LAPORAN PRAKTIKUM
Modul 14
GRAPH

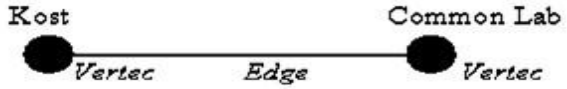
Disusun Oleh:
Alvin Bagus Firmansyah - 2311104070
Kelas
SE-07-02
Dosen :
Wahyu Andi Saputra, S.PD, M.Eng,

PROGRAM STUDI S1 SOFTWARE ENGINEERING
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY
PURWOKERTO
2024

1. **Tujuan**
   1. Memahami konsep *graph*
   2. Mengimplementasikan *graph* dengan menggunakan *pointer*.
2. **Tolls**
   Algoritma Breadth First Search (BFS) dan Depth First Search (DFS)
3. **Lamdasan Teori**

   14.1 *Pengertian*
   Graph merupakan himpunan tidak kosong dari *node* (*vertec*) dan garis penghubung (*edge*). Contoh sederhana tentang *graph*, yaitu antara Tempat Kost Anda dengan

Common Lab. Tempat Kost Anda dan *Common Lab* merupakan *node* (*vertec*). Jalan yang menghubungkan tempat Kost dan *Common Lab* merupakan garis penghubung antara keduanya (*edge*).

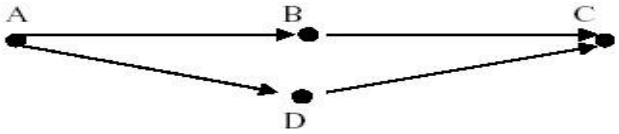


Gambar *Graph* Kost dan *Common Lab*

14.2 *Jenis-Jenis Graph*

14.2.1 Graph Berarah (Directed Graph)

Merupakan *graph* dimana tiap *node* memiliki *edge* yang memiliki arah, kemana *node* tersebut dihubungkan.

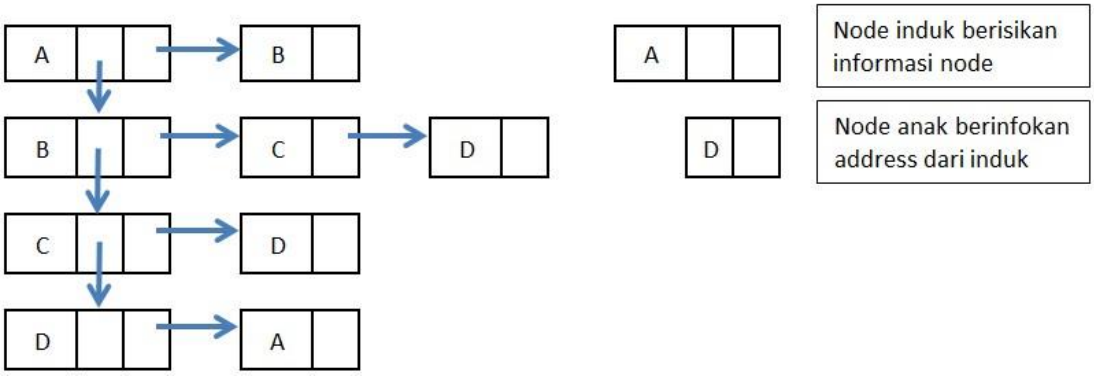


Gambar 14-2 *Graph* Berarah (*Directed Graph*)

A. Representasi *Graph*

Pada dasarnya representasi dari *graph* berarah sama dengan *