Tecnológico Nacional de México

Campus Orizaba

ESTRUCTURA DE DATOS

Carrera:

Ingeniería en sistemas computacionales

Tema : Árbol Binario

Alumnos:

Brayan Hernández Acevedo

Cesar Ríos Méndez

Pedro Sentíes Robles

Pedro Tezoco Cruz

Grupo:3g2B

Fecha 30/05/23

Introducción

Los árboles binarios son una estructura de datos fundamental en ciencias de la computación y se utilizan para organizar y almacenar datos de manera jerárquica. En un árbol binario, cada nodo puede tener como máximo dos hijos: un hijo izquierdo y un hijo derecho.

La estructura de un árbol binario se compone de nodos que están conectados entre sí mediante enlaces o referencias. El nodo en la parte superior se conoce como "raíz", mientras que los nodos que no tienen hijos se denominan "hojas". Los nodos que tienen un hijo se llaman "nodos internos". Cada nodo puede contener un valor o dato, que puede ser cualquier tipo de información deseada.

La propiedad clave de los árboles binarios es que el hijo izquierdo de un nodo tiene un valor menor o igual al valor del nodo padre, mientras que el hijo derecho tiene un valor mayor. Esta propiedad es conocida como "propiedad de ordenación" y es fundamental para la búsqueda y el ordenamiento eficiente de los datos almacenados en el árbol.

Los árboles binarios son ampliamente utilizados en la implementación de estructuras de datos como los árboles de búsqueda binarios (BST, por sus siglas en inglés), que permiten realizar búsquedas rápidas y eficientes. También se utilizan en la implementación de algoritmos de ordenamiento, como el ordenamiento rápido (quicksort) y el ordenamiento por fusión (merge sort).

Además de la propiedad de ordenación, los árboles binarios pueden tener otras características y variantes. Por ejemplo, los árboles de búsqueda binarios balanceados, como el árbol AVL o el árbol rojo-negro, mantienen una altura balanceada para garantizar un rendimiento óptimo en operaciones de búsqueda, inserción y eliminación.

Métodos para la creación de un árbol Binario

Este método llamado "insertarArbol" se utiliza para insertar un nuevo nodo en un árbol binario

1. Se crea un nuevo nodo llamado "p" y se le asigna la información proporcionada como parámetro "info".
2. Se verifica si el árbol está vacío llamando al método "arbolVacio()". Si el árbol está vacío, el nuevo nodo "p" se establece como la raíz del árbol.
3. Si el árbol no está vacío, se busca el padre del nuevo nodo "p" en el árbol llamando al método "buscaPadre(raiz, p)". El padre se almacena en el nodo "padre".
4. Se compara la información del nuevo nodo "p" con la información del nodo padre. Si la información de "p" es mayor o igual que la información del padre, el nuevo nodo se agrega como hijo derecho del padre. Si es menor, se agrega como hijo izquierdo.

```
public void insertarArbol(T info) {
        Nodito<T> p = new Nodito(info);
        if (arbolVacio()) {
            raiz = p;
        } else {
            Nodito<T> padre = buscaPadre(raiz, p);
            if (p.getInfo().compareTo(padre.getInfo()) >= 0) {
                padre.setDer(p);
            } else {
                padre.setIzq(p);
            }
        }
    }
```

Este método llamado "buscaPadre" se utiliza para encontrar el padre de un nodo en un árbol binario de búsqueda.

