**DASAR ANALISIS ALGORITMA GRAF**

**STUDI KASUS: BREADTH FIRST SEARCH DAN DEPTH FIRST SEACRH MATA**

**PRAKTIKUM MATA KULIAH ANALISIS ALGORITMA**



Disusun oleh:

Nama : Muhammad Fadillah Arsa (140810170005)

Muhammad Siradj Al-Fauzi (140810170021)

Muhammad Hanif (140810170033)

Yoga Prambudi Sunaryudanto Putro (140810170065)

Kelas : A – Teknik Informatika 2017

**PROGRAM STUDI TEKNIK INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS PADJADJARAN**

**JATINANGOR**

**2019**

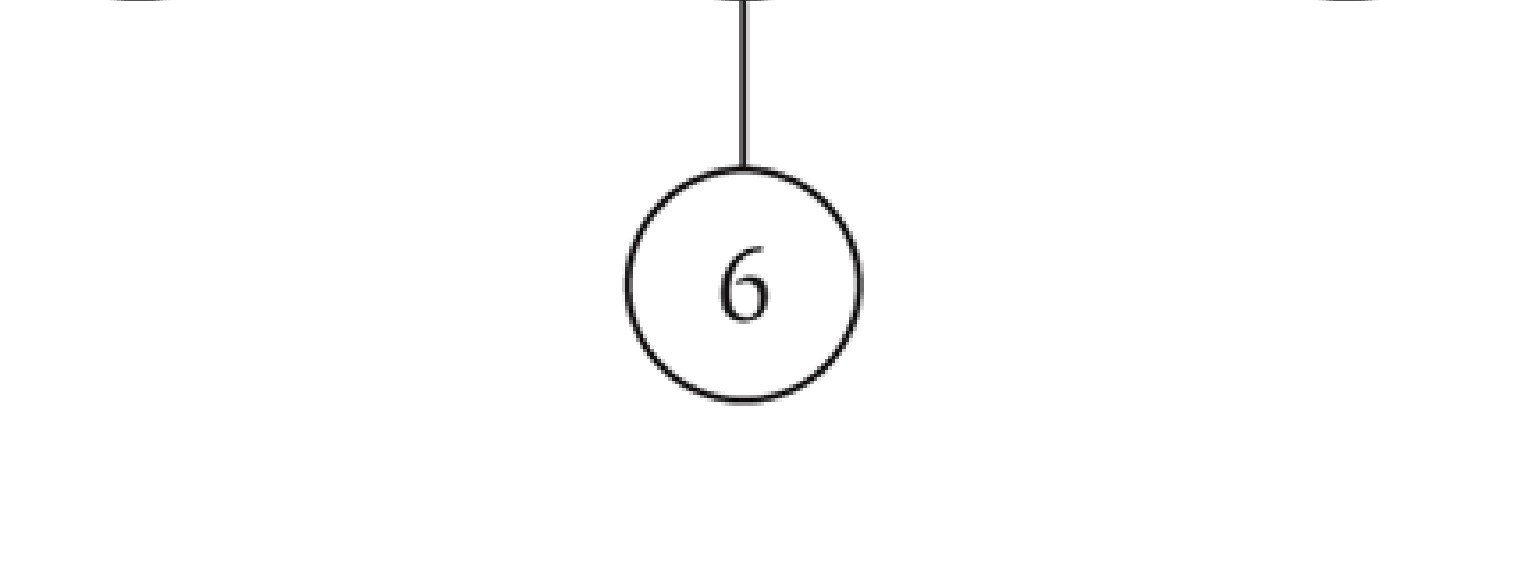
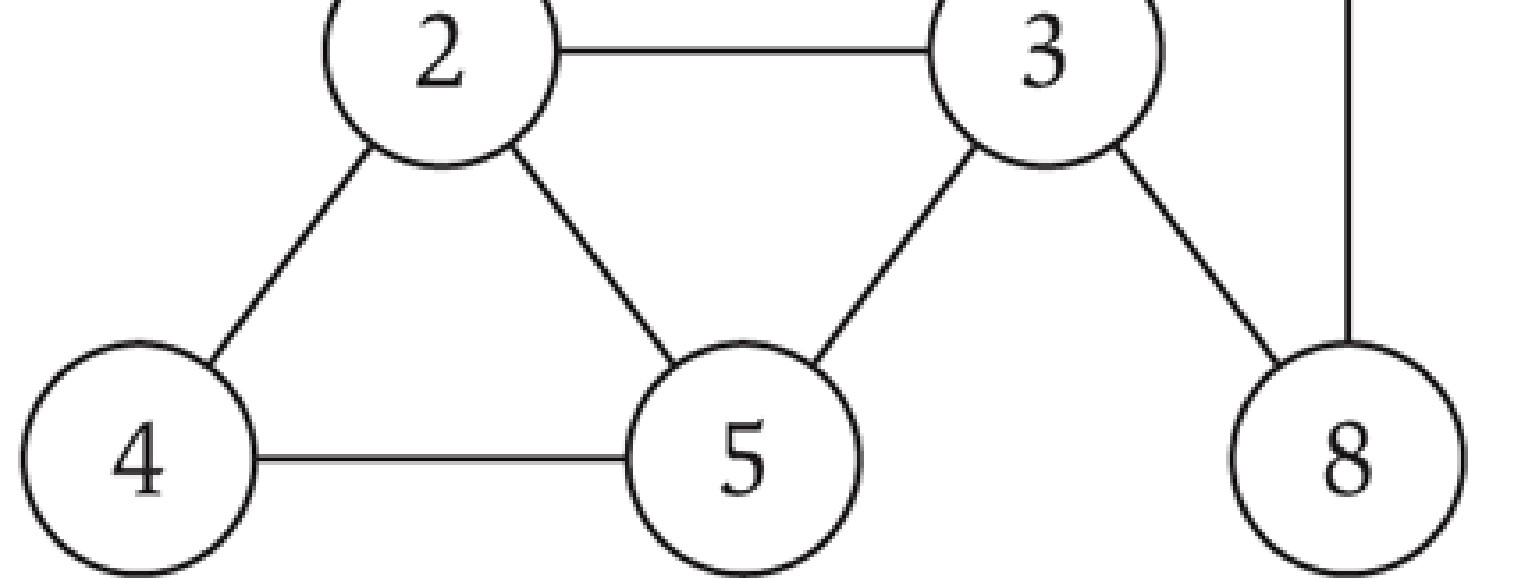
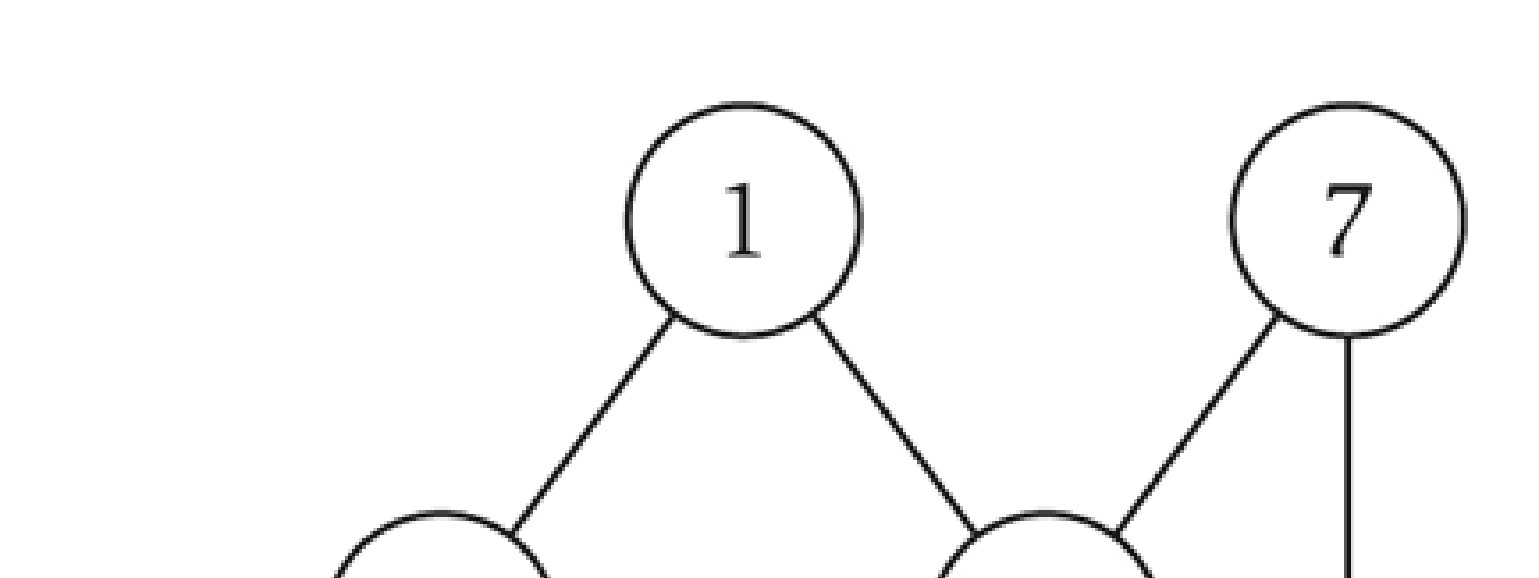
# Pendahuluan

Graf Tak Berarah (Undirected Graf)

(Undirected) graph: G=(V,E)

* V = sekumpulan node (vertex, simpul, titik, sudut)
* E = sekumpulan edge (garis, tepi)
* Menangkap hubungan berpasangan antar objek. • Parameter ukuran Graf: n = |V|, m=|E|

|  |
| --- |
| V = {1,2,3,4,5,6,7,8}  E = {(1,2), (1,3), (2,3), (2,4), (2,5), (3,5),(3,7), (3,8), (4,5), (5,6), (7,8)} n = 8 M = 11 |



Dalam pemrograman, Graf dapat direpresentasikan dengan ***adjacency matrix*** dan ***adjacency list***

## Representasi Graf dengan Adjacency Matrix

**Adjacency Matrix:** n-ke-n matriks dengan = 1 jika (u,v) adalah sebuah garis

* Dua representasi dari setiap sisi
* Ruang berukuran sebesar
* Memeriksa apakah (u, v) edge membutuhkan waktu Θ(1)
* Mengidentifikasi semua tepi membutuhkan Θ ( ) waktu



## Representasi Graf dengan Adjacency List Adjacency List: node diindeks sebagai list

* Dua representasi untuk setiap sisi
* Ukuran ruang m + n
* Memeriksa apakah (u, v) edge membutuhkan O (deg (u)). Degree = jumlah tetangga u.
* Mengidentifikasi semua tepi membutuhkan Θ(m + n).



Array List

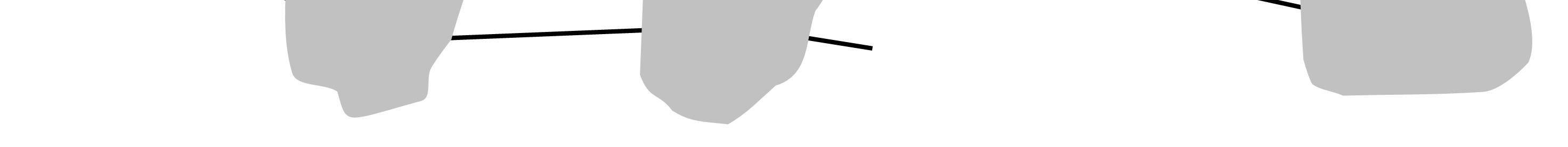
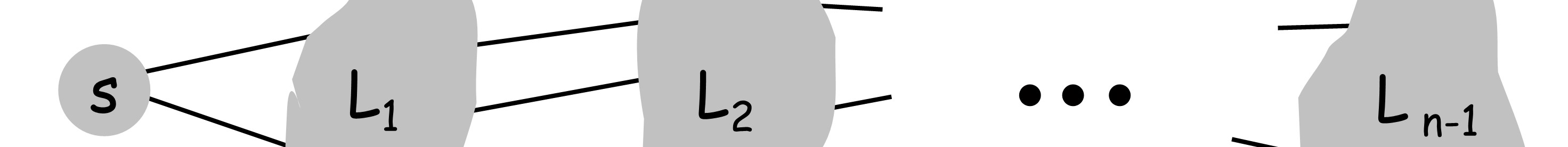
## Breadth First Search

Intuisi BFS. Menjelajahi alur keluar dari s ke semua arah yang mungkin, tambahkan node satu “layer" sekaligus.

### Algoritma BFS

* = { }
* = semua tetangga dari L
* = atau , dan yang mempunyai edge ke sebuah node di

* + 1 = semua node yang bukan milik layer sebelumnya, dan yang memiliki edge ke node di



Gambar 1. Ilustrasi algoritma BFS

### Teorema 1.0

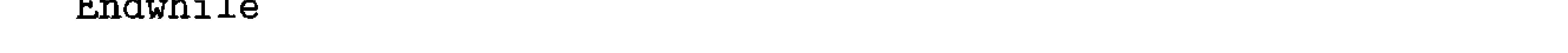
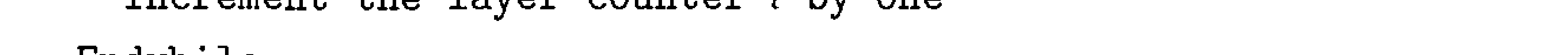
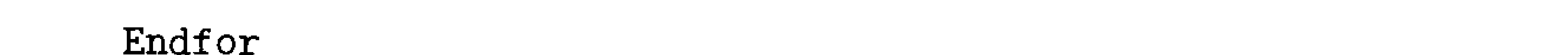
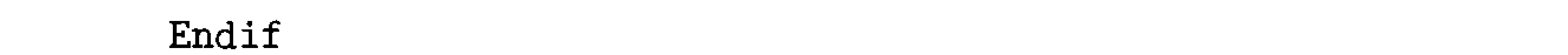
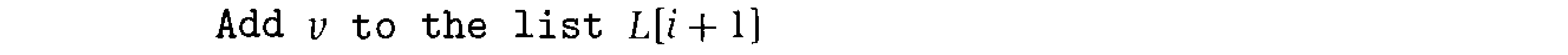
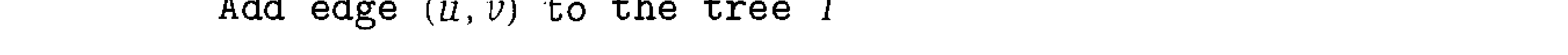
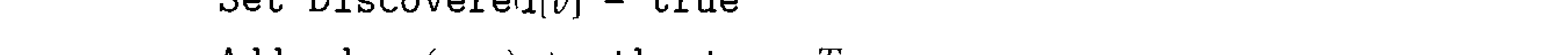
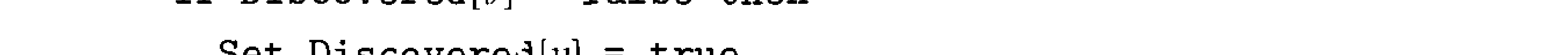
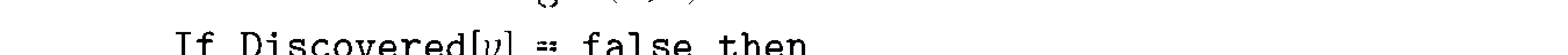
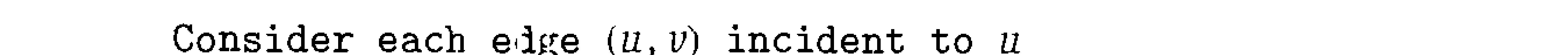
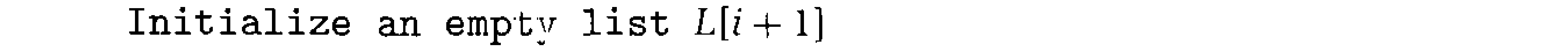
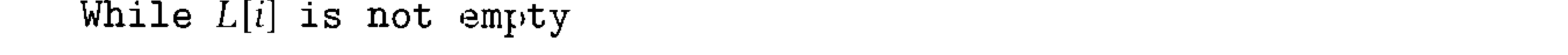
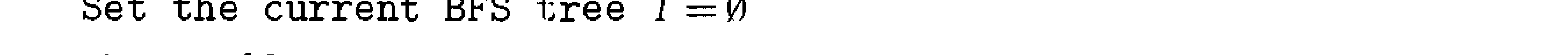
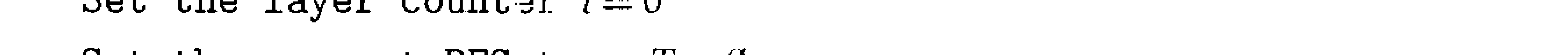
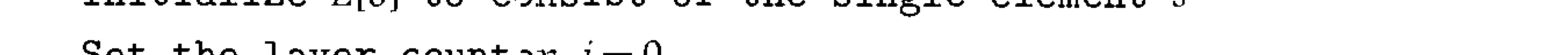
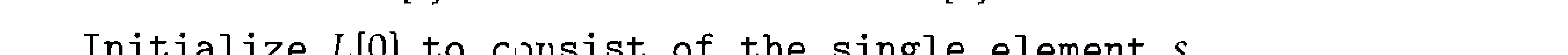
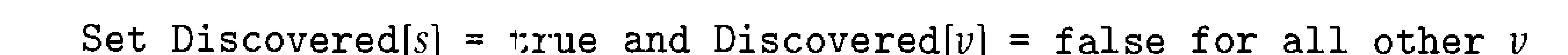
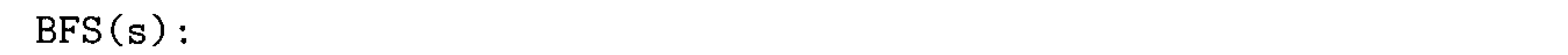
Untuk setiap , terdiri dari semua node pada jarak tepat ke dari . Ada path dari ke jika *t* muncul di beberapa layer.



Gambar 2. Ilustrasi pembentukan tree BFS dari undirected Graf

Implementasi BFS dalam Koding Program

* Adjacency list adalah representasi struktur data paling ideal untuk BFS
* Algoritma memeriksa setiap ujung yang meninggalkan node satu per satu. Ketika kita memindai edge yang meninggalkan u dan mencapai edge(u, v), kita perlu tahu apakah node v telah ditemukan sebelumnya oleh pencarian.
* Untuk menyederhanakan ini, kita maintain array yang ditemukan dengan panjang n dan mengatur Discovered [v] = true segera setelah pencarian kita pertama kali melihat v. Algoritma BFS membangun lapisan node L1, L2, ..., di mana Li adalah set node pada jarak dari sumber .
* Untuk mengelola node dalam layer , kami memiliki daftar [ ] untuk setiap = 0,1,2, ….

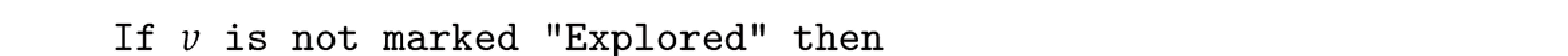
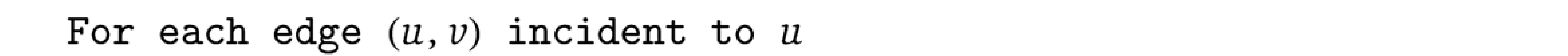
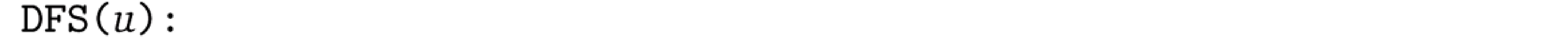


### Depth First Search

Algoritma BFS muncul, khususnya, sebagai cara tertentu mengurutkan node yang kita kunjungi — dalam lapisan berurutan, berdasarkan pada jarak node lain dari s. Metode alami lain untuk menemukan node yang dapat dijangkau dari s adalah pendekatan yang mungkin Anda dilakukan jika grafik G benar-benar sebuah labirin dari kamar yang saling berhubungan dan kita berjalan-jalan di dalamnya.

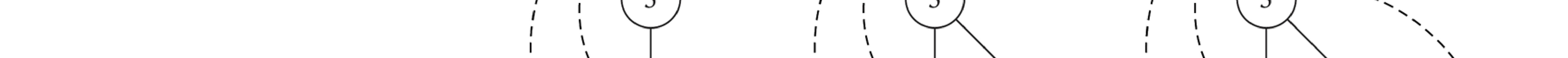
Kita akan mulai dari s dan mencoba edge pertama yang mengarah ke ke node v. Kita kemudian akan mengikuti edge pertama yang mengarah keluar dari v, dan melanjutkan dengan cara ini sampai kita mencapai "jalan buntu" —sebuah node di mana Anda sudah menjelajahi semua tetangganya. Kita kemudian akan mundur sampai kita mencapai node dengan tetangga yang belum dijelajahi, dan melanjutkan dari sana. Kita menyebutnya Depth-first search (DFS), karena ini mengeksplorasi G dengan masuk sedalam mungkin dan hanya mundur jika diperlukan.

DFS juga merupakan implementasi khusus dari algoritma component-growing generik yang dijelaskan sebelumnya. Kita dapat memulai DFS dari titik awal mana pun tetapi mempertahankan pengetahuan global tentang node yang telah dieksplorasi.



Untuk menerapkan ini pada problem konektivitas s-t, kita cukup mendeklarasikan semua node pada awalnya untuk tidak dieksplorasi, dan memanggil DFS (s).

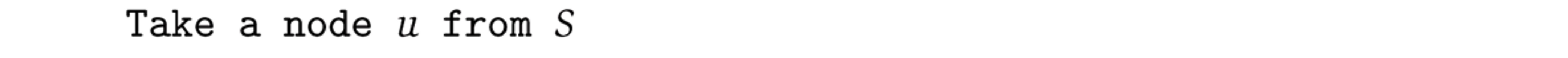
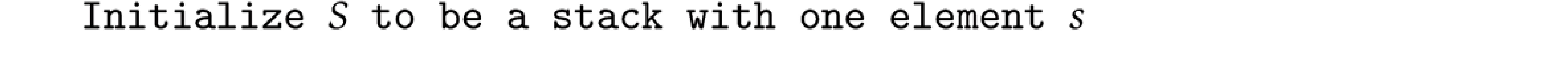
Ada beberapa kesamaan dan beberapa perbedaan mendasar antara DFS dan BFS. Kesamaan didasarkan pada fakta bahwa mereka berdua membangun komponen terhubung yang mengandung s, dan bahwa mereka mencapai tingkat efisiensi yang serupa secara kualitatif. Sementara DFS akhirnya mengunjungi set node yang sama persis seperti BFS, ia biasanya melakukannya dalam urutan yang sangat berbeda; menyelidiki jalan panjang, berpotensi menjadi sangat jauh dari s, sebelum membuat cadangan untuk mencoba lebih dekat node yang belum dijelajahi.



Gambar 3. Ilustrasi pembentukan tree DFS dari undirected graph

### Implementasi BFS dalam Koding Program

Implementasi DFS paling ideal adalah dengan menggunakan stack. Adapun algoritma DFS dengan stack adalah sebagai berikut:



**Tugas Anda**

1. Dengan menggunakan *undirected graph* dan *adjacency matrix* berikut, buatlah koding programmnya menggunakan bahasa C++.



**JAWABAN:**

#include<iostream>

using namespace std;

int isiMatriks[10][10]; //Index matriks 0

void tampilkanMatrix(int v) {

int i, j;

for(i = 0; i < v; i++) {

for(j = 0; j < v; j++) {

cout << isiMatriks[i][j] << " ";

}

cout << endl;

}

}

void tambah\_tepi(int u, int v) { //fungsi untuk menambahkan tepi

isiMatriks[u][v] = 1;

isiMatriks[v][u] = 1;

}

main(){

int v = 8; //ada 8 vertex di graf

tambah\_tepi (0, 1); //1-2

tambah\_tepi (0, 2); //1-3

tambah\_tepi (1, 2); //2-3

tambah\_tepi (1, 3); //2-4

tambah\_tepi (1, 4); //2-5

tambah\_tepi (2, 4); //3-5

tambah\_tepi (2, 6); //3-7

tambah\_tepi (2, 7); //3-8

tambah\_tepi (3, 3); //4-4

tambah\_tepi (3, 4); //4-5

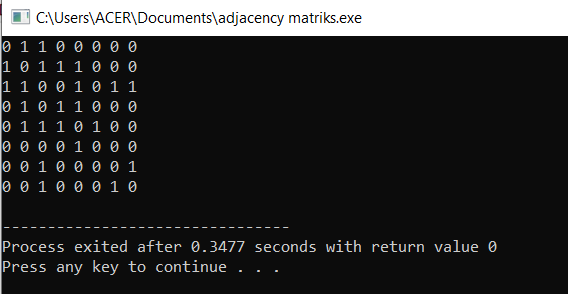
tambah\_tepi (4, 5); //5-6

tambah\_tepi (6, 7); //7-8

tampilkanMatrix (v);

}

**SCREENSHOT HASIL:**



1. Dengan menggunakan *undirected graph* dan representasi *adjacency list*, buatlah koding programmnya menggunakan bahasa C++.



**JAWABAN:**

#include<iostream>

#include<windows.h>

using namespace std;

struct adjacent{

int nodeAdj;

adjacent\* nextAdj;

};

struct elemen{

int node;

elemen\* next;

adjacent\* firstAdj;

};

typedef elemen\* pointerNode;

typedef adjacent\* pointerAdj;

typedef pointerNode list;

void createListNode(list& first){

first = NULL;

}

void createNode(pointerNode& pBaru,int simpul){

pBaru = new elemen;

pBaru->node = simpul;

pBaru->next = NULL;

pBaru->firstAdj = NULL;

}

void createAdjacent(pointerAdj& pBaru,int simpul){

pBaru = new adjacent;

pBaru->nodeAdj = simpul;

pBaru->nextAdj = NULL;

}

void insertAdjacent(pointerNode& curNode,pointerAdj pBaruAdj){

pointerAdj last;

if(curNode->firstAdj == NULL){

curNode->firstAdj = pBaruAdj;

}else{

last = curNode->firstAdj;

while(last->nextAdj != NULL){

last = last->nextAdj;

}

last->nextAdj = pBaruAdj;

}

}

void insertElement(list& first, pointerNode pBaruNode, int size){

pointerNode last;

pointerAdj pBaruAdj;

if(first == NULL){

first = pBaruNode;

}else{

last = first;

while(last->next != NULL){

last = last->next;

}

last->next = pBaruNode;

}

if(size>0){

cout<<"Masukan node yang berhubungan dengan "<<pBaruNode->node<<" : "<<endl;

}

for(int i = 0; i < size; i++){

int simpul;

cin>>simpul;

createAdjacent(pBaruAdj,simpul);

insertAdjacent(pBaruNode,pBaruAdj);

}

}

void output(list first){

pointerNode pOut;

pointerAdj pOutAdj;

if(first == NULL){

cout<<"Tidak ada Node"<<endl;

}else{

pOut = first;

while(pOut != NULL){

cout<<"Parent = "<<pOut->node<<endl;

if(pOut->firstAdj == NULL){

cout<<"Tidak ada adjacency"<<endl;

}else{

pOutAdj = pOut->firstAdj;

cout<<"Child = ";

while(pOutAdj != NULL){

cout<<pOutAdj->nodeAdj<<" ";

pOutAdj = pOutAdj->nextAdj;

}

}

cout<<endl;

pOut = pOut->next;

}

}

}

int main(){

list first;

pointerNode node;

createListNode(first);

createNode(node,1);

insertElement(first,node,2);

createNode(node,2);

insertElement(first,node,4);

createNode(node,3);

insertElement(first,node,5);

createNode(node,4);

insertElement(first,node,2);

createNode(node,5);

insertElement(first,node,4);

createNode(node,6);

insertElement(first,node,1);

createNode(node,7);

insertElement(first,node,2);

createNode(node,8);

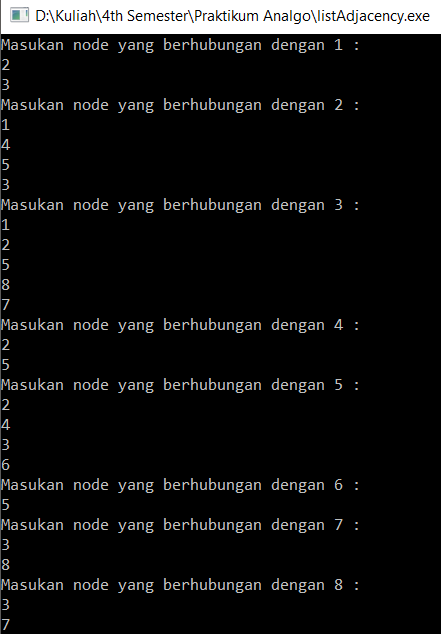
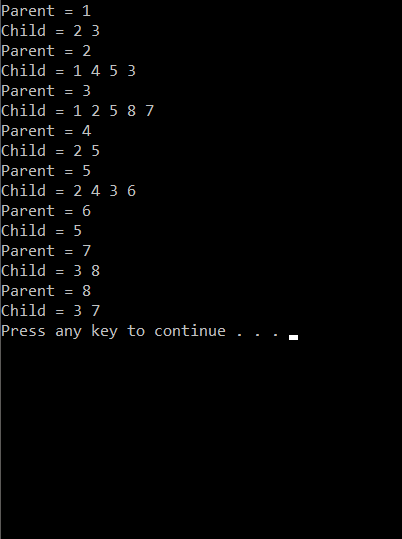
insertElement(first,node,2);

output(first);

system("pause");

}

**SCREENSHOT HASIL:**



1. Buatlah program Breadth First Search dari algoritma BFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree BFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!



**JAWABAN:**

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for(int i = 0; i < V; i++)

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(8);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(2, 5);

g.addEdge(2, 3);

g.addEdge(3, 7);

g.addEdge(3, 8);

g.addEdge(4, 5);

g.addEdge(5, 3);

g.addEdge(5, 6);

g.addEdge(7, 8);

cout << "Following is Breadth First Traversal "

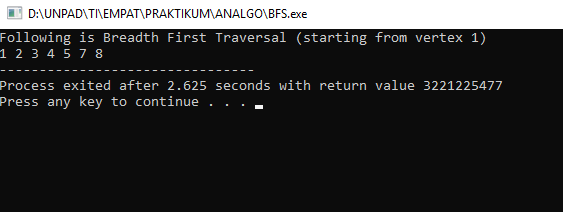
<< "(starting from vertex 1) \n";

g.BFS(1);

return 0;

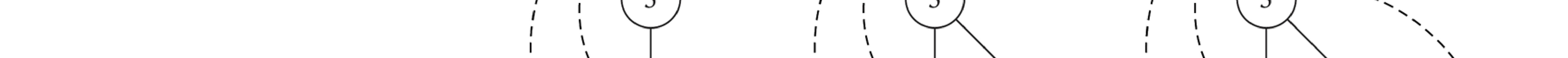
}

**SCREENSHOT HASIL DAN ANALISIS:**

****Karena Big-O dari BFS adalah O(V+E) dimana V itu jumlah vector dan E itu adalah jumlah edges maka Big-O = O(n) dimana n = v+e

Maka dari itu Big-Ө nya adalah Ө(n).

1. Buatlah program Depth First Search dari algoritma DFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree DFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!



**JAWABAN:**

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include<iostream>

#include<list>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing

// adjacency lists

list<int> \*adj;

// A recursive function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFSUtil(int v, bool visited[])

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

void Graph::DFS(int v)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// Call the recursive helper function

// to print DFS traversal

DFSUtil(v, visited);

}

int main()

{

// Create a graph given in the above diagram

Graph g(8);

g.addEdge(0, 1);//

g.addEdge(1, 0);//

g.addEdge(0, 2);//

g.addEdge(2, 0);//

g.addEdge(1, 2);

g.addEdge(2, 1);

g.addEdge(1, 3);

g.addEdge(3, 1);

g.addEdge(1,4 );

g.addEdge(4,1 );

g.addEdge(2, 4);

g.addEdge(4, 2);

g.addEdge(3, 4);

g.addEdge(4, 3);

g.addEdge(4, 5);

g.addEdge(5, 4);

g.addEdge(2, 6);

g.addEdge(6, 2);

g.addEdge(2, 7);

g.addEdge(7, 2);

g.addEdge(6, 7);

g.addEdge(7, 6);

cout << "Following is Depth First Traversal"

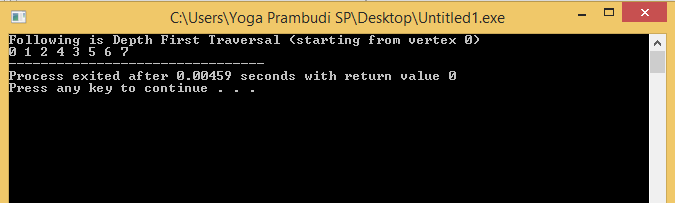
" (starting from vertex 0) \n";

g.DFS(0);

return 0;

}

**SCREENSHOT HASIL DAN ANALISIS:**



Big O

kompleksitas total DFS () adalah

O (V + E).

O(n)

Dengan V = Jumlah Verteks

Dan E = Jumlah Edges