



**EDUCACIÓN**  
SECRETARÍA DE EDUCACIÓN PÚBLICA



**TECNM**  
TECNOLÓGICO NACIONAL DE  
**MÉXICO**



## **Tecnológico Nacional de México Campus Orizaba**

### **Estructura de Datos**

### **Ingeniería en Sistemas Computacionales**

#### **Tema 4: Estructuras no Lineales**

#### **Integrantes:**

**Muñoz Hernández Vania Lizeth – 21011009**  
**Romero Ovando Karyme Michelle – 21011037**  
**Flores Domínguez Ángel Gabriel – 21010951**  
**Castillo Solís Luis Ángel – 21010932**

**Grupo:  
3g2B**

**Fecha de entrega: 02/Jun /2023**

## 1. Introducción

Las estructuras de datos son elementos fundamentales para organizar y manipular grandes volúmenes de información de manera eficiente. En las unidades anteriores hemos utilizado estructuras de datos lineales, como arreglos y listas enlazadas, que almacenan los elementos de forma secuencial.

Sin embargo, en muchos casos, las estructuras lineales no son suficientes para modelar relaciones complejas y jerárquicas entre los datos. Aquí es donde entran en juego las estructuras no lineales, que permiten representar y organizar la información de una manera más flexible y dinámica.

Las estructuras no lineales son aquellas en las que los elementos no están dispuestos de forma secuencial, sino que pueden estar conectados entre sí de diversas formas, creando relaciones más complejas. Estas estructuras son especialmente útiles cuando se necesita representar relaciones jerárquicas, como árboles o grafos, donde un elemento puede tener múltiples hijos o estar conectado a varios otros elementos.

Un ejemplo común de una estructura no lineal es el árbol binario, que consta de nodos interconectados en una estructura jerárquica, donde cada nodo puede tener hasta dos hijos. Los árboles binarios son utilizados en la implementación de algoritmos de búsqueda y ordenamiento.

Otra estructura no lineal importante es el grafo, que consta de nodos interconectados llamados vértices, donde cada vértice puede estar conectado a otros vértices a través de aristas. Los grafos son utilizados para modelar relaciones entre entidades en redes sociales, rutas de transporte, sistemas de información geográfica y muchas otras aplicaciones.

Las estructuras no lineales ofrecen una mayor flexibilidad y capacidad de representación en comparación con las estructuras lineales. Permiten modelar relaciones más complejas entre los datos y facilitan la implementación de algoritmos y operaciones específicas, como la búsqueda, inserción, eliminación y recorrido de los elementos.

## 2. Competencia específica

Conoce, comprende y aplica eficientemente estructuras de datos, estructuras no lineales y búsqueda para la optimización del rendimiento de soluciones a problemas del mundo real.

## 3. Marco Teórico

### 4.1 Árboles

Los árboles son una estructura de datos no lineal fundamental en el campo de la informática y la programación. Se caracterizan por su organización jerárquica, donde un elemento principal, llamado raíz, se conecta a varios elementos secundarios, conocidos como nodos, a través de enlaces llamados ramas o aristas. A su vez, estos nodos pueden estar conectados a otros nodos, formando subárboles.

La estructura de un árbol se asemeja a un árbol invertido, donde la raíz se encuentra en la parte superior y las ramas se extienden hacia abajo, dividiéndose en ramas más pequeñas. Cada nodo puede tener cero o más hijos, excepto los nodos terminales, llamados hojas, que no tienen hijos.

Los árboles se utilizan para representar relaciones jerárquicas y anidadas entre los elementos. Son especialmente útiles en situaciones donde los datos deben organizarse en niveles o categorías, como la representación de estructuras empresariales, sistemas de archivos, árboles genealógicos y estructuras de datos de búsqueda y ordenamiento.

Algunos conceptos importantes relacionados con los árboles son:

1. Raíz: Es el nodo principal del árbol, desde donde se inicia la estructura.
2. Nodo: Representa un elemento dentro del árbol y puede tener cero o más hijos.
3. Rama o arista: Es el enlace que conecta un nodo con sus hijos.
4. Hoja: Son los nodos terminales del árbol, que no tienen hijos.
5. Subárbol: Es un árbol que se encuentra dentro de otro árbol más grande.
6. Nivel: Indica la generación o distancia de un nodo desde la raíz. La raíz se encuentra en el nivel 0, sus hijos directos en el nivel 1, y así sucesivamente.
7. Altura: Es la longitud máxima del camino desde un nodo hasta una hoja. La altura de la raíz se considera 0.

8. Árbol binario: Es un tipo especial de árbol en el que cada nodo tiene como máximo dos hijos, denominados hijo izquierdo e hijo derecho.

Los árboles ofrecen varias ventajas en comparación con otras estructuras de datos. Permiten una organización eficiente de los datos, facilitan la búsqueda y el acceso a elementos específicos, y son útiles en la implementación de algoritmos de búsqueda, ordenamiento y recorrido. Además, los árboles pueden adaptarse y crecer dinámicamente a medida que se agregan o eliminan elementos.

#### 4.1.1 Clasificación de árboles

Árbol binario: Un árbol binario es un tipo de árbol en el que cada nodo puede tener como máximo dos hijos. Cada nodo se denomina nodo padre, y los dos nodos que se derivan directamente de él se denominan nodo hijo izquierdo y nodo hijo derecho.

Árbol binario de búsqueda: Un árbol binario de búsqueda (BST, por sus siglas en inglés) es un tipo de árbol binario en el que para cada nodo, todos los valores de los nodos en su subárbol izquierdo son menores que el valor del nodo, y todos los valores de los nodos en su subárbol derecho son mayores que el valor del nodo. Esta propiedad facilita la búsqueda y la inserción de elementos en el árbol de manera eficiente.

Árbol AVL: Un árbol AVL es un tipo de árbol binario de búsqueda en el que se garantiza que la diferencia de alturas entre el subárbol izquierdo y el subárbol derecho de cada nodo sea como máximo 1. Esto se logra mediante operaciones de rotación que mantienen el equilibrio del árbol. Los árboles AVL están diseñados para asegurar una búsqueda y una inserción eficientes con un tiempo de ejecución equilibrado.

Árbol B: Un árbol B es un árbol de búsqueda multidimensional utilizado en estructuras de datos para organizar grandes conjuntos de datos de manera eficiente. Los árboles B están diseñados para minimizar el número de accesos a disco en operaciones de lectura y escritura.