1. Se declara una variable llamada "padre" e inicialmente se establece como null. Esta variable se utilizará para almacenar el padre del nodo buscado.
2. Se inicia un bucle while que se ejecutará mientras el nodo actual no sea null. Este bucle se utiliza para recorrer el árbol en busca del padre del nodo.
3. Dentro del bucle, se asigna el nodo actual a la variable "padre", ya que este es el padre potencial del nodo buscado.
4. Luego, se compara la información del nodo buscado "p" con la información del nodo padre "padre" utilizando el método "compareTo()". Si la información de "p" es mayor o igual que la información del padre, se sigue buscando en el subárbol derecho y se actualiza el nodo actual con el hijo derecho del padre. Si es menor, se sigue buscando en el subárbol izquierdo y se actualiza el nodo actual con el hijo izquierdo del padre.
5. El bucle continúa hasta que se llegue a un punto donde el nodo actual sea null, lo que indica que se ha encontrado la ubicación donde se insertará el nuevo nodo. En este punto, el valor de "padre" será el padre del nodo buscado.
6. Finalmente, se devuelve el nodo "padre", que es el padre del nodo buscado.

```
public Nodito<T> buscaPadre(Nodito<T> actual, Nodito<T> p) {
        Nodito<T> padre = null;
        while (actual != null) {
                padre = actual;
                if (p.getInfo().compareTo(padre.getInfo()) >= 0) {
                        actual = padre.getDer();
                } else {
                        actual = padre.getIzq();
                }
        }
        return padre;
}
```

Este método llamado "preOrden" se utiliza para realizar un recorrido preorden en un árbol binario.

1. El método recibe como parámetro un nodo "r", que representa la raíz del subárbol en el que se realizará el recorrido.
2. Se verifica si el nodo "r" no es nulo. Si el nodo no es nulo, se realiza el recorrido preorden. En caso contrario, se devuelve una cadena vacía.
3. En el recorrido preorden, se realiza la siguiente concatenación de cadenas:
    - El valor de la información del nodo actual "r".
    - Un guion "-" como separador.
    - La llamada recursiva a "preOrden" pasando el hijo izquierdo del nodo actual "r".
    - Un guion "-" como separador.
    - La llamada recursiva a "preOrden" pasando el hijo derecho del nodo actual "r".
    - La concatenación se realiza de izquierda a derecha, lo que significa que se procesa primero el nodo actual, luego el subárbol izquierdo y finalmente el subárbol derecho.
4. Este proceso se repite recursivamente para cada nodo no nulo en el árbol, hasta que se llegue a las hojas y se retorne una cadena vacía.

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

Para los siguientes métodos de recorridos sucede algo similar pero el orden cambia dependiendo del tipo de recorrido:

```
    public String inOrden1(Nodito<T> r) {
        if (r != null) {
            return inOrden1(r.getIzq()) + "-" + r.getInfo() + "-" + inOrden1(r.getDer());
        } else {
```

```
                return "";
            }
        }
        public String inOrden2(Nodito<T> r) {
            if (r != null) {
                return inOrden2(r.getDer()) + "-" + r.getInfo() + "-" + inOrden2(r.getIzq());
            } else {
                return "";
            }
        }
        public String posOrden(Nodito<T> r) {
            if (r != null) {
                return posOrden(r.getIzq()) + "-" + posOrden(r.getDer()) + "-" + r.getInfo();
            } else {
                return "";
            }
        }
```

Este método llamado "buscaDato" se utiliza para buscar un dato específico en un árbol binario de búsqueda.

1. El método recibe como parámetro un nodo "nodo", que representa la raíz del subárbol en el que se realizará la búsqueda, y un entero "dato", que es el valor que se desea buscar.
2. Se verifica si el nodo "nodo" es nulo. Si el nodo es nulo, significa que se ha llegado a una hoja y el dato no se encuentra en el árbol. En ese caso, se devuelve false.
3. Se compara el valor del dato buscado con la información del nodo actual "nodo". Si son iguales, se ha encontrado el dato y se devuelve true.
4. Si el dato buscado es mayor que la información del nodo actual, se realiza una llamada recursiva a "buscaDato" pasando el hijo derecho del nodo actual. Esto se hace porque en un árbol binario de búsqueda, los valores mayores se encuentran en los subárboles derechos.

