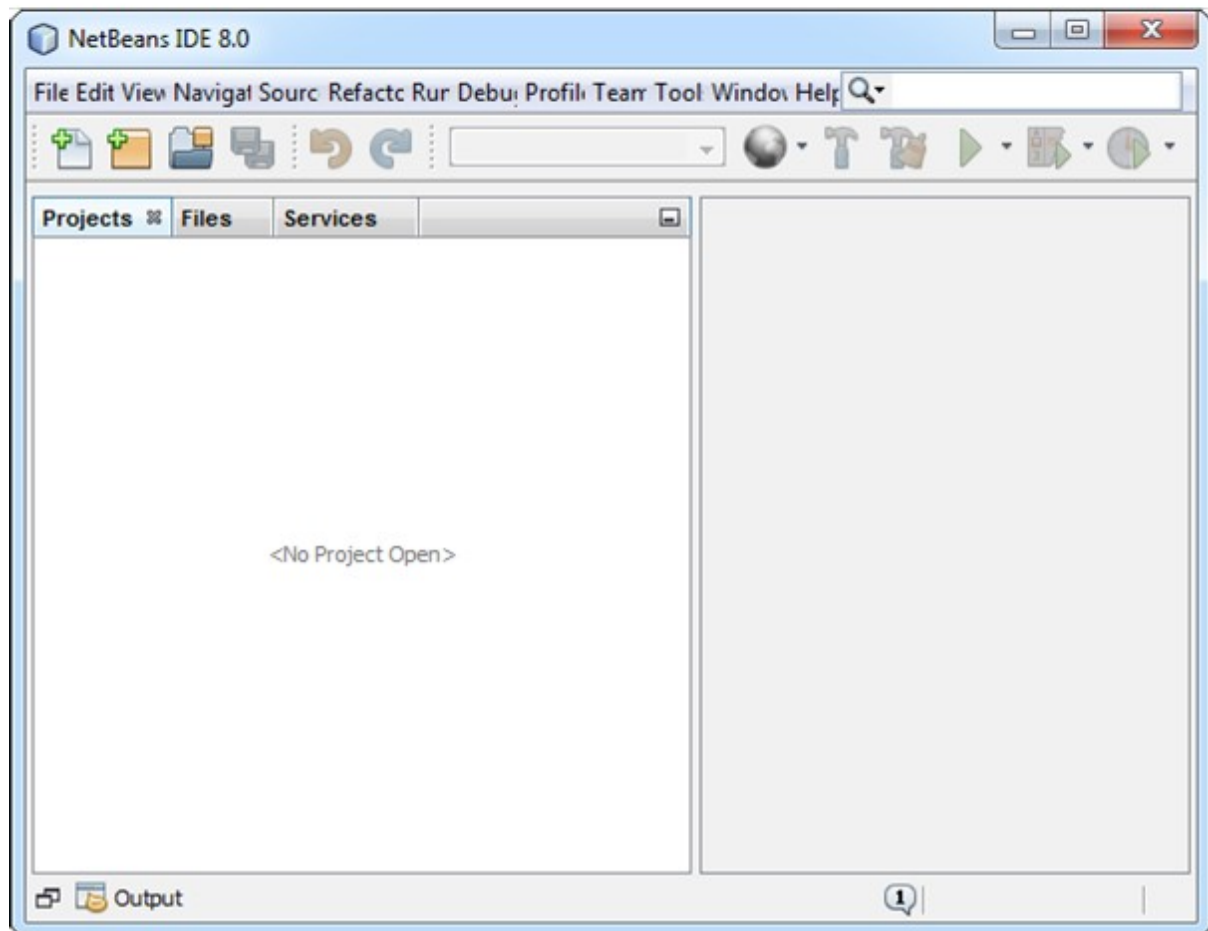
En la imagen de abajo, observamos el diagrama UML que será implementado para esta aplicación de ejemplo, correspondiente a arboles binarios.



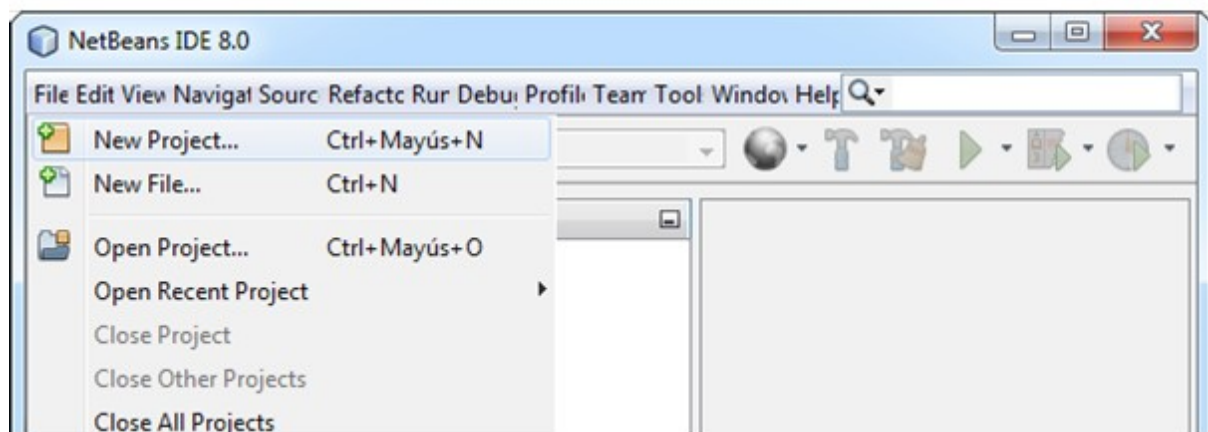


## 8.2 Creación del proyecto de ventana en *NetBeans*

➤ Inicie el IDE de *NetBeans* con lo cual se muestra una pantalla como siguiente:

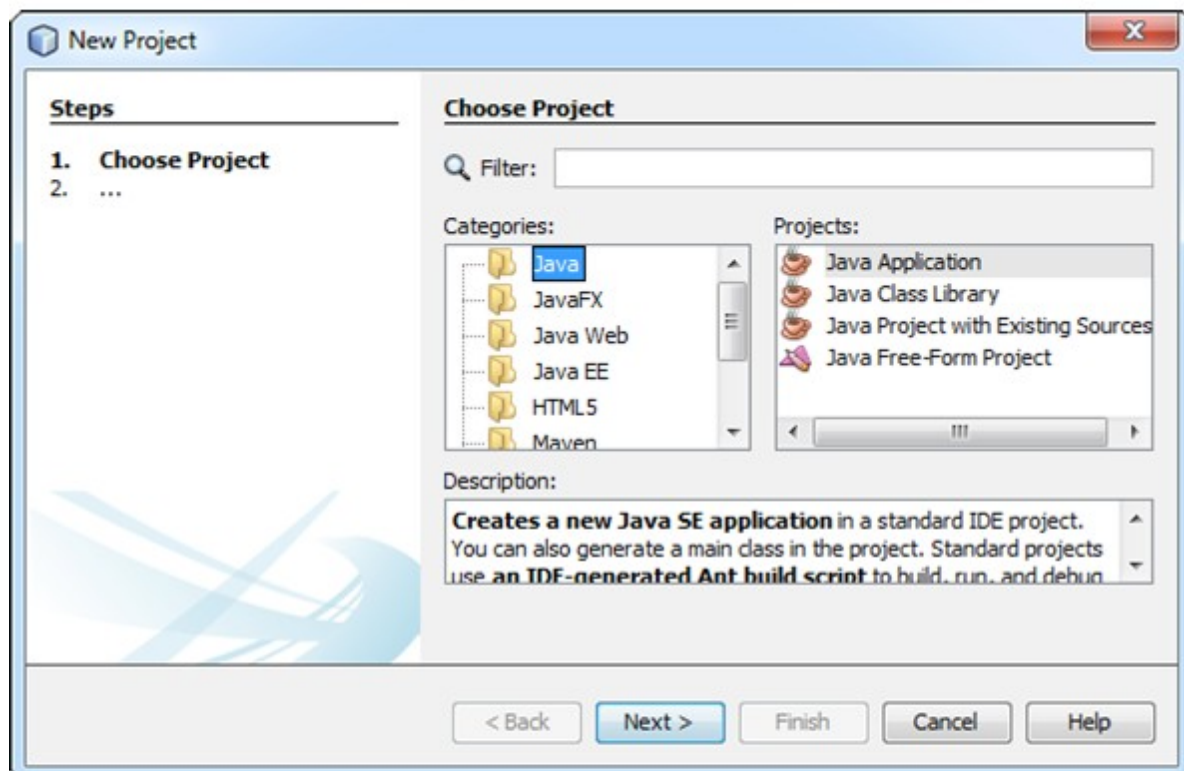


➤ Para crear el proyecto entre por la opción **File + New Project**, tal como lo indica la siguiente imagen:

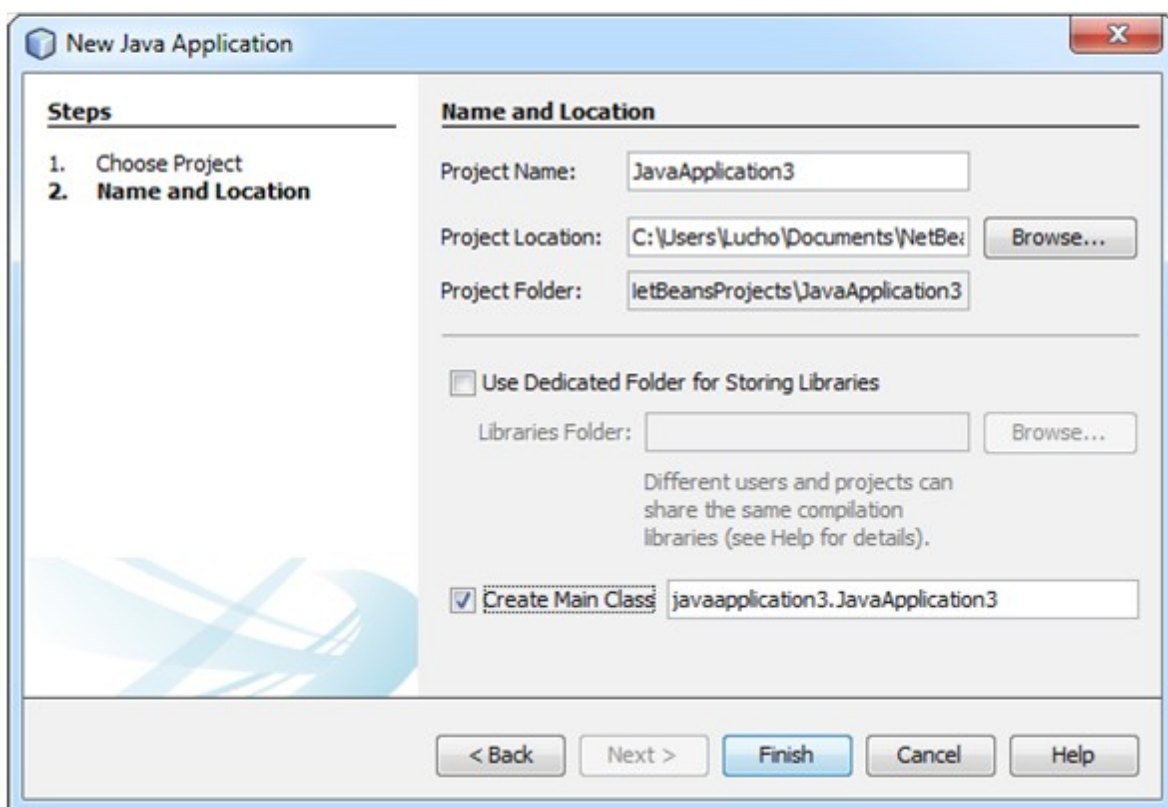




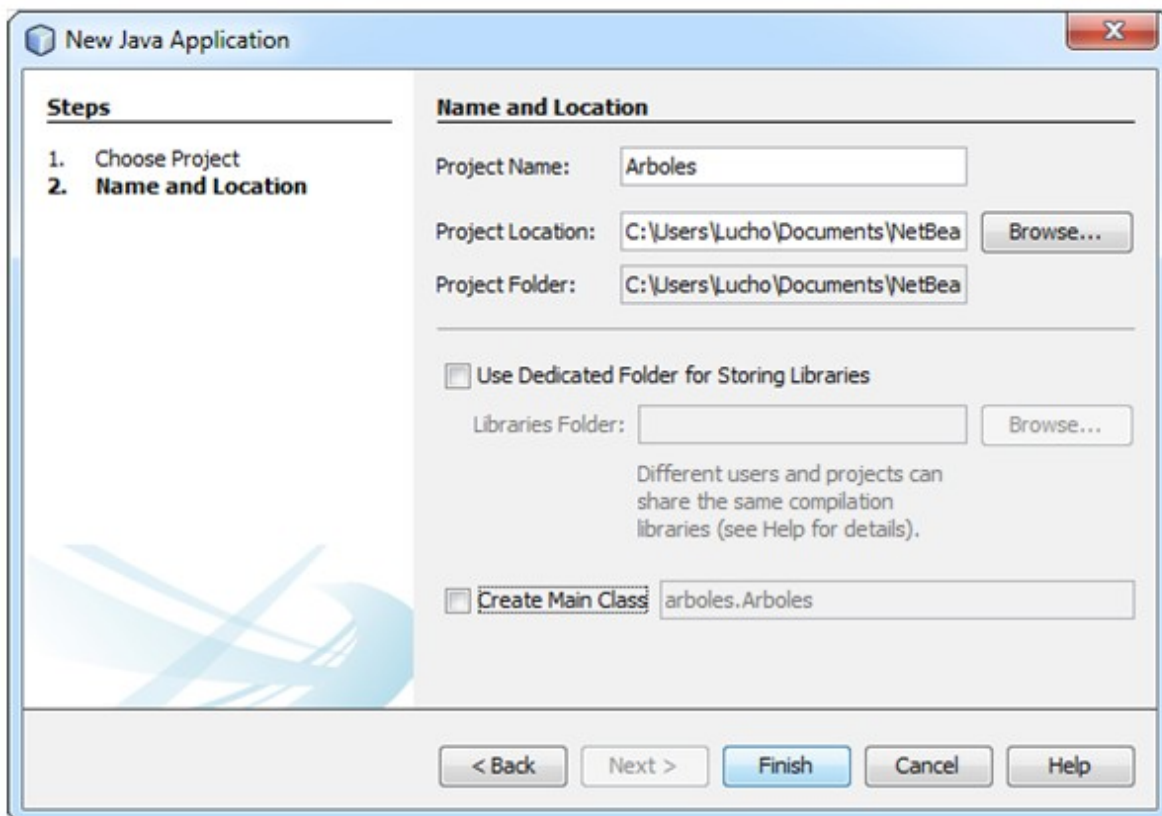
➡ Presentándose la ventana siguiente:



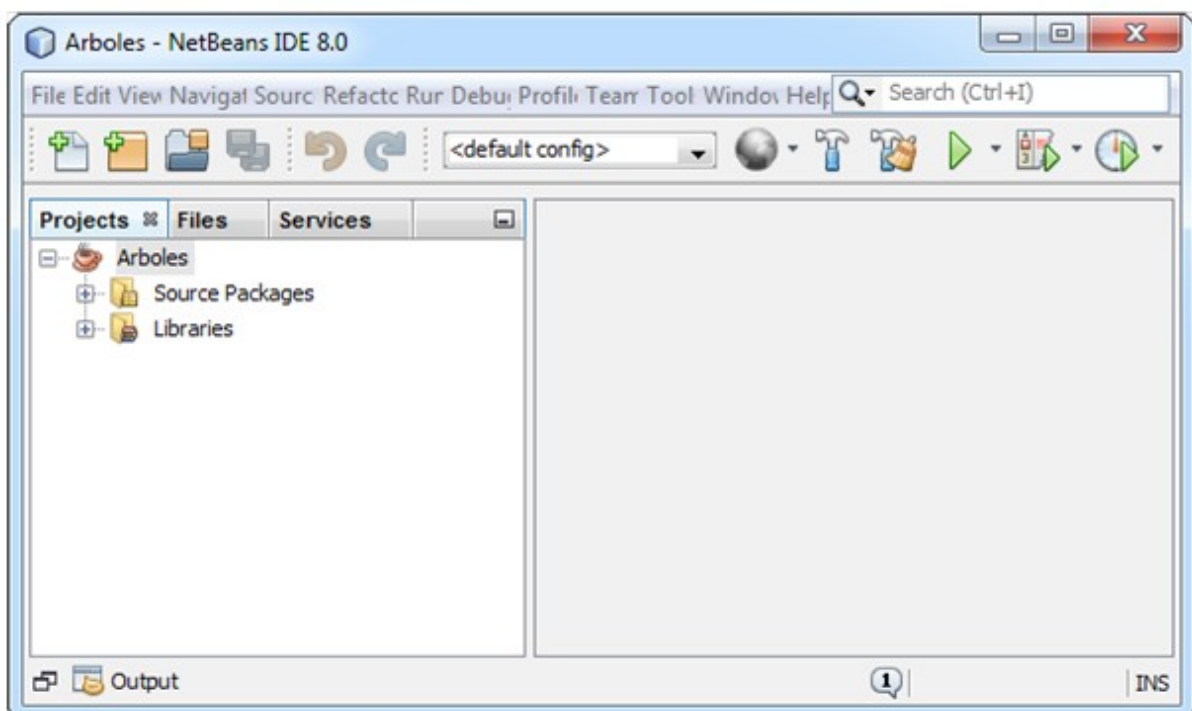
➡ Seleccione el nodo Java en panel de **Categories**, en el panel proyectos el nodo **Java Application** y pulse el botón **Next**; con ello se ve la ventana siguiente:



- Ponga **Arboles** como nombre del proyecto, desmarque la casilla de verificación **Create Main Class** y haga click en el botón **Finish**, como se indica en la imagen de abajo :

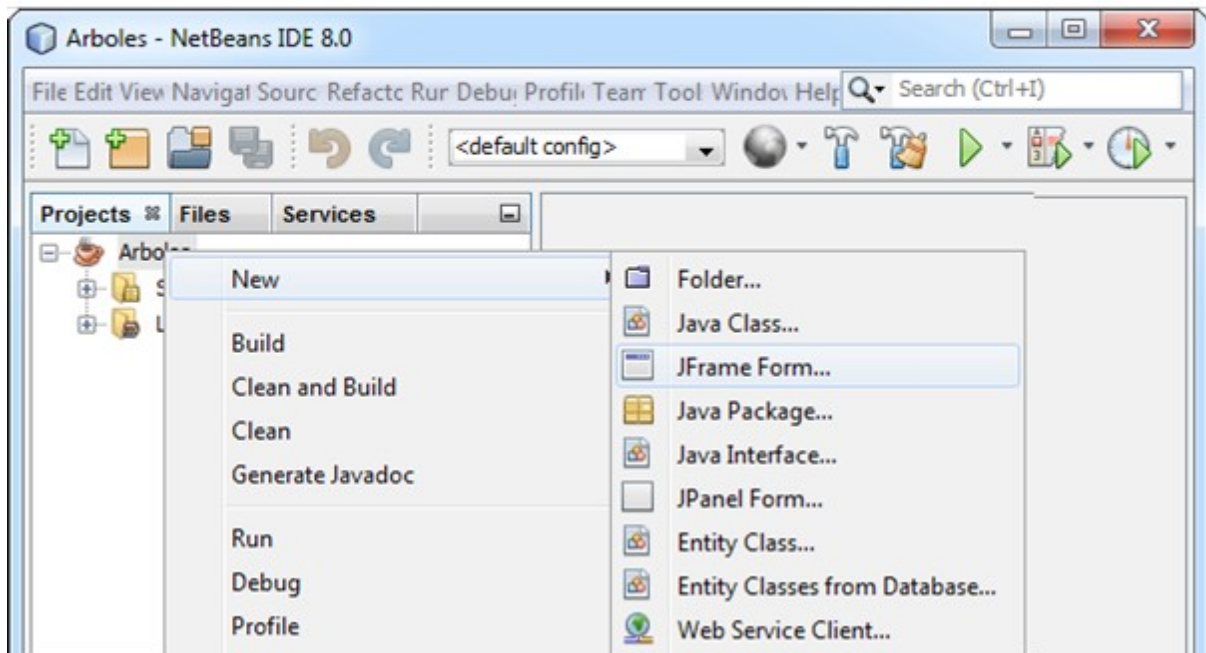


- El proyecto ahora se verá como lo muestra la siguiente imagen:

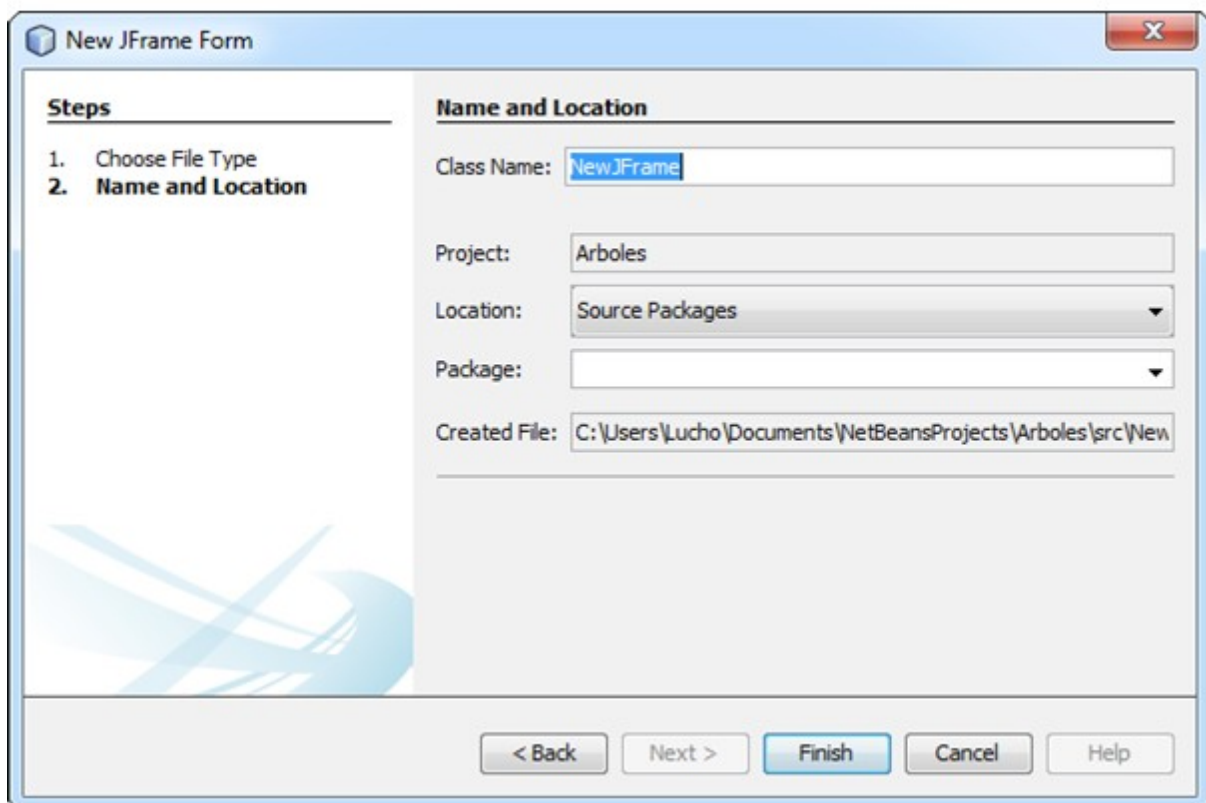


### 8.3 Creación de la ventana para el proyecto

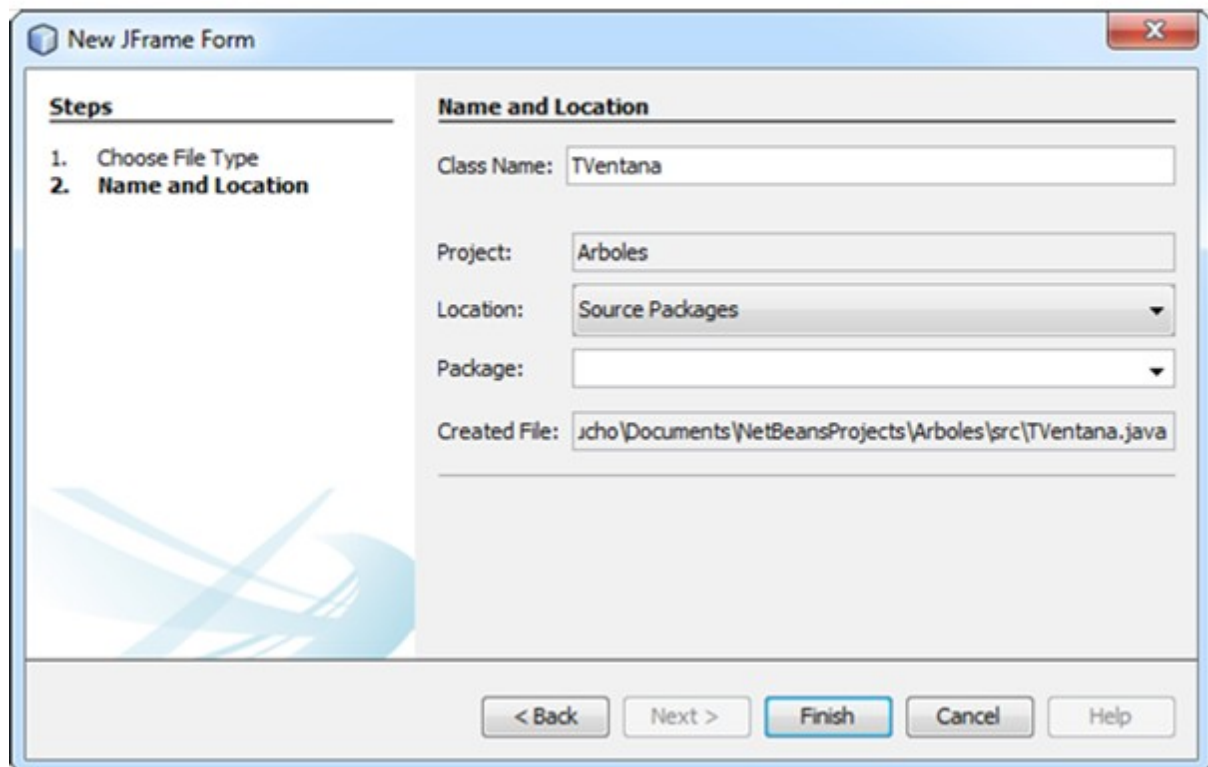
- ➡ Para crear la nueva ventana, haga click derecho en el nombre del paquete del proyecto y seleccione **New + JFrame Form** como se aprecia en esta imagen:



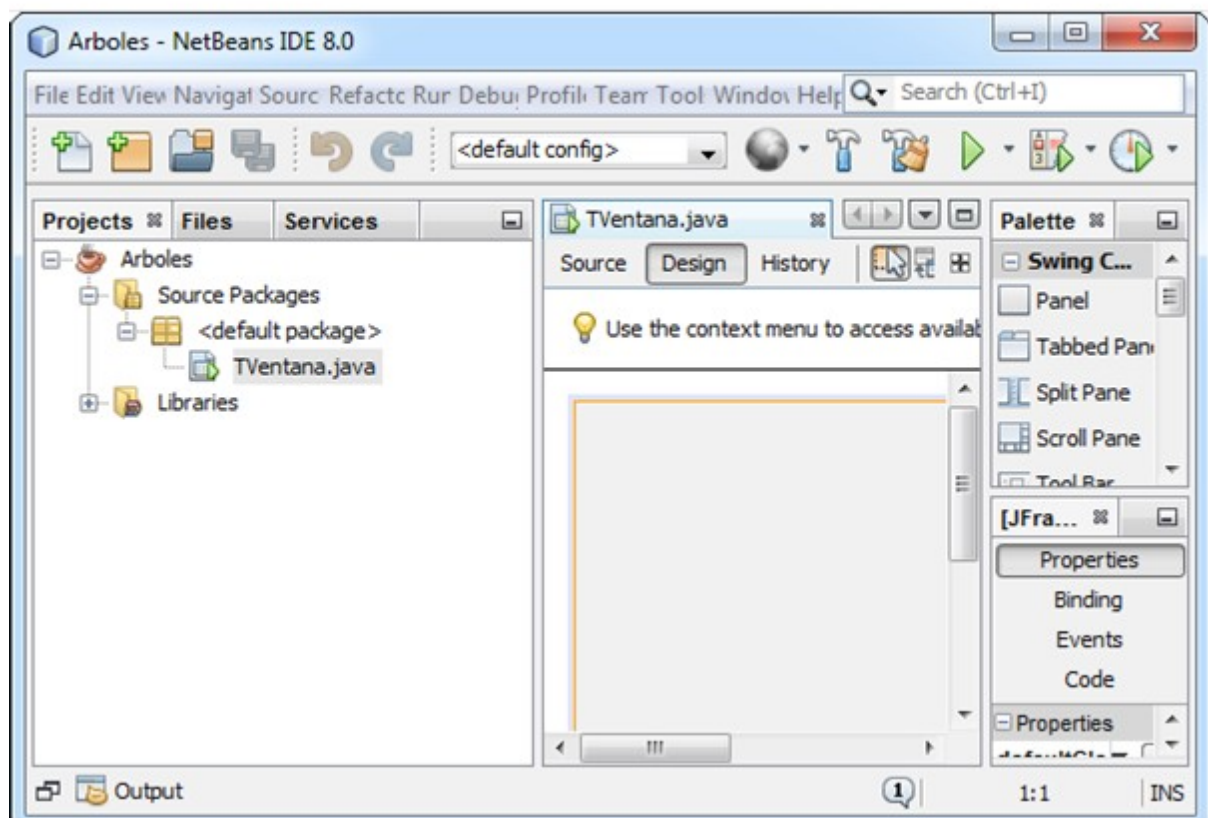
- ➡ Con ello se despliega la siguiente ventana:



- Ponga **TVentana** como nombre de la clase para la ventana y haga click en el botón **Finish**.



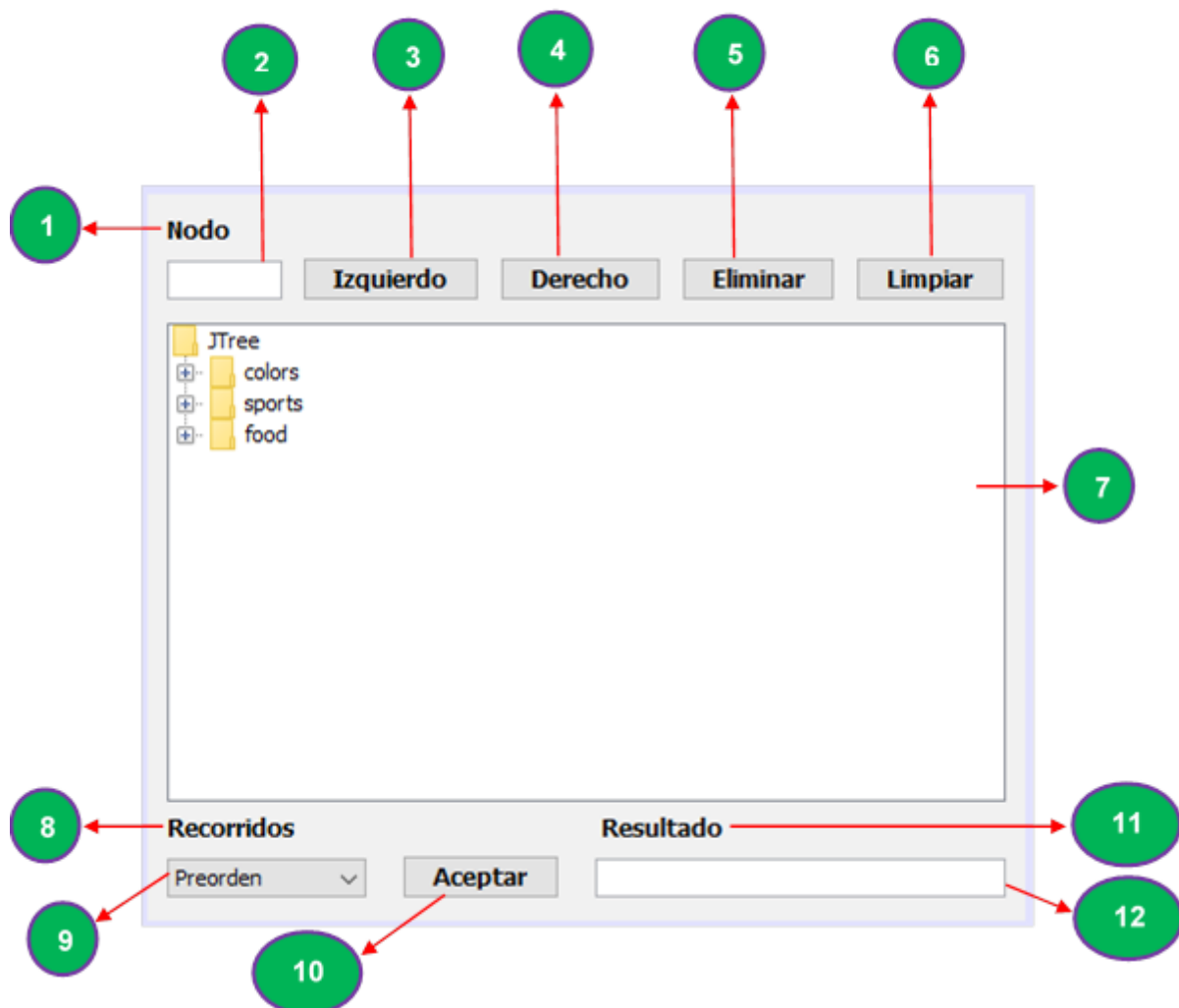
- Ahora su proyecto tendrá el siguiente aspecto:





## 8.4 Diseño de la ventana para el proyecto

➡ El diseño de la ventana es el que se muestra en la siguiente imagen:



➡ Para diseñar la ventana ponga en su propiedad **Title** el texto **Ejemplo arboles binarios** y asegúrese de arrastrar los componentes adecuados hasta la ventana, según la imagen anterior, configurando los ilustrados con números según la siguiente tabla:

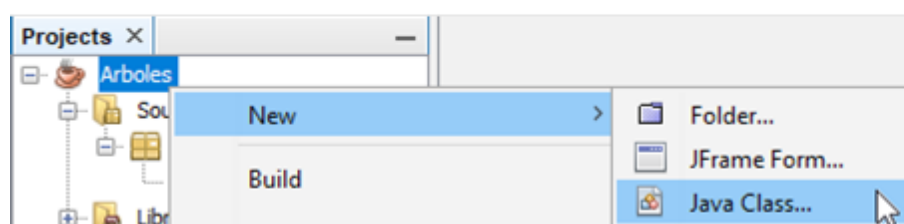
Nº	Componente	Nombre Variable	Otras propiedades	
1	JLabel		text	Nodo
			font	font   Tahoma 12 Bold
2	JTextField	tf1		
3	JButton	B1	text	Izquierdo
			font	font   Tahoma 12 Bold



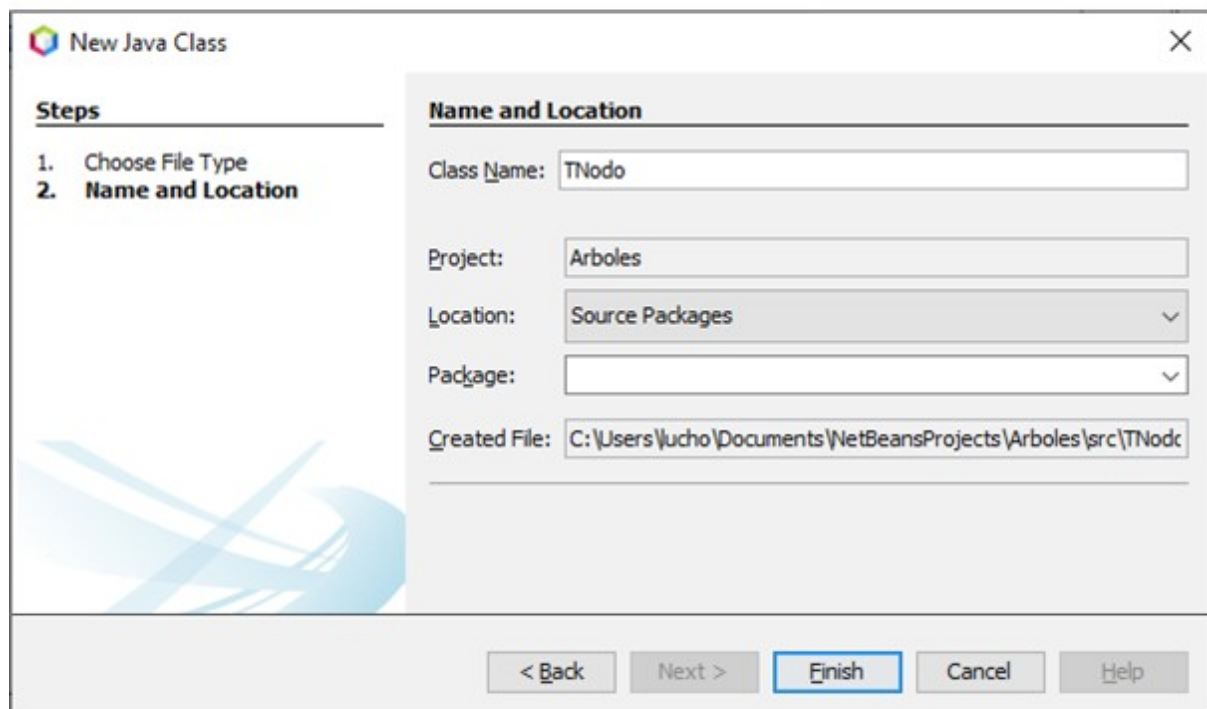
Nº	Componente	Nombre Variable	Otras propiedades	
4	JButton	B2	text	Derecho
			font	font Tahoma 12 Bold
5	JButton	B3	text	Borrar
			font	font Tahoma 12 Bold
6	JButton	B4	text	Limpiar
			font	font Tahoma 12 Bold
7	JTree	Arb		
8	JLabel		text	Recorridos
			font	font Tahoma 12 Bold
9	JComboBox	cb1	<div> <div>model</div> <div></div> </div>	Click al botón de tres puntos y asignar los valores de listados abajo:
			Preorden InOrden PostOrden	
10	JButton	B5	text	Aceptar
			font	font Tahoma 12 Bold
11	JLabel		text	Resultado
			font	font Tahoma 12 Bold
12	JTextField	tf2	<div> <div>editable</div> <div><input type="checkbox"/></div> </div>	(False, desmarcar el checkbox)

### 8.5 Creación clase *TNodo*

- Haga click derecho sobre el proyecto **Arboles** y luego sobre las opciones **New + Java Class**.

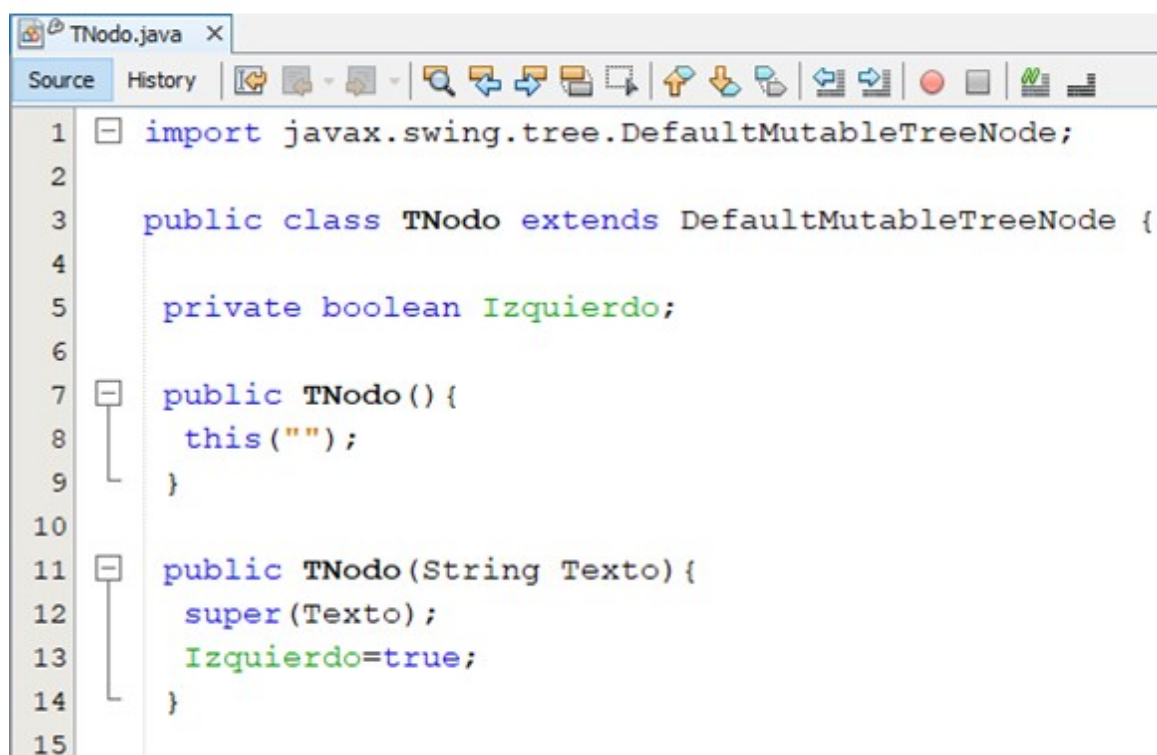


- En la ventana que se despliega ponga **TNodo** en la entrada **Class Name** y haga click en el botón **Finish**.



## 8.6 Implementación clase **TNodo**

Ahora asegúrese que el código de la clase **TNodo** sea como sigue:



```
1 import javax.swing.tree.DefaultMutableTreeNode;
2
3 public class TNodo extends DefaultMutableTreeNode {
4
5     private boolean Izquierdo;
6
7     public TNodo() {
8         this("");
9     }
10
11     public TNodo(String Texto) {
12         super(Texto);
13         Izquierdo=true;
14     }
15
```

```

16  public void setIzquierdo(boolean Izq) {
17      Izquierdo=Izq;
18  }
19
20  public boolean getIzquierdo() {
21      return Izquierdo;
22  }
23
24  protected TNode Buscar(boolean Izq) {
25      int i;
26      TNode Nod;
27      Nod=null;
28      for(i=0;i<getChildCount();i++){
29          Nod=(TNode)getChildAt(i);
30          if(Nod.getIzquierdo()==Izq){
31              break;
32          }
33          else{
34              Nod=null;
35          }
36      }
37      return Nod;
38  }
39
40  public TNode HijoIzquierdo() {
41      return Buscar(true);
42  }
43
44  public TNode HijoDerecho() {
45      return Buscar(false);
46  }
47
48  public boolean AgregarIzq(TNode Nod) {
49      if(HijoIzquierdo()==null){
50          Nod.setIzquierdo(true);
51          add(Nod);
52          return true;
53      }
54      else{
55          return false;
56      }
57  }

```

```

58
59  public boolean AgregarDer (TNodo Nod) {
60      if(HijoDerecho()==null){
61          Nod.setIzquierdo(false);
62          add(Nod);
63          return true;
64      }
65      else{
66          return false;
67      }
68  }
69
70  public String getTexto(){
71      return getUserObject() + "";
72  }
73
74  @Override
75  public String toString(){
76      String Texto;
77      Texto=getTexto();
78      if(getParent()==null){
79          return Texto;
80      }
81      else
82          if(Izquierdo){
83              return Texto + " (I) ";
84          }
85          else{
86              return Texto + " (D) ";
87          }
88  }
89
90  }

```

## 8.7 Implementación métodos de la ventana

- ➡ Active la vista de fuente (*Source*) de la ventana, antes de la declaración de la clase **TVentana**, añada las líneas para importar clases indicadas en el marco rojo.

```

import javax.swing.JOptionPane;
import javax.swing.tree.DefaultTreeModel;

```

```

public class TVentana extends javax.swing.JFrame {

```



- ➡ Debajo de la declaración de la clase **TVentana**, añada la declaración del atributo para el modelo del árbol, enmarcada en rojo.

```
public class TVentana extends javax.swing.JFrame {  
    private DefaultTreeModel Mod;
```

- ➡ Más abajo en el constructor de la de la clase **TVentana**, inicialice y limpie el modelo, tal como se muestra en las líneas enmarcadas en rojo; además debajo de este constructor agregue los métodos indicados (desde el método *VerMensaje*):

```
public TVentana() {  
    initComponents();  
    Mod=(DefaultTreeModel)Arb.getModel();  
    Mod.setRoot(null);  
}  
  
private void VerMensaje(String Texto){  
    JOptionPane.showMessageDialog(null,Texto);  
}  
  
private TNodo getRaiz(){  
    if(Mod.getRoot()!=null){  
        return (TNodo)Mod.getRoot();  
    }  
    else{  
        return null;  
    }  
}  
  
private TNodo Seleccionado(){  
    if(Arb.getSelectionPath()!=null){  
        return (TNodo)Arb.getLastSelectedPathComponent();  
    }  
    else{  
        return null;  
    }  
}  
  
private void Actualizar(){  
    Mod.reload();  
    tf1.setText("");  
    tf1.grabFocus();  
}
```



```

private boolean AgregarRaiz() {
    if(getRaiz()==null) {
        Mod.setRoot(new TNode(tf1.getText()));
        Actualizar();
        return true;
    }
    else{
        return false;
    }
}

private void Mostrar(TNode Nod) {
    tf2.setText(tf2.getText() + " " + Nod.getText());
}

private void PreOrden(TNode Nod) {
    if(Nod!=null) {
        Mostrar(Nod);
        PreOrden(Nod.HijoIzquierdo());
        PreOrden(Nod.HijoDerecho());
    }
}

private void InOrden(TNode Nod) {
    if(Nod!=null) {
        InOrden(Nod.HijoIzquierdo());
        Mostrar(Nod);
        InOrden(Nod.HijoDerecho());
    }
}

private void PostOrden(TNode Nod) {
    if(Nod!=null) {
        PostOrden(Nod.HijoIzquierdo());
        PostOrden(Nod.HijoDerecho());
        Mostrar(Nod);
    }
}

```

- Ahora active la vista de diseño (**Desing**) de la ventana, haga doble click en el botón **Izquierdo**; asegúrese que el código para el evento click del botón sea así:

```
private void B1ActionPerformed(java.awt.event.ActionEvent evt) {  
    TNode Padre;  
    if(!AgregarRaiz()){  
        Padre=Seleccionado();  
        if(Padre==null){  
            VerMensaje("Selecione Nodo Padre");  
        }  
        else  
            if(Padre.AgregarIzq(new TNode(tf1.getText()))){  
                Actualizar();  
            }  
            else{  
                VerMensaje("El padre ya tiene hijo izquierdo");  
            }  
        }  
    }  
}
```

- Active nuevamente la vista de diseño, haga doble click en el botón **Derecho**; asegúrese que el código para el evento click del botón sea como sigue:

```
private void B2ActionPerformed(java.awt.event.ActionEvent evt) {  
    TNode Padre;  
    if(!AgregarRaiz()){  
        Padre=Seleccionado();  
        if(Padre==null){  
            VerMensaje("Selecione Nodo Padre");  
        }  
        else  
            if(Padre.AgregarDer(new TNode(tf1.getText()))){  
                Actualizar();  
            }  
            else{  
                VerMensaje("El padre ya tiene hijo derecho");  
            }  
        }  
    }  
}
```

- Vaya otra vez a la vista de diseño (*Desing*), haga doble click en el botón **Eliminar**, asegúrese que el código para el evento click del botón sea este:

```
private void B3ActionPerformed(java.awt.event.ActionEvent evt) {  
    TNode Nod;  
    Nod=Seleccionado();  
    if(Nod!=null){  
        if(Nod!=getRaiz()){  
            Mod.removeNodeFromParent(Nod);  
        }  
    }  
    else{  
        VerMensaje("Seleccione nodo a eliminar");  
    }  
}
```

- Diríjase nuevamente a la vista de diseño de la ventana, haga doble click en el botón **Limpiar** asegúrese que el código para el evento click sea el siguiente:

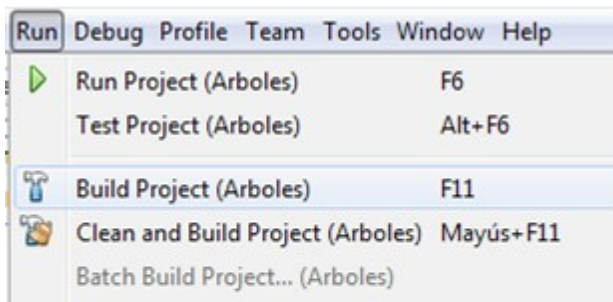
```
private void B4ActionPerformed(java.awt.event.ActionEvent evt) {  
    Mod.setRoot(null);  
    Actualizar();  
}
```

- Finalmente en la vista diseño, haga doble click en el botón **Aceptar** e implemente el evento click del botón así:

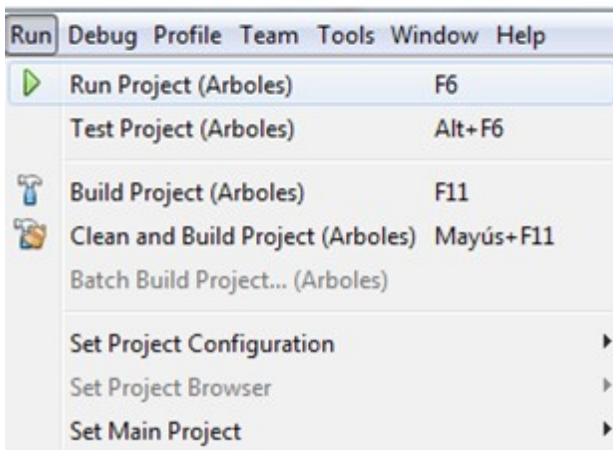
```
private void B5ActionPerformed(java.awt.event.ActionEvent evt) {  
    TNode Raiz;  
    Raiz=getRaiz();  
    if(Raiz==null){  
        tf2.setText("");  
        switch(cb1.getSelectedIndex()){  
            case 0: PreOrden(Raiz); break;  
            case 1: InOrden(Raiz); break;  
            case 2: PostOrden(Raiz);break;  
        }  
    }  
    else{  
        VerMensaje("Arbol Vacio");  
    }  
}
```

## 8.8 Compilación y ejecución del programa.

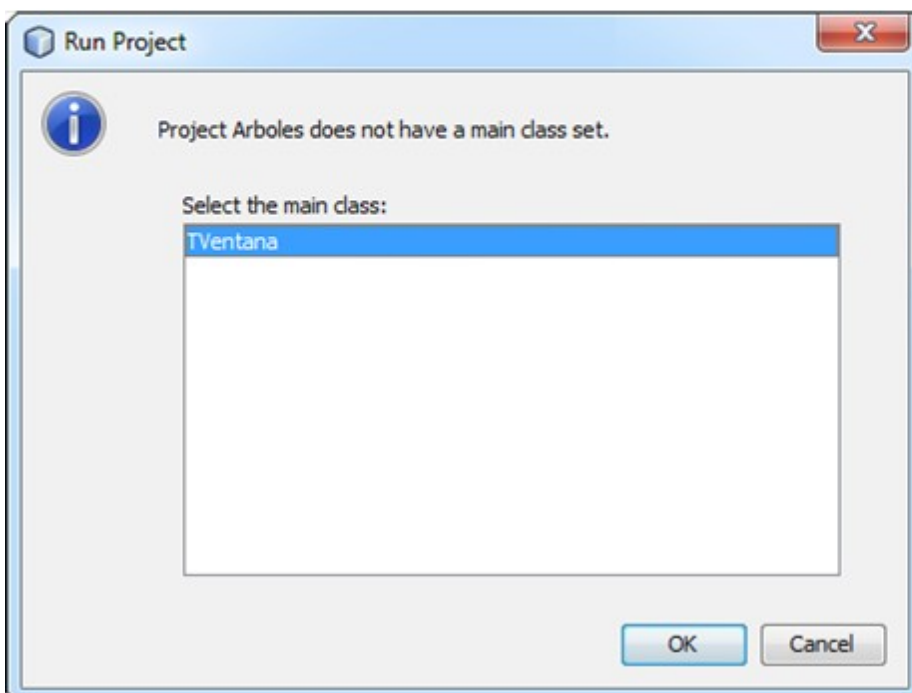
- ➡ Compile con la opción **Run + Build Project** del menú o pulse la tecla F11.



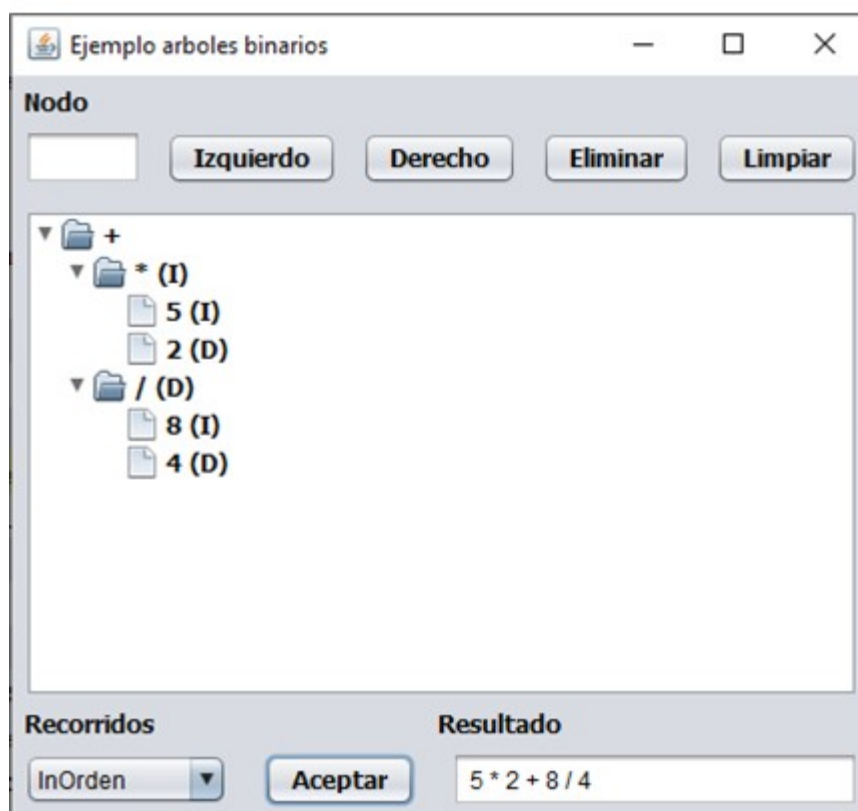
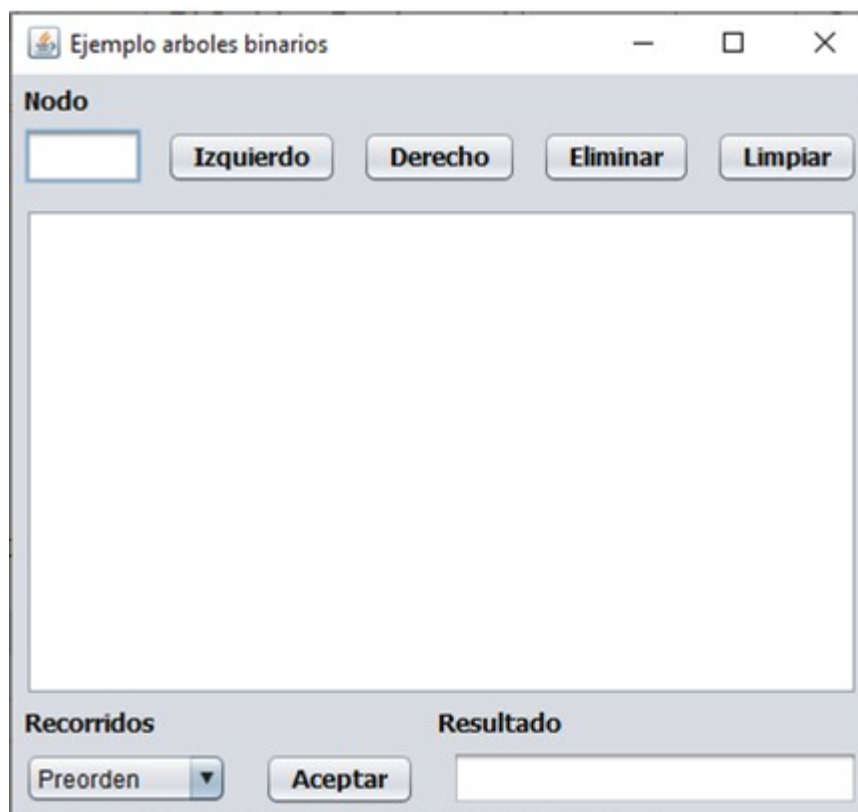
- ➡ Corrija los errores comparando con el código fuente anterior y corra el programa con la opción **Run + Run Project** del menú principal, o pulsando la tecla F6:



- ➡ Si le aparece la siguiente ventana, en ella haga click al botón **Ok**.



Las siguientes son imágenes del programa en ejecución:





## Actividades propuestas

Como complemento al anterior ejemplo desarrolle las siguientes actividades:

1. Para que se le asigna **null** al modelo del árbol (línea `Mod.setRoot(null);` )
2. Para que es el método *getUserObject* de un *DefaultMutableTreeNode*?
3. Para que son los métodos *getFirstChild()*, *getLastChild()*, *getChildCount()* y *getChildAt()* de la clase *DefaultMutableTreeNode*
4. Explique para que se usa el método *removeNodeFromParent* de la clase ***DefaultTreeModel***.
5. Consulte en qué consiste un árbol binario equilibrado, árbol binario balanceado y árbol de búsqueda; cite ejemplos para cada uno de los casos.
6. En el proyecto de ejemplo, añada un botón con el título "*Borrar hijos*", de modo que solo borre los hijos (recursivamente) del nodo seleccionado sin borrar a este.
7. En el proyecto anterior, añada un botón con el título "*Contar*", de modo que cuente cuantos nodos contienen un valor entero par, que además sea múltiplo de tres o cinco.
8. Diseñe en UML e implemente en Java, las clases necesarias para un árbol binario, cuyos nodos almacenan un número entero como atributo; de modo que se permita:
  - a) Calcular el número de niveles del árbol.
  - b) Calcular la altura de un nodo dado.
  - c) Obtener la profundidad de un nodo dato.
  - d) Cuántos nodos hijos izquierdos tiene un valor que es múltiplo de 5.
9. Desarrolle un programa similar al ejemplo presentado anteriormente, pero que además permita realizar las siguientes operaciones:
  - a) Buscar un nodo del árbol dato el texto que lo describe (la etiqueta).
  - b) No aceptar dos nodos con el mismo texto.
  - c) Indicar cuántos nodos hijos tiene el nodo seleccionado en el árbol.

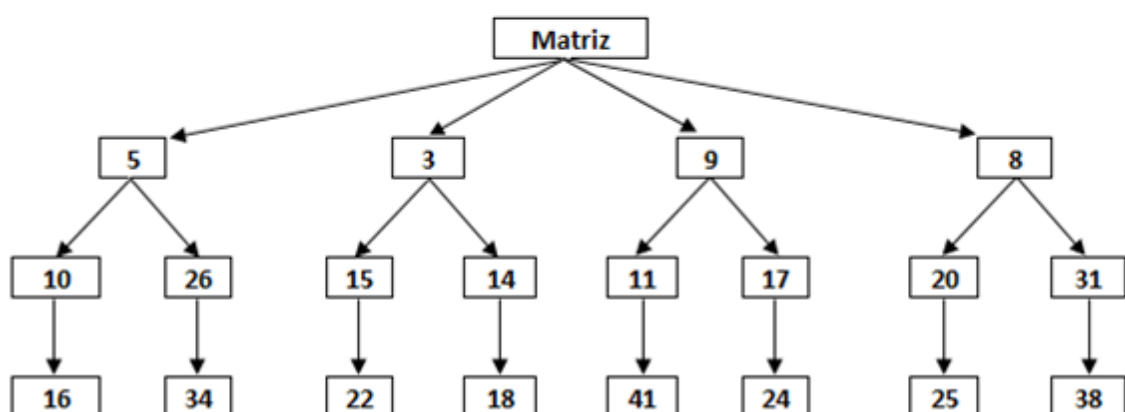
10. Crear un programa de ventana similar al presentado como ejemplo, que sea capaz de llenar el árbol (**JTree**) con los datos de una matriz de tamaño MxN (M número de filas y N número de columnas) de modo que:

- ♦ El usuario pueda ingresar el número de filas.
- ♦ El usuario pueda ingresar el número de columnas.
- ♦ El árbol tendrá una sola raíz, pero será un árbol de grado general.
- ♦ Cada fila de la matriz será un nodo hijo de la raíz del árbol; dichos nodos hijos tendrán como título (contenido, etiqueta o texto) el valor de la primera celda de la fila; en consecuencia el nodo raíz tendrá tantos hijos como filas tenga la matriz.
- ♦ Cada nodo creado por fila, tendrá una cantidad de nodos hijos igual a la mitad de del número de columnas menos uno; es decir :  $(N-1) / 2$
- ♦ El contenido de los nodos del punto anterior, son los valores de las celdas de la primera mitad de las columnas de la matriz.
- ♦ Cada uno de estos últimos nodos tendrá un solo nodo hijo, correspondiente a un único elemento de la segunda mitad de la columna en cuestión de la matriz.

La siguiente imagen muestra un caso de la matriz de 4x5:

5	10	26	16	34
3	15	14	22	18
9	11	17	41	24
8	20	31	25	38

La siguiente imagen es otra ilustración de cómo se organiza el árbol según la matriz de ejemplo:



La imagen de abajo muestra cómo debe quedar el árbol en la ventana después de llenarse con los datos de la matriz anterior de ejemplo:

