Penjelasan BFS :

```java
import java.util.LinkedList;
import java.util.Queue;

public class BFSTraversal {
    private int nodes;
    private LinkedList<Integer> adj[];
    private Queue<Integer> vertexQueue;

    @SuppressWarnings("unchecked")
    public BFSTraversal(int vertex) {
        nodes = vertex;
        adj = new LinkedList[nodes];

        for (int i = 0; i < vertex; i++) {
            adj[i] = new LinkedList<>();
        }

        vertexQueue = new LinkedList<Integer>();
    }

    public void addEdge(int source, int dest) {
        adj[source].add(dest);
    }

    public void getAdj() {
        for (int i = 0; i < nodes; i++) {
            if (adj[i].size() > 0) {
                System.out.print(convert(i) + " --> ");
                for (int j = 0; j < adj[i].size(); j++) {
                    System.out.print(convert(adj[i].get(j)) + " ");
                }
                System.out.println();
            }
        }
    }
```

1. Import Libraries: Kode mengimpor kelas LinkedList dan Queue dari paket java.util. Dua struktur data ini akan digunakan untuk menyimpan informasi tentang graf dan antrian yang dibutuhkan dalam algoritma BFS.

2. Variabel Kelas:
- nodes: Menyimpan jumlah simpul dalam graf.
- adj: Menyimpan daftar adjasensi, yaitu daftar node-node yang terhubung dengan setiap node.
- vertexQueue: Antrian simpul yang akan diproses dalam algoritma BFS.

3. Konstruktor (BFSTraversal(int vertex)):
- Menginisialisasi nodes dengan jumlah simpul yang diberikan.
- Menginisialisasi adj sebagai array dari LinkedList, yang mewakili daftar adjasensi setiap simpul.
- Menginisialisasi vertexQueue sebagai LinkedList kosong untuk menyimpan simpul yang akan diproses.

4. Metode addEdge(int source, int dest):
- Menambahkan edge antara simpul source dan dest dengan menambahkan dest ke daftar adjasensi dari simpul source.

5. Metode getAdj():
- Mengiterasi melalui semua simpul dan mencetak daftar adjasensi masing-masing simpul dalam format yang terbaca.
  Kode ini adalah implementasi dari algoritma Breadth-First Search (BFS) dalam bahasa pemrograman Java. Berikut adalah penjelasan untuk setiap bagian dari kode:

```
37
38      public String convert(int s) {
39          String[] alphabet = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
40          return alphabet[s];
41      }
42
43
44      public void bfs(int source) {
45          boolean traversalOrder[] = new boolean[nodes];
46
47          int visitedVertex = 0;
48          traversalOrder[source] = true;
49          vertexQueue.add(source);
50
51
52          while (!vertexQueue.isEmpty()) {
53              visitedVertex = vertexQueue.poll();
54              System.out.print(convert(visitedVertex) + " ");
55
56              for (int tetangga : adj[visitedVertex]) {
57                  if (!traversalOrder[tetangga]) {
58                      traversalOrder[tetangga] = true;
59                      vertexQueue.add(tetangga);
60                  }
61              }
62          }
63      }
64
65  }
```

6. Method `convert(int s)` : Method ini menerima sebuah integer `s` dan mengembalikan karakter dalam array alphabet yang memiliki indeks sesuai dengan nilai `s` . Jadi, jika `s` adalah 0, maka akan mengembalikan "A", jika 1 maka "B", dan seterusnya.

7. Method `bfs(int source)` : Method ini adalah implementasi dari algoritma BFS. Ia menerima parameter integer `source` yang merupakan simpul awal dari graf yang akan di-traverse.

- traversalOrder[] : Array boolean yang digunakan untuk menyimpan status traversal dari setiap simpul. Jika suatu simpul sudah di-traverse, maka nilainya akan diubah menjadi true.

- visitedVertex : Variabel integer yang digunakan untuk menyimpan simpul yang sedang di-traverse pada iterasi saat ini.

- vertexQueue : Antrian simpul yang akan di-traverse. Dimulai dengan menambahkan simpul awal source ke dalam antrian.

- Selama antrian simpul tidak kosong (!vertexQueue.isEmpty()), lakukan langkah-langkah berikut:

- Ambil simpul pertama dari antrian (`visitedVertex = vertexQueue.poll()).

- Cetak karakter yang mewakili simpul yang sedang di-traverse menggunakan method `convert(visitedVertex)`.

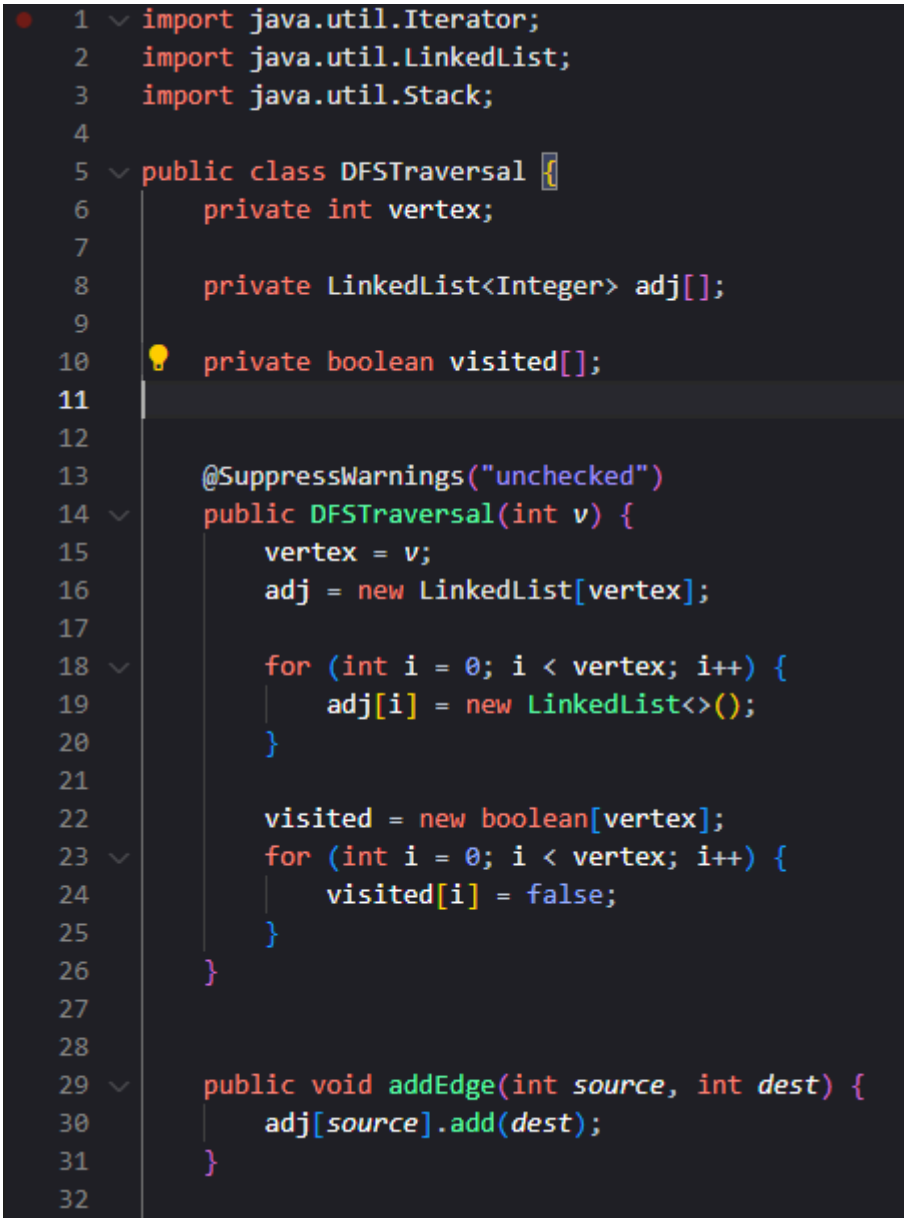
- Iterasi melalui semua tetangga dari simpul yang sedang di-traverse (for (int tetangga : adj[visitedVertex])). Adj merupakan representasi dari graf yang berisi daftar tetangga dari setiap simpul.

- Jika tetangga tersebut belum pernah di-traverse (!traversalOrder[tetangga]), maka tandai simpul tersebut sebagai telah di-traverse (traversalOrder[tetangga] = true) dan tambahkan simpul tersebut ke dalam antrian (vertexQueue.add(tetangga)).

- Algoritma berakhir ketika semua simpul yang dapat dijangkau dari source telah di-traverse.

- Kode yang diberikan mungkin merupakan bagian dari sebuah kelas yang belum sepenuhnya ditampilkan di sini. Terdapat beberapa variabel seperti nodes dan adj yang digunakan dalam method bfs(int source), namun deklarasi dan inisialisasi mereka tidak terlihat dalam potongan kode yang diberikan.

Penjelasan DFS :



```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Stack;

public class DFSTraversal {
    private int vertex;

    private LinkedList<Integer> adj[];

    private boolean visited[];


    @SuppressWarnings("unchecked")
    public DFSTraversal(int v) {
        vertex = v;
        adj = new LinkedList[vertex];

        for (int i = 0; i < vertex; i++) {
            adj[i] = new LinkedList<>();
        }

        visited = new boolean[vertex];
        for (int i = 0; i < vertex; i++) {
            visited[i] = false;
        }
    }


    public void addEdge(int source, int dest) {
        adj[source].add(dest);
    }

```

Ini adalah kelas Java yang digunakan untuk melakukan penelusuran DFS (Depth First Search) pada sebuah grafik. Mari kita jelaskan bagian-bagian utamanya:

1. Variabel dan Struktur Data:

private int vertex; : Variabel yang menyimpan jumlah simpul (vertex) dalam graf. private LinkedList<Integer> adj[];: Array dari LinkedList, di mana setiap elemen array merupakan daftar adjacency (daftar simpul terhubung) dari sebuah simpul dalam graf. Ini digunakan untuk menyimpan daftar tetangga dari setiap simpul.

Private boolean visited[];: Array boolean yang digunakan untuk melacak apakah sebuah simpul telah dikunjungi selama penelusuran DFS.

2. Konstruktor :

public DFSTraversal (int v): Konstruktor untuk kelas DFSTraversal, menerima parameter v yang merupakan jumlah simpul dalam graf. Di dalam konstruktor ini, array adj diinisialisasi untuk menyimpan daftar adjacency untuk setiap simpul, dan array visited diinisialisasi untuk melacak apakah simpul tersebut telah dikunjungi atau belum.

3. Metode addEdge:

public void addEdge(int source, int dest): Metode ini digunakan untuk menambahkan edge (sambungan) antara dua simpul. Ini dilakukan dengan menambahkan simpul tujuan (dest) ke dalam daftar adjacency dari simpul sumber (source).

```
34      public String convert(int s) {
35          String alphabet[] = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
36          return alphabet[s];
37      }
38
39
40      public void dfs(int source) {
41
42          Stack<Integer> stack = new Stack<>();
43
44          stack.push(source);
45
46          while (stack.empty() == false) {
47              source = stack.peek();
48              stack.pop();
49
50              if (visited[source] == false) {
51                  System.out.print(convert(source) + " ");
52                  visited[source] = true;
53              }
54
55              Iterator<Integer> it = adj[source].iterator();
56
57              while (it.hasNext()) {
58                  int v = it.next();
59                  if (!visited[v]) {
60                      stack.push(v);
61                  }
62              }
63          }
64      }
65
66  }
67
```

Method convert(int s): Ini adalah method yang mengonversi indeks integer menjadi huruf berdasarkan indeks tersebut. Method ini mengambil indeks s sebagai argumen dan mengembalikan huruf sesuai dengan indeks tersebut dari array alphabet.
Method dfs(int source): Ini adalah method utama untuk melakukan pencarian DFS. Method ini mengambil node awal source sebagai argumen.
Baris 5-7: Membuat sebuah stack kosong untuk menyimpan node yang akan diperiksa.
Baris 9: Menambahkan node awal ke dalam stack.
Baris 11-25: Memulai loop selama stack tidak kosong. Di dalam loop ini:
Baris 12-14: Mengambil node teratas dari stack dan menghapusnya dari stack. Node ini akan menjadi node yang sedang diperiksa.

Baris 16-19: Jika node yang sedang diperiksa belum pernah dikunjungi sebelumnya (ditandai dengan visited[source] == false), maka cetak huruf yang sesuai dengan node tersebut dan tandai node tersebut sebagai sudah dikunjungi.
Baris 21-24: Mengiterasi melalui semua node yang terhubung dengan node yang sedang diperiksa. Jika node tersebut belum pernah dikunjungi, masukkan ke dalam stack untuk diperiksa nanti.