Árbol Trie: Un árbol Trie, también conocido como árbol de prefijos, es una estructura de datos en la que cada nodo representa un carácter y los nodos a lo largo de un camino desde la raíz hasta un nodo hoja forman una palabra o un prefijo de una palabra. Los árboles Trie son útiles para almacenar y buscar palabras o cadenas de caracteres de manera eficiente.

#### 4.1.2 Operaciones básicas sobre árboles binarios

Las operaciones básicas sobre árboles binarios incluyen:

1. Inserción: Permite agregar un nuevo nodo al árbol binario. La inserción se realiza siguiendo una regla específica dependiendo del tipo de árbol binario. En un árbol binario de búsqueda, por ejemplo, se inserta el nuevo nodo en el lugar correcto según su valor.
2. Eliminación: Permite eliminar un nodo existente del árbol binario. Al igual que con la inserción, la eliminación sigue una regla específica según el tipo de árbol binario. En un árbol binario de búsqueda, por ejemplo, se deben considerar diferentes casos dependiendo de la cantidad de hijos que tenga el nodo a eliminar.
3. Búsqueda: Permite buscar un valor específico en el árbol binario. La búsqueda se realiza comparando el valor deseado con los valores de los nodos a medida que se recorre el árbol. En un árbol binario de búsqueda, se puede aplicar la regla de búsqueda para moverse hacia el subárbol izquierdo o derecho según el valor del nodo actual.
4. Recorrido: Los recorridos son operaciones que permiten visitar todos los nodos del árbol en un orden específico. Hay tres tipos comunes de recorridos en árboles binarios:
  - Recorrido en orden (in-order traversal): Se visita primero el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho.
  - Recorrido en preorden (pre-order traversal): Se visita primero el nodo actual, luego el subárbol izquierdo y finalmente el subárbol derecho.
  - Recorrido en postorden (post-order traversal): Se visita primero el subárbol izquierdo, luego el subárbol derecho y finalmente el nodo actual.

Estas son las operaciones básicas más comunes que se realizan sobre árboles binarios. Cada una de estas operaciones puede tener variaciones y particularidades dependiendo de las implementaciones y los requisitos específicos de la aplicación.

#### 4.1.3 Aplicaciones

Los árboles se utilizan en una amplia variedad de aplicaciones en estructuras de datos debido a sus propiedades y características. Algunas de las aplicaciones más comunes de los árboles en estructuras de datos son las siguientes:

1. Árboles de búsqueda: Los árboles binarios de búsqueda y sus variantes, como los árboles AVL y los árboles rojinegros, son utilizados para implementar estructuras de búsqueda eficientes. Estas estructuras permiten realizar búsquedas, inserciones y eliminaciones de elementos en tiempo logarítmico, lo que es especialmente útil para aplicaciones que requieren recuperación y manipulación de datos rápidas, como bases de datos y diccionarios.
2. Árboles de expresiones: Los árboles binarios se utilizan para representar y evaluar expresiones matemáticas o lógicas. Cada nodo del árbol representa un operador o un valor, y los subárboles representan las subexpresiones. Esto permite evaluar las expresiones de manera eficiente siguiendo las reglas y precedencias de los operadores.
3. Árboles de compresión: Los árboles se utilizan en algoritmos de compresión de datos, como el árbol de Huffman. Estos árboles se construyen asignando códigos de bits a cada símbolo de entrada en función de su frecuencia de aparición. La estructura del árbol se utiliza para asignar códigos de longitud variable de manera eficiente, lo que permite la compresión y descompresión de datos.
4. Árboles de directorios: Los sistemas de archivos utilizan árboles para organizar y representar la estructura jerárquica de los directorios y archivos en una computadora. Cada nodo del árbol representa un directorio y puede tener nodos hijos que representan subdirectorios o archivos.
5. Árboles de juegos: Los árboles se utilizan en algoritmos de búsqueda de juegos, como el árbol de juego minimax, que se utiliza en juegos como el ajedrez o el go. Los nodos del árbol representan posiciones de juego y los bordes representan las jugadas posibles. Estos árboles permiten encontrar la mejor jugada posible considerando todas las posibles ramificaciones del árbol.

Estas son solo algunas de las aplicaciones comunes de los árboles en estructuras de datos. Los árboles también se utilizan en otras áreas, como algoritmos de grafos, análisis sintáctico de lenguajes de programación y algoritmos de enrutamiento en redes, entre otros. Su versatilidad y eficiencia los convierten en una herramienta poderosa en el diseño de estructuras de datos y algoritmos.

## 4.2 Grafos

Los grafos son una representación abstracta de una colección de elementos llamados nodos (también conocidos como vértices) que están conectados entre sí mediante enlaces llamados aristas. Los grafos se utilizan para modelar relaciones

entre elementos y se pueden aplicar en una amplia variedad de problemas y aplicaciones.

Cuando se habla de árboles en la estructura de datos, se puede considerar un árbol como un tipo especial de grafo acíclico, es decir, un grafo que no tiene ciclos. En un árbol, hay un nodo especial llamado raíz, desde el cual se extienden las ramas (aristas) hacia los demás nodos del árbol. Cada nodo, excepto la raíz, tiene un único nodo padre, y los nodos sin hijos se llaman nodos hoja.

La estructura de árbol impone ciertas restricciones y propiedades sobre la organización y conectividad de los nodos. En particular, no puede haber ciclos en un árbol, lo que significa que no puede haber rutas cerradas a través de las aristas. Además, todos los nodos deben ser accesibles desde la raíz siguiendo una única ruta.

Los árboles se utilizan ampliamente en estructuras de datos y algoritmos. Proporcionan una forma eficiente de organizar y buscar datos jerárquicamente, y se aplican en áreas como bases de datos, sistemas de archivos, algoritmos de búsqueda y recorrido, análisis sintáctico de lenguajes de programación y muchas otras aplicaciones.

#### 4.2.1 Representación de grafos

Existen diferentes formas de representar grafos. Las dos representaciones más comunes son:

1. Matriz de adyacencia: En esta representación, se utiliza una matriz bidimensional para indicar las conexiones entre los nodos. Si un grafo tiene  $N$  nodos, entonces se crea una matriz  $N \times N$ , donde el elemento en la posición  $(i, j)$  representa la existencia de una arista entre los nodos  $i$  y  $j$ . Si hay una arista entre los nodos  $i$  y  $j$ , el valor en la posición  $(i, j)$  puede ser 1 o un peso asociado a esa arista. Si no hay una arista, el valor puede ser 0 o algún otro valor que indique la falta de conexión. Esta representación es eficiente para verificar rápidamente si hay una conexión directa entre dos nodos, pero puede ser ineficiente en términos de espacio si el grafo es grande y disperso.
2. Lista de adyacencia: En esta representación, se utiliza una lista de enlaces para cada nodo del grafo. Cada nodo tiene asociada una lista que contiene los nodos adyacentes a él, es decir, los nodos a los que está directamente conectado por

una arista. Esta lista puede implementarse como un array, una lista enlazada u otra estructura de datos apropiada. Además, cada elemento de la lista puede almacenar información adicional, como pesos de las aristas. Esta representación es eficiente en términos de espacio para grafos dispersos, ya que solo se almacena información sobre las conexiones existentes. Además, es útil para recorrer rápidamente todos los vecinos de un nodo en particular.

