INSTITUTO TECNOLÓGICO NACIONAL DE MEXICO CAMPUS ORIZABA

Materia:

Estructura De Datos

Maestra

Martínez Castillo María Jacinta

Integrantes:

Hernández Angón Diego
Morales Valdeolivar Gael Octavio

Grupo:

3PM – 4PM HRS

Clave:

3a3A

Especialidad:

Ing. informática

Fecha De Entrega

02/06/2023

REPORTE DE PRACTICAS
UNIDAD 4

Introducción

Estas estructuras de datos juegan un papel crucial en la representación y organización de información de manera jerárquica y conectada.

Un árbol es una estructura no lineal compuesta por nodos interconectados de forma jerárquica. Cada árbol tiene un nodo raíz que es el punto de partida, ya partir de ahí se ramifica en diferentes niveles formando subárboles. Cada nodo puede tener hijos, que son nodos descendientes directos, y también puede tener padres, que son nodos antecesores. Los árboles se utilizan ampliamente en la organización de datos, como en la representación de árboles genealógicos, estructuras de directorios en sistemas operativos y en la implementación de algoritmos como los árboles de búsqueda binaria y los árboles AVL.

Por otro lado, los gráficos son estructuras que representan relaciones entre objetos. Un grafo está compuesto por nodos (también llamados vértices) y aristas, que son las conexiones entre los nodos. Los gráficos pueden ser dirigidos, donde las aristas tienen una dirección definida, o no dirigidos, donde las aristas no tienen dirección. Además, los gráficos pueden ser ponderados, es decir, los aristas tienen un valor asociado. Los gráficos se utilizan en una amplia variedad de aplicaciones, como la representación de redes sociales, sistemas de transporte, diagramas de flujo y algoritmos de rutas más cortas.

Competencia especifica.

Específica(s):

Comprende y aplica estructuras no lineales para la solución de problemas.

Genéricas:

- Habilidad para buscar y analizar información proveniente de fuentes diversas.
- La comprensión y manipulación de ideas y pensamientos.
- Metodologías para solución de problemas,

organización del tiempo y para el aprendizaje.

- Habilidad en el manejo de equipo de cómputo
- Capacidad para trabajar en equipo.

- Capacidad de aplicar los conocimientos en la práctica.

Marco teórico

4.1 Arboles

4.1.1 Clasificación de árboles

Concepto de arboles

En ciencias de la computación y en informática, un árbol es un tipo abstracto de datos (TAD) ampliamente usado que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados.

Una estructura de datos de árbol se puede definir de forma recursiva (localmente) como una colección de nodos (a partir de un nodo raíz), donde cada nodo es una estructura de datos con un valor, junto con una lista de referencias a los nodos (los hijos), con la condición de que ninguna referencia esté duplicada ni que ningún nodo apunte a la raíz.

Alternativamente, un árbol se puede definir de manera abstracta en su conjunto como un árbol ordenado, con un valor asignado a cada nodo. Ambas perspectivas son útiles: mientras que un árbol puede ser analizado matemáticamente, realmente es representado como una estructura de datos en la que se trabaja con cada nodo por separado (en lugar de como una lista de nodos y una lista de adyacencia entre nodos, como un grafo). Mirando a un árbol como conjunto, se puede hablar del nodo padre de un nodo dado, pero en general se habla de una estructura de datos de un nodo dado que sólo contiene la lista de sus hijos sin referencia a su padre (si lo hay).

Tipos de arboles

1. **Árbol general**

Si no se coloca ninguna restricción en la jerarquía del árbol, un árbol se llama árbol general. Cada nodo puede tener un número infinito de hijos en el árbol general. El árbol es el superconjunto de todos los demás árboles.

2. **Árbol binario**

   El árbol binario es el tipo de árbol en el que se pueden encontrar la mayoría de los dos hijos para cada padre. Los niños son conocidos como el niño izquierdo y el niño derecho. Esto es más popular que la mayoría de los otros árboles. Cuando se aplican ciertas restricciones y características en un árbol binario, también se utilizan otras como árbol AVL, BST (árbol de búsqueda binario), árbol RBT, etc. Cuando avancemos, explicaremos todos estos estilos en detalle.

3. **Árbol de búsqueda binaria**

   Es una extensión de árbol binario con varias restricciones opcionales. El valor secundario izquierdo de un nodo en BST debe ser menor o igual que el valor primario y el valor secundario derecho siempre debe ser mayor o igual que el valor primario. Esta propiedad del árbol de búsqueda binaria lo hace ideal para las operaciones de búsqueda, ya que podemos determinar con precisión en cada nodo si el valor está en el subárbol izquierdo o derecho. Por eso se nombra el Árbol de búsqueda.

4. **Árbol AVL**

   El árbol AVL es un árbol de búsqueda binario que se autoequilibra. En nombre de los inventores Adelson-Velshi y Landis, se le da el nombre AVL. Este fue el primer árbol que se equilibró dinámicamente. Se asigna un factor de equilibrio para cada nodo en el árbol AVL, en función de si el árbol está equilibrado o no. La altura de los niños del nodo es como máximo 1. AVL vine. En el árbol AVL, el factor de equilibrio correcto es 1, 0 y -1. Si el árbol tiene un nuevo nodo, se rotará para garantizar que esté equilibrado. Luego se rotará. Las operaciones comunes, como la visualización, la inserción y la eliminación, llevan tiempo O (log n) en el árbol AVL. Se aplica principalmente cuando se trabaja con operaciones de búsqueda.

