Instituto tecnológico de Orizaba

Sistemas Computacionales

Estructura de Datos

Unidad 6

Profesora: María Jacinta Martínez Castillo

Integrantes del equipo:

Moran De la Cruz Aziel 21011006

Jiménez Jiménez Carlos Yael 21010975

Sebastián Brito García 21010929

Bandala Hernández Sebastián 21010921

Introducción

En el campo de las bases de datos, una estructura fundamental para organizar y gestionar datos de manera eficiente es el árbol. Los árboles son una forma de representar jerarquías y relaciones entre datos, ofreciendo diversas ventajas en términos de búsqueda, inserción y eliminación de información. En el contexto de la Estructura de Base de Datos, el estudio de los árboles es crucial para comprender cómo se organizan los datos y cómo se optimizan las consultas.

Esta introducción tiene como objetivo brindar una visión general del tema de árboles en el contexto de la Estructura de Base de Datos. Exploraremos los conceptos fundamentales de los árboles, su importancia en las bases de datos y los diferentes tipos de árboles utilizados en este campo. Además, nos adentraremos en las operaciones y funciones relacionadas con los árboles, como la inserción, eliminación, búsqueda y recorrido de nodos. También analizaremos las aplicaciones prácticas de los árboles en las bases de datos, como los índices y la organización jerárquica de datos.

Además, abordaremos aspectos de rendimiento y optimización en el uso de árboles, incluyendo técnicas para reducir la altura del árbol, mejorar la eficiencia en las operaciones y garantizar el equilibrio y balanceo adecuado de la estructura.

En resumen, el estudio de los árboles en la Estructura de Base de Datos es esencial para comprender cómo se organizan y manipulan los datos en entornos de bases de datos. Los árboles ofrecen un enfoque eficiente y flexible para la gestión de información, permitiendo realizar consultas de manera rápida y optimizada. A lo largo de este estudio, exploraremos los conceptos clave, las operaciones y las aplicaciones de los árboles, brindando una base sólida para el diseño y desarrollo de bases de datos eficientes y escalables.

Competencia específica:

1. Diseñar y representar correctamente diferentes tipos de árboles utilizados en bases de datos.
2. Identificar y aplicar las operaciones fundamentales en los árboles, como la inserción, eliminación, búsqueda y recorrido de nodos, con el fin de realizar manipulaciones eficientes de los datos almacenados.
3. Evaluar y seleccionar el tipo de árbol más adecuado para un escenario de base de datos específico, considerando los requisitos de rendimiento, escalabilidad y eficiencia en las operaciones de consulta.
4. Analizar y explicar las técnicas de optimización de árboles, como el equilibrio y balanceo de la estructura, la reducción de la altura del árbol y el uso de índices, con el objetivo de mejorar el rendimiento y la eficiencia en la manipulación de datos.
5. Aplicar los conocimientos adquiridos sobre árboles en la resolución de problemas y casos prácticos relacionados con la gestión y organización de datos en bases de datos.

Marco Teórico:

Los árboles son estructuras de datos fundamentales en el campo de la informática y juegan un papel crucial en la organización y manipulación de datos en sistemas de bases de datos. En la materia de Estructura de Base de Datos, se estudian diversos tipos de árboles y sus aplicaciones en la gestión eficiente de información.

Un árbol es una estructura jerárquica compuesta por nodos interconectados mediante enlaces o aristas. Cada árbol tiene un nodo raíz, a partir del cual se ramifican los demás nodos. Cada nodo puede tener cero o más hijos, pero solo puede tener un padre, excepto el nodo raíz que no tiene padre. Los nodos sin hijos se denominan hojas.

Uno de los tipos de árboles ampliamente utilizados en bases de datos es el árbol binario de búsqueda. En este tipo de árbol, cada nodo tiene como máximo dos hijos: un hijo izquierdo y un hijo derecho. Los valores almacenados en los nodos siguen una propiedad de orden, donde los valores en el subárbol izquierdo son menores que el valor del nodo padre, y los valores en el subárbol derecho son mayores.

En el contexto de bases de datos, los árboles se utilizan para implementar índices que aceleran las operaciones de búsqueda y acceso a los datos almacenados en las tablas. Los índices basados en árboles permiten una búsqueda eficiente utilizando claves o valores asociados a los registros, reduciendo la complejidad de búsqueda y mejorando el rendimiento del sistema.

La optimización de los árboles en las bases de datos implica técnicas como el equilibrio y el balanceo de la estructura para minimizar la altura del árbol y garantizar un acceso eficiente a los datos. Además, se utilizan algoritmos y estrategias de optimización para mejorar la velocidad de las operaciones de inserción, eliminación y búsqueda en los árboles.

En resumen, el estudio de los árboles en la materia de Estructura de Base de Datos proporciona a los estudiantes los conocimientos necesarios para comprender y utilizar estas estructuras en la organización y manipulación de datos en sistemas de bases de datos. Los árboles permiten una gestión eficiente de la información, mejorando el rendimiento y la eficiencia en las operaciones de consulta y acceso a los datos almacenados.