Existen otras representaciones más especializadas dependiendo del tipo de grafo y de las operaciones que se deseen realizar sobre él, como la matriz de incidencia, la lista de aristas, entre otras.

#### 4.2.2 Operaciones básicas

Las operaciones básicas en la manipulación de grafos incluyen:

1. Inserción de nodo: Agregar un nuevo nodo al grafo.
2. Inserción de arista: Establecer una conexión entre dos nodos existentes en el grafo. La arista puede tener un peso asociado o ser no ponderada.
3. Eliminación de nodo: Eliminar un nodo existente del grafo, junto con todas las aristas conectadas a él.
4. Eliminación de arista: Eliminar una arista específica entre dos nodos existentes en el grafo.
5. Búsqueda de nodo: Buscar la existencia de un nodo en el grafo.
6. Búsqueda de arista: Buscar la existencia de una arista entre dos nodos en el grafo.
7. Recorrido de grafo: Visitar todos los nodos del grafo siguiendo una estrategia de recorrido específica. Los recorridos comunes incluyen el recorrido en profundidad (DFS) y el recorrido en amplitud (BFS).
8. Verificación de conectividad: Determinar si dos nodos en el grafo están conectados por un camino o una arista.
9. Obtención de vecinos de un nodo: Obtener todos los nodos adyacentes a un nodo dado en el grafo.
10. Obtención de grados de los nodos: Determinar el grado de un nodo, que representa el número de aristas que están conectadas a ese nodo.

Estas operaciones básicas son fundamentales para manipular y trabajar con grafos en estructuras de datos. Las implementaciones específicas de estas operaciones pueden variar según la representación elegida para el grafo.



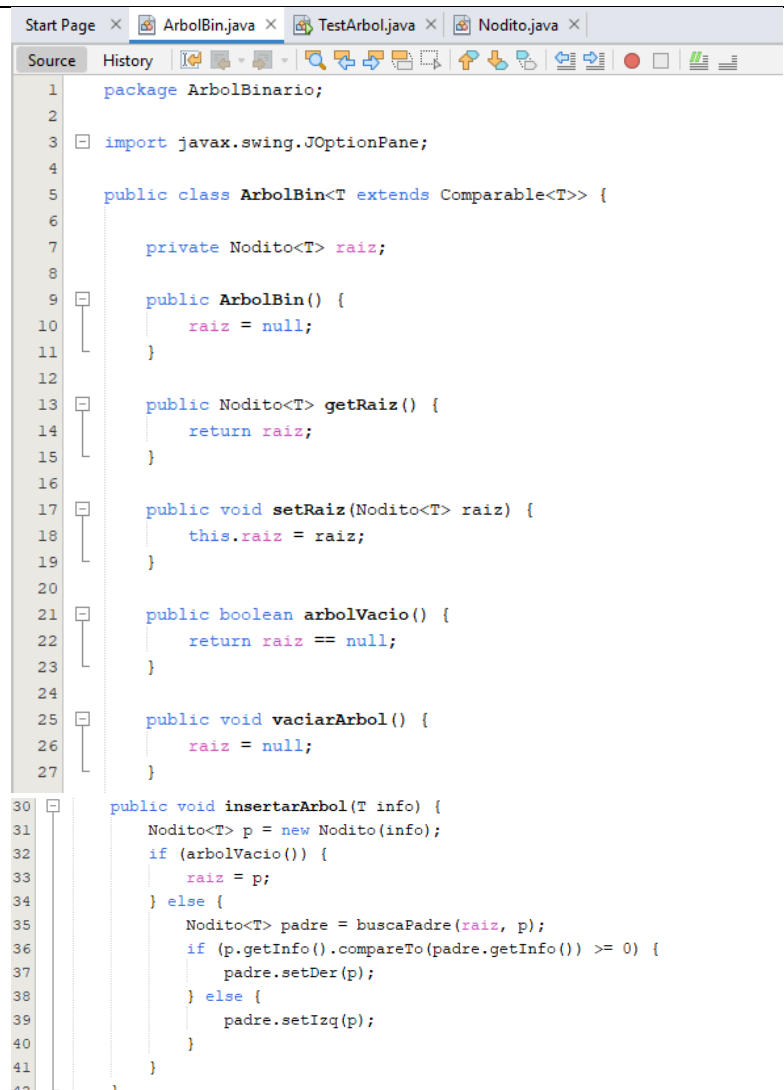
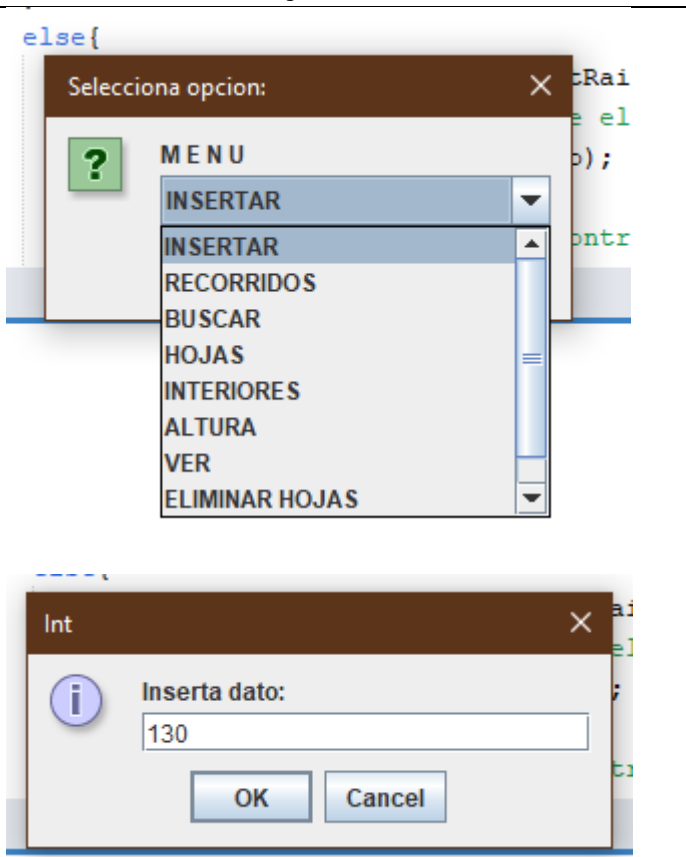
## 4. Material y Equipo