5. **Árbol rojo-negro**

Otro tipo de árbol de equilibrio automático es rojo-negro. El nombre rojo-negro se da porque el árbol rojo-negro tiene rojo o negro pintado en cada nodo de acuerdo con las propiedades del árbol rojo-negro. Mantiene el equilibrio del bosque. Aunque este árbol no es un árbol completamente equilibrado, la operación de búsqueda solo toma tiempo O (log n). Cuando los nuevos nodos se agregan en Red-Black Tree, los nodos se rotarán nuevamente para mantener las propiedades del Red-Black Tree.

6. **Árbol N-ary**

   El número máximo de hijos en este tipo de árbol que puede tener un nodo es N. Un árbol binario es un árbol de dos años, como máximo 2 hijos en cada nodo de árbol binario. Un árbol N-ary completo es un árbol donde los hijos de un nodo son 0 o N.

7. **Árboles de decisión**

   Son utilizados en aprendizaje automático y minería de datos. Representan una estructura de árbol donde cada nodo interno corresponde a una característica o atributo, y los nodos hoja representan las clases o resultados. Se utilizan para tomar decisiones basadas en un conjunto de características o atributos.

8. **Árboles de segmentos**

   Se utilizan para resolver problemas relacionados con consultas de intervalos en secuencias de datos. Cada nodo representa un intervalo y los nodos hijos representan subdivisiones de ese intervalo. Son útiles para realizar operaciones de consulta y actualización eficientes en un rango específico.

9. **Árboles Trie**

   También conocidos como árboles de prefijos, se utilizan principalmente para representar y buscar palabras o cadenas de caracteres. Cada nodo representa un carácter y los caminos desde la raíz hasta los nodos hoja forman todas las palabras o prefijos posibles.

10. **Árboles B**

    Son árboles balanceados en los que cada nodo puede tener más de dos hijos. Están diseñados para manejar grandes cantidades de datos y se utilizan comúnmente en bases de datos y sistemas de archivos.

4.1.2 Operaciones Básicas Sobre Arboles Binarios

Inserción:

La operación de inserción en un árbol binario consiste en agregar un nuevo nodo al árbol respetando la estructura jerárquica y las reglas de ordenamiento.

Siguiendo los siguientes pasos básicos para realizar la inserción:

1. Comenzar desde la raíz del árbol: Si el árbol está vacío, el nuevo nodo se convierte en la raíz del árbol. En caso contrario, se realiza la búsqueda del lugar adecuado para insertar el nuevo nodo.
2. Comparar el valor del nuevo nodo con el valor del nodo actual: Si el valor del nuevo nodo es menor que el valor del nodo actual, se desciende al subárbol izquierdo. Si es mayor, se desciende al subárbol derecho.
3. Repetir el paso 2 hasta encontrar una posición vacía: Se continúa descendiendo por el árbol, comparando y moviéndose hacia el subárbol izquierdo o derecho según corresponda, hasta encontrar un nodo que no tenga un hijo en la dirección necesaria.
4. Insertar el nuevo nodo: Una vez se encuentra una posición vacía, se crea un nuevo nodo con el valor deseado y se enlaza desde el nodo actual en la dirección correspondiente (izquierda o derecha).

```java
public void insertarArbol(T info) {
    Nodito p=new Nodito(info);
    if(arbolVacio()) {
        raiz=p;
    }

    else {
        Nodito padre=buscaPadre(raiz, p);

        if((int)p.info>=(int)padre.info)
            padre.setDer(p);
        else
            padre.setIzq(p);
    }
}
```

Recorridos:

Un recorrido en un árbol binario es una operación que consiste en visitar todos sus vértices o nodos, de tal manera que cada vértice se visite una sola vez. Se distinguen tres tipos de recorrido:

- Inorden:

  En el recorrido inorden, primero recorre el subárbol izquierdo, luego visita la raíz y, por último, va al subárbol derecho.

  hijo izquierdo — raíz — hijo derecho

  El recorrido inorden se utiliza comúnmente para obtener una secuencia ordenada de los elementos almacenados en un árbol binario. Por ejemplo, si los nodos representan valores numéricos, el recorrido inorden devuelve una secuencia ordenada ascendente.

```java
public String inOrden(Nodito r) {
    if(r!=null) {
        return inOrden(r.izq) +" - "+ r.info +" - "+inOrden(r.der);
    }
    else
        return"";

    }
```

- Pre-orden

  En el recorrido preorden, primero visita la raíz, luego recorre el subárbol izquierdo y por último va a subárbol derecho.

raíz — hijo izquierdo — hijo derecho

El recorrido preorden es útil para realizar una copia del árbol, imprimir la estructura jerearquica del árbol o ejecutar alguna operación en cada nodo antes de visitar subárboles.

```
public String preorden(Nodito r) {
    if(r!=null) {
        return r.info+" - "+preorden(r.izq)+" - "+preorden(r.der);
    }
    else
        return"";

    }
```

- Post-orden

  En el recorrido postorden, Primero recorre el subárbol izquierdo, luego recorre el subárbol derecho y por último, visita la raíz.

  hijo izquierdo– hijo derecho — raíz

  El recorrido postorden se utiiza comúnmente en algoritmos de eliminación de nodos, liberación de memoria o para realizar alguna operación en cada nodo después de visitar los subárboles.

```
84
85    public String posOrden(Nodito r) {
86        if(r!=null) {
87            return posOrden(r.izq) +" - "+posOrden(r.der)+" - "+ r.info;
88        }
89        else
90            return"";
91
92        }
93
```

Estos recorridos son algoritmos recursivos, lo que significa que se invocan a si mismos para recorrer los subárboles. Son fundamentales para realizar operaciones en arboles binarios, como búsqueda, inserción, eliminación o impresión de los elementos.

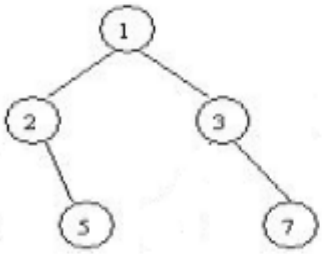


figura 12.5a

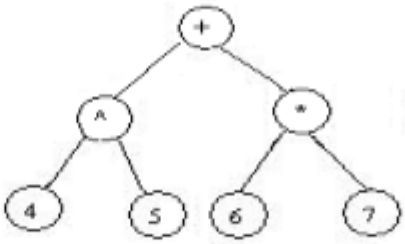


figura 12.5b

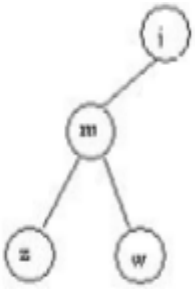


figura 12.5c

RECORRIDOS	Figura 12.5a	Figura 12.5b	Figura 12.5c
INORDEN	2-5-1-3-7	4^5+6*7	z m n i
PREORDEN	1-2-5-3-7	+^4 5*6 7	i m z n
POSORDEN	5-2-7-3-1	4 5 ^ 6 7*+	z n m i

Búsqueda:

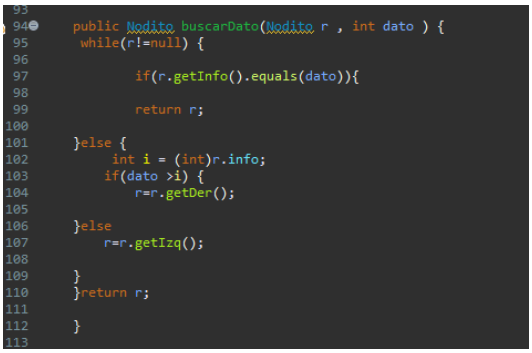
La operación de búsqueda en un árbol binario implica encontrar un nodo específico que contiene un valor deseado. La búsqueda se realiza siguiendo un algoritmo recursivo o iterativo que compara el valor buscado con el valor de los nodos del árbol y se dirige hacia el subárbol izquierdo o derecho según sea necesario.

Pasos básicos para realizar la búsqueda en un árbol binario:

1. Comenzar desde la raíz del árbol: La búsqueda comienza en la raíz del árbol y se compara el valor buscado con el valor del nodo actual.
2. Comparar el valor buscado con el valor del nodo actual: Si el valor buscado es igual al valor del nodo actual, se ha encontrado el nodo deseado y se puede realizar cualquier acción requerida (por ejemplo, devolver el nodo o realizar alguna operación sobre él).
3. Determinar el subárbol a explorar: Si el valor buscado es menor que el valor del nodo actual, se debe realizar la búsqueda en el subárbol izquierdo. Si el valor buscado es mayor, se debe realizar la búsqueda en el subárbol derecho.
4. Repetir los pasos anteriores: Dependiendo del subárbol en el que se deba realizar la búsqueda, se repiten los pasos 2 y 3 hasta que se encuentre el

nodo deseado o se llegue a una posición vacía (indicando que el valor buscado no está presente en el árbol).

Este proceso es recursivo, ya que cuando nos movemos a un nodo hijo, podemos considerar a este como el nodo raíz de un nuevo árbol.



```
93
94    public Nodito buscarDato(Nodito r , int dato ) {
95     while(r!=null) {
96
97            if(r.getInfo().equals(dato)){
98
99            return r;
100
101   }else {
102        int i = (int)r.info;
103       if(dato >i) {
104           r=r.getDer();
105
106   }else
107       r=r.getIzq();
108
109   }
110   }return r;
111
112   }
113
```

Eliminación:

La operación de eliminación en un árbol binario implica la eliminación de un nodo específico y la reorganización del árbol para mantener su estructura y propiedades. La eliminación en un árbol binario puede ser un poco más compleja que la inserción o la búsqueda, ya que implica considerar diferentes casos según la estructura del árbol y el nodo que se desea eliminar.

Pasos básicos para realizar la eliminación en un árbol binario:

1. Buscar el nodo a eliminar: El primer paso es buscar el nodo que se desea eliminar. Utilizando el algoritmo de búsqueda para encontrar el nodo con el valor deseado en el árbol.
2. Manejar los casos de eliminación:

   Caso 1: El nodo a eliminar es una hoja (no tiene hijos): En este caso, simplemente se elimina el nodo del árbol.

   Caso 2: El nodo a eliminar tiene un solo hijo: En este caso, se reemplaza el nodo a eliminar por su hijo, asegurándose de mantener las referencias correctas en el árbol.

Caso 3: El nodo a eliminar tiene dos hijos: En este caso, se debe encontrar el sucesor inorden o el predecesor inorden del nodo (el nodo más cercano en valor que está a la izquierda o a la derecha del nodo a eliminar). Luego, se reemplaza el valor del nodo a eliminar por el valor del sucesor o predecesor inorden y se procede a eliminar el sucesor o predecesor inorden en su posición original.

3. Actualizar el árbol: Después de eliminar el nodo, se deben actualizar las referencias en el árbol para mantener su estructura y propiedades, asegurando que los nodos restantes estén correctamente enlazados.

Mostrar Hojas:

Para mostrar las hojas de un árbol binario, se debe recorrer el árbol e identificar los nodos que son hojas, es decir, aquellos que no tienen hijos.