Materiales utilizados

Los materiales utilizados para esta práctica son:

Computadora

NetBeans o Eclipse

Desarrollo

Clase Tools:

```
public static String MenuArboles(String Menu_Ejecutable) {
                String opcion[]=
{"INSERTAR","RECORRIDO","BUSCAR","HOJAS","INTERIORES","GRADO","ALTU
RA","VER","SALIR"};
                  int n;
                  n = JOptionPane.showOptionDialog(null,"SELECCIONA DANDO
CLICK","M E N U", JOptionPane.NO_OPTION,
JOptionPane.QUESTION_MESSAGE, null, opcion, opcion[0]);
                  return (opcion[n]);
          }
```

Este método proporciona una forma interactiva de seleccionar una operación específica relacionada con los árboles, mostrando un menú y devolviendo la opción seleccionada por el usuario.

Clase ArbolBin:

```
public class ArbolBin <T> {
        private Nodito<T> raiz;

   public ArbolBin() {
      raiz = null;
   }

   public Nodito<T> getRaiz() {
      return raiz;
   }

   public void setRaiz(Nodito<T> raiz) {
      this.raiz = raiz;
   }

   public boolean arbolVacio() {
      return raiz == null;
   }
```

Estos métodos pertenecen a la clase genérica ArbolBin, que representa un árbol binario.

1. El constructor ArbolBin(): Inicializa un árbol binario estableciendo la raíz como nula.
2. El método getRaiz(): Devuelve el nodo raíz del árbol.
3. El método setRaiz(Nodito<T> raiz): Establece el nodo raíz del árbol con el valor especificado.
4. El método arbolVacio(): Verifica si el árbol está vacío, es decir, si la raíz es nula. Retorna true si el árbol está vacío, o false en caso contrario.

```
public void vaciarArbol() {
      raiz = null;
   }
```

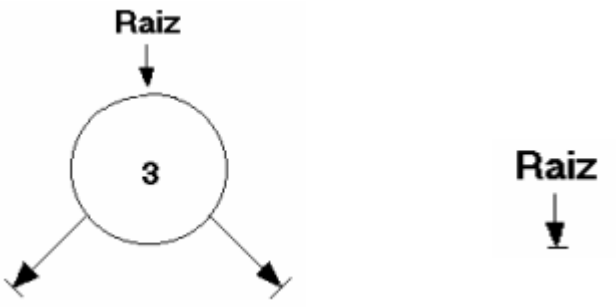
Procedimiento Borrar Este es posiblemente el procedimiento más complicado de la asignatura, hay una gran cantidad de casos particulares y el caso general es bastante

complicado, sería bueno que los casos particulares fueran tratados en procedimientos aparte a la hora de hacer la práctica. Se distinguirán los casos particulares y el caso general. Para implementar este procedimiento hay que tener en cuenta, a grandes rasgos, cinco casos:

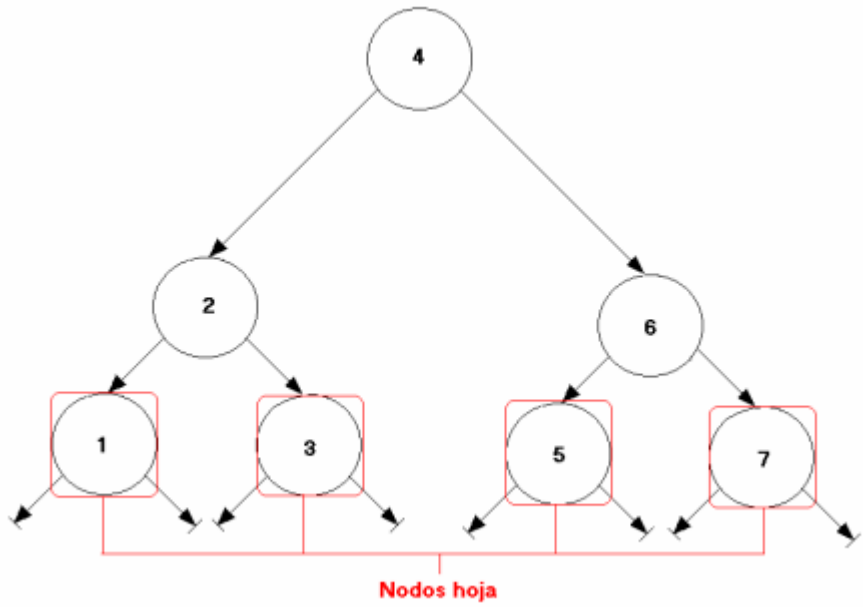
- Árbol vacío.
- Árbol con un solo elemento.
- El nodo a borrar es un nodo hoja.
- El nodo a borrar tiene un único hijo, en suyo caso éste ocupará el lugar del nodo que vamos a borrar.
- El nodo a borrar tiene dos hijos. En este caso hay que buscar un nodo que pueda ocupar el lugar del que vamos a borrar de forma que se respete la estructura del árbol: para ello, el nodo sustituto debe ser el mayor de los menores (el situado más a la derecha del hijo izquierdo de la raíz del árbol), o bien, el menor de los mayores (el situado más a la izquierda del hijo derecho de la raíz del árbol)

