# MODUL VI

**GRAPH**

# TUJUAN

# Tujuan dari praktikum ini adalah:

1. Mahasiswa dapat memahami konsep *graph*.
2. Mahasiswa dapat mengimplementasikan *graph* dengan bahasa pemrograman *java*.
3. Mahasiswa mampu mengimplementasikan DFS dan BFS Transversal pada *graph*.

**6.2 DASAR TEORI**

* + 1. **Pengertian *Graph***

*Graph* adalah kumpulan *node* (simpul) di dalam bidang dua dimensi yang dihubungkan dengan sekumpulan garis (sisi). *Graph* dapat digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut. Representasi visual dari *graph* adalah dengan menyatakan objek sebagai *node*, bulatan atau titik (*vertex*), sedangkan hubungan antara objek dinyatakan dengan garis (*edge*) [1].

G = (V, E)

Dengan :

G = *Graph*

V = Simpul atau *vertex*, atau *node*, atau Titik

E = Busur atau *edge*

|  |
| --- |
| Capture |

Gambar 6.1Ilustrasi *graph*

Terdapat beberapa istilah yang berkaitan dengan *graph* yaitu [6]:

1. *Vertex*

*Vertex* adalah himpunan *node* atau titik pada sebuah *graph*.

1. *Edge*

*Edge* adalah himpunan garis yang menghubungkan tiap *node* atau *vertex*.

1. *Adjacent*

Dua buah simpul disebut *adjacent* bila ada busur yang menghubungkan kedua simpul tersebut.

1. *Incident*

Jika e merupakan busur dengan simpul-simpulnya adalah v dan w yang ditulis e = (v,w), maka v dan w disebut “terletak” pada e dan e disebut *incident* dengan v dan w.

1. *Degree*

*Degree* adalah jumlah busur yang *incident* dengan simpul.

1. *Path*

*Path* adalah serangkaian simpul-simpul yang berbeda, yang *adjacent* secara berturut-turut dari simpul satu ke simpul berikutnya.

* + 1. **Jenis-jenis *Graph***

Berikutinijenis– jenis *graph* [6]:

1. *Graph* tak berarah (*undirected graph* atau *non-directed graph*)

*Graph* tak berarah adalah *graph* dimana urutan simpul dalam sebuah busur tidak dipentingkan.

|  |
| --- |
|  |

Gambar 6.2*Graph* tak berarah

1. *Graph* berarah (*directed graph*)

*Graph* berarah adalah *graph* dimana urutan simpul dalam sebuah busur memiliki arti atau dipentingkan.

|  |
| --- |
|  |

Gambar 6.3*Graph* berarah

* + 1. **Metode Pencarian *Vertex***

Pencarian *vertex* adalah proses umum dalam *graph*. Terdapat 2 metode pencarian, yakni *Depth First Search* (DFS) dan *Breadth First Search* (BFS).

1. *Depth First Search* (DFS)

Pencarian dengan metode ini dilakukan dari *node* awal secara mendalam hingga yang paling akhir (*dead-end*) atau sampai ditemukan. Dengan kata lain, simpul cabang atau anak yang terlebih dahulu dikunjungi [5].

|  |
| --- |
| 3 |

Gambar 6.4*Depth first search*

Proses pencarian dilakukan dengan mengunjungi cabang terlebih dahulu hingga tiba di simpul terakhir. Jika tujuan yang diinginkan belum tercapai maka pencarian dilanjutkan ke cabang sebelumnya, turun ke bawah jika memang masih ada cabangnya. Begitu seterusnya hingga diperoleh tujuan akhir (*goal*). *Depth first search* memiliki kelebihan diantaranya adalah cepat mencapai kedalaman ruang pencarian [4].

1. *Breadth First Search* (BFS)

*Breadth first search* (BFS) merupakan pencarian yang dilakukan dengan mengunjungi tiap-tiap *node* secara sistematis pada setiap *level* hingga keadaan tujuan (*goal state*) ditemukan. Atau dengan kata lain, penelusuran yang dilakukan adalah dengan mengunjungi tiap-tiap *node* pada *level* yang sama hingga ditemukan *goal state*-nya [1].

|  |
| --- |
| 4 |

Gambar 6.5*Breadth first search*

Pengimplementasian BFS dapat ditelusuri dengan menggunakan daftar (*list*), *open* dan *closed*, untuk menelusuri gerakan pencarian di dalam ruang keadaan. Keuntungan menggunakan *breadth first search* ini, diantaranya adalah tidak akan menemui jalan buntu dan jika ada satu solusi maka *breadth first search* akan menemukannya, dan jika ada lebih dari satu solusi maka solusi minimum akan ditemukan [1].

# 6.3 PERMASALAHAN

1. Buatlah sebuah *graph* berdasarkan gambar berikut :

|  |
| --- |
|  |

Gambar 6.6*Graph* jurnal

Dengan ketentuan :

1. Menampilkan *adjacency list* dari *graph* pada **gambar 6.6**.
2. Menampilkan secara BFS dari suatu *vertex* tertentu.
3. Menampilkan secara DFS dari suatu *vertex* tertentu.

# 6.4 HASIL PEMBAHASAN

# Program Data Mahasiswa

# Algoritma

1. Menginisialisasi suatu objek untuk menyimpan barisan data dalam bentuk susunan graf.
2. Memasukkan data secara tetap di dalam program beserta data setelahnya dalam susunan data.
3. Menampilkan data dalam bentuk suatu daftar dimana setiap menampilkan data akan ditampilkan juga data yang berhubungan langsung atau berada setelah data tersebut.
4. Memasukkan data yang akan menjadi data pertama tampil dan ditampilkan satu per satu hingga semua tampil dengan metode *breadth first search* dengan pola menampilkan data yang dimasukkan lalu data sebelah kiri data dicari lalu data data sebelah kanan hingga data terakhir telah tampil semua.
5. Memasukkan data yang akan menjadi data pertama tampil dan ditampilkan satu per satu hingga semua data tampil dengan metode *depth first* *search* dengan pola pencarian dan menampilkan data dari data yang dimasukkan lalu menuju data sebelah kiri data masukkan hingga data paling kiri dari data masukkan dilanjutkan dengan data kanan setelah data paling kiri telah tampil.

# *Source code*

|  |
| --- |
| package JURNAL\_MODUL\_6;  import java.util.Iterator;  import java.util.LinkedList;  import java.util.Scanner;  public class Graph { static class Graf {  int Vertex;  LinkedList<Integer> adjListArray[];  Graf(int V){ this.Vertex = V; adjListArray = new LinkedList[V]; for(int i = 0; i < V ; i++){  adjListArray[i] = new LinkedList<>(); }  }  }   static void addEdge(Graf graph, int sc, int tuj) {  graph.adjListArray[sc].add(tuj);}   static void show(Graf graph){  for(int ver = 0; ver < graph.Vertex; ver++) {  System.*out*.println("Adjacency list dari vertex "+ ver);System.*out*.print("head");for(Integer pCrawl: graph.adjListArray[ver]){System.*out*.print(" -> "+pCrawl);}  System.*out*.println("\n"); }  }   static void BFS(int key, Graf grap){  boolean visited[] = new boolean[grap.Vertex]; LinkedList<Integer> queue = new LinkedList<Integer>();  visited[key]=true; queue.add(key);  while (queue.size() != 0){  key = queue.poll();  System.*out*.print(key+" "); Iterator<Integer> i = grap.adjListArray[key].listIterator();  while (i.hasNext())  {  int n = i.next();  if (!visited[n])  {  visited[n] = true;queue.add(n);}  }  }  }   static void deepth(int v,boolean visit[],Graf graph){  visit[v] = true;  System.*out*.print(v+" ");Iterator<Integer> i = graph.adjListArray[v].listIterator();  while (i.hasNext())  {  int n = i.next();  if (!visit[n])  deepth(n, visit,graph);}  }   static void DFS(int v, Graf grap{  boolean visited[] = new boolean[grap.Vertex]; deepth(v, visited,grap);}   public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  int V = 7 ; Graf gra = new Graf(V); addEdge(gra, 0, 1); addEdge(gra, 1, 2);  addEdge(gra, 1, 5);  addEdge(gra, 2, 0); addEdge(gra, 2, 3);  addEdge(gra, 3, 4);  addEdge(gra, 4, 0); addEdge(gra, 4, 6);  addEdge(gra, 5, 6);  System.*out*.println("Adjacency list");  *show*(gra); System.*out*.println("\nBFS");  System.*out*.println("\nMasukkan data awal penelusuran : ");int key = sc.nextInt();  System.*out*.println("\nHasil penelusuran : ");  *BFS*(key, gra); System.*out*.println("\nDFS"); System.*out*.println("\nMasukkan data awal penelusuran : ");  key = sc.nextInt();System.*out*.println("\nHasil penelusuran : ");  *DFS*(key, gra); } } |

# Hasil program

|  |
| --- |
|  |

# Gambar 6.7 Hasil *run* program data mahasiswa dalam menambah data

Pada **Gambar 6.7** membuktikan bahwa pada kasus diatas, merupakan program *graph* dimana ditampilkan secara *adjacency list* dan ditampilkan secara BFS (*Breadth First Search*) dan DFS (*Deepth First Search*) yang dimulai dari *vertex* 0.

# 6.5 ANALISA

**6.5.1 Program Data Mahasiswa**

|  |
| --- |
| import java.util.Iterator; |

Potongan *code* “import java.util.Iterator;” merupakan sebuah pemanggilan *method* “Iterator”. *Code* “Iterator” berfungsi untuk *mengimport iterator* pada program, *iterator* berfungsi untuk mengakses dan menampilkan nilai dari seluruh elemen pada *ArrayList*.

|  |
| --- |
| import java.util.LinkedList; |

Potongan *code* “import java.util.LinkedList;” merupakan sebuah pemanggilan variabel *method* “LinkedList”. *Code* “LinkedList” berfungsi untuk memasukan dan menyimpan sekumpulan elemen. Masing-masing elemen pada *linked list* disimpan dalam sebuah *node*.

|  |
| --- |
| import java.util.Scanner; |

Potongan *code* “import java.util.Scanner;” merupakan sebuah pemanggilan variabel *method* “Scanner”. *Code* “Scanner” berfungsi untuk memasukan data langsung melalui *keybord*. Dan juga *method* tersebut dapat memasukkan data secara dinamis.

|  |
| --- |
| public class Graph { |

Potongan *code* “public class Graph {” berfungsi untuk mendeklarasikan sebuah *class* pada *java* yang bernama “Graph” yang bersifat *public*. Fungsi *class* tersebut untuk menampung *method-method* yang digunakan di program.

|  |
| --- |
| static class Graf {  int Vertex;  LinkedList<Integer> adjListArray[];  Graf(int V){ this.Vertex = V; adjListArray = new LinkedList[V]; for(int i = 0; i < V ; i++){  adjListArray[i] = new LinkedList<>();}  }  } |

Potongan *code* “static class Graf” merupakan sebuah *static class* baru yang bernama “Graf”. Dimana di dalam *class* terdapat “int Vertex; LinkedList<Integer> adjListArray[];Graf(int V){ this.Vertex = V; adjListArray = new LinkedList[V]; for(int i = 0; i < V ; i++){ adjListArray[i] = new LinkedList<>();}}}”. *Code* “int Vertex;” merupakan variabel *vertex* dengan tipe data “int” yang berfungsi untuk menyimpan nilai dari *vertex* pada program *graph*. *Code* “LinkedList<Integer> adjListArray[];” dengan tipe data “LinkedList” isinya dengan tipe data *integer* berfungsi sebagai tipe data pada variabel *array* “adjListArray[]” yang berfungsi untuk menyimpan data dalam *array*. *Code* “Graf(int V){” berfungsi sebagai konstraktor dengan parameter variabel “V” yang bertipe data *integer* dimana di dalamanya terdapat “this.Vertex = V” berfungsi untuk menyimpan data *vertex* dimana *this* berfungsi sebagai pembeda dalam penemamaan variabel. *Code* “for(int i = 0; i < V ; i++){ adjListArray[i] = new LinkedList<>();” merupakan perulangan berfungsi untuk meneruskan data sebanyak *vertex* yang di masukkan dalam *array* yang bernama “adjListArray[i]” dengan perwakilan variabel “i” yang dimana *array* tersebut dimasukkan ke dalam *linked list* dimana nantinya *vertex* akan ditampilkan secara *adjacency list*.

|  |
| --- |
| static void addEdge(Graf graph, int sc, int tuj) {  graph.adjListArray[sc].add(tuj);} |

Potongan *code* “static void addEdge(Graf graph, int sc, int tuj){ graph.adjListArray[sc].add(tuj);}” merupakan sebuah *method* yang bernama “addEdge” yang bersifat *static void* dengan parameter variabel bernama “graph” dengan tipe data “Graf”, nilai bertipe *integer* dengan nama variabel “sc” dan bertipe data *integer* dengan nama variabel “tuj”.*Code* “ {graph.adjListArray[sc].add(tuj);}” berfungsi untuk menempatkan atau memasangkan vertex awal dengan variabel “sc” pada array dengan *vertex* tujuan dengan variabel “tuj” dengan ditambahkan dengan method “add” dimana method tersebut terdapat pada *linked list.*

|  |
| --- |
| static void show(Graf graph){  for(int ver = 0; ver < graph.Vertex; ver++) {  System.*out*.println("Adjacency list dari vertex "+ ver);System.*out*.print("head");for(Integer pCrawl: graph.adjListArray[ver]){System.*out*.print(" -> "+pCrawl);}  System.*out*.println("\n"); }  } |

Potongan *code* “static void show(Graf graph)” merupakan *method* bernama “show” bersifat “static” dan “void” dengan parameter “graph” dengan tipe data “Graf” berfungsi untuk menampilkan *graph* dari *method - method* yang ada pada *class* “Graph”. *Code* “for(int ver = 0; ver < graph.Vertex; ver++)” berfungsi untuk meneruskan data *vertex* sesuai jumlah *vertex* pada *graph.* “System.*out*.println("Adjacency list dari vertex "+ ver);System.*out*.print("head");” dimana perulangan dengan variabel “ver” mengulang menampilkan kata “Adjacency list dari vertex "+ ver” dimana pada *code* tersebut ditampilkan *vertex graph* secara *adjacency list* dengan menunjukkan “head” sebagai pada *vertex* pertama. *Code* “for(Integer pCrawl: graph.adjListArray[ver]){System.*out*.print("->"+pCrawl);}” dimana “pCrawl” berfungsi sebagai penunjuk urutan pasangan *vertex* dari awal urutan pasangan *vertex* hingga pasangan terakhir *vertex* dengan symbol “->”. *Code* “System.*out*.println("\n");}}” berfungsi sebagai membuat garis baru pada program.

|  |
| --- |
| static void BFS(int key, Graf grap){  boolean visited[] = new boolean[grap.Vertex]; LinkedList<Integer> queue = new LinkedList<Integer>();  visited[key]=true; queue.add(key);  while (queue.size() != 0){  key = queue.poll();  System.*out*.print(key+" "); Iterator<Integer> i = grap.adjListArray[key].listIterator();  while (i.hasNext())  {  int n = i.next();  if (!visited[n])  {  visited[n] = true;queue.add(n);}  }  }  } |

Potongan *code* “static void BFS(int key, Graf grap){” merupakan *method* bernama “BFS” bersifat “static” dan “void” dengan parameter “grap” dengan tipe data “Graf”berfungsi untuk menampilkan *graph* secara BFS (*Breadth First Search*). *Code* “boolean visited[] = new boolean[grap.Vertex];” berfungsi untuk menyimpan data *vertex* yang telah dikunjungi. “LinkedList<Integer> queue = new LinkedList<Integer>();” berfungsi untuk mendeklarasikan satu objek yaitu “queue” untuk memanggil *method linked list. Code* “visited[key]=true;” dimana data yang telah dikunjungi bernilai benar . *Code* “queue.add(key);” berfungsi untuk menambahkan data *vertex* pada urutan *vertex*. *Code* “while (queue.size() != 0){ key = queue.poll();System.*out*.print(key+" ");” merupakan perulangan kondisi ukuran antrian tidak kosong maka data akan dipindahkan ke dalam antrian urutan lalu menampilkannya. *Code* “Iterator<Integer> i = grap.adjListArray[key].listIterator();” berfungsi untuk membuat objek *iterator* yang bertipe data *integer* yaitu “i” untuk memodifikasi *vertex* awal dengan *listiterator*. *Code* “while (i.hasNext()){int n = i.next();if (!visited[n]){ visited[n] = true; queue.add(n);}}}” merupakan perulangan untuk mengecek kondisi apakah objek *iterator* masih mempunyai nilai pada selanjutnya atau tidak. Lalu menampilkan elemen selanjutnya dan apabila elemen yang ditampung di variabel “n” pernah dikunjungi maka ditambahkan ke dalam urutan.

|  |
| --- |
| static void deepth(int v,boolean visit[],Graf graph){  visit[v] = true;  System.*out*.print(v+" ");Iterator<Integer>i=graph.adjListArray[v].listIterator();  while (i.hasNext())  {  int n = i.next();  if (!visit[n])  *deepth*(n, visit,graph);}  } |

Potongan *code* “static void deepth(int v,boolean visit[],Graf graph){” merupakan *method* bernama “deepth” bersifat “static” dan “void” dengan parameter “v” bertipe data *integer,* “visit[]” bertipe data *boolean*  dan “grap” dengan konstraktor “Graf”berfungsi untuk menampilkan graph secara DFS (*deepth first search*). Code “visit[v] = true;” dimana data atau vertex yang telah dikunjungi bernilai benar. “System.*out*.print(v+" ");”berfungsi untuk menampilkan *vertex* yang telah dikunjungi. “Iterator<Integer>I =graph.adjListArray[v].listIterator();” berfungsi untuk membuat objek *iterator* yang bertipe data *integer* yaitu “v” untuk memodifikasi *vertex* awal dengan *listiterator*. *Code* “while (i.hasNext()){int n = i.next();if (!visited[n]){ *deepth*(n, visit,graph);}}}” merupakan perulangan untuk mengecek kondisi apakah objek *iterator* masih mempunyai nilai pada selanjutnya atau tidak. Lalu menampilkan *vertex* secara DFS yaitu mengecek dari *vertex* awal lalu ke elemen kiri dan terakhir ke elemen kanan.

|  |
| --- |
| static void DFS(int v, Graf grap{  boolean visited[] = new boolean[grap.Vertex];  *deepth*(v, visited,grap);} |

Potongan *code* “static void DFS(int v, Graf grap{” merupakan sebuah *method* yang bernama “DFS” yang bersifat *static void* dengan parameter variabel bernama “graph” dengan tipe data “Graf” dan “v” bertipe data *integer.* *Code* “boolean visited[] = new boolean[grap.Vertex];” berfungsi untuk menyimpan data *vertex* yang telah dikunjungi. *Code* “*deepth*(v, visited,grap); }” berfungsi untuk menampilkan *vertex* secara DFS yaitu mengecek dari *vertex* awal lalu ke elemen kiri dan terakhir ke elemen kanan.

|  |
| --- |
| public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  int V = 7 ; Graf gra = new Graf(V);  *addEdge*(gra, 0, 1);  *addEdge*(gra, 1, 2);  *addEdge*(gra, 1, 5);  *addEdge*(gra, 2, 0);  *addEdge*(gra, 2, 3);  *addEdge*(gra, 3, 4);  *addEdge*(gra, 4, 0);  *addEdge*(gra, 4, 6);  *addEdge*(gra, 5, 6);  System.*out*.println("Adjacency list");  *show*(gra); System.*out*.println("\nBFS");  System.*out*.println("\nMasukkan data awal penelusuran : ");int key = sc.nextInt();  System.*out*.println("\nHasil penelusuran : ");  *BFS*(key, gra); System.*out*.println("\nDFS"); System.*out*.println("\nMasukkan data awal penelusuran : ");  key = sc.nextInt();System.*out*.println("\nHasil penelusuran : ");  *DFS*(key, gra); } } |

Potongan *code* “public static void main(String [] ar{gs)” adalah suatu fungsi utama atau “main” yang bertipe data “void” dimana semua aktivitas dalam fungsi inilah yang akan dieksekusi terlebih dahulu oleh program. *Code* “Scanner sc = new Scanner(System.*in*);” berfungsi untuk membuat konstraktor dengan nama “sc” yang berfungsi untuk memasukkan data. “int V = 7 ;” berfungsi untuk menampung banyak *vertex graph.* “Graf gra = new Graf(V);” berfungsi untuk memanggil kelas “Graf” beserta *method – method* yang ada di dalamnya. *Code* “*addEdge*(gra, 0, 1);” berfungsi untuk mengurutkan pasangan antara *vertex* awal yaitu 0 dengan *vertex* tujuan yaitu 1. *Code* “*addEdge*(gra, 1, 2);” berfungsi untuk mengurutkan pasangan antara *vertex* awal yaitu 1 dengan *vertex* tujuan yaitu 2. *Code* “*show*(gra);” berfungsi untuk menampilkan *vertex* *graph* dengan *adjacency list. Code* “System.*out*.println("\nBFS");” berfungsi untuk menampilkan “BFS”. *Code* “System.*out*.println("\nMasukkan data awal penelusuran : ");” berfungsi untuk menampilkan “Masukkan data awal penelusuran :”. *Code* “int key = sc.nextInt();” berfungsi untuk memasukkan *vertex* awal. *Code* “System.*out*.println("\nHasil penelusuran : ");” berfungsi untuk menampilkan “nHasil penelusuran :”. *Code* “*BFS*(key, gra);” berfungsi untuk menampilkan urutan *vertex* sesuai dari *vertex* awal secara BFS. *Code* “int key = sc.nextInt();” berfungsi untuk memasukkan *vertex* awal. *Code* “System.*out*.println("\nHasil penelusuran : ");” berfungsi untuk menampilkan “nHasil penelusuran :”. *Code* “*DFS*(key, gra);” berfungsi untuk menampilkan urutan *vertex* sesuai dari *vertex* awal secara DFS.

# 6.6 KESIMPULAN

Dari praktikum yang telah di laksanakan dapat diambil beberapa kesimpulan, yaitu sebagai berikut:

1. Konsep *graph* yaitu sekumpulan *vertex* atau titik yang tidak memiliki *parent* atau tidak ada *vertex* atau titik awal. *Graph* ini merupakan pendekatan untuk mencari jarak terdekat untuk sampai ke tujuan, dimana setiap *node* atau titik dihubungkan dengan titik lainnya melalui sebuah jalur atau bisa juga disebut *edge* dan kemudian akan dilakukan pencarian untuk menentukan jarak terpanjang dan terpendek dari hubungan tersebut.
2. Pengimplementasian *graph* dalam dalam bahasa pemprograman *java* dilakukan dengan terlebih dahulu memahami konsep dari *graph*. Lalu, membentuk kelas-kelas dan metode-metode yang mendukung terbentuknya *graph* sesuai konsep.
3. Pengimplementasian DFS dan BFS Traversal pada *graph* dilakukan dengan bantuan metode-metode tertentu yaitu metode-metode yang mendukung pengaplikasian *stack* dan *queue*.