1. Comienza desde la raíz del árbol: Comienza el recorrido desde la raíz del árbol.
2. Realiza un recorrido en orden: Realiza un recorrido en orden (inorden) recursivo en el árbol.
3. Identifica las hojas: Durante el recorrido, verifica si el nodo actual no tiene hijos (es una hoja). Si es así, muestra o almacena el valor del nodo, ya que es una hoja del árbol.
4. Repite para los subárboles: Luego, repite los pasos 2 y 3 para el subárbol izquierdo y el subárbol derecho, realizando un recorrido en orden en cada subárbol.

Recordando que las hojas son los nodos finales del árbol, por lo que en árboles balanceados, la cantidad de hojas se incrementa en cada nivel más bajo del árbol.

```
163     public String imprimirHojas(Nodito nodo) {
164         String cad = "";
165         if (nodo != null) {
166             if (nodo.izq == null && nodo.der == null) {
167                 cad = nodo.info + " ";
168             }
169             return cad + imprimirHojas(nodo.izq) + imprimirHojas(nodo.der);
170         } else {
171             return "";
172         }
173
174     }
175
176
177
178
179
180
181 }
```

Mostrar Nodos Interiores

Para mostrar los nodos interiores de un árbol binario, se debe recorrer el árbol e identificar los nodos que no son hojas, es decir, aquellos que tienen al menos un hijo.

1. Comienza desde la raíz del árbol: Comienza el recorrido desde la raíz del árbol.
2. Realiza un recorrido en orden: Realiza un recorrido en orden (inorden) recursivo en el árbol.
3. Identifica los nodos interiores: Durante el recorrido, verifica si el nodo actual tiene al menos un hijo. Si es así, muestra o almacena el valor del nodo, ya que es un nodo interior del árbol.
4. Repite para los subárboles: Luego, repite los pasos 2 y 3 para el subárbol izquierdo y el subárbol derecho, realizando un recorrido en orden en cada subárbol.

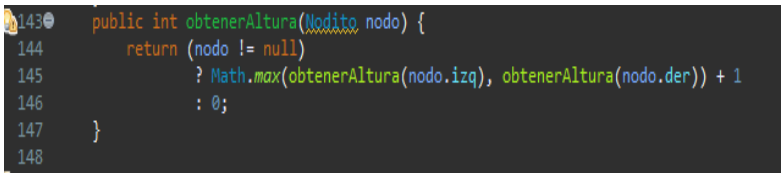
Los nodos interiores son aquellos que no son hojas, por lo que, en árboles balanceados, la cantidad de nodos interiores se reduce a medida que te acercas a los niveles más bajos del árbol.

```
public String imprimirInteriores(Nodito nodo) {
    String cad = "";
    if (nodo != null) {

        if (nodo.izq != null || nodo.der != null) {
            if(nodo.info!=raiz.info)
            cad = nodo.info + " ";
        }
        return cad + imprimirInteriores(nodo.izq) + imprimirInteriores(nodo.der);
    } else {
        return "";
    }
}
```

Mostrar Altura:

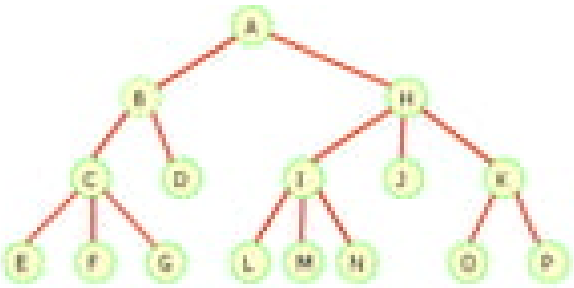
La altura de un árbol binario es la longitud del camino más largo desde la raíz del árbol hasta una de sus hojas. En otras palabras, es el número máximo de aristas que hay entre la raíz y cualquier hoja del árbol. La altura de un árbol binario se puede calcular de manera recursiva utilizando el siguiente algoritmo:

1. Si el árbol está vacío (es decir, la raíz es nula), su altura es 0.
2. Si el árbol no está vacío, la altura se calcula como 1 + el máximo entre la altura del subárbol izquierdo y la altura del subárbol derecho.



4.1.3 Aplicaciones

Los árboles representan las estructuras no lineales y dinámicas de datos más importantes en Computación. Dinámicas porque las estructuras de árbol pueden cambiar durante la ejecución de un programa. No lineales, puesto que a cada elemento del árbol pueden seguirle varios elementos. La definición de árbol es la siguiente: es una estructura jerárquica aplicada sobre una colección de Elementos u objetos llamados nodos; uno de los cuales es conocido como raíz. Además se crea una relación o parentesco entre los nodos dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ancestro, etc. Los árboles tienen una gran variedad de aplicaciones. Por ejemplo, se pueden utilizar para representar fórmulas matemáticas, para organizar adecuadamente la información, para construir un árbol genealógico, en la toma de decisiones, para el análisis de circuitos eléctricos y para numerar los capítulos y secciones de un libro. A los árboles ordenados de grado dos se les conocen como árboles binarios ya que cada nodo del árbol no tendrá más de dos descendientes directos. Las aplicaciones de los árboles binarios son muy variadas ya que se les puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos



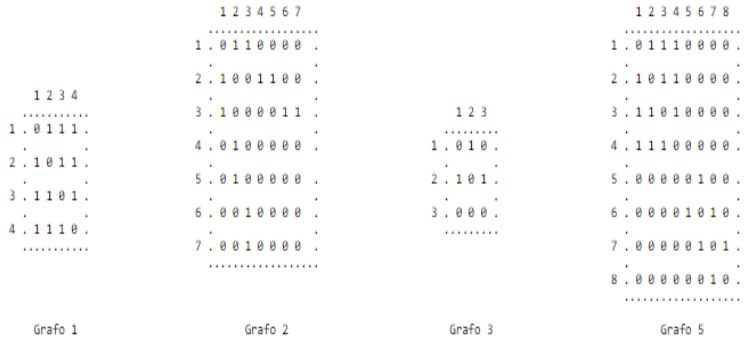
4.2 Grafos

4.2.1 Representación de grafos

Los grafos se pueden representar de diversas maneras. La representación correcta debe elegirse según la aplicación que se le dará al grafo, en particular, teniendo en cuenta las funciones a realizar sobre el mismo. Las siguientes son representaciones posibles de grafos:

Matrices de Adyacencia (Adjacency Matrices)

Son matrices cuadradas de tantas filas y columnas como vértices tenga el grafo a representar. En las celdas de la matriz se indica si existe un arco entre los vértices que determinan la celda.