El material y equipo que se necesita para llevar a cabo la práctica son:

- ✓ Computadora
- ✓ Software y versión usados
- ✓ Materiales de apoyo para el desarrollo de la práctica

## 5. Resultados

🚦 EJERCICIO DE ÁRBOLES.

| Clase  | Ejecución   |
|--|---|
|  <pre>1 package ArbolBinario; 2 3 import javax.swing.JOptionPane; 4 5 public class ArbolBin&lt;T extends Comparable&lt;T&gt;&gt; { 6 7     private Nodito&lt;T&gt; raiz; 8 9     public ArbolBin() { 10         raiz = null; 11     } 12 13     public Nodito&lt;T&gt; getRaiz() { 14         return raiz; 15     } 16 17     public void setRaiz(Nodito&lt;T&gt; raiz) { 18         this.raiz = raiz; 19     } 20 21     public boolean arbolVacio() { 22         return raiz == null; 23     } 24 25     public void vaciarArbol() { 26         raiz = null; 27     } 28 29     public void insertarArbol(T info) { 30         Nodito&lt;T&gt; p = new Nodito(info); 31         if (arbolVacio()) { 32             raiz = p; 33         } else { 34             Nodito&lt;T&gt; padre = buscaPadre(raiz, p); 35             if (p.getInfo().compareTo(padre.getInfo()) &gt;= 0) { 36                 padre.setDer(p); 37             } else { 38                 padre.setIzq(p); 39             } 40         } 41     } 42 }</pre> |  |

```

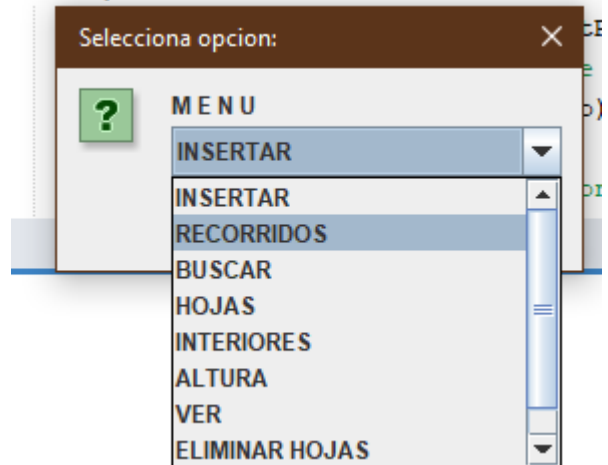
44 //BuscaPadre
45 public Nodito<T> buscaPadre(Nodito<T> actual, Nodito<T> p) {
46     Nodito<T> padre = null;
47
48     while (actual != null) {
49         padre = actual;
50         if (p.getInfo().compareTo(padre.getInfo()) >= 0) {
51             actual = padre.getDer();
52         } else {
53             actual = padre.getIzq();
54         }
55     }
56
57     return padre;
58 }
59
61 public String preorden(Nodito<T> r) {
62     if (r != null) {
63         return r.getInfo() + "-" + preorden(r.getIzq()) + "-" + preorden(r.getDer());
64     } else {
65         return "";
66     }
67 }
68
69 //Inorden
70 public String Inorden(Nodito<T> r) {
71     if (r != null) {
72         return Inorden(r.getIzq()) + "-" + r.getInfo() + "-" + Inorden(r.getDer());
73     } else {
74         return "";
75     }
76 }
77 //Inorden2
78 public String Inorden2(Nodito<T> r) {
79     if (r != null) {
80         return Inorden2(r.getDer()) + " - " + r.getInfo() + Inorden2(r.getIzq());
81     } else {
82         return "";
83     }
84 }
85
86 public String postOrden(Nodito<T> r) {
87     if (r != null) {
88         return postOrden(r.getIzq()) + " - " + postOrden(r.getDer()) + r.getInfo();
89     } else {
90         return "";
91     }
92 }
93 //BUSCAR
94 public static boolean buscar(Nodito<Integer> nodo, int dato) {
95     if (nodo == null) {
96         return false;
97     }
98     if (dato == nodo.getInfo()) {
99         return true;
100     } else if (dato > nodo.getInfo()) {
101         return buscar(nodo.getDer(), dato);
102     } else {
103         return buscar(nodo.getIzq(), dato);
104     }
105 }
106

```

```

else{

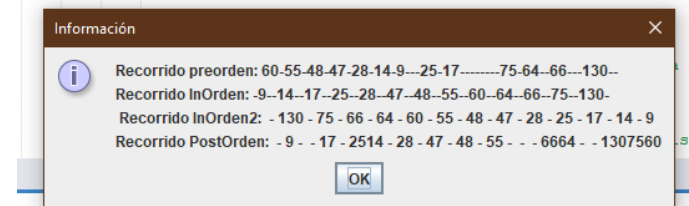
```



```

if (arb.arbolVacio()) {
    Tools.imprime("El arbol esta vacio");
}

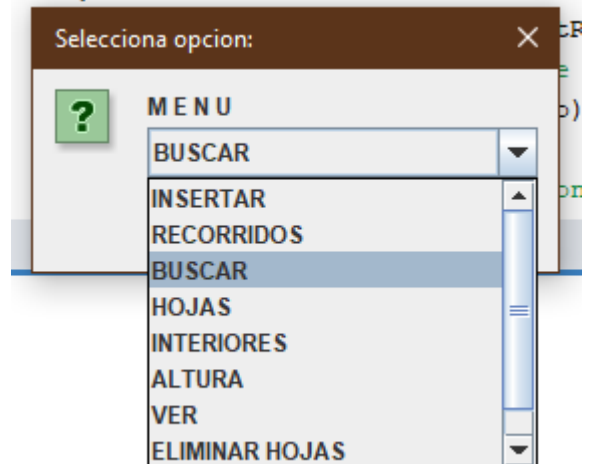
```



```

else{