Caso 1: Borrar la raíz

Es el más sencillo, tan solo hay que hacer free del nodo y poner Raiz a null:

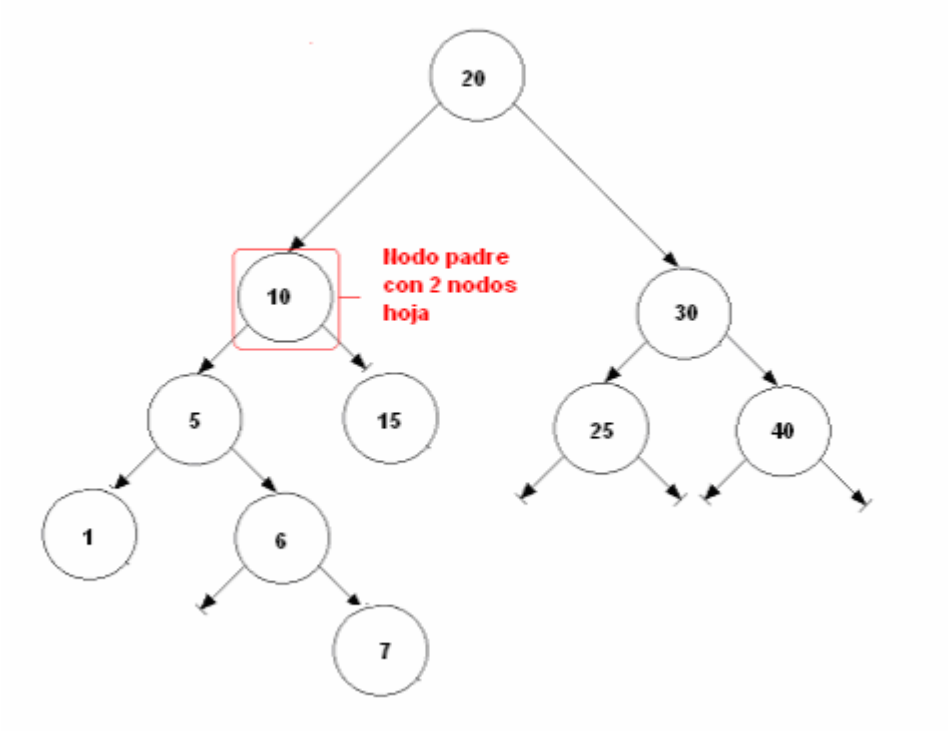


Caso 2: Borrar un nodo hoja



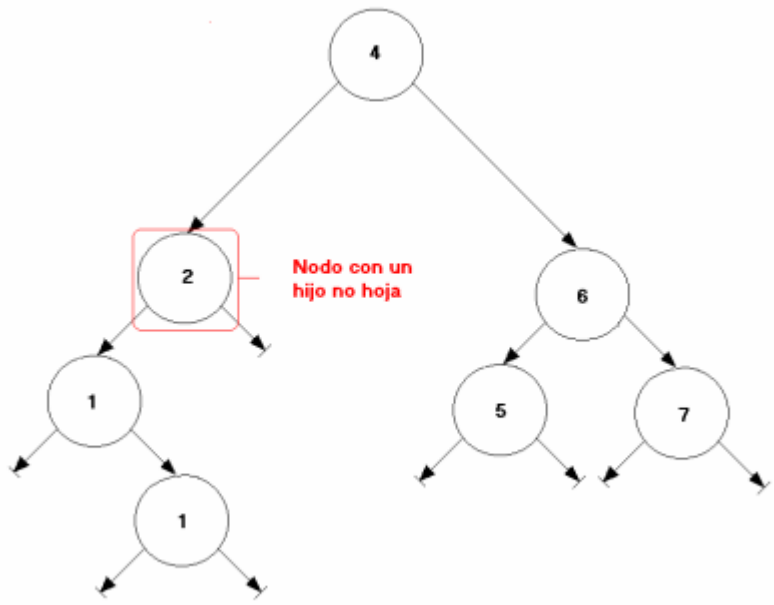
Los nodos hoja se borran también fácilmente, basta con hacer free y poner el anterior a null.

Caso 3: Borrar un nodo padre de dos nodos hoja



En este caso debemos encontrar un sustituto ideal para reemplazar al padre con dos hijos. Hay dos posibilidades: 1) El sustituto ideal puede ser el mayor del sub-arbol izquierdo del padre. (en nuestro ejemplo el 7) 2) El sustituto ideal puede ser el menor del sub-arbol derecho del padre. (en nuestro ejemplo el 15) Para el árbol de la figura anterior sería más conveniente la primera opción porque el árbol resultante estaría más “equilibrado” que si resolvemos utilizando la opción 2. Para elegir una opción u otra podríamos hacer una función que nos calcule cual de los sub-arboles del padre tiene mayor número de elementos.

Caso 4: Borrar un nodo padre con un solo hijo que no es hoja



Para borrar este nodo su sustituto ideal es el hijo izquierdo. En caso de tener sólo el hijo derecho, el sustituto ideal es el hijo derecho.

```
public void insertarArbol(T info) {
        Nodito<T> p = new Nodito<T>(info);
        if (arbolVacio()) {
            raiz = p;
        } else {
            Nodito<T> padre = buscaPadre(raiz, p);
            if (p.getInfo().hashCode() >= padre.getInfo().hashCode()) {
                padre.setDer(p);
            } else {
                padre.setIzq(p);
            }
        }
    }
```

El método "insertarArbol" se utiliza para insertar un nuevo nodo en un árbol binario. Toma como parámetro un objeto de tipo "T" que representa la información que se desea insertar.
El método crea un nuevo nodo "p" con la información proporcionada. Luego verifica si el árbol está vacío, es decir, si la raíz es nula. Si el árbol está vacío, el nuevo nodo se establece como la raíz del árbol.
En caso contrario, se busca el padre del nuevo nodo utilizando el método auxiliar "buscaPadre". Este método se encarga de encontrar la posición adecuada para insertar el nuevo nodo en el árbol. Se compara el valor hash de la información del nuevo nodo con el valor hash de la información del nodo padre. Si el valor hash del nuevo nodo es mayor o igual al valor hash del padre, se inserta el nuevo nodo como hijo derecho del padre. De lo contrario, se inserta como hijo izquierdo.

```
public Nodito<T> buscaPadre(Nodito<T> actual, Nodito<T> p) {
        Nodito<T> padre = null;
        while (actual != null) {
            padre = actual;
            if (p.getInfo().hashCode() >= padre.getInfo().hashCode()) {
                actual = padre.getDer();
            } else {
                actual = padre.getIzq();
            }
        }
        return padre;
    }
```

El método "buscaPadre" se utiliza para encontrar el nodo padre de un nuevo nodo que se va a insertar en un árbol binario. Toma como parámetros dos nodos: "actual" que representa el nodo actual en la búsqueda y "p" que es el nuevo nodo que se desea insertar.
El método utiliza un bucle while para recorrer el árbol y encontrar el lugar adecuado para insertar el nuevo nodo. Inicialmente, se inicializa el nodo "padre" como nulo.
Dentro del bucle, se actualiza el nodo "padre" con el valor del nodo "actual" en cada iteración. Luego, se compara el valor hash de la información del nuevo nodo "p" con el valor hash de la información del nodo padre. Si el valor hash del nuevo nodo es

mayor o igual al valor hash del padre, se mueve hacia el hijo derecho del padre asignando el nodo derecho del padre al nodo "actual". De lo contrario, se mueve hacia el hijo izquierdo asignando el nodo izquierdo del padre al nodo "actual".
El bucle continúa hasta que se alcanza un nodo nulo, es decir, se ha llegado a una posición vacía donde se puede insertar el nuevo nodo. En ese momento, se devuelve el nodo "padre" que representa el nodo padre del nuevo nodo que se va a insertar.

```
public String preorden(Nodito<T> r) {
      if (r != null) {
         return r.getInfo() + "-" + preorden(r.getIzq()) + "-" + preorden(r.getDer());
      } else {
         return " ";
      }
   }
```

El método "preorden" se utiliza para realizar el recorrido en preorden de un árbol binario. Toma como parámetro el nodo raíz del árbol (representado por "r") y devuelve una cadena de caracteres que representa el recorrido en preorden del árbol.
El recorrido en preorden consiste en visitar primero el nodo actual, luego recorrer el subárbol izquierdo y finalmente recorrer el subárbol derecho.
En el método, se realiza lo siguiente:

- Se verifica si el nodo "r" no es nulo. Si es nulo, significa que se ha alcanzado una hoja del árbol y se devuelve una cadena vacía.
- Si el nodo "r" no es nulo, se concatena la información del nodo actual ("r.getInfo()") con los resultados de los recorridos en preorden del subárbol izquierdo ("preorden(r.getIzq())") y del subárbol derecho ("preorden(r.getDer())").
- La concatenación se realiza utilizando el carácter "-" como separador entre la información del nodo actual y los resultados de los subárboles.
- El resultado final es una cadena que representa el recorrido en preorden del árbol binario.

```
public String inOrden(Nodito<T> r) {
      if (r != null) {
         return inOrden(r.getIzq()) + "-" + r.getInfo() + "-" + inOrden(r.getDer());
      } else {
         return " ";
      }
   }
```

El método "inOrden" se utiliza para realizar el recorrido en inorden de un árbol binario. Toma como parámetro el nodo raíz del árbol (representado por "r") y devuelve una cadena de caracteres que representa el recorrido en inorden del árbol.
El recorrido en inorden consiste en recorrer el subárbol izquierdo, visitar el nodo actual y luego recorrer el subárbol derecho.
En el método, se realiza lo siguiente:

- Se verifica si el nodo "r" no es nulo. Si es nulo, significa que se ha alcanzado una hoja del árbol y se devuelve una cadena vacía.
- Si el nodo "r" no es nulo, se concatena el resultado del recorrido en inorden

del subárbol izquierdo ("inOrden(r.getIzq())") con la información del nodo actual ("r.getInfo()") y los resultados del recorrido en inorden del subárbol derecho ("inOrden(r.getDer())").