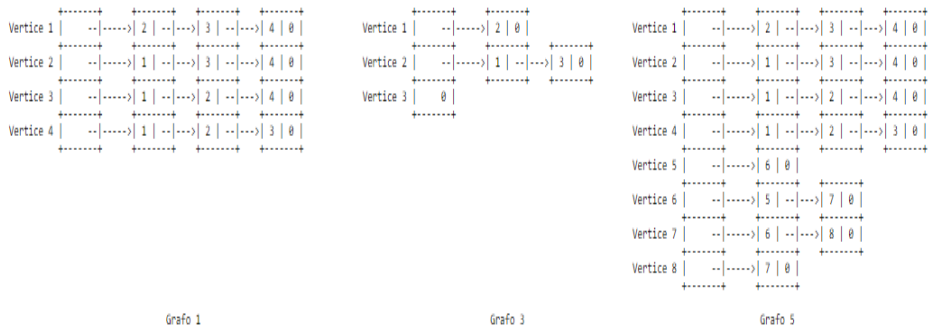


Observar que en todos los grafos la diagonal de la matriz tiene solo ceros. Observar que para los grafos no direccionados la submatriz superior es espejo de la inferior, respecto de la diagonal. Observar que en los grafos direccionados, normalmente, difieren la submatriz superior de la inferior. Observar que solo existe una señal,

indicando la existencia o no del arco, tratandose de una representación inadecuada para almacenar información perteneciente al arco, como podría ser la distancia entre dos ciudades, si los vértices representan a las mismas.

Listas de Adyacencia (Adjacency Matrices)

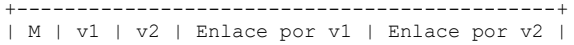
Partiendo de los nodos de cada vértice, evoluciona una lista que informa los nodos que son adyacentes del inicial.

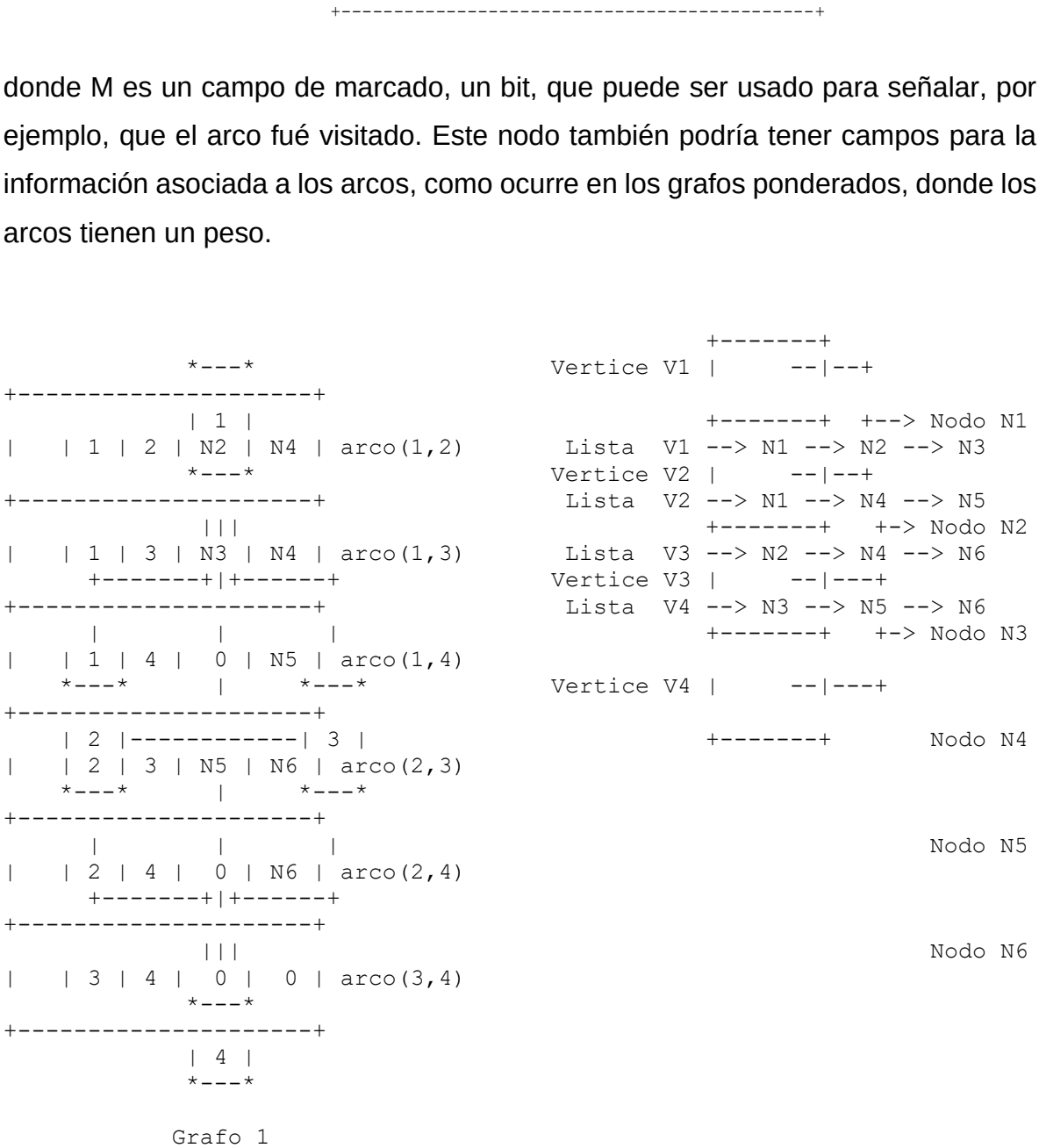


Observar que en el caso de grafos no direccionados un arco aparece dos veces. Observar que el el caso de grafos direccionados un arco aparece solo una vez. Observar que si los arcos conllevan información, es facil administrarla en el caso de los digrafos, pero no así en los grafos.

Multilistas de Adyacencia (Adjacency Multilist)

Los nodos de una multilista de adyacencia tiene enlaces a varias listas, hecho del cual proviene su nombre. Esta representación se basa en que existe un nodo, y solo uno, por cada arco, aunque este nodo enlace varias listas. La estructura del nodo es :





Una secuencia de construcción del grafo requeriría, primero, crear los nodos de los vértices, segundo, crear los nodos de los arcos, y tercero, enlazar todos los nodos donde participa un determinado vértice, recurriendo hasta agotar los vértices.

4.2.2 Operaciones básicas

En la estructura de datos de los grafos, las operaciones básicas que se pueden realizar son:

1. **Agregar un vértice**: Permite añadir un nuevo nodo o vértice al grafo. Se crea un nuevo objeto de vértice y se agrega a la estructura de datos del grafo.

2. **Eliminar un vértice:** Consiste en eliminar un vértice existente del grafo, junto con todas las aristas que estén conectadas a él. Esto implica eliminar el objeto de vértice de la estructura de datos del grafo y actualizar las conexiones entre los demás vértices.

3. **Agregar una arista:** Se utiliza para establecer una conexión entre dos vértices existentes en el grafo. Se crea un objeto de arista y se agrega a la estructura de datos del grafo, estableciendo la relación entre los dos vértices correspondientes.

4. **Eliminar una arista:** Implica eliminar la conexión entre dos vértices del grafo. Se busca y elimina el objeto de arista correspondiente de la estructura de datos del grafo, pero los vértices en sí permanecen en la estructura.

5. **Buscar un vértice:** Consiste en buscar un vértice específico dentro de la estructura de datos del grafo. Esto se puede realizar de manera secuencial, verificando cada vértice en la estructura de datos, o utilizando algoritmos más eficientes, como la búsqueda en profundidad (DFS) o la búsqueda en anchura (BFS).

6. **Recorrer el grafo:** Permite visitar todos los vértices y aristas del grafo de manera sistemática. Hay diferentes algoritmos para realizar esto, como el

recorrido en profundidad (DFS) y el recorrido en anchura (BFS), que utilizan estructuras de datos como pilas o colas para realizar el recorrido.

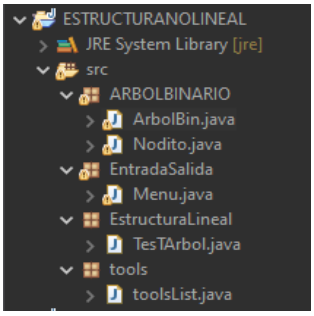
7. **Verificar la existencia de una arista:** Se utiliza para comprobar si existe una conexión directa entre dos vértices en el grafo. Esto implica buscar en la estructura de datos del grafo si hay una arista que conecte los dos vértices específicos.

8. **Obtener los vecinos de un vértice:** Consiste en obtener todos los vértices adyacentes a un vértice específico en el grafo. Se recorre la estructura de datos del grafo para encontrar las aristas que están conectadas al vértice y, a partir de ellas, se obtienen los vértices vecinos.

Estas son las operaciones básicas que se realizan en la estructura de datos de los grafos. La implementación de estas operaciones puede variar dependiendo del tipo de representación del grafo, como listas de adyacencia, matrices de adyacencia, entre otras.

Desarrollo de practica

Para el presente reporte se creo la siguiente estructura de paquetes y clases para el correcto orden y ejecución del programa



Se creo en el paquete de ARBOLBINARIO una clase llamada ArbolBin donde se crearon los métodos

```java
package ARBOLBINARIO;

public class ArbolBin<T>{
private Nodito raiz;

    public ArbolBin() {
        raiz=null;
    }

    public Nodito getRaiz() {
        return raiz;

    }

    public void setRaiz (Nodito raiz ) {
        this.raiz=raiz;

    }

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

    public void vaciarArbol() {
        raiz=null;
    }

```

```java
    public void insertarArbol(T info) {
        Nodito p=new Nodito(info);
        if(arbolVacio()) {
            raiz=p;
        }

        else {
            Nodito padre=buscaPadre(raiz, p);

            if((int)p.info>=(int)padre.info)
                padre.setDer(p);
            else
                padre.setIzq(p);
        }
    }

    public Nodito buscaPadre(Nodito actual, Nodito p) {
    Nodito padre=null;
    while (actual!=null) {
        padre=actual;
        if((int)p.info>=(int)padre.info)

         actual=padre.getDer();
         else
             actual=padre.getIzq();

    }
    return padre;
    }

    public String preorden(Nodito r) {
        if(r!=null) {
            return r.info+" - "+preorden(r.izq)+" - "+preorden(r.der);
        }
        else
            return"";

        }
```

```java
    public String inOrden(Nodito r) {
        if(r!=null) {
            return inOrden(r.izq) +" - "+ r.info +" - "+inOrden(r.der);
        }
        else
            return"";

        }

    public String enOrden(Nodito r) {
        if(r!=null) {
            return inOrden(r.der) +" - "+ r.info +" - "+inOrden(r.izq);
        }
        else
            return"";

        }

    public String posOrden(Nodito r) {
        if(r!=null) {
            return posOrden(r.izq) +" - "+posOrden(r.der)+" - "+ r.info;
        }
        else
            return"";

        }
```

```java
    public Nodito buscarDato(Nodito r , int dato ) {
     while(r!=null) {

            if(r.getInfo().equals(dato)){

            return r;

    }else {
         int i = (int)r.info;
        if(dato >i) {
            r=r.getDer();

    }else
        r=r.getIzq();

    }
    }return r;

    }

    public static void printRecursive(Nodito node, int level) {
        if (node == null) return;

        printRecursive(node.der, level + 1);

        if (level != 0) {
                for (int i = 0; i < level - 1; i++) {
                        System.out.print("|\t");
                }
                System.out.println("|-------" + node.info);
        } else {
                System.out.println(node.info);
        }
        printRecursive(node .izq, level + 1);
}
```

```java

    public void imprimirArbol(Nodito nodo, String prefijo, boolean esUltimo) {
        if (nodo == null) {
            return;
        }

        imprimirArbol(nodo.der, prefijo + (esUltimo ? "    " : "|   "), false);

        System.out.print(prefijo);
        System.out.print(esUltimo ? "|__" : "|--");
        System.out.println(nodo.info);

        imprimirArbol(nodo.izq, prefijo + (esUltimo ? "|   " : "    "), true);
    }
    public int obtenerAltura(Nodito nodo) {
        return (nodo != null)
                ? Math.max(obtenerAltura(nodo.izq), obtenerAltura(nodo.der)) + 1
                : 0;
    }

    public String imprimirInteriores(Nodito nodo) {
        String cad = "";
        if (nodo != null) {

            if (nodo.izq != null || nodo.der != null) {
                if(nodo.info!=raiz.info)
                cad = nodo.info + " ";
            }
            return cad + imprimirInteriores(nodo.izq) + imprimirInteriores(nodo.der);
        } else {
            return "";
        }
    }
```

```java
    public String imprimirHojas(Nodito nodo) {
        String cad = "";
        if (nodo != null) {
            if (nodo.izq == null && nodo.der == null) {
                cad = nodo.info + " ";
            }
            return cad + imprimirHojas(nodo.izq) + imprimirHojas(nodo.der);
        } else {
            return "";
        }

    }



}
```

Despues se creo otra clase llamada Nodito deltro del mismo paquete

busquedas.java test.java metodos.java *TDA.java

```java
package ARBOLBINARIO;

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

    public void setIzq(Nodito izq) {
        this.izq = izq;
    }

    public Nodito getDer() {
        return der;
    }

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

Se creo otro paquete llamado EntradaSalida este contiene una clase llamada menu esta es donde mandamos a llamar a los metodos y se crea codigo con ayuda de los tools para solicitar dato al usuario

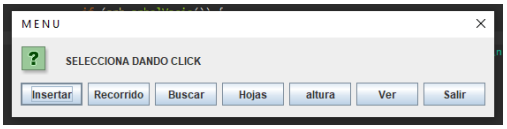
```java
package EntradaSalida;
import java.util.Scanner;
public class Menu {

    public static void main(String[] args) {
        menu();
    }

public static void menu() {


    ArbolBin<Integer> arb ;
    arb = new ArbolBin<>();
    String op;

        do {
            op=toolsList.boton("Insertar,Recorrido,Buscar,Hojas,altura,Ver,Salir");

            switch (op) {
            case "Insertar":

                arb.insertarArbol(toolsList.leerInt("Dato"));

                System.out.println("Arbol: \n");
                arb.imprimirArbol(arb.getRaiz(), " ", false);
                break;
            case "Recorrido":

                toolsList.imprimePantalla("Preorden"+arb.preorden(arb.getRaiz())+"\n in orden IZQ a DER"+arb.inOrden(arb.getRaiz())+
                        "\n en orden DER a IZQ"+arb.enOrden(arb.getRaiz())+"\n pos orden"+arb.posOrden(arb.getRaiz()));
                break;
```

```java
            case "Buscar":
                if(arb.arbolVacio()) {
                    toolsList.imprimeErrorMsg("vacio");
                } else {
                    int datoBuscar = toolsList.leerInt("Ingresa valor a buscar");
                    Nodito  r= arb.buscarDato(arb.getRaiz(), datoBuscar);

                    if(r!=null) {
                        toolsList.imprimeMsg("Si existe el dato\n"+ datoBuscar);

                    }else {
                        toolsList.imprimeErrorMsg("No existe el dato");
                    }
                }
                break;
            case "Imprimir":
                if(arb.arbolVacio()) {
                    toolsList.imprimeErrorMsg("vacio");
                } else {
                    System.out.println("La estructura del arbol es la sig: \n");
                    arb.imprimirArbol(arb.getRaiz(), "", false);

                }
                break;

           case "altura":
               if (arb.arbolVacio()) {
                    toolsList.imprimeErrorMsg("Arbol vacio");
                } else {
                    toolsList.imprimePantalla("La altura del arbol es: \n" + arb.obtenerAltura(arb.getRaiz()));
                }
                break;
```

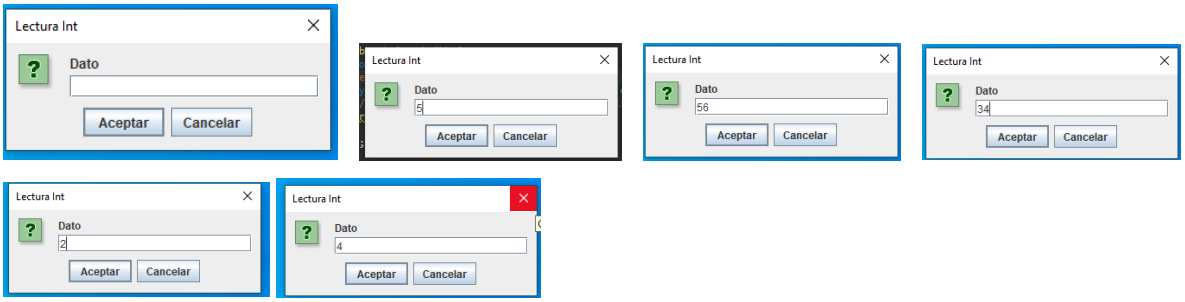
```
68          case "Ver":
69              if (arb.arbolVacio()) {
70                  toolsList.imprimeErrorMsg("Arbol vacio");
71              } else {
72                  System.out.println("La estructura del arbol es la siguiente: \n");
73                  // arb.imprimirArbol(arb.getRaiz(), " ", false);
74                  arb.printRecursive(arb.getRaiz(), 0);
75              }
76              break;
77
78          case "Hojas":
79              if (arb.arbolVacio()) {
80                  toolsList.imprimeErrorMsg("Arbol vacio");
81              } else {
82                  toolsList.imprimePantalla("Las hojas del arbol son: \n" + arb.imprimirHojas(arb.getRaiz()));
83                  toolsList.imprimePantalla("Los interiores del arbol son: \n" + arb.imprimirInteriores(arb.getRaiz()));
84
85              }
86              break;
87
88           default:
89              toolsList.imprimePantalla("Saliendo...");
90          }
91      } while (!op.equalsIgnoreCase("Salir"));
92      }
93  }
94
```

Bueno nuestro programa funciona asi

Se les mostrara un menu



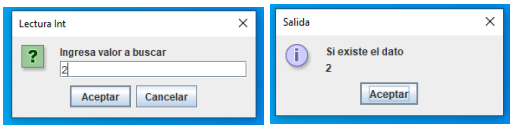
Debes seleccionar insertar le pedimos al usuario que inserte los valores que quiere para el arbol



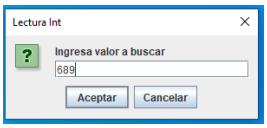
Ya que le metimos datos nos vamos a que nos muestre recorridos y nos mostrara los datos en in-orden pre-orden pos-orden en-orden

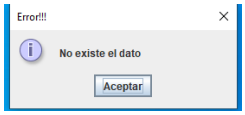
Tambien podemos buscar dato en el arbol

Le ingresaremos un valor que insertamos nosotros y nos pondara un mensaje que si existe

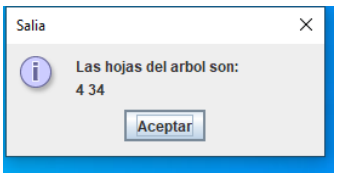


Y ahora ingresamos otro valor y nos manda un mensaje que no existe ya que este dato pues no lo ingresamos

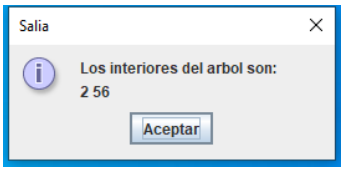




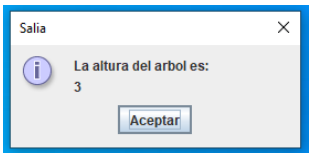
seleccionamos Mostrar hoja y si nos mustra las hojas del arbol



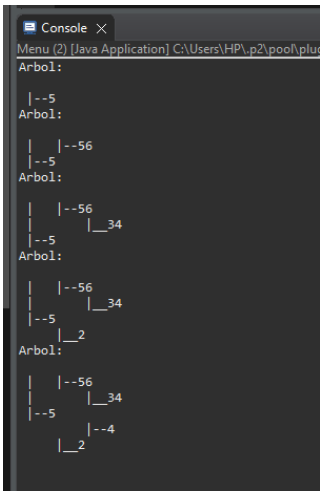
Tambien nos mustra los interiores de arbol



Y la atura del arbol



Y en la consola nos imprime el arbol



Recursos materiales y equipo.

• Computadora.

• Software IDE Eclipse For Java Developers

• Lectura de los materiales de apoyo

Conclusión

Tanto los árboles como los gráficos ofrecen un conjunto diverso de algoritmos y técnicas para realizar operaciones de búsqueda, inserción, eliminación y recorrido de datos de manera eficiente.

los árboles son una forma especializada de grafos que siguen una estructura jerárquica, mientras que los grafos son estructuras más generales que permiten representar una amplia gama de relaciones. Tanto los árboles como los grafos son herramientas esenciales en ciencias de la computación y se utilizan en numerosas aplicaciones, algoritmos y problemas de optimización.

Bibliografías

- (S/fb). www.uv.mx. Recuperado el 1 de junio de 2023, de https://www.uv.mx/personal/ermeneses/files/2021/08/Clase8-Arboles.pdf
- (S/fc). Uva.es. Recuperado el 1 de junio de 2023, de https://www.infor.uva.es/~cvaca/asigs/doceda/tema4.pdf
- (S/f). Unam.mx. Recuperado el 2 de junio de 2023, de http://fcaenlinea1.unam.mx/anexos/1566/1566_u4_anexo3.pdf