5. Si el dato buscado es menor que la información del nodo actual, se realiza una llamada recursiva a "buscaDato" pasando el hijo izquierdo del nodo actual. Esto se hace porque en un árbol binario de búsqueda, los valores menores se encuentran en los subárboles izquierdos.
6. El proceso de búsqueda se repite recursivamente hasta que se encuentre el dato o se llegue a una hoja nula en el árbol. En cada llamada recursiva, se desciende por el subárbol derecho o izquierdo según corresponda.

```
public static boolean buscaDato(Nodito<Integer> nodo, int dato) {
        if (nodo == null) {
                return false;
        }
        if (dato == nodo.getInfo()) {
                return true;
        } else if (dato > nodo.getInfo()) {
                return buscaDato(nodo.getDer(), dato);
        } else {
                return buscaDato(nodo.getIzq(), dato);
        }}
```

Este método llamado "imprimeNodosF" se utiliza para imprimir los nodos hoja de un árbol binario.

1. Se declara una variable de tipo String llamada "cadena" e inicialmente se establece como una cadena vacía. Esta variable se utilizará para concatenar los nodos hoja.
2. Se verifica si el nodo "nodo" no es nulo. Si el nodo no es nulo, se procede con la impresión de los nodos hoja. En caso contrario, se devuelve la cadena vacía.
3. Dentro del bloque de código condicional, se realiza la siguiente verificación:
    - Se verifica si el nodo actual "nodo" no tiene hijos, es decir, tanto el hijo izquierdo como el hijo derecho son nulos. Esto indica que el nodo es una hoja.
    - Si el nodo es una hoja, se concatena la información del nodo convertida a String con una coma y un espacio ", ".
    - Si el nodo no es una hoja, se realizan llamadas recursivas a "imprimeNodosF" pasando el hijo izquierdo y el hijo derecho del nodo actual "nodo". Esto se hace para continuar explorando los subárboles y encontrar más nodos hoja.

4. La concatenación se realiza de izquierda a derecha, lo que significa que primero se procesan los nodos hoja en el subárbol izquierdo y luego en el subárbol derecho.
5. El proceso recursivo se repite hasta que se hayan explorado todos los nodos del árbol. Los nodos hoja encontrados se concatenan en la variable "cadena".
6. Finalmente, se devuelve la cadena que contiene los valores de los nodos hoja separados por comas y espacios.

```
public String imprimeNodosF(Nodito<T> nodo) {
        String cadena = "";
        if (nodo != null) {
                if (nodo.getIzq() == null && nodo.getDer() == null) {
                        cadena += nodo.getInfo().toString() + ", ";
                } else {
                        cadena += imprimeNodosF(nodo.getIzq());
                        cadena += imprimeNodosF(nodo.getDer());
                }
        }
        return cadena;
}
```

Este método llamado "alturaArbol" se utiliza para calcular la altura de un árbol binario. La altura de un árbol se define como la longitud del camino más largo desde la raíz hasta una hoja.

1. El método recibe como parámetro un nodo "nodo", que representa la raíz del subárbol para el cual se calculará la altura.
2. Se verifica si el nodo "nodo" es nulo. Si el nodo es nulo, significa que se ha llegado a una hoja y la altura es 0. En ese caso, se devuelve 0.
3. Si el nodo no es nulo, se realiza el cálculo de la altura. Se realiza lo siguiente:
   Se realiza una llamada recursiva a "alturaArbol" pasando el hijo izquierdo del nodo actual "nodo". Esto se hace para calcular la altura del subárbol izquierdo.
   Se realiza una llamada recursiva a "alturaArbol" pasando el hijo derecho del nodo actual "nodo". Esto se hace para calcular la altura del subárbol derecho.
   Se utiliza la función "Math.max()" para obtener el máximo entre las alturas calculadas del subárbol izquierdo y derecho.

Se suma 1 al máximo obtenido, ya que se considera el nodo actual en el cálculo de la altura.
El resultado final es la altura del árbol.