- La concatenación se realiza utilizando el carácter "-" como separador entre los resultados del subárbol izquierdo, la información del nodo actual y los resultados del subárbol derecho.
- El resultado final es una cadena que representa el recorrido en inorden del árbol binario.

```
public String inOrden2(Nodito<T> r) {
      StringBuilder resultado = new StringBuilder();
      inOrdenRec(r, resultado);
      return resultado.toString();
   }
```

El método "inOrden2" realiza un recorrido en inorden de un árbol binario y devuelve una cadena de caracteres que representa dicho recorrido.
En lugar de concatenar las cadenas recursivamente como se hace en el método "inOrden", este método utiliza un objeto StringBuilder para construir la cadena de resultado de manera eficiente.
El método utiliza una función auxiliar llamada "inOrdenRec" que se encarga de realizar el recorrido en inorden de manera recursiva. Toma como parámetros el nodo actual del árbol ("r") y el objeto StringBuilder donde se construirá el resultado ("resultado").
El método principal "inOrden2" inicializa un objeto StringBuilder llamado "resultado" y luego invoca a "inOrdenRec" pasándole el nodo raíz del árbol y el objeto "resultado". Finalmente, se devuelve el resultado obtenido convertido a una cadena de caracteres utilizando el método "toString()" de StringBuilder.
La función "inOrdenRec" realiza lo siguiente:

- Verifica si el nodo actual ("r") no es nulo.
- Si el nodo no es nulo, realiza recursivamente el recorrido en inorden del subárbol izquierdo llamando a "inOrdenRec" pasando el subárbol izquierdo y el objeto "resultado".
- Agrega la información del nodo actual al objeto "resultado" utilizando el método "append" de StringBuilder.
- Realiza recursivamente el recorrido en inorden del subárbol derecho llamando a "inOrdenRec" pasando el subárbol derecho y el objeto "resultado".

```
public String postOrden(Nodito<T> r) {
      if (r != null) {
         return postOrden(r.getIzq()) + "-" + postOrden(r.getDer()) + "-" + r.getInfo();
      } else {
         return " ";
      }
   }
```

El método "postOrden" realiza un recorrido en postorden de un árbol binario y devuelve una cadena de caracteres que representa el recorrido.
El recorrido en postorden sigue la estrategia de visitar primero los subárboles izquierdo y derecho, y luego el nodo actual.
El método toma como parámetro el nodo raíz del árbol ("r") y realiza lo siguiente:

- Verifica si el nodo actual no es nulo.
- Si el nodo no es nulo, realiza recursivamente el recorrido en postorden del subárbol izquierdo llamando a "postOrden" pasando el subárbol izquierdo.
- Realiza recursivamente el recorrido en postorden del subárbol derecho llamando a "postOrden" pasando el subárbol derecho.
- Concatena las cadenas resultantes de los recorridos de los subárboles izquierdo y derecho, junto con la información del nodo actual, utilizando el operador de concatenación "+".
- Retorna la cadena resultante.

```
public Nodito<T> buscarDato(T dato) {
      return buscarDatoRec(dato, raiz);
   }
```

El método "buscarDato" busca un nodo específico en un árbol binario y devuelve el nodo que contiene el dato buscado.
El método toma como parámetro el dato a buscar ("dato") y realiza lo siguiente:

- Llama a otro método llamado "buscarDatoRec" pasando el dato a buscar y la raíz del árbol como argumentos.
- El método "buscarDatoRec" es un método auxiliar recursivo que se encarga de realizar la búsqueda del dato en el árbol binario.
- Retorna el resultado de la búsqueda que devuelve el método "buscarDatoRec".

```
public Nodito<T> buscarDatoRec(T dato, Nodito<T> r) {
      if (r == null || dato.equals(r.getInfo())) {
         return r;
      } else if (dato.hashCode() >= r.getInfo().hashCode()) {
         return buscarDatoRec(dato, r.getDer());
      } else {
         return buscarDatoRec(dato, r.getIzq());
      }
   }
```

El método "buscarDatoRec" es un método auxiliar recursivo utilizado por el método "buscarDato" para buscar un nodo específico en un árbol binario.
El método toma dos parámetros: el dato a buscar ("dato") y el nodo actual en el que se está realizando la búsqueda ("r").
El método realiza lo siguiente:

- Verifica si el nodo actual es nulo o si el dato buscado es igual al dato contenido en el nodo actual. En cualquiera de estos casos, retorna el nodo actual.
- Si el dato buscado tiene un valor hash mayor o igual al valor hash del dato contenido en el nodo actual, realiza una llamada recursiva a "buscarDatoRec" pasando el dato buscado y el subárbol derecho (obtenido mediante "r.getDer()") como argumentos.
- En caso contrario, realiza una llamada recursiva a "buscarDatoRec" pasando el dato buscado y el subárbol izquierdo (obtenido mediante "r.getIzq()") como argumentos.

