Universidad Nacional de San Agustín
Facultad de Ingeniería de Producción y Servicios
Escuela Profesional de Ingeniería de Sistemas

Curso:
Laboratorio de EDAT
Tema:
Hashing
Docente:
Edith Pamela Rivero Tupac
Alumnos:
Diego Gustavo Montana Neyra
Arequipa-Perú
2021

Matriz de Adyacencia

La clase *Vertice* tiene como atributos una lista de aristas, un dato y una etiqueta para señalar si el vértice fue sido visitado o no.

```java
public class Vertice<T> {

    protected T data;
    protected ListLinked<Arista<T>> listAdy;
    protected int label; //0=no visitado, 1=visitado

    public Vertice(T data) {
        this.data = data;
        this.listAdy = new ListLinked<Arista<T>>();
    }

    public boolean equals(Object obj) {
        if(obj instanceof Vertice<?>) {
            Vertice<T> v = (Vertice<T>) obj;
            return this.data.equals(v.data);
        }
        return false;
    }

    public String toString() {
        return this.data + "-->" + this.listAdy.toString() + "\n";
    }

}
```

La clase *Arista* tiene como atributos una referencia a un vértice de destino, un peso y una etiqueta, para señalar si una etiqueta fue descubierta o si no, o si se trata de un back o un cross.

```java
package Grafos;

public class Arista<T> {

    protected Vertice<T> refDestino;
    protected int peso;
    protected int label; //0=inexplorada, 1=descubierta, 2=back v cross

    public Arista(Vertice<T> refDestino) {
        this(refDestino,-1);
    }

    public Arista(Vertice<T> refDestino, int peso) {
        this.refDestino = refDestino;
        this.peso = peso;
    }

    public boolean equals(Object obj) {
        if(obj instanceof Vertice<?>) {
            Arista<T> v = (Arista<T>) obj;
            return this.refDestino.equals(v.refDestino);
        }
        return false;
    }

    public String toString() {
        if(this.peso > -1) return this.refDestino.data + "[" + this.peso + "], ";
        else return refDestino.data + ", ";
    }

}
```

El grafo contiene como atributos una lista de elementos de la clase *Vertice*.

El método para insertar un vértice comprueba primero que no exista un vértice igual insertado antes, recorriendo toda la lista de vértices.

```
package Grafos;

public class GraphLink<T> {

    protected ListLinked<Vertice<T>> listVertice;

    public GraphLink() {
        this.listVertice = new ListLinked<Vertice<T>>();
    }

    public void insertVertice(T data) {
        //Encapsulamos el dato en un vertice
        Vertice<T> nuevo = new Vertice<T>(data);
        if (this.listVertice.search(nuevo) != null) {
            System.out.println("Vertice ya fue insertado anteriormente...");
            return;
        }
        this.listVertice.insertToBegin(nuevo);
    }
```

El método para insertar una arista requiere de dos vértices como parámetros que deben existir previamente en el grafo. Primero se comprueba que los vértices existan, luego se comprueba que la arista a insertar no exista previamente.

Finalmente insertamos la arista en la lista de adyacentes, como se trata de un grafo no dirigido se tiene que insertar dos veces, en ambos sentidos.

```
public void insertArista(T verOrigen, T verDestino) {
    //Encapsulamos los datos en vertices
    Vertice<T> refOrigen = this.listVertice.search(new Vertice<T>(verOrigen));
    Vertice<T> refDestino = this.listVertice.search(new Vertice<T>(verDestino));

    if(refOrigen == null || refDestino == null) {
        System.out.println("Vertice origen y/o destino no existen...");
        return;
    }
    if(refOrigen.listAdy.search(new Arista<T>(refDestino)) != null) {
        System.out.println("Arista ya fue insertada anteriormente");
    }
    refOrigen.listAdy.insertToBegin(new Arista<T>(refDestino));
    refDestino.listAdy.insertToBegin(new Arista<T>(refOrigen));
}
```

Algoritmo DFS

El método DFS requiere que primero se inicialicen las etiquetas en cero, tanto de vértices como de aristas.

El método DFSRec es recursivo, y recibe como parámetro un vértice. Se recorre el arreglo de aristas, hasta hallar una que no fue visitada previamente, cuando se encuentra la arista se marca según corresponda, lo mismo con el vértice opuesto a esta arista.

```
private void initLabel() {
    Node<Vertice<T>> auxNodeV = this.listVertice.first;
    for(; auxNodeV != null; auxNodeV = auxNodeV.getNext()) {
        auxNodeV.data.label = 0;//Se inicializa vertices
        Node<Arista<T>> auxNodeAr = auxNodeV.data.listAdy.first;
        for(; auxNodeAr != null; auxNodeAr = auxNodeAr.getNext())
            auxNodeAr.data.label = 0;// Se inicializa aristas
    }
}

public void DFS(T data) {
    Vertice<T> vertV = this.listVertice.search(new Vertice<T>(data));
    if(vertV == null) {
        System.out.println("Vertice no existe para hacer DFS");
        return;
    }
    initLabel();
    DFSRec(vertV);
}

private void DFSRec(Vertice<T> vertV) {
    vertV.label = 1;//El vertice enviado se marca como visitado
    System.out.print(vertV.data + ", ");
    Node<Arista<T>> nodeAr = vertV.listAdy.first;
    for(; nodeAr != null; nodeAr = nodeAr.getNext()) {
        if(nodeAr.data.label == 0) {
            Vertice<T> vertW = nodeAr.data.refDestino;//opuesto
            if(vertW.label == 0) {
                nodeAr.data.label = 1;//La arista se marca como descubierta
                DFSRec(vertW);
            }
            else
                nodeAr.data.label = 2;
        }
    }
}
```

Algoritmo BFS

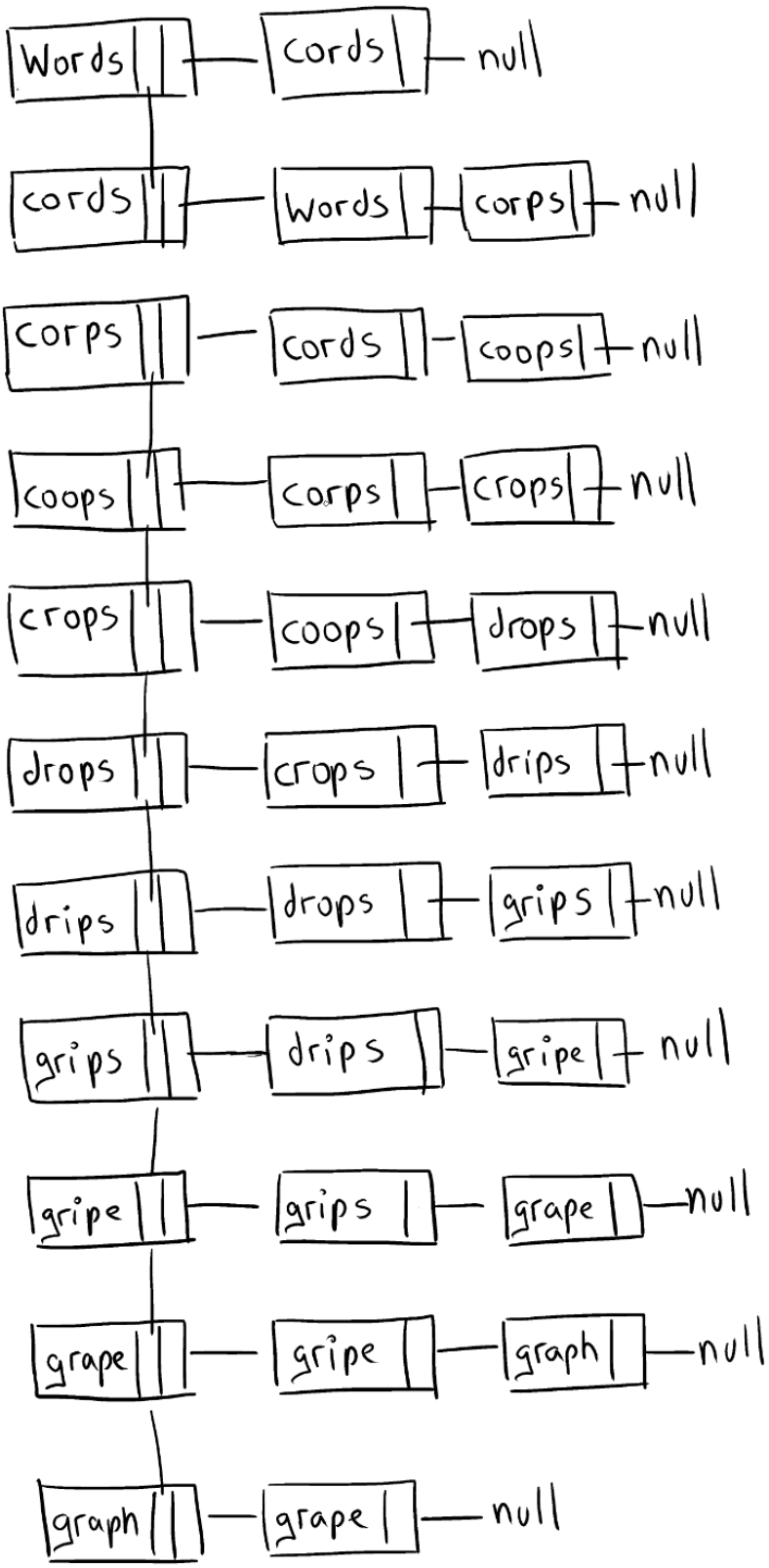
El algoritmo es similar con la diferencia de que el nodo visitado se guarda en una cola. El ciclo while recorre entre las colas creadas.