4. El proceso de cálculo de la altura se repite recursivamente para cada nodo no nulo en el árbol, hasta que se llegue a las hojas y se retorne la altura total del árbol.

```
public int alturaArbol(Nodito<T> nodo) {
        if (nodo == null) {
                return 0;
        } else {
                int alturaIzq = alturaArbol(nodo.getIzq());
                int alturaDer = alturaArbol(nodo.getDer());

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

Este método llamado "buscInteriores" se utiliza para buscar y retornar una cadena que contiene los valores de los nodos interiores de un árbol binario. Los nodos interiores son aquellos que no son la raíz y que tienen al menos un hijo.

1. Se declara una variable de tipo String llamada "cadena" e inicialmente se establece como una cadena vacía. Esta variable se utilizará para concatenar los valores de los nodos interiores.
2. Se verifica si el nodo "nodo" no es nulo. Si el nodo no es nulo, se procede con la búsqueda de los nodos interiores. En caso contrario, se devuelve la cadena vacía.
3. Dentro del bloque de código condicional, se realiza la siguiente verificación:
   Se verifica si el nodo actual "nodo" no es la raíz del árbol y si tiene al menos un hijo. Esto se hace verificando que tanto el hijo izquierdo como el hijo derecho no sean nulos.
   Si se cumple la condición, se concatena la información del nodo convertida a String con una coma y un espacio ", ".
   Luego, se realizan llamadas recursivas a "buscInteriores" pasando el hijo izquierdo y el hijo derecho del nodo actual "nodo". Esto se hace para continuar buscando nodos interiores en los subárboles.

4. La concatenación se realiza de izquierda a derecha, lo que significa que primero se procesan los nodos en el subárbol izquierdo y luego en el subárbol derecho.
5. El proceso recursivo se repite hasta que se hayan explorado todos los nodos del árbol. Los nodos interiores encontrados se concatenan en la variable "cadena".
6. Finalmente, se devuelve la cadena que contiene los valores de los nodos interiores separados por comas y espacios.

```
public String buscInteriores(Nodito<T> nodo) {
        String cadena = "";
        if (nodo != null) {
            if (nodo != raiz && (nodo.getIzq() != null || nodo.getDer() != null))
{
                cadena += nodo.getInfo().toString() + ", ";
            }
            cadena += buscInteriores(nodo.getIzq());
            cadena += buscInteriores(nodo.getDer());
        }
        return cadena;
    }
```

Este método llamado "dibujarArbol" se utiliza para visualizar el árbol binario de forma gráfica, imprimiendo una representación en forma de árbol en la consola.

1. Se verifica si el árbol está vacío utilizando el método "arbolVacio()". Si el árbol está vacío, se imprime un mensaje indicando que el árbol está vacío y se retorna sin realizar más acciones.
2. Se crea un objeto StringBuilder llamado "sb" que se utilizará para construir la representación gráfica del árbol.
3. Se llama al método "dibujarNodo" pasando la raíz del árbol, el objeto StringBuilder "sb", una cadena vacía como prefijo y un valor booleano "true" para indicar que el nodo es el hijo izquierdo de su padre.
4. Se imprime el contenido del objeto StringBuilder "sb" utilizando el método "ImprimeMensaje" de la clase Tools. Se concatena la cadena "Binario" y se pasa como parámetro la constante JOptionPane.INFORMATION_MESSAGE para indicar el tipo de mensaje.
5. El método privado "dibujarNodo" es llamado recursivamente para construir la representación gráfica del árbol. Recibe como parámetros el nodo actual

"nodo", el objeto StringBuilder "sb", una cadena "prefijo" que se utiliza para agregar espacios y líneas de conexión, y un valor booleano "esIzquierdo" que indica si el nodo es el hijo izquierdo de su padre.

6. Dentro del método "dibujarNodo", se verifica si el nodo actual "nodo" no es nulo. Si no es nulo, se procede a agregar la representación del nodo al objeto StringBuilder "sb".
    - Se concatena el prefijo con la línea de conexión correspondiente (├── si es el hijo izquierdo, └── si es el hijo derecho).
    - Se agrega la información del nodo convertida a String.
    - Se agrega un salto de línea "\n" para separar los nodos.
7. Luego, se realiza una llamada recursiva al método "dibujarNodo" para el hijo derecho del nodo actual, pasando el prefijo correspondiente (con espacios adicionales si es el hijo izquierdo) y actualizando el valor de "esIzquierdo" a "false".
8. A continuación, se realiza otra llamada recursiva al método "dibujarNodo" para el hijo izquierdo del nodo actual, pasando el prefijo correspondiente (con espacios adicionales si es el hijo izquierdo) y actualizando el valor de "esIzquierdo" a "true".
9. El proceso recursivo se repite hasta que se hayan explorado todos los nodos del árbol, construyendo la representación gráfica del árbol en el objeto StringBuilder "sb".

```
public void dibujarArbol() {
        if (arbolVacio()) {
            Tools.ImprimeMensaje("El árbol está vacío.");
            return;
        }
        StringBuilder sb = new StringBuilder();
        dibujarNodo(getRaiz(), sb, "", true);
        Tools.ImprimeMensaje(sb.toString() + "Binario" +
JOptionPane.INFORMATION_MESSAGE);
    }


    private void dibujarNodo(Nodito<T> nodo, StringBuilder sb, String prefijo,
boolean esIzquierdo) {
        if (nodo != null) {
```

```
                sb.append(prefijo).append(esIzquierdo ? "├── " : "└──
").append(nodo.getInfo().toString()).append("\n");

                dibujarNodo(nodo.getDer(), sb, prefijo + (esIzquierdo ? "│   " : "
"), false);

                dibujarNodo(nodo.getIzq(), sb, prefijo + (esIzquierdo ? "│   " : "
"), true);

            }

        }
```

Conclusión

En conclusión, los árboles binarios son una estructura de datos fundamental en informática y ciencias de la computación. Estos árboles se caracterizan por tener un máximo de dos hijos por nodo, conocidos como hijo izquierdo y hijo derecho.

Los árboles binarios ofrecen varias ventajas y aplicaciones en el campo de la informática. Algunas de estas ventajas incluyen:

Búsqueda eficiente: Los árboles binarios permiten realizar búsquedas eficientes, ya que el tiempo de búsqueda en un árbol binario balanceado es O(log n). Esto los convierte en una estructura de datos ideal para almacenar y buscar datos de manera rápida.

Ordenamiento: Los árboles binarios también se utilizan en algoritmos de ordenamiento, como el árbol de búsqueda binaria. Estos algoritmos aprovechan la estructura jerárquica de los árboles para ordenar los datos de manera eficiente.

Manipulación de datos: Los árboles binarios permiten realizar operaciones de inserción, eliminación y búsqueda de elementos de manera eficiente. Estas operaciones son fundamentales en muchas aplicaciones y algoritmos.

Representación de jerarquías: Los árboles binarios se utilizan para representar estructuras jerárquicas, como el árbol genealógico, la estructura de directorios en un sistema de archivos o la estructura de un árbol sintáctico en análisis gramatical.

Sin embargo, también es importante tener en cuenta algunas consideraciones al utilizar árboles binarios:

Balanceo: Para garantizar un rendimiento óptimo, es importante mantener el árbol balanceado. Un árbol desequilibrado puede afectar negativamente la eficiencia de las operaciones.

Implementación adecuada: La elección de la implementación del árbol binario es crucial. Existen diferentes variantes de árboles binarios, como árboles de búsqueda binaria, árboles AVL o árboles rojo-negro, cada uno con sus propias características y eficiencia en diferentes escenarios.

En resumen, los árboles binarios son una estructura de datos versátil y poderosa que se utiliza en una amplia variedad de aplicaciones. Su capacidad para realizar búsquedas eficientes y representar jerarquías los convierte en una herramienta fundamental en el campo de la informática y las estructuras de datos. Al comprender los conceptos y las operaciones asociadas con los árboles binarios, los desarrolladores pueden aprovechar su potencial para mejorar el rendimiento y la eficiencia de sus aplicaciones.