```

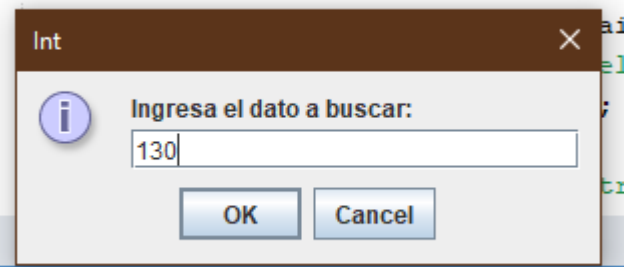


```

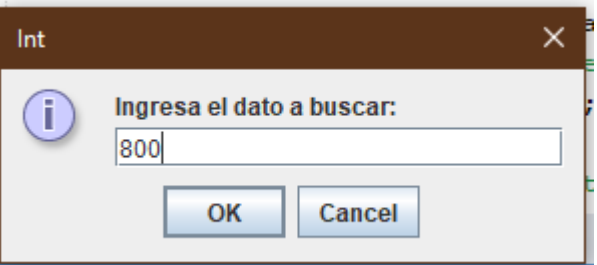
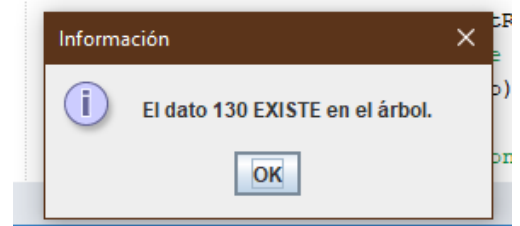
107 public String hojas(Nodito<T> nodo) {
108     String cadena = "";
109     if (nodo != null) {
110         if (nodo.getIzq() == null && nodo.getDer() == null) {
111             cadena += nodo.getInfo().toString() + ", ";
112         } else {
113             cadena += hojas(nodo.getIzq());
114             cadena += hojas(nodo.getDer());
115         }
116     }
117     return cadena;
118 }
119
120 //Calcula la altura
121 public int altura(Nodito<T> nodo) {
122     if (nodo == null) {
123         return 0;
124     } else {
125         int alturaIzq = altura(nodo.getIzq());
126         int alturaDer = altura(nodo.getDer());
127         return Math.max(alturaIzq, alturaDer) + 1;
128     }
129 }
130
131 public String buscoInteriores(Nodito<T> nodo) {
132     String cadena = "";
133     if (nodo != null) {
134         if (nodo != raiz && (nodo.getIzq() != null || nodo.getDer() != null)) {
135             cadena += nodo.getInfo().toString() + ", ";
136         }
137         cadena += buscoInteriores(nodo.getIzq());
138         cadena += buscoInteriores(nodo.getDer());
139     }
140     return cadena;
141 }
142
143 public void mostrarArbol() {
144     String arbolVisual = generarArbolVisual(raiz, "", "");
145     JOptionPane.showMessageDialog(null, arbolVisual, "Árbol Binario", JOptionPane.INFORMATION_MESSAGE);
146 }
147
148 private String generarArbolVisual(Nodito<T> nodo, String prefijo, String rama) {
149     StringBuilder sb = new StringBuilder();
150
151     if (nodo != null) {
152         sb.append(prefijo);
153         sb.append(rama);
154         sb.append(nodo.getInfo());
155         sb.append("\n");
156
157         String nuevoPrefijo = prefijo + "|   ";
158         String ramaIzq = "├─ ";
159         String ramaDer = "└─ ";
160
161         sb.append(generarArbolVisual(nodo.getDer(), nuevoPrefijo, ramaDer));
162         sb.append(generarArbolVisual(nodo.getIzq(), nuevoPrefijo, ramaIzq));
163     }
164     return sb.toString();
165 }
166
167 public void eliminarHojas(int dato) {
168     raiz = eliminarHojasRecursivo(raiz, dato);
169 }
170
171 private Nodito<T> eliminarHojasRecursivo(Nodito<T> nodo, int dato) {
172     if (nodo == null) {
173         return null;
174     }
175     if (nodo.getIzq() == null && nodo.getDer() == null && nodo.getInfo().equals(dato)) {
176         return null;
177     }
178     nodo.setIzq(eliminarHojasRecursivo(nodo.getIzq(), dato));
179     nodo.setDer(eliminarHojasRecursivo(nodo.getDer(), dato));
180     return nodo;
181 }
182
183 public boolean buscarNodo(T dato) {
184     return buscarNodoRecursivo(raiz, dato);
185 }
186
187

```

```
else{
```



```
else{
```

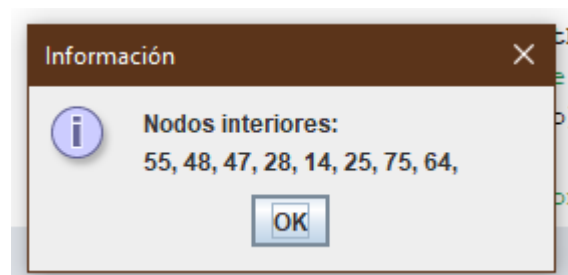
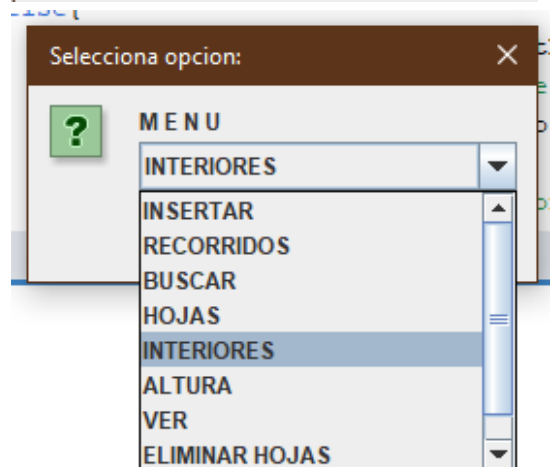
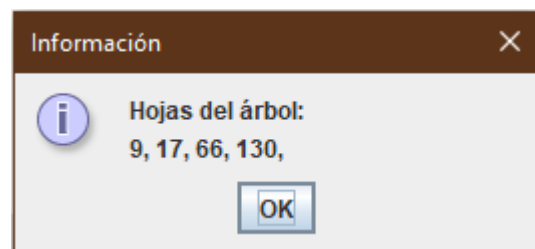
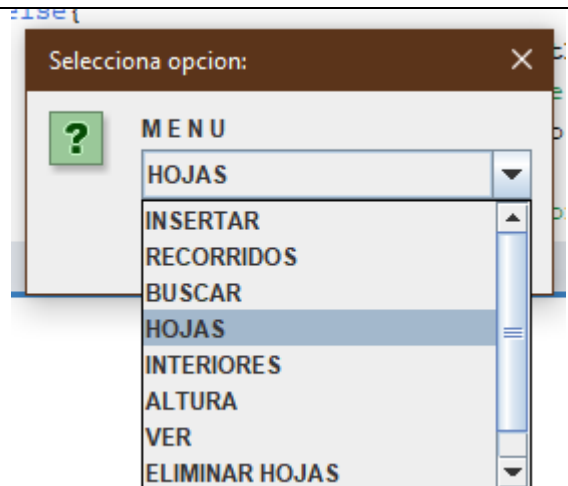


```

public boolean buscarNodo(T dato) {
    return buscarNodoRecursivo(raiz, dato);
}

private boolean buscarNodoRecursivo(Nodito<T> nodo, T dato) {
    if (nodo == null) {
        return false;
    }
    if (dato.compareTo(nodo.getInfo()) == 0) {
        return true;
    } else if (dato.compareTo(nodo.getInfo()) > 0) {
        return buscarNodoRecursivo(nodo.getDer(), dato);
    } else {
        return buscarNodoRecursivo(nodo.getIzq(), dato);
    }
}
}

```



```

1  package ArbolBinario;
2  public class Nodito <T>{
3
4      private T info;
5      private Nodito isq;
6      private Nodito der;
7
8      public Nodito(T dato) {
9          this.info = dato;
10         this.isq = null;
11         this.der = null;
12     }
13
14     public T getInfo() {
15         return info;
16     }
17
18     public void setInfo(T info) {
19         this.info = info;
20     }
21
22     public Nodito getIsq() {
23         return isq;
24     }
25
26     public void setIsq(Nodito isq) {
27         this.isq = isq;
28     }
29
30     public Nodito getDer() {
31         return der;
32     }
33
34     public void setDer(Nodito der) {
35         this.der = der;
36     }
37 }

```

Selecciona opcion:



M E N U

ALTURA  
INSERTAR  
RECORRIDOS  
BUSCAR  
HOJAS  
INTERIORES  
ALTURA  
VER  
ELIMINAR HOJAS

Información



Altura de árbol: 8

OK

Selecciona opcion:



M E N U

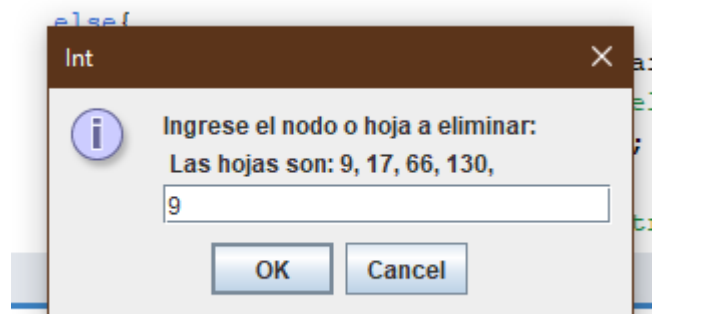
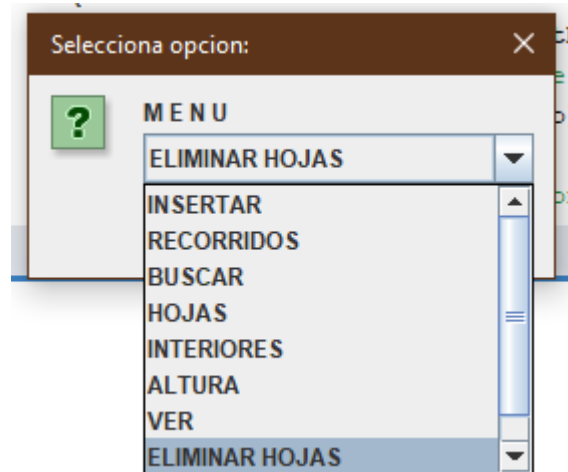
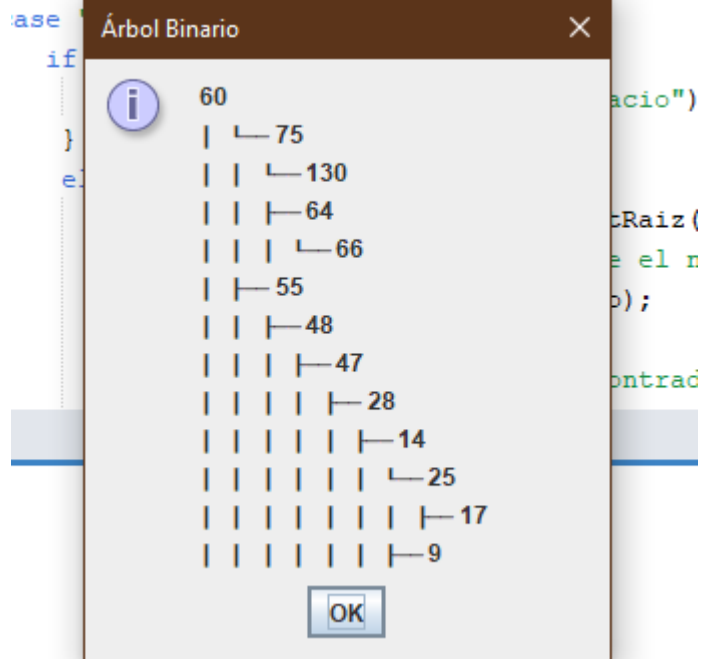
VER  
INSERTAR  
RECORRIDOS  
BUSCAR  
HOJAS  
INTERIORES  
ALTURA  
VER  
ELIMINAR HOJAS