La recursión continúa hasta encontrar el nodo que contiene el dato buscado o hasta

llegar a un nodo nulo, en cuyo caso se devuelve nulo.

```
public void imprimirNodosTerminales() {
      StringBuilder mensaje = new StringBuilder();
      nodosTerminalesX(raiz, mensaje);
      Tools.imprimirMSJE(mensaje.toString());
   }
```

El método "imprimirNodosTerminales" se encarga de imprimir en consola los nodos terminales de un árbol binario.
El método realiza lo siguiente:

- Crea un objeto StringBuilder llamado "mensaje" para almacenar los nodos terminales.
- Llama al método auxiliar "nodosTerminalesX" pasando como argumentos la raíz del árbol y el objeto "mensaje".
- Después de que el método auxiliar completa su ejecución, imprime el contenido del objeto "mensaje" utilizando el método "imprimirMSJE" de la clase Tools.

El método auxiliar "nodosTerminalesX" es responsable de recorrer el árbol de manera recursiva y agregar los nodos terminales encontrados al objeto "mensaje".

```
 private void nodosTerminalesX(Nodito<T> nodo, StringBuilder mensaje) {
      if (nodo == null) {
         return;
      }
      if (nodo.getIzq() == null && nodo.getDer() == null) {
         mensaje.append("Dato: ").append(nodo.getInfo()).append("\n");
      }
      nodosTerminalesX(nodo.getIzq(), mensaje);
      nodosTerminalesX(nodo.getDer(), mensaje);
   }
```

El método "nodosTerminalesX" es un método auxiliar utilizado por el método "imprimirNodosTerminales" para encontrar y almacenar los nodos terminales de un árbol binario.
El método realiza lo siguiente:

- Verifica si el nodo actual es nulo. Si es así, termina la recursión.
- Comprueba si el nodo actual no tiene hijos izquierdo y derecho, es decir, es un nodo terminal.
    - En caso afirmativo, agrega una línea al objeto "mensaje" que contiene la información del nodo actual.
- Realiza llamadas recursivas a los hijos izquierdo y derecho del nodo actual para continuar la búsqueda de nodos terminales.

```
public void imprimirNodosInteriores() {
      StringBuilder mensaje = new StringBuilder();
      nodosInterioresX(raiz, mensaje);
      Tools.imprimirMSJE(mensaje.toString());
   }
```

El método "imprimirNodosInteriores" se utiliza para imprimir los nodos interiores de un árbol binario. Un nodo interior es aquel que tiene al menos un hijo.
El método realiza lo siguiente:

- Crea un objeto StringBuilder llamado "mensaje" para almacenar los nodos interiores encontrados.
- Llama al método auxiliar "nodosInterioresX" pasando la raíz del árbol y el objeto "mensaje".
- Llama al método "imprimirMSJE" de la clase Tools, pasando como argumento la representación en texto del objeto "mensaje".

El método auxiliar "nodosInterioresX" realiza una búsqueda recursiva en el árbol para encontrar los nodos interiores y los agrega al objeto "mensaje".

```
 private void nodosInterioresX(Nodito<T> nodo, StringBuilder mensaje) {
       if (nodo == null || (nodo.getIzq() == null && nodo.getDer() == null)) {
          return;
       }
       if (nodo != raiz) {
          mensaje.append("Dato: ").append(nodo.getInfo()).append("\n");
       }
       nodosInterioresX(nodo.getIzq(), mensaje);
       nodosInterioresX(nodo.getDer(), mensaje);
    }
```

El método "nodosInterioresX" es un método auxiliar utilizado para encontrar y agregar los nodos interiores de un árbol binario a un objeto StringBuilder llamado "mensaje".
El método realiza lo siguiente:

- Verifica si el nodo actual es nulo o si es un nodo hoja (no tiene hijos izquierdo ni derecho). En caso afirmativo, termina la ejecución del método.
- Si el nodo actual no es la raíz del árbol (es decir, no es igual a "raiz"), agrega el dato del nodo al objeto "mensaje".
- Llama recursivamente al método "nodosInterioresX" pasando el hijo izquierdo del nodo actual y el objeto "mensaje".
- Llama recursivamente al método "nodosInterioresX" pasando el hijo derecho del nodo actual y el objeto "mensaje".

```
public int altura() {
       return obtenerAltura(raiz);
    }
```

El método "altura" devuelve la altura del árbol binario.
El método realiza lo siguiente:

- Llama al método "obtenerAltura" pasando la raíz del árbol.
- El método "obtenerAltura" es un método auxiliar recursivo que calcula la altura del árbol. Recibe un nodo como parámetro y retorna la altura del subárbol con ese nodo como raíz.
- Si el nodo pasado como parámetro es nulo, se considera que el subárbol tiene altura cero.
- Si el nodo no es nulo, se calcula la altura del subárbol izquierdo y la altura del subárbol derecho. La altura del subárbol es el máximo entre estas dos alturas

más uno (por el propio nodo).
- Al finalizar la recursión, se obtiene la altura total del árbol.

```
private int obtenerAltura(Nodito<T> nodo) {
    if (nodo == null) {
        return 0;
    }

    int alturaIzq = obtenerAltura(nodo.getIzq());
    int alturaDer = obtenerAltura(nodo.getDer());

    return Math.max(alturaIzq, alturaDer) + 1;
}
```

El método "obtenerAltura" es un método auxiliar recursivo que calcula la altura de un subárbol con un nodo dado como raíz.
El método realiza lo siguiente:

- Comprueba si el nodo pasado como parámetro es nulo. Si es así, se considera que el subárbol tiene una altura de cero.
- Si el nodo no es nulo, se calcula la altura del subárbol izquierdo llamando recursivamente al método "obtenerAltura" con el nodo izquierdo del nodo actual.
- Se calcula la altura del subárbol derecho llamando recursivamente al método "obtenerAltura" con el nodo derecho del nodo actual.
- La altura del subárbol es el máximo entre la altura del subárbol izquierdo y la altura del subárbol derecho, más uno (por el propio nodo).
- Al finalizar la recursión, se obtiene la altura total del subárbol.

Clase Nodito:

```
public class Nodito <T> {
    private T info;
    Nodito <T> izq;
    Nodito <T> der;

    public Nodito (T dato) {
        this.info=dato;
        this.izq=null;
        this.der=null;
    }

    public T getInfo() {
        return info;
    }

    public void setInfo(T info) {
        this.info = info;
    }

    public Nodito <T> getIzq() {
        return izq;
    }
```

```
	public void setIzq(Nodito <T> izq) {
		this.izq = izq;
	}

	public Nodito <T> getDer() {
		return der;
	}

	public void setDer(Nodito <T> der) {
		this.der = der;
	}
}
```

La clase "Nodito" es una clase genérica que representa un nodo en una estructura de árbol binario.
La clase tiene los siguientes atributos:

- "info": Representa la información almacenada en el nodo. Es de tipo genérico T.
- "izq": Representa el enlace al nodo hijo izquierdo.
- "der": Representa el enlace al nodo hijo derecho.

La clase tiene los siguientes métodos:

- Constructor: Recibe un dato de tipo genérico T y crea un nuevo nodo con dicho dato. Inicializa los enlaces izquierdo y derecho como nulos.
- Métodos getter y setter: Permiten acceder y modificar los valores de los atributos "info", "izq" y "der".

Clase Menu_Ejecutable:

```
public class Menu_Ejecutable <T>{
	// metodo que de la figura del arbol, nodos interiores, hijos, buscar
		public static void main(String[] args) {
			ArbolBin  <Integer> arb = new ArbolBin <Integer>();
			String op;
			do {

op=Tools.MenuArboles("Insertar,Recorrido,Buscar,Hojas,Grado,Altura,Salir");
				switch(op) {

				case "INSERTAR":
				arb.insertarArbol(Tools.leerInt("Dato:"));break;

				case "RECORRIDO": Tools.imprimirMSJE("InPreorden:
"+arb.preorden(arb.getRaiz())
				+"\nInOrden: "+arb.inOrden(arb.getRaiz())
				+"\nInOrden2: "+arb.inOrden2(arb.getRaiz())
				+"\nPostOrden: "+arb.postOrden(arb.getRaiz()));break;

				case "BUSCAR" : if(arb.arbolVacio())
Tools.imprimirMSJE("Lista Vacia");
```

```
                                else {
                                int dato=Tools.leerInt("Ingresa el dato:");
                                if(arb.buscarDato(dato)==null) {
                                        Tools.imprimirMSJE("Dato no encontrado");
                                } else
                                        Tools.imprimirMSJE("Dato encontrado: "+dato);
                                }break;

                                case "HOJAS" : if(arb.arbolVacio())
Tools.imprimirMSJE("Lista Vacia");
                                else {
                                arb.imprimirNodosTerminales();
                                }break;

                                case "INTERIORES" : if(arb.arbolVacio())
Tools.imprimirMSJE("Lista Vacia");
                                else {
                                arb.imprimirNodosInteriores();
                                }break;

                                case "ALTURA" : if(arb.arbolVacio())
Tools.imprimirMSJE("Lista Vacia");
                                else {
                                Tools.imprimirMSJE("Altura: "+arb.altura());
                                }break;
                                }
                        } while(!op.equals("SALIR"));
                }
        }
```

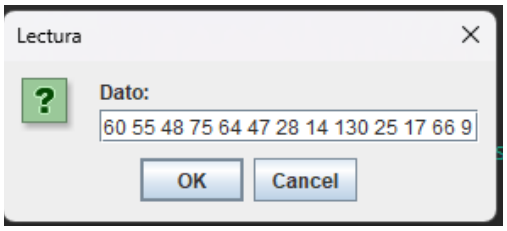
La clase "Menu_Ejecutable" es una clase que contiene el método principal "main" y permite ejecutar un programa interactivo para manipular un árbol binario.
El método "main" realiza lo siguiente:

1. Crea una instancia de la clase "ArbolBin" parametrizada con el tipo de dato Integer.
2. Muestra un menú de opciones utilizando el método estático "MenuArboles" de la clase "Tools" para que el usuario seleccione una acción a realizar.
3. Según la opción seleccionada por el usuario, realiza una serie de operaciones:
    - "INSERTAR": Pide al usuario ingresar un dato entero y lo inserta en el árbol utilizando el método "insertarArbol" de la clase "ArbolBin".
    - "RECORRIDO": Imprime los recorridos del árbol (preorden, inorden, inorden2 y postorden) utilizando los métodos correspondientes de la clase "ArbolBin".
    - "BUSCAR": Pide al usuario ingresar un dato entero y busca dicho dato en el árbol utilizando el método "buscarDato" de la clase "ArbolBin".
    - "HOJAS": Imprime los nodos terminales (hojas) del árbol utilizando el método "imprimirNodosTerminales" de la clase "ArbolBin".
    - "INTERIORES": Imprime los nodos interiores (no hojas) del árbol utilizando el método "imprimirNodosInteriores" de la clase "ArbolBin".

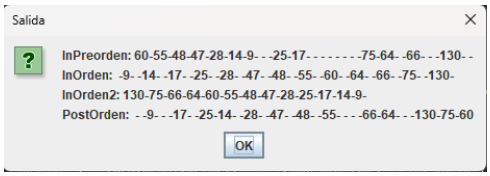
- "ALTURA": Calcula la altura del árbol utilizando el método "altura" de la clase "ArbolBin".

4. El programa se ejecuta en un bucle hasta que el usuario seleccione la opción "SALIR", momento en el cual el programa termina su ejecucion.

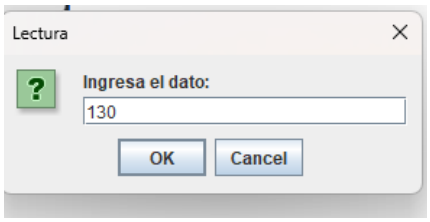
Insertar:

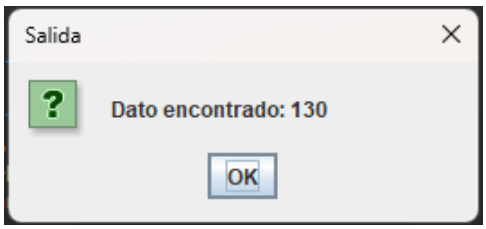


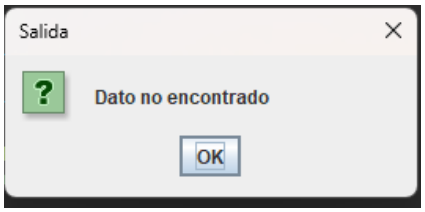
Recorrido:



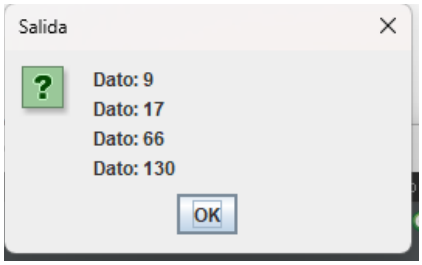
Buscar:



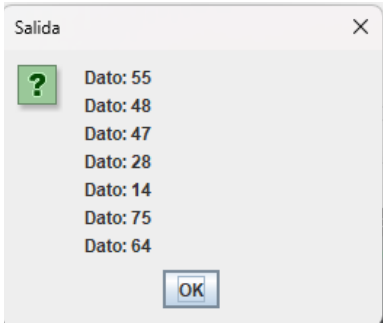




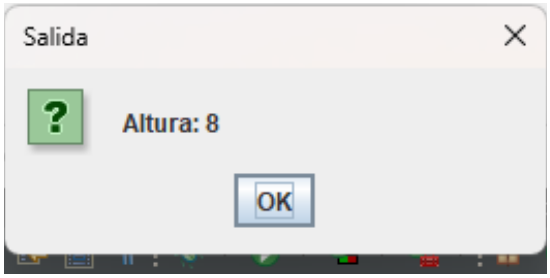
Hojas:



Interiores:



Altura:



Conclusión

En conclusión, los árboles son una estructura fundamental en la materia de Estructura de Base de Datos. Estos árboles, permiten organizar y gestionar datos de manera eficiente, optimizando el tiempo de búsqueda y las operaciones de inserción y eliminación. El uso de árboles en bases de datos proporciona ventajas significativas, como una organización estructurada de los datos, mejoras en el rendimiento de las operaciones de búsqueda y acceso a los datos, y la capacidad de implementar índices que aceleran la recuperación de información. La optimización de los árboles implica equilibrar y balancear la estructura, minimizar la altura del árbol y aplicar algoritmos y estrategias de optimización. Estas técnicas garantizan un acceso eficiente a los datos y una manipulación rápida de los mismos. El conocimiento y dominio de los árboles son fundamentales para diseñar y desarrollar bases de datos eficientes y escalables. Los árboles son herramientas poderosas para organizar grandes volúmenes de datos y mejorar el rendimiento de las consultas, lo que contribuye a la eficiencia y la efectividad de los sistemas de bases de datos. En resumen, los árboles son una pieza clave en el mundo de las bases de datos, proporcionando una estructura jerárquica eficiente para la organización y gestión de datos. Su aplicación correcta y optimización contribuyen a la mejora del rendimiento y la eficiencia de los sistemas de bases de datos, lo que beneficia a las organizaciones en la administración y acceso a su información.: