


| | | |
|---|---|--|
|  | Carátula para entrega de prácticas | |
| Facultad de Ingeniería | Laboratorio de docencia | |

Laboratorios de computación salas A y B

Profesor : Edgar Tista García

Asignatura: EDA 2

Grupo: 4

No. Práctica : 6-7

Integrante: Chagoya Gonzalez Leonardo

No. Equipo de Cómputo: N/A

No. de Lista o Brigada: #Lista 08

Semestre: 2022-2

Fecha de entrega: 14 de febrero del 2022

Observaciones:

CALIFICACIÓN: _____

Objetivos

Objetivo General: El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para comprender el algoritmo de búsqueda por expansión y profundidad.

Objetivo de Clase: El alumno será capaz de implementar y comprender los grafos, así como los recorridos

EJERCICIO 1

Clase Graph:

Atributos

V es un valor de tipo entero el cual indicará el número de vértices de nuestro grafo.

adjArray es un arreglo unidimensional en el cual cada uno de sus elementos será una lista ligada.

Métodos

Graph(int v) este método es un constructor en el cual recibe como parámetro un entero que será el número de vértices contenido en el grafo, posteriormente de asignar el número de vértices, se hace la instancia de nuestro arreglo de listas de tamaño v y se realiza la instancia de cada una de las listas que contendrá el arreglo.

addEdge(int v, int w) es un método que recibe como parámetros dos enteros y añade en la lista ubicada en la posición v el elemento w, para posteriormente añadir en la lista ubicada en la posición w el elemento v. Gráficamente esto se puede apreciar como la conexión de dos vértices por medio de una arista .

printGraph imprime la lista de adyacencia del grafo construido.

Clase Principal

En la clase principal se encuentra el método main en el cual se define un entero V y se inicializa con un valor de 5, posteriormente se realiza la instancia de nuestro objeto graph perteneciente a la clase Graph, pasandole como parametro el valor de V, indicando que nuestro grafo tendrá 5 vértices.

Posteriormente se añaden las aristas entre los vértices utilizando el método addEdge y se imprime el grafo construido

De esta implementación es importante mencionar que se está creando un grafo no dirigido pues no hay ninguna restricción que nos indique hacia donde apunta la arista, además de ello es un grafo no ponderado pues ninguna de las aristas (conexiones de los vértices) se le indica algún valor. Además el rango de valores numéricos que se pueden ingresar para llamar a los nodos está delimitado de 0 hasta (v-1) pues recordemos que nuestro grafo se trata de un arreglo de tamaño v que contiene listas, en donde cada índice es una lista ligada (se puede ver como un nodo) haciendo que si se ingresa un nodo con valor mayor o igual a v el programa truena.

Capturas de Ejecución

Con los elementos proporcionados

```
public static void main(String[] args) {  
    // TODO code application logic here  
    int V=5;  
    Graph graph= new Graph(V);  
    graph.addEdge(0,1);  
    graph.addEdge(0,4);  
    graph.addEdge(1,2);  
    graph.addEdge(1,3);  
    graph.addEdge(1,4);  
    graph.addEdge(2,3);  
    graph.addEdge(3,4);  
    graph.printGraph(graph);  
}
```

Lista de Adyacencia del vertice 0

0
->1
->4

Lista de Adyacencia del vertice 1

1
->0
->2
->3
->4

Lista de Adyacencia del vertice 2

2
->1
->3

Lista de Adyacencia del vertice 3

3
->1
->2
->4

Lista de Adyacencia del vertice 4

4
->0
->1
->2

Para un grafo no dirigido construido por el Usuario

*****PRACTICA 6-7*****

- 1.-Grafo no dirigido
- 2.-Grafo dirigido
- 3.-Ponderado Dirigido
- 4.-Salir

Opcion: 1

*****Grafo no dirigido*****

Construccion

Ingrese el numero de nodos que contendra el grafo: 7

Ingrese la cantidad de aristas que contendra el grafo: 8

Arista Num 1

Ingrese el nombre de un nodo (valor numerico): 0

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 1

Arista Num 2

Ingrese el nombre de un nodo (valor numerico): 0

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 4

Arista Num 3

Ingrese el nombre de un nodo (valor numerico): 2

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 3

Arista Num 4

Ingrese el nombre de un nodo (valor numerico): 3

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 4

Arista Num 5

Ingrese el nombre de un nodo (valor numerico): 4

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 5

Arista Num 6

Ingrese el nombre de un nodo (valor numerico): 4

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 1

Arista Num 7

Ingrese el nombre de un nodo (valor numerico): 1

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 5

Arista Num 8

Ingrese el nombre de un nodo (valor numerico): 1

Ingrese el nombre del nodo con quien tendra conexion por medio de una arista: 6

Lista de Adyacencia del grafo construido

Lista de Adyacencia del vertice 0

0
->1
->4

Lista de Adyacencia del vertice 1

1
->0
->4
->5
->6

Lista de Adyacencia del vertice 2

2
->3

Lista de Adyacencia del vertice 3

3
->2
->4

Lista de Adyacencia del vertice 4

4
->0
->3
->5
->1

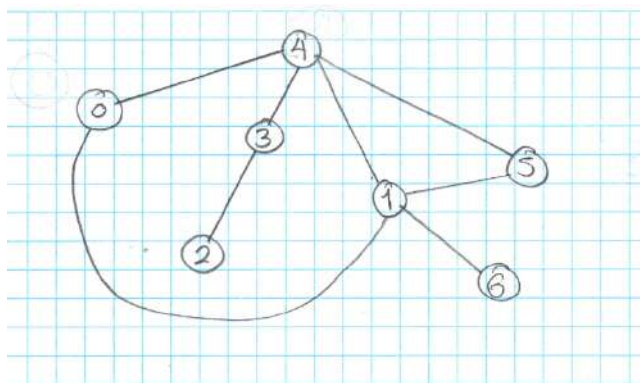
Lista de Adyacencia del vertice 5

5
->4
->1

Lista de Adyacencia del vertice 6

6
->1

Gráficamente el grafo construido se ve de la siguiente forma



EJERCICIO 2

Para la implementación de un grafo dirigido en la clase graph se agrego el método addEdgeDireccion(int inicio, int destino) basándose en el método addEdge de la misma clase este nuevo método recibe dos enteros, el primero indicará el nodo sobre el cual sale la arista y el segundo nodo será el nodo que recibe la arista. La dificultad de este ejercicio estuvo en imaginar cómo implementar el pensamiento de nodo inicio y nodo destino, para realizar esto únicamente se agrega en la lista ubicada en la posición dado por inicio dentro del arreglo, el elemento destino.

```
void addEdgeDireccion(int inicio, int destino) {  
    adjArray[inicio].add(destino);  
}
```

adjArray[inicio] dará una lista como resultado refiriéndose al nodo del que parte la arista y una vez ubicado en esa lista agrega un valor numérico refiriéndose al nodo que le llega la arista.

A diferencia del método addEdge que agrega la relación de la arista de forma mutua para los nodos.

```
void addEdge(int v, int w) {  
    adjArray[v].add(w); //en la lista que se encuentra en la posicion v añade el elemento w  
    adjArray[w].add(v); //en la lista que se encuentra en la posicion w añade el elemento v  
}
```

Para la impresión de la lista de adyacencia del grafo construido se reutilizó el código del método printGraph que fue proporcionado en la práctica.

Capturas de Ejecución

*****PRACTICA 6-7*****

- 1.-Grafo no dirigido
- 2.-Grafo dirigido
- 3.-Ponderado Dirigido
- 4.-Salir

Opcion: 2

*****Grafo Dirigido*****

Construccion

Ingrese el numero de nodos que contendra el grafo: 5

Ingrese la cantidad de aristas que contendra el grafo: 8

Arista Num 1

Ingrese el nodo del que parte la arista: 0

Ingrese el nodo destino: 3

Arista Num 2

Ingrese el nodo del que parte la arista: 3

Ingrese el nodo destino: 3

Arista Num 3

Ingrese el nodo del que parte la arista: 2

Ingrese el nodo destino: 0

Arista Num 4

Ingrese el nodo del que parte la arista: 1

Ingrese el nodo destino: 0

Arista Num 5

Ingrese el nodo del que parte la arista: 1

Ingrese el nodo destino: 2

Arista Num 6

Ingrese el nodo del que parte la arista: 4

Ingrese el nodo destino: 1

Arista Num 7

Ingrese el nodo del que parte la arista: 2

Ingrese el nodo destino: 4

Arista Num 8

Ingrese el nodo del que parte la arista: 4

Ingrese el nodo destino: 2

Lista de Adyacencia del Grafo Construido

Lista de Adyacencia del vertice 0

0
->3

Lista de Adyacencia del vertice 1

1
->0
->2

Lista de Adyacencia del vertice 2

2
->0
->4

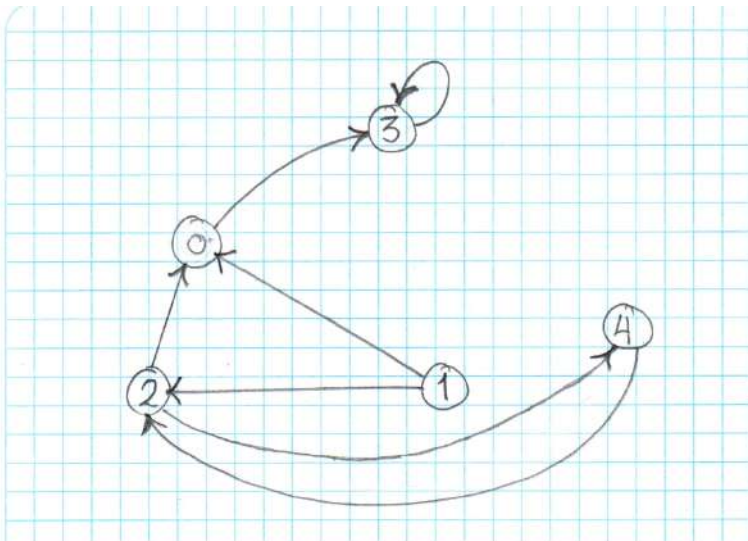
Lista de Adyacencia del vertice 3

3
->3

Lista de Adyacencia del vertice 4

4
->1
->2

El grafo construido se puede ver de la siguiente manera



EJERCICIO 3

Para la realización de este ejercicio se utilizó un grafo dirigido ya que el enunciado solicitaba que al momento de construir el grafo el usuario indicará el nodo origen y el nodo destino. Al tratarse de un grafo ponderado las conexiones entre nodos deben de tener un valor, dicho valor también será cargado por el usuario.

La dificultad que se presentó estuvo en implementar la idea de colocar a cada arista un valor, para solucionar esto se utilizó una especie de arreglos de listas paralelas, utilizando dos arreglos de listas a cada elemento del arreglo le corresponderá una lista, su posición de la lista indicará el nodo de salida y en una lista contendrá los nodos a los cuales se les hace llegar la arista mientras que en la otra representará el valor de dicha arista.

Para implementar los puntos anteriores se utilizó una nueva clase llamada GraphPonderado en esta clase se tienen 3 atributos y 3 métodos:

Atributos

V- es un entero que indicará el número de nodos, vértices que contendrá nuestro grafo.

addArray- es un arreglo unidimensional en el cual cada uno de sus elementos será una lista ligada que representa a un nodo.

valores- es un arreglo de listas en donde la posición del arreglo estará relacionada con el nodo de add array y en la lista obtenida en dicha posición se encontrarán los valores para las aristas que llegan de ese nodo.

Métodos

GraphPonderado(int v) es un método constructor el cual recibe un entero que indicará el número de nodos del grafo se hacen las instancias para los arreglos de addArray y valores e inmediatamente se hacen las instancias para cada uno de los elementos de dichos arreglos (instancia listas en cada uno de estos arreglos).

addEdgeDireccion(int inicio, int destino, int valor) es un método que recibe tres parámetros el nodo del que sale la arista el nodo destino y el valor que contendrá dicha arista. Para realizar la relación de los nodos es igual que en grafos dirigidos y

tomando la idea de arreglo de listas paralelas el valor de dicha arista debe encontrarse en la misma posición solo que en nuestra lista valores.

```
void addEdgeDireccion(int inicio, int destino, int valor){  
    adjArray[inicio].add(destino);  
    valores[inicio].add(valor);  
}
```

printGraph(GraphPonderado graph) es un método que recibe como parámetro un objeto de la clase GraphPonderado en dicho método se recorre cada uno de los elementos del arreglo e imprime el contenido de las listas ubicados en ellos tanto para adjArray como para valores esto mediante dos ciclos for anidados, el primero se ubicará en el índice del arreglo y el segundo se encargará de recorrer la lista ubicada en dicho índice.

Ejecución

*****PRACTICA 6-7*****

- 1.-Grafo no dirigido
- 2.-Grafo dirigido
- 3.-Ponderado Dirigido
- 4.-Salir

Opcion: 3

*****GRAFO PONDERADO DIRIGIDO*****

Construccion

Ingrese el numero de nodos que contendra el grafo: 5

Ingrese la cantidad de aristas que contendra el grafo: 8

Arista Num 1

Ingrese el nodo del que parte la arista: 0

Ingrese el nodo destino: 3

Ingrese el valor para esta arista: 10

Arista Num 2

Ingrese el nodo del que parte la arista: 3

Ingrese el nodo destino: 3

Ingrese el valor para esta arista: 20

Arista Num 3

Ingrese el nodo del que parte la arista: 2

Ingrese el nodo destino: 0

Ingrese el valor para esta arista: 30

Arista Num 4

Ingrese el nodo del que parte la arista: 1

Ingrese el nodo destino: 0

Ingrese el valor para esta arista: 40

Arista Num 5

Ingrese el nodo del que parte la arista: 1

Ingrese el nodo destino: 2

Ingrese el valor para esta arista: 50

Arista Num 6

Ingrese el nodo del que parte la arista: 4

Ingrese el nodo destino: 1

Ingrese el valor para esta arista: 60

Arista Num 7

Ingrese el nodo del que parte la arista: 2

Ingrese el nodo destino: 4

Ingrese el valor para esta arista: 70

Arista Num 8

Ingrese el nodo del que parte la arista: 4

Ingrese el nodo destino: 2

Ingrese el valor para esta arista: 80

Lista de Adyacencia del vertice 0
->3 valor de la arista 10

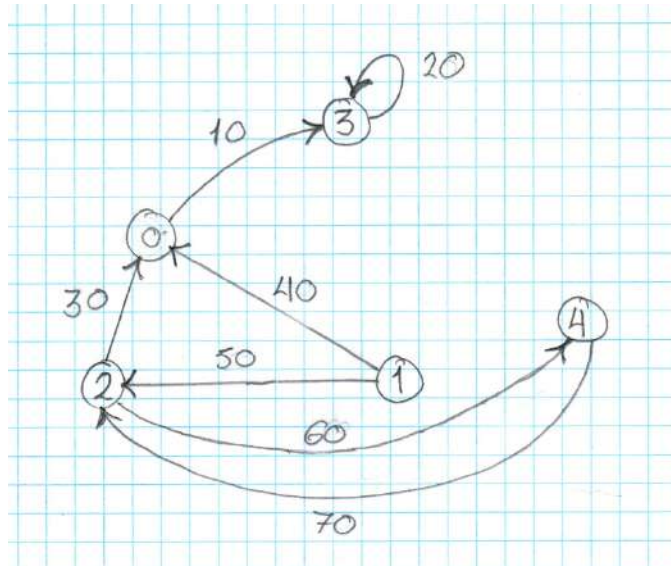
Lista de Adyacencia del vertice 1
->0 valor de la arista 40
->2 valor de la arista 50

Lista de Adyacencia del vertice 2
->0 valor de la arista 30
->4 valor de la arista 70

Lista de Adyacencia del vertice 3
->3 valor de la arista 20

Lista de Adyacencia del vertice 4
->1 valor de la arista 60
->2 valor de la arista 80

El nodo construido se puede ver de la siguiente forma.