Menú:

```

1 package ArbolesPrincipal;
2 import EntradaSalida.Tools;
3 import javax.swing.JOptionPane;
4 import ArbolBinario.ArbolBin;
5 public class TestArbol {
6     //MENU DE CLASE ARBOL BIN
7     public static void testArbol(String menu){
8         ArbolBin<Integer> arb = new ArbolBin();
9         //int op;
10        String op;
11        do {
12            op = desplegable(menu);
13            switch (op) {
14                case "INSERTAR":
15                    arb.insertarArbol(Tools.leeInt("Inserta dato: "));
16                    break;
17                case "RECORRIDOS":
18                    if (arb.arbolVacio()) {
19                        Tools.imprime("Árbol vacio");
20                    } else {
21                        Tools.imprime("Recorrido preorden: " + arb.preorden(arb.getRaiz())
22                            + "\nRecorrido InOrden: " + arb.inorden(arb.getRaiz())
23                            + "\n Recorrido InOrden2: " + arb.inorden2(arb.getRaiz())
24                            + "\nRecorrido PostOrden: " + arb.postOrden(arb.getRaiz()) );
25                    }
26                    break;
27                case "BUSCAR":
28                    if (arb.arbolVacio()) {
29                        Tools.imprime("Árbol vacio");
30                    } else {
31                        int dato = Tools.leeInt("Ingresa el dato a buscar: ");
32                        boolean encontrado = arb.buscar(arb.getRaiz(), dato);
33                        //nos devuelve un true o false, es decir una "confirmación" de nue
34                        if (encontrado) {
35                            Tools.imprime("El dato " + dato + " EXISTE en el árbol.");
36                        } else {
37                            Tools.imprime("El dato " + dato + " NO existe en el árbol.");
38                        }
39                    }
40                    break;
41                case "HOJAS":
42                    if (arb.arbolVacio()) {
43                        Tools.imprime("Árbol vacio");
44                    } else {
45                        String hojas = arb.hojas(arb.getRaiz());
46                        Tools.imprime("Hojas del árbol:\n" + hojas);
47                    }
48                    break;
49                case "INTERIORES":
50                    if (arb.arbolVacio()) {
51                        Tools.imprime("Árbol vacio");
52                    } else {
53                        String interiores = arb.buscInteriores(arb.getRaiz());
54                        Tools.imprime("Nodos interiores:\n" + interiores);
55                    }
56                    break;
57                case "ALTURA":
58                    if (arb.arbolVacio()) {
59                        Tools.imprime("Árbol vacio");
60                    } else {
61                        int alturaArbol = arb.altura(arb.getRaiz());
62                        Tools.imprime("Altura de árbol: " + alturaArbol);
63                    }
64                    break;
65                case "VER":
66                    if (arb.arbolVacio()) {
67                        Tools.imprime("Árbol vacio");
68                    } else {
69                        arb.mostrarArbol();
70                    }
71                    break;
72                case "ELIMINAR HOJAS":
73                    if (arb.arbolVacio()) {
74                        Tools.imprime("El árbol esta vacio");
75                    }
76                    else{
77                        String hojas = arb.hojas(arb.getRaiz());
78                        int dato = Tools.leeInt("Ingresa el nodo o hoja a eliminar: \n Las hojas son: " + hojas);
79                        boolean exis=arb.buscarNodo(dato);
80                        if (exis == false) {
81                            Tools.imprime("Dato no encontrado en la lista");
82                        }
83                    }

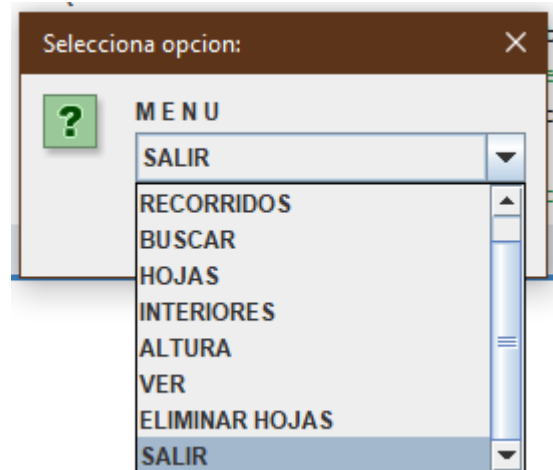
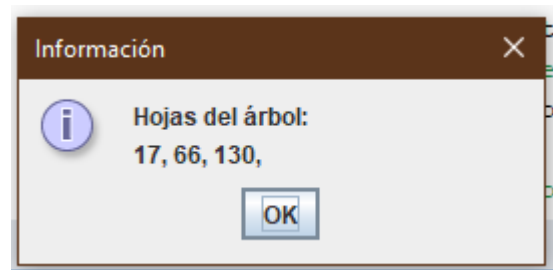
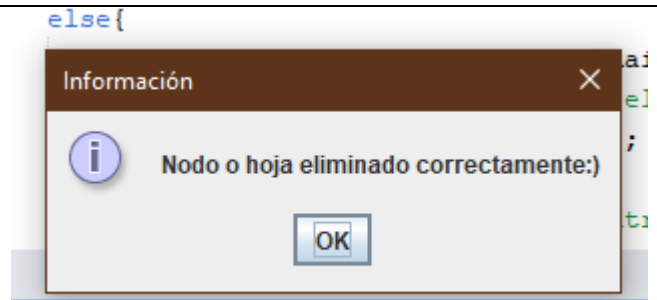
```



```

84         else{
85             arb.eliminarHojas(dato);
86             Tools.imprime("Nodo o hoja eliminado correctamente:");
87         }
88     }
89     }
90     break;
91 }
92 }
93 }while (! op.equals("SALIR"));
94 }
95 }
96
97 public static String desplegable(String menu) {
98     String valores[] = menu.split(",");
99     String res = (String) JOptionPane.showInputDialog(null,"M E N U", "Selecciona opcion:",JOptionPane.QUESTION_MESSAGE, null, null, null);
100     return (res);
101 }
102
103 public static void main(String[] args) {
104     String menu = "INSERTAR, RECORRIDOS, BUSCAR, HOJAS, INTERIORES, ALTURA, VER, ELIMINAR HOJAS, SALIR";
105     testArbol(menu);
106 }
107
108 }

```



Output - Arboles (run)

```

run:
BUILD SUCCESSFUL (total time: 20 minutes 15 seconds)

```

## Funcionamiento

Para este proyecto se crearon 4 clases para el funcionamiento de este tema, ArbolBin, Nodito, TestArbol y Tools, esta última es solamente para las entradas y salidas de datos, a continuación se dará una explicación de cada clase y los componentes que se fueron creando para que el programa funcionara correctamente.

**La primera clase fue Nodito, que va a representar un nodo de un árbol binario, aquí una explicación de la clase:**

Declaración de variables:

- La clase tiene tres variables miembros privadas: "info" de tipo genérico T, que representa la información almacenada en el nodo, y "izq" y "der" de tipo "Nodito", que representan los nodos hijo izquierdo y derecho, respectivamente.

1. Constructor:

- El constructor de la clase recibe un dato como parámetro y se utiliza para inicializar las variables del nodo. El dato se asigna a la variable "info" y los nodos hijo izquierdo y derecho se establecen inicialmente como nulos.

2. Métodos getter y setter:

- Los métodos "getInfo", "getIzq" y "getDer" se utilizan para obtener los valores de las variables "info", "izq" y "der", respectivamente.
- Los métodos "setInfo", "setIzq" y "setDer" se utilizan para establecer los valores de las variables "info", "izq" y "der", respectivamente.

En resumen, la clase "Nodito" representa un nodo de un árbol binario y proporciona métodos para acceder y modificar la información del nodo y los nodos hijo izquierdo y derecho.

**A continuación, se explica las diferentes partes y métodos de la clase ArbolBin:**

1. Declaración de variables:

- La clase tiene una variable miembro privada llamada "raiz" de tipo "Nodito<T>", que representa la raíz del árbol.

2. Constructor:

- El constructor de la clase inicializa la raíz del árbol como nula.

3. Métodos getter y setter:

- Los métodos "getRaiz" y "setRaiz" se utilizan para obtener y establecer el valor de la raíz del árbol, respectivamente.



#### 4. Métodos básicos:

- El método "arbolVacio" verifica si el árbol está vacío, es decir, si la raíz es nula.
- El método "vaciarArbol" establece la raíz del árbol como nula, eliminando todos los nodos del árbol.

#### 5. Método de inserción:

- El método "insertarArbol" se utiliza para insertar un nuevo nodo en el árbol. El nodo se crea a partir de un dato proporcionado como parámetro y se coloca en la posición correcta del árbol según su valor. Si el árbol está vacío, el nuevo nodo se convierte en la raíz.

#### 6. Método de búsqueda de padre:

- El método "buscaPadre" se utiliza para buscar el nodo padre del nodo proporcionado como parámetro. Comienza desde la raíz y desciende por el árbol comparando los valores de los nodos para determinar si debe ir hacia la izquierda o hacia la derecha. El método devuelve el nodo padre encontrado.

#### 7. Métodos de recorrido:

- Los métodos "preorden", "Inorden", "Inorden2" y "postOrden" implementan los recorridos preorden, inorden, inorden inverso y postorden, respectivamente. Estos métodos realizan un recorrido recursivo del árbol, visitando los nodos en el orden correspondiente y devolviendo una cadena con la información de los nodos visitados como se vio en clase.

#### 8. Método de búsqueda:

- El método estático "buscar" se utiliza para buscar un dato específico en el árbol. Comienza desde un nodo dado y compara el dato con el valor del nodo actual. Si el dato es igual, se devuelve verdadero. Si el dato es mayor, la búsqueda continúa por la rama derecha del árbol. Si el dato es menor, la búsqueda continúa por la rama izquierda del árbol. Si el nodo actual es nulo, se devuelve falso.

#### 9. Otros métodos:

- El método "hojas" devuelve una cadena que contiene la información de todas las hojas (nodos sin hijos) del árbol.
- El método "altura" calcula la altura del árbol. Utiliza un enfoque recursivo para obtener la altura máxima entre las alturas de los subárboles izquierdo y derecho, y luego agrega 1 a ese valor.
- El método "buscInteriores" devuelve una cadena con la información de los nodos interiores del árbol (nodos que no son la raíz ni hojas).

- El método "mostrarArbol" genera una representación visual del árbol en forma de cadena y lo muestra en una ventana emergente mediante JOptionPane.

#### 10. Métodos de eliminación y búsqueda de nodo:

- El método "eliminarHojas" elimina todas las hojas del árbol que contengan un valor igual al dato proporcionado.
- El método "buscarNodo" busca un nodo con un valor específico en el árbol utilizando un enfoque recursivo.

### **Ahora se explica cada parte de la clase TestArbol:**

#### Importaciones de paquetes:

Se importan los paquetes necesarios para utilizar las clases y métodos requeridos en el programa, como "EntradaSalida.Tools" para la entrada y salida de datos, "javax.swing.JOptionPane" para mostrar mensajes y obtener entradas del usuario, y "ArbolBinario.ArbolBin" para utilizar la clase "ArbolBin" definida en otra clase.

#### Método "testArbol":

Este método es responsable de ejecutar las operaciones del árbol binario según la opción seleccionada por el usuario. Recibe un parámetro de tipo String llamado "menu", que representa las opciones del menú separadas por comas.

Dentro de un bucle "do-while", se muestra un menú desplegable utilizando la clase JOptionPane, donde el usuario puede seleccionar una opción.

Dependiendo de la opción seleccionada, se ejecuta el código correspondiente utilizando la instancia del objeto "arb" de la clase "ArbolBin".

#### Método "desplegable":

Este método se utiliza para mostrar un menú desplegable utilizando la clase JOptionPane.

Recibe un parámetro de tipo String llamado "menu", que contiene las opciones del menú separadas por comas. El método muestra el menú desplegable utilizando JOptionPane.showInputDialog y devuelve la opción seleccionada por el usuario.

### Método "main":

Este es el punto de entrada del programa.

Define una variable de tipo String llamada "menu" que contiene las opciones del menú separadas por comas.

Llama al método "testArbol" pasando el menú como argumento para iniciar el programa y permitir al usuario interactuar con el árbol binario.

En resumen, la clase "TestArbol" es la clase principal que contiene el punto de entrada del programa y se encarga de mostrar un menú al usuario para realizar operaciones en un árbol binario utilizando la clase "ArbolBin". Cada opción del menú ejecuta el código correspondiente utilizando métodos de la clase "ArbolBin".

## **6. Conclusiones**

En conclusión, los árboles son una estructura de datos muy importante en programación. Te permiten organizar y buscar datos de manera eficiente. Son como una especie de árbol invertido, donde cada nodo puede tener hijos. Puedes usar diferentes tipos de árboles, como los de búsqueda binaria, para buscar datos rápidamente.

Las características principales de los árboles incluyen la presencia de un nodo raíz, nodos internos y hojas, así como las relaciones jerárquicas entre ellos. Estas relaciones jerárquicas permiten establecer un orden entre los elementos almacenados en el árbol, lo que facilita la búsqueda y recuperación de datos.

En general, podemos decir que los árboles son estructuras de datos poderosas y flexibles que juegan un papel fundamental en la representación y manipulación eficiente de datos en muchas aplicaciones. Su capacidad para organizar datos de manera jerárquica y proporcionar un acceso rápido a la información los convierte en una herramienta invaluable en el campo de la informática.

## 7. Bibliografía

### Bibliografía

Alduncin, K. (11 de 07 de 2011). *karenalduncin*. Obtenido de Arboles Binarios/Grafos/Estructura de Datos: <https://karenalduncin.wordpress.com/2011/07/11/arboles-binariosgrafosestructura-de-datos/>

Cormen, T. H. (2009). *Introducción a los algoritmos*. MIT press.

*dlsiAsignaturas*. (26 de 01 de 2023). Obtenido de <https://www.dlsi.ua.es/asignaturas/p2ir/teoria/l03/lessonh.html>

oblancarte. (22 de 08 de 2014). *ESTRUDIANTES*. Obtenido de Estructura de datos – Árboles: <https://www.oscarblancarteblog.com/2014/08/22/estructura-de-datos-arboles/>