LAPORAN PRAKTIKUM
STRUKTUR DATA

MODUL 14

GRAPH

Disusun Oleh :
NAMA : DEVI YULIANA
NIM : 103112400151

Dosen
FAHRUDIN MUKTI WIBOWO
STRUKTUR DATA

PROGRAM STUDI S1 TEKNIK INFORMATIKA
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY PURWOKERTO
2025

A. Dasar Teori

Graph merupakan salah satu struktur data yang digunakan untuk merepresentasikan hubungan antar objek. Dalam Modul 14 dijelaskan bahwa graph terdiri atas himpunan node (vertex) serta garis penghubung antar node yang disebut edge. Hubungan antara dua lokasi, misalnya tempat kost dan Common Lab, adalah contoh sederhana dari graph, di mana setiap lokasi dianggap sebagai node dan jalan yang menghubungkannya dianggap sebagai edge. Modul juga membedakan graph menjadi dua jenis, yaitu graph berarah dan graph tidak berarah. Pada graph berarah, suatu node dapat terhubung ke node lain hanya melalui arah tertentu, sedangkan pada graph tidak berarah, hubungan antar node bersifat dua arah sehingga jika A terhubung ke B, maka B juga terhubung ke A.

Representasi graph dalam modul dilakukan menggunakan **multilist**, karena struktur ini bersifat dinamis dan mampu menampung perubahan jumlah node maupun edge secara fleksibel. Setiap node pada graph direpresentasikan sebagai elemen list yang menyimpan informasi node, status kunjungannya (visited), pointer ke daftar edge (firstEdge), serta pointer ke node berikutnya (next). Sementara itu, setiap edge direpresentasikan sebagai elemen list lain yang berisi pointer menuju node tetangga serta pointer ke edge berikutnya. Dengan pendekatan multilist, setiap node dapat memiliki daftar tetangga yang terhubung dengannya, sehingga struktur graph dapat dibangun secara efisien tanpa batasan ukuran tertentu seperti pada adjacency matrix.

Breadth First Search (BFS) adalah salah satu metode penelusuran graph yang dijelaskan dalam modul. BFS menelusuri graph secara melebar (level by level), dimulai dari node awal kemudian mengunjungi semua node yang bertetangga langsung dengan node tersebut sebelum berpindah ke level berikutnya. Proses penelusuran BFS dilakukan menggunakan struktur data queue sehingga node yang pertama kali dimasukkan akan menjadi node yang pertama kali diproses. Algoritma BFS bekerja dengan memasukkan node awal ke dalam queue, kemudian secara berulang mengambil node terdepan, menandainya sebagai telah dikunjungi, dan memasukkan seluruh tetangganya yang belum dikunjungi ke dalam queue. Proses ini terus berlangsung hingga queue kosong, menandakan seluruh node yang dapat dicapai dari node awal telah dikunjungi. Untuk memastikan penelusuran tidak mengunjungi node yang sama berulang kali atau terjebak dalam siklus, setiap node memiliki atribut visited yang di-reset sebelum proses BFS dimulai.

B. Guided (berisi screenshot source code & output program disertai penjelasannya)

graph.h

```
#ifndef GRAF_H_INCLUDE
#define GRAF_H_INCLUDED
#include <iostream>

using namespace std;
typedef char infoGraph;

struct ElmNode;
struct ElmEdge;
typedef ElmNode *adrNode;
typedef ElmEdge *adrEdge;
struct ElmNode {
    infoGraph info;
    int visited;
    adrEdge firstEdge;
    adrNode next;
};

struct ElmEdge {
adrNode node;
adrEdge next;
};

struct Graph {
adrNode first;
};

void createGraph(Graph &G);
adrNode AllocatedNode(infoGraph X);
adrEdge AllocatedEdge(adrNode N);
void insertNode(Graph &G, infoGraph X);
void FindNode(Graph G, infoGraph X);
void ConnectNode(Graph &G, infoGraph A, infoGraph B);
void printInfoGraph(Graph G);
void ResetVisited(Graph &G);
void printDFS(Graph &G, adrNode N);
void printBFS(Graph &G, adrNode N);
#endif
```

graph.cpp

```
#include "graf.h"
#include <queue>
#include <stack>
 void createGraph(Graph &G)
{
    G.first = NULL;
}  adrNode AllocatedNode(infoGraph X)
{
    adrNode P = new ElmNode;
    P->info = X;
    P->visited = 0;
    P->firstEdge = NULL;     P->next = NULL;     return P;
}  adrEdge AllocatedEdge(adrNode N)
{
    adrEdge P = new ElmEdge;
    P->node = N;     P->next = NULL;     return P;
}  void InsertNode(Graph &G, infoGraph X)
{
    adrNode P = AllocatedNode(X);
    P->next = G.first;
    G.first = P;
}  adrNode findNode(Graph G, infoGraph X)
{     adrNode P = G.first;     while (P != NULL) {         if (P->info ==
X)             return P;
        P = P->next;
    }     return NULL;
}  void ConnectNode(Graph &G, infoGraph A, infoGraph B)
{     adrNode N1 = findNode(G, A);     adrNode N2 = findNode(G,
B);     if (N1 == NULL || N2 == NULL) {         cout << "Node tidak
ditemukan!\n";         return;
    }
    adrEdge E1 = AllocatedEdge(N2);
    E1->next = N1->firstEdge;
    N1->firstEdge = E1;

    adrEdge E2 = AllocatedEdge(N1);
    E2->next = N2->firstEdge;
    N2->firstEdge = E2;
}
```

```
 void printInfoGraph(Graph G)
{     adrNode P = G.first;     while (P != NULL)
    {         cout << P->info << "->";         adrEdge E = P-
>firstEdge;         while (E != NULL)
        {
            cout << E->node->info << " ";
            E = E->next;
        }
        cout << endl;
        P = P->next;
    }
}  void ResetVisited(Graph &G)
{     adrNode P = G.first;     while (P != NULL)
    {
        P->visited = 0;
        P = P->next;
    }
}  void printDFS(Graph &G, adrNode N)
{     if (N == NULL)         return;

    N->visited = 1;     cout << N->info << " ";     adrEdge E = N-
>firstEdge;     while (E != NULL)
    {         if (E->node->visited == 0)
        {
            printDFS(G, E->node);
        }
        E = E->next;
    }
}  void printBFS(Graph &G, adrNode N)
{
if (N == NULL)         return;

    queue<adrNode> Q;     Q.push(N);
    while (!Q.empty())
    {
        adrNode curr = Q.front();         Q.pop();         if (curr-
>visited == 0)
        {             curr->visited = 1;             cout << curr->info
<< " ";             adrEdge E = curr->firstEdge;             while (E !=
NULL)
```

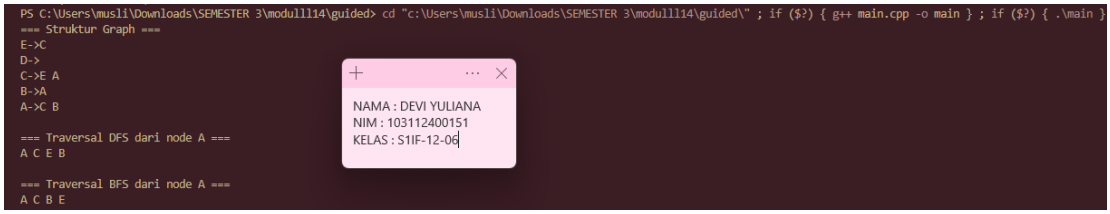
```
            {                   if (E->node->visited == 0)
                {
                    Q.push(E->node);
                }
                E = E->next;
            }
        }
    }
}
```

main.cpp

```
# #include <iostream>
#include "graf.h"
#include "graf.cpp"

using namespace std;

int main() {

    Graph G;
    createGraph(G);
    InsertNode(G, 'A');
    InsertNode(G, 'B');
    InsertNode(G, 'C');
    InsertNode(G, 'D');
    InsertNode(G, 'E');
    ConnectNode(G, 'A', 'B');
    ConnectNode(G, 'A', 'C');
    ConnectNode(G, 'C', 'E');
    cout << "=== Struktur Graph ===" << endl;
    printInfoGraph(G);
    cout << "\n=== Traversal DFS dari node A ===" << endl;
    ResetVisited(G);
    printDFS(G, findNode(G, 'A'));
    cout << "\n\n=== Traversal BFS dari node A ===" << endl;
    ResetVisited(G);
    printBFS(G, findNode(G, 'A'));
    cout << endl;
```

```
    return 0;
}
```

Screenshots Output



Deskripsi :

Program ini membangun sebuah graph tidak berarah dengan menggunakan struktur multilist. Setiap node pada graph disimpan dalam struktur ElmNode yang berisi informasi karakter, penanda kunjungan (visited), pointer ke daftar edge, dan pointer ke node berikutnya. Setiap edge direpresentasikan oleh ElmEdge, yang menyimpan node tujuan serta pointer edge berikutnya. Dengan struktur ini, graph dapat menampung hubungan antar-node secara dinamis.

Fungsi-fungsi dasar mencakup createGraph untuk membuat graph baru, AllocatedNode dan AllocatedEdge untuk mengalokasikan node dan edge baru, InsertNode untuk menambahkan node, serta findNode dan ConnectNode untuk mencari dan menghubungkan node. Fungsi printInfoGraph digunakan untuk menampilkan setiap node beserta tetangganya dalam bentuk adjacency list.

Program juga mengimplementasikan dua metode penelusuran graph. Fungsi printDFS melakukan penelusuran Depth First Search secara rekursif dengan mengunjungi node sedalam mungkin sebelum kembali. Sedangkan fungsi printBFS menggunakan struktur queue untuk melakukan penelusuran Breadth First Search, yaitu mengunjungi node secara melebar berdasarkan urutan level. Fungsi ResetVisited digunakan untuk mengatur ulang status kunjungan sebelum DFS dan BFS dijalankan.

Pada fungsi main, program membuat graph berisi node A, B, C, D, dan E, lalu menghubungkan beberapa node seperti A–B, A–C, dan C–E. Setelah itu, program menampilkan struktur graph, kemudian menjalankan penelusuran DFS dan BFS dari node A untuk menunjukkan perbedaan hasil kedua metode penelusuran tersebut.

C. Unguided/Tugas (berisi screenshot source code & output program disertai penjelasannya)

Unguided 1

graph.h

```
#ifndef GRAPH_H_INCLUDED
#define GRAPH_H_INCLUDED

#include <iostream>
using namespace std;

typedef char infoGraph;

struct ElmNode;
struct ElmEdge;

typedef ElmNode* adrNode;
typedef ElmEdge* adrEdge;

struct ElmNode {
    infoGraph info;
    int visited;        // disiapkan untuk latihan DFS/BFS berikutnya
    adrEdge firstEdge;  // pointer ke list edge
    adrNode next;       // pointer ke node berikutnya
};

struct ElmEdge {
    adrNode node;       // node tetangga
    adrEdge next;       // edge berikutnya dari node yang sama
};

struct Graph {
    adrNode first;      // node pertama dalam graph
};

// Primitif yang diminta soal
void CreateGraph(Graph &G);
void InsertNode(Graph &G, infoGraph X);
void ConnectNode(adrNode N1, adrNode N2);
void PrintInfoGraph(Graph G);
```

```
// Fungsi bantu (boleh dipakai untuk implementasi)
adrNode AllocateNode(infoGraph X);
adrEdge AllocateEdge(adrNode N);
adrNode FindNode(Graph G, infoGraph X);

#endif
```

graph.cpp

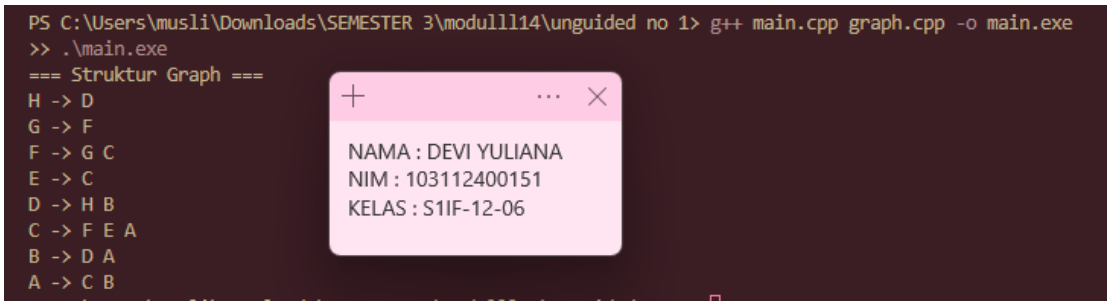
```
#include "graph.h"

void CreateGraph(Graph &G) {
    G.first = NULL;
}

// Alokasi node baru
adrNode AllocateNode(infoGraph X) {
    adrNode P = new ElmNode;
    P->info = X;
    P->visited = 0;
    P->firstEdge = NULL;
    P->next = NULL;
    return P;
}

// Alokasi edge baru
adrEdge AllocateEdge(adrNode N) {
    adrEdge E = new ElmEdge;
    E->node = N;
    E->next = NULL;
    return E;
}

// Insert node ke list node (InsertFirst)
void InsertNode(Graph &G, infoGraph X) {
    adrNode P = AllocateNode(X);
    P->next = G.first;
    G.first = P;
}
```

```
// Mencari node berdasarkan info
adrNode FindNode(Graph G, infoGraph X) {
    adrNode P = G.first;
    while (P != NULL) {
        if (P->info == X) return P;
        P = P->next;
    }
    return NULL;
}

// Menghubungkan dua node (graph tak berarah)
void ConnectNode(adrNode N1, adrNode N2) {
    if (N1 == NULL || N2 == NULL) return;

    // edge dari N1 ke N2
    adrEdge E1 = AllocateEdge(N2);
    E1->next = N1->firstEdge;
    N1->firstEdge = E1;

    // edge dari N2 ke N1 (karena tak berarah)
    adrEdge E2 = AllocateEdge(N1);
    E2->next = N2->firstEdge;
    N2->firstEdge = E2;
}

// Cetak struktur graph: setiap node dan tetangganya
void PrintInfoGraph(Graph G) {
    adrNode P = G.first;
    while (P != NULL) {
        cout << P->info << " -> ";
        adrEdge E = P->firstEdge;
        while (E != NULL) {
            cout << E->node->info << " ";
            E = E->next;
        }
        cout << endl;
        P = P->next;
    }
}
```

main.cpp

```cpp
#include <iostream>
#include "graph.h"

using namespace std;

int main() {
    Graph G;
    CreateGraph(G);

    // Membuat node A-H (sesuai gambar di modul)
    InsertNode(G, 'A');
    InsertNode(G, 'B');
    InsertNode(G, 'C');
    InsertNode(G, 'D');
    InsertNode(G, 'E');
    InsertNode(G, 'F');
    InsertNode(G, 'G');
    InsertNode(G, 'H');

    // Menghubungkan node (sesuai ilustrasi graph di modul)
    ConnectNode(FindNode(G, 'A'), FindNode(G, 'B'));
    ConnectNode(FindNode(G, 'A'), FindNode(G, 'C'));
    ConnectNode(FindNode(G, 'B'), FindNode(G, 'D'));
    ConnectNode(FindNode(G, 'D'), FindNode(G, 'H'));
    ConnectNode(FindNode(G, 'C'), FindNode(G, 'E'));
    ConnectNode(FindNode(G, 'C'), FindNode(G, 'F'));
    ConnectNode(FindNode(G, 'F'), FindNode(G, 'G'));

    cout << "=== Struktur Graph ===" << endl;
    PrintInfoGraph(G);

    return 0;
}
```

Screenshots Output



Deskripsi :

Graph yang dibentuk terdiri dari sekumpulan node yang masing-masing memiliki informasi berupa karakter (tipe char). Setiap node disusun dalam sebuah linked list utama, sementara setiap node juga memiliki daftar edge yang direpresentasikan sebagai linked list kedua yang berisi node-node yang bertetangga dengannya. Representasi multilist ini dipilih karena sifatnya yang dinamis, sehingga memudahkan penambahan node dan edge tanpa memerlukan ukuran struktur yang tetap. Program dimulai dengan mendefinisikan struktur dasar graph, yaitu ElmNode dan ElmEdge. Struktur ElmNode menyimpan informasi node, penanda kunjungan (*visited*), pointer menuju daftar edge pertama, dan pointer node berikutnya. Sementara itu, ElmEdge digunakan untuk menyimpan node tetangga dari suatu node serta pointer ke edge berikutnya.

Dengan kedua struktur ini, program dapat menyusun graph tak berarah secara fleksibel. Selanjutnya, program menyediakan beberapa fungsi dasar untuk membentuk graph. Fungsi CreateGraph digunakan untuk menginisialisasi graph kosong, sedangkan InsertNode berfungsi untuk menambahkan node baru ke dalam graph. Fungsi FindNode digunakan untuk mencari node tertentu berdasarkan karakter informasinya. Hubungan antar dua node dibentuk melalui fungsi ConnectNode, yang menambahkan edge dua arah (karena graph tak berarah) antara node yang bersangkutan. Dengan demikian, jika node A dihubungkan dengan node B, maka A tercatat sebagai tetangga B dan B tercatat sebagai tetangga A.

Program juga menyediakan fungsi PrintInfoGraph untuk menampilkan struktur graph yang telah dibentuk. Fungsi ini mencetak setiap node beserta daftar node tetangganya, sesuai dengan representasi multilist. Dengan menjalankan program ini, pengguna dapat melihat bagaimana node-node pada graph terhubung satu sama lain secara dinamis.

Unguided 2

graph.h

```
#ifndef GRAPH_H_INCLUDED
#define GRAPH_H_INCLUDED

#include <iostream>
```

```cpp
using namespace std;

typedef char infoGraph;

struct ElmNode;
struct ElmEdge;

typedef ElmNode* adrNode;
typedef ElmEdge* adrEdge;

// STRUCT NODE
struct ElmNode {
    infoGraph info;
    int visited;
    adrEdge firstEdge;
    adrNode next;
};

// STRUCT EDGE
struct ElmEdge {
    adrNode node;
    adrEdge next;
};

// STRUCT GRAPH
struct Graph {
    adrNode first;
};

// ===== PRIMITIF ADT (LATIHAN 1) =====
void CreateGraph(Graph &G);
adrNode AllocateNode(infoGraph X);
adrEdge AllocateEdge(adrNode N);
void InsertNode(Graph &G, infoGraph X);
adrNode FindNode(Graph G, infoGraph X);
void ConnectNode(Graph &G, infoGraph A, infoGraph B);
void PrintInfoGraph(Graph G);

// ====== LATIHAN NOMOR 2 (DFS) ======
void ResetVisited(Graph &G);
```

```
void PrintDFS(Graph &G, adrNode N);

#endif
```

graph.cpp

```
#include "graph.h"

// ========== FUNGSI DASAR GRAPH ==========

void CreateGraph(Graph &G) {
    G.first = NULL;
}

adrNode AllocateNode(infoGraph X) {
    adrNode P = new ElmNode;
    P->info      = X;
    P->visited   = 0;
    P->firstEdge = NULL;
    P->next      = NULL;
    return P;
}

adrEdge AllocateEdge(adrNode N) {
    adrEdge E = new ElmEdge;
    E->node = N;
    E->next = NULL;
    return E;
}

void InsertNode(Graph &G, infoGraph X) {
    adrNode P = AllocateNode(X);
    P->next   = G.first;
    G.first   = P;
}

adrNode FindNode(Graph G, infoGraph X) {
    adrNode P = G.first;
    while (P != NULL) {
```

```
        if (P->info == X) return P;
        P = P->next;
    }
    return NULL;
}

void ConnectNode(Graph &G, infoGraph A, infoGraph B) {
    adrNode N1 = FindNode(G, A);
    adrNode N2 = FindNode(G, B);

    if (N1 == NULL || N2 == NULL) {
        cout << "Node tidak ditemukan!\n";
        return;
    }

    // karena graph TIDAK berarah → dua arah
    adrEdge E1 = AllocateEdge(N2);
    E1->next   = N1->firstEdge;
    N1->firstEdge = E1;

    adrEdge E2 = AllocateEdge(N1);
    E2->next   = N2->firstEdge;
    N2->firstEdge = E2;
}

void PrintInfoGraph(Graph G) {
    adrNode P = G.first;
    while (P != NULL) {
        cout << P->info << " -> ";
        adrEdge E = P->firstEdge;
        while (E != NULL) {
            cout << E->node->info << " ";
            E = E->next;
        }
        cout << endl;
        P = P->next;
    }
}

// ========== BAGIAN PENTING: NOMOR 2 (DFS) ==========
```

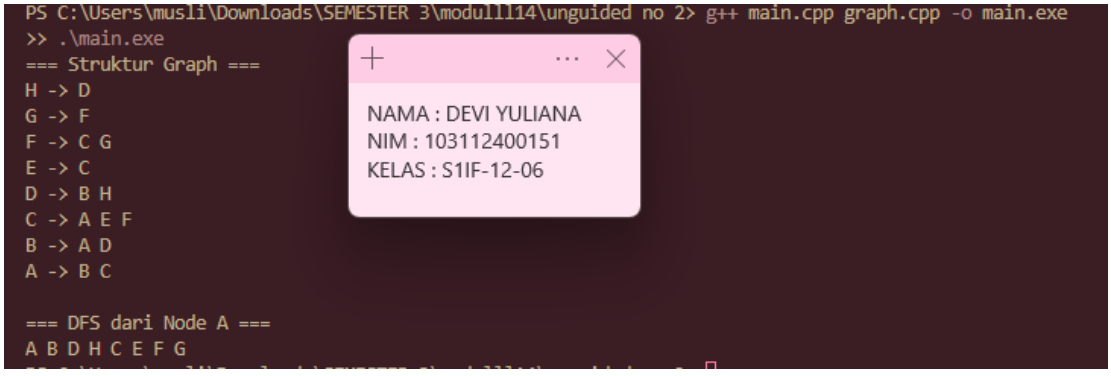
```
// Reset visited sebelum DFS
void ResetVisited(Graph &G) {
    adrNode P = G.first;
    while (P != NULL) {
        P->visited = 0;
        P = P->next;
    }
}

// === PrintDFS (LATIHAN NOMOR 2) ===
void PrintDFS(Graph &G, adrNode N) {
    if (N == NULL) return;

    N->visited = 1;
    cout << N->info << " ";

    adrEdge E = N->firstEdge;
    while (E != NULL) {
        if (E->node->visited == 0) {
            PrintDFS(G, E->node);  // REKURSIF → DFS
        }
        E = E->next;
    }
}
```

main.cpp

```
#include <iostream>
#include "graph.h"

using namespace std;

int main() {
    Graph G;
    CreateGraph(G);

    // Membuat node A–H (contoh dari modul)
    InsertNode(G, 'A');
```

```
    InsertNode(G, 'B');
    InsertNode(G, 'C');
    InsertNode(G, 'D');
    InsertNode(G, 'E');
    InsertNode(G, 'F');
    InsertNode(G, 'G');
    InsertNode(G, 'H');

    // Menghubungkan sesuai gambar modul
    ConnectNode(G, 'A', 'B');
    ConnectNode(G, 'A', 'C');
    ConnectNode(G, 'B', 'D');
    ConnectNode(G, 'D', 'H');
    ConnectNode(G, 'C', 'E');
    ConnectNode(G, 'C', 'F');
    ConnectNode(G, 'F', 'G');

    cout << "=== Struktur Graph ===" << endl;
    PrintInfoGraph(G);

    cout << "\n=== DFS dari Node A ===" << endl;
    ResetVisited(G);
    PrintDFS(G, FindNode(G, 'A'));

    cout << endl;
    return 0;
}
```

Screenshots Output



Deskripsi :

Program ini mengimplementasikan algoritma Depth First Search (DFS) untuk menelusuri graph tidak berarah yang direpresentasikan dengan multilist. Penelusuran dimulai dari satu node awal, kemudian bergerak ke tetangga pertama yang belum dikunjungi, dan terus masuk lebih dalam hingga tidak ada lagi node baru yang dapat ditelusuri. Proses ini dilakukan secara rekursif dan setiap node ditandai dengan variabel visited untuk mencegah kunjungan ulang.

Unguided 3

graph.h

```
#ifndef GRAPH_H_INCLUDED
#define GRAPH_H_INCLUDED

#include <iostream>
using namespace std;

typedef char infoGraph;

struct ElmNode;
struct ElmEdge;

typedef ElmNode* adrNode;
typedef ElmEdge* adrEdge;

struct ElmNode {
    infoGraph info;
    int visited;
    adrEdge firstEdge;
    adrNode next;
};

struct ElmEdge {
    adrNode node;
    adrEdge next;
};

struct Graph {
    adrNode first;
};
```

```
// ====== PRIMITIF GRAPH (LATIHAN 1) ======
void CreateGraph(Graph &G);
adrNode AllocateNode(infoGraph X);
adrEdge AllocateEdge(adrNode N);
void InsertNode(Graph &G, infoGraph X);
adrNode FindNode(Graph G, infoGraph X);
void ConnectNode(Graph &G, infoGraph A, infoGraph B);
void PrintInfoGraph(Graph G);

// ====== LATIHAN NOMOR 3: BFS ======
void ResetVisited(Graph &G);
void PrintBFS(Graph &G, adrNode N);

#endif
```

graph.cpp

```
#include "graph.h"
#include <queue>

// ========== ADT GRAPH DASAR ==========

void CreateGraph(Graph &G) {
    G.first = NULL;
}

adrNode AllocateNode(infoGraph X) {
    adrNode P = new ElmNode;
    P->info = X;
    P->visited = 0;
    P->firstEdge = NULL;
    P->next = NULL;
    return P;
}

adrEdge AllocateEdge(adrNode N) {
    adrEdge E = new ElmEdge;
    E->node = N;
    E->next = NULL;
    return E;
}
```

```
void InsertNode(Graph &G, infoGraph X) {
    adrNode P = AllocateNode(X);
    P->next = G.first;
    G.first = P;
}

adrNode FindNode(Graph G, infoGraph X) {
    adrNode P = G.first;
    while (P != NULL) {
        if (P->info == X) return P;
        P = P->next;
    }
    return NULL;
}

void ConnectNode(Graph &G, infoGraph A, infoGraph B) {
    adrNode N1 = FindNode(G, A);
    adrNode N2 = FindNode(G, B);

    if (!N1 || !N2) {
        cout << "Node tidak ditemukan!\n";
        return;
    }

    // Tidak berarah – dua arah
    adrEdge E1 = AllocateEdge(N2);
    E1->next = N1->firstEdge;
    N1->firstEdge = E1;

    adrEdge E2 = AllocateEdge(N1);
    E2->next = N2->firstEdge;
    N2->firstEdge = E2;
}

void PrintInfoGraph(Graph G) {
    adrNode P = G.first;
    while (P != NULL) {
        cout << P->info << " -> ";
        adrEdge E = P->firstEdge;
        while (E != NULL) {
```

```
            cout << E->node->info << " ";
            E = E->next;
        }
        cout << endl;
        P = P->next;
    }
}

// ========== RESET VISITED ==========

void ResetVisited(Graph &G) {
    adrNode P = G.first;
    while (P != NULL) {
        P->visited = 0;
        P = P->next;
    }
}

// ========== LATIHAN NOMOR 3: BFS ==========

void PrintBFS(Graph &G, adrNode N) {
    if (N == NULL) return;

    queue<adrNode> Q;
    Q.push(N);

    while (!Q.empty()) {
        adrNode curr = Q.front();
        Q.pop();

        if (curr->visited == 0) {
            curr->visited = 1;
            cout << curr->info << " ";

            adrEdge E = curr->firstEdge;
            while (E != NULL) {
                if (E->node->visited == 0) {
                    Q.push(E->node);
                }
                E = E->next;
            }
```

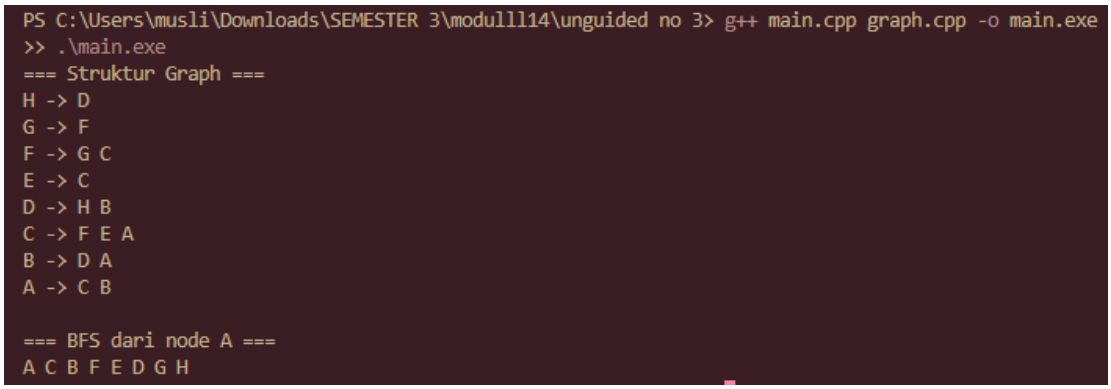
```
        }
    }
}
```

main.cpp

```cpp
#include <iostream>
#include "graph.h"

using namespace std;

int main() {
    Graph G;
    CreateGraph(G);

    // Membuat node A-H
    InsertNode(G, 'A');
    InsertNode(G, 'B');
    InsertNode(G, 'C');
    InsertNode(G, 'D');
    InsertNode(G, 'E');
    InsertNode(G, 'F');
    InsertNode(G, 'G');
    InsertNode(G, 'H');

    // Menghubungkan sesuai modul
    ConnectNode(G, 'A', 'B');
    ConnectNode(G, 'A', 'C');
    ConnectNode(G, 'B', 'D');
    ConnectNode(G, 'D', 'H');
    ConnectNode(G, 'C', 'E');
    ConnectNode(G, 'C', 'F');
    ConnectNode(G, 'F', 'G');

    cout << "=== Struktur Graph ===" << endl;
    PrintInfoGraph(G);

    cout << "\n=== BFS dari node A ===" << endl;
    ResetVisited(G);
    PrintBFS(G, FindNode(G, 'A'));    // INI JAWABAN LATIHAN 3
```

```
    cout << endl;
    return 0;
}
```

Screenshots Output



Deskripsi :

Program ini menerapkan algoritma Breadth First Search (BFS) untuk menelusuri graph secara melebar menggunakan struktur data queue. BFS mengunjungi node awal, lalu seluruh tetangga langsungnya sebelum berpindah ke node pada level berikutnya. Setiap node yang telah dikunjungi ditandai dengan `visited`, sehingga tidak diproses lebih dari sekali.

D. Kesimpulan

Berdasarkan praktikum graph pada Modul 14, dapat disimpulkan bahwa graph merupakan struktur data penting yang mampu merepresentasikan hubungan antar objek secara fleksibel. Dengan menggunakan multilist sebagai bentuk representasi, graph dapat dibangun secara dinamis karena node dan edge dapat ditambahkan tanpa batasan ukuran. Selain itu, penggunaan multilist memudahkan implementasi key operations seperti penambahan node, pencarian node, serta pembuatan hubungan antar node.

Pada latihan penelusuran, metode Breadth First Search (BFS) memberikan gambaran bagaimana graph dapat dijelajahi secara sistematis dengan pendekatan melebar. BFS memanfaatkan struktur data queue untuk menjelajahi node berdasarkan level kedekatan dari node awal, sehingga seluruh node tetangga dikunjungi terlebih dahulu sebelum melanjutkan ke node yang lebih jauh. Atribut visited memainkan peran penting dalam mencegah pengulangan kunjungan terhadap node yang sama dan menghindari terjadinya loop. Melalui implementasi BFS pada graph tidak berarah, mahasiswa memperoleh pemahaman mengenai cara kerja

penelusuran graph serta bagaimana representasi multilist mendukung proses tersebut. Dengan demikian, latihan ini memberikan dasar yang kuat untuk mempelajari algoritma graph yang lebih kompleks seperti penentuan jarak terpendek, spanning tree, maupun topological sorting.

E. Referensi

Anita Sindar, R. M. S. (2019). *Struktur Data Dan Algoritma Dengan C++ (Vol. 1). CV. AA. RIZKY*.