Ejercicio 4

Para el algoritmo de BFS se utiliza el método llamado BFS al cual se le pasa como parámetro un entero indicando que este será el nodo inicial, dentro del método se instancia un arreglo de tipo booleano del tamaño de nodos, es importante mencionar que el valor del índice en el arreglo corresponde al nombre del nodo, en el grafo llamado *visited* este arreglo nos ayudará para saber cuáles nodos ya se encuentran marcados (como se vio en el video de clases) en caso de que el nodo ya esté marcado tendrá un true, en caso contrario tendrá un false en la posición correspondiente. De igual forma se instancia una lista ligada de enteros que contendrá todos aquellos nodos sobre los cuales se va realizando la búsqueda.

En un inicio el método marca como visitado el nodo inicial pasado como parámetro y agrega dicho nodo a la queue utiliza un iterador sobre el arreglo de listas que representa el grafo de tal forma que recorre el grafo revisando los nodos adyacentes al nodo en turno y agregandolos y marcandolos como visitados a las respectivas estructuras, tal y como se vio en clase, al final solo se muestra la forma en como fueron visitados cada uno de los nodos pero dicho orden jamás es guardado en una estructura este es el único cambio con respecto a lo visto en clase ya que en la clase utilizamos una lista de nodos visitados mientras que aquí es un arreglo de booleanos para indicar que nodo ya ha sido visitado

Capturas de Ejecución

Se construye un grafo no dirigido

```
//Para ejercicio 4
graph = new Graph(9);
graph.addEdge(0,1);
graph.addEdge(0,2);
graph.addEdge(0,3);
graph.addEdge(1,4);
graph.addEdge(1,5);
graph.addEdge(2,6);
graph.addEdge(3,8);
graph.addEdge(4,7);
graph.addEdge(5,7);
graph.printGraph(graph);
```

Lista de Adyacencia del vertice 0

0
->1
->2
->3

Lista de Adyacencia del vertice 1

1
->0
->4
->5

Lista de Adyacencia del vertice 2

2
->0
->6

Lista de Adyacencia del vertice 3

3
->0
->8

Lista de Adyacencia del vertice 4

4
->1
->7

Lista de Adyacencia del vertice 5

5
->1
->7

Lista de Adyacencia del vertice 6

6
->2

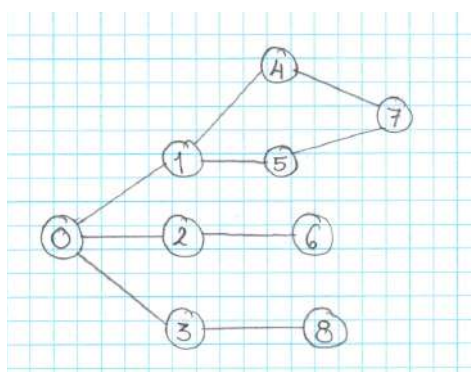
Lista de Adyacencia del vertice 7

7
->4
->5

Lista de Adyacencia del vertice 8

8
->3

Se puede ver de la siguiente manera



Y aplicando los recorridos en dos nodos distintos tenemos los siguientes resultados donde se verifica que cada uno de los nodos fueron visitados.

```
Recorrido de BFS iniciando en nodo 0 para un grafo no dirigido
0 1 2 3 4 5 6 8 7
Recorrido de BFS iniciando en nodo 8 para un grafo no dirigido
8 3 0 1 2 4 5 6 7 BUILD SUCCESSFUL (total time: 0 seconds)
```

Si ahora el grafo construido previamente lo convertimos como un grafo dirigido los recorridos no siempre tendrán que cumplirse pues no será posible en determinados casos acceder a un nodo. Por ejemplo

Se construye el siguiente grafo dirigido

```
// new Graph
graph = new Graph(9);
graph.addEdgeDireccion(0,1);
graph.addEdgeDireccion(0,2);
graph.addEdgeDireccion(0,3);
graph.addEdgeDireccion(1,4);
graph.addEdgeDireccion(1,5);
graph.addEdgeDireccion(2,6);
graph.addEdgeDireccion(3,8);
graph.addEdgeDireccion(4,7);
graph.addEdgeDireccion(5,7);
graph.printGraph(graph);
```

Lista de Adyacencia del vertice 0

```
0
->1
->2
->3
```

Lista de Adyacencia del vertice 1

```
1
->4
->5
```

Lista de Adyacencia del vertice 2

```
2
->6
```

Lista de Adyacencia del vertice 3

```
3
->8
```

Lista de Adyacencia del vertice 4

```
4
->7
```

Lista de Adyacencia del vertice 5

```
5
->7
```

Lista de Adyacencia del vertice 6

```
6
```

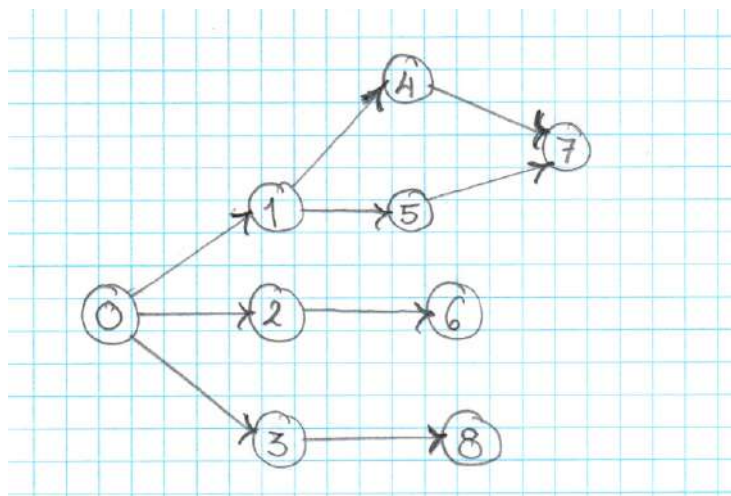
Lista de Adyacencia del vertice 7

```
7
```

Lista de Adyacencia del vertice 8

```
8
```

Gráficamente el grafo se puede representar de la siguiente manera.



Como se puede observar si se iniciara el recorrido desde el nodo 0 sería posible acceder a todos los elementos de dicho grafo por la forma en cómo se encuentran las aristas, sin embargo intentando recorrer el grafo partiendo desde cualquier otro nodo es imposible recorrer el grafo, poniendo a prueba el algoritmo de recorrido BFS se arrojan los siguientes resultados que confirman lo dicho.

```
Recorrido de BFS iniciando en nodo 0 para un grafo no dirigido  
0 1 2 3 4 5 6 8 7
```

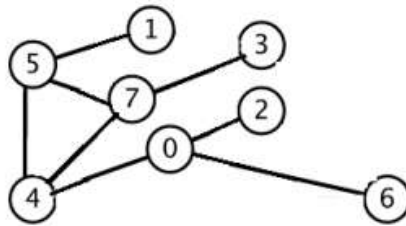
```
Recorrido de BFS iniciando en nodo 8 para un grafo no dirigido  
8
```

```
Recorrido de BFS iniciando en nodo 2 para un grafo no dirigido  
2 6
```

```
Recorrido de BFS iniciando en nodo 5 para un grafo no dirigido  
5 7 BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejercicio 5

Crea una clase principal con el siguiente grafo



Aplicando DFS en 3 diferentes nodos iniciales

Nodo inicial 1

DFS
Nodo Inicial 1 visitados {1, 5}

| |
|---|
| 1 |
|---|

DFS sobre 5 visitados {1, 5}

| | |
|---|---|
| 1 | 5 |
|---|---|

DFS sobre 4 visitados {1, 5, 4}

| | | |
|---|---|---|
| 1 | 5 | 4 |
|---|---|---|

DFS sobre 0 visitados {1, 5, 4, 0}

| | | | |
|---|---|---|---|
| 1 | 5 | 4 | 0 |
|---|---|---|---|

DFS sobre 2 visitados {1, 5, 4, 0}

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 4 | 0 | 2 |
|---|---|---|---|---|

No hay mas adyacentes

| | | | |
|---|---|---|---|
| 1 | 5 | 4 | 0 |
|---|---|---|---|

DFS sobre 6 visitados {1, 5, 4, 0, 6}

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 4 | 0 | 6 |
|---|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | 5 | 4 | 0 |
|---|---|---|---|

no hay mas nodos adyacentes

| | | |
|---|---|---|
| 1 | 5 | 4 |
|---|---|---|

no hay mas nodos adyacentes

Dfs sobre 7

1 | 5 | 4 | 7

visitados {1, 5, 4, 0, 6, 7}

Dfs sobre 3

1 | 5 | 4 | 7 | 3

visitados {1, 5, 4, 0, 6, 7, 3}

1 | 5 | 4 | 7

no hay mas nodos adyacentes

1 | 5 | 4

no hay mas nodos adyacentes

1 | 5

no hay mas nodos adyacentes

1

no hay mas "

visitados {1, 5, 4, 0, 6, 7}

Nodo Inicial 6

Nodo inicial 6

6

visitados {6}

Dfs sobre 0

6 | 0

visitados {6, 0}

Dfs sobre 2

6 | 0 | 2

visitados {6, 0, 2}

6 | 0 |

visitados {6, 0, 2}

Dfs sobre 4

6 | 0 | 4

visitados {6, 0, 4}

Dfs sobre 5

6 | 0 | 4 | 5

visitados { 6, 0, 4, 5 }

Dfs sobre 1

6 | 0 | 4 | 5 | 1

visitados { 6, 0, 4, 5, 1 }

6 | 0 | 4 | 5

Dfs sobre 7

6 | 0 | 4 | 5 | 7

visitados { 6, 0, 4, 5, 1, 7 }

Dfs sobre 3

6 | 0 | 4 | 5 | 7 | 3

visitados { 6, 0, 4, 5, 1, 7, 3 }

6 | 0 | 4 | 5 | 7

6 | 0 | 4 | 5

6 | 0 | 4

6 | 0

6

visitados { 6, 0, 4, 5, 1, 7, 3 }

Nodo Inicial 4

Nodo inicial 4

4

visitados { 4, }

Dfs sobre 0

4 | 0

visitados { 4, 0 }

Dfs sobre 2

4 | 0 | 2

visitados { 4, 0, 2 }

4 | 0

Dfs sobre 6

4 | 0 | 6

visitados {4, 0, 2, 6}

4 | 0

4

Dfs sobre 5

4 | 5

visitados {4, 0, 2, 6, 5}

Dfs sobre 1

4 | 5 | 1

visitados {4, 0, 2, 6, 5, 1}

4 | 5

Dfs sobre 7

4 | 5 | 7

visitados {4, 0, 2, 6, 5, 1, 7}

Dfs sobre 3

4 | 5 | 7 | 3

visitados {4, 0, 2, 6, 5, 1, 7}

4 | 5 | 7

4 | 5

4

visitados {4, 0, 2, 6, 5, 1, 7}

Construcción del Grafo

Lista de Adyacencia del vertice 0

0

->4

->2

->6

Lista de Adyacencia del vertice 1

1

->5

Lista de Adyacencia del vertice 2

2

->0

Lista de Adyacencia del vertice 3

3

->7

Lista de Adyacencia del vertice 4

4

->5

->0

Lista de Adyacencia del vertice 5

5

->1

->4

->7

Lista de Adyacencia del vertice 6

6

->0

Lista de Adyacencia del vertice 7

7

->5

```
Aplicando DFS con el nodo inicial en 1: 1 5 4 0 2 6 7 3
Aplicando DFS con el nodo inicial en 4: 4 5 1 7 3 0 2 6
Aplicando DFS con el nodo inicial en 6: 6 0 4 5 1 7 3 2
```

A simple vista el algoritmo proporcionado en esta práctica puede parecer idéntico al algoritmo visto en clase sin embargo las salidas de recorrido son diferentes en cada caso esto se debe a que el algoritmo no contempla una prioridad de nodos dándole mayor prioridad al nodo que contenga un valor menor es allí donde radica la mayor diferencia de ambos algoritmos.

Conclusiones

Durante la realización de los ejercicios se pusieron en práctica los conceptos de grafos y recorridos sobre ellos. Se considera que cada uno de los ejercicios se completaron de forma satisfactoria. La principal y primer dificultad que presente fue el imaginar la implementación de grafos ya que para estructuras vistas con anterioridad Java tenía sus propias implementaciones en clases y aquí fueron vistos a través de una lista de Adyacencia, sin embargo me gusto de igual forma porque permite emplear nuestro pensamiento lógico al darnos algoritmos de recorridos y verificar su funcionamiento reforzando los contenidos vistos en clase.