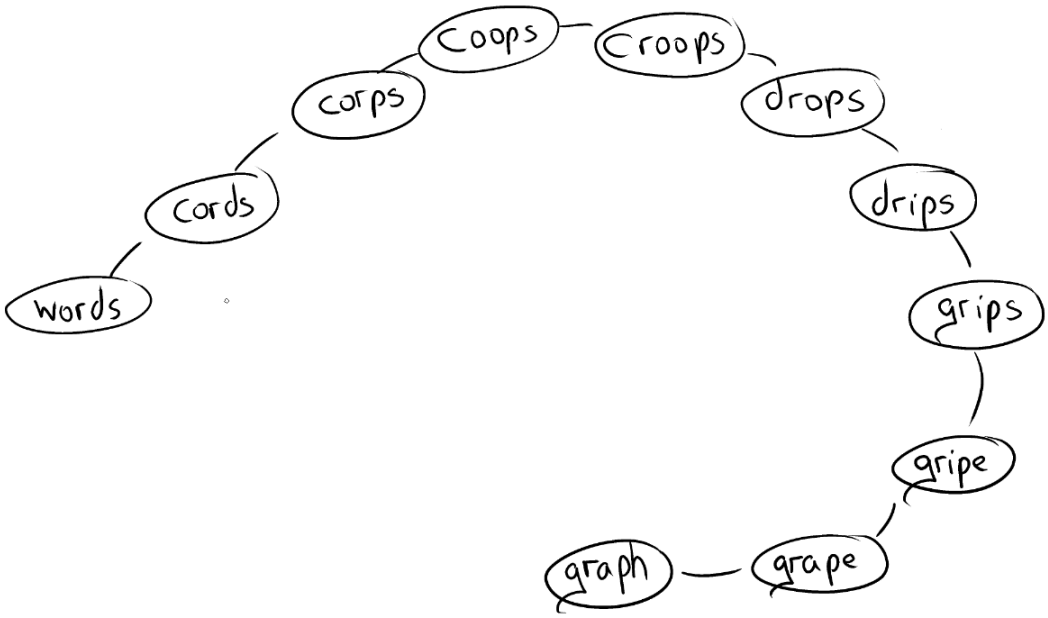
Se selecciona un vértice de la cola y se hace el recorrido por su lista de adyacencia como en el anterior caso, a diferencia de que si se encuentra una arista no visitada, el nodo se guarda en una cola diferente a la inicial.

```
public void BFS(T data) {
    Vertice<T> vertV = this.listVertice.search(new Vertice<T>(data));
    if(vertV == null) {
        System.out.println("Vertice no existe para hacer DFS");
        return;
    }
    initLabel();
    BFS(vertV);
}


//metodo incompleto
private void BFS(Vertice<T> vertV) {
    QueueLink<Vertice<T>> cola = new QueueLink<Vertice<T>>();
    cola.enqueue(vertV);
    vertV.label = 1;//El vertice enviado se marca como visitado
    System.out.print(vertV.data + ", ");

    while(!cola.isEmpty()) {//tendria que iterar entre colas
        Vertice<T> auxV = cola.dequeue();
        Node<Arista<T>> nodeAr = auxV.listAdy.first;
        for(; nodeAr != null; nodeAr = nodeAr.getNext()) {
            Vertice<T> vertW = nodeAr.data.refDestino;//opuesto
            if(vertW.label == 0) {
                nodeAr.data.label = 1;//La arista se marca como descubierta
                auxV.label = 1;
                //tendria que insertar en un nueva cola
            }
            else
                nodeAr.data.label = 2;
        }
    }
}
```

Ejercicio 5

En este caso los elementos se colocan unos tras otros en la lista.




coops
croops
corps
drops
cords
drips
words
grips
gripe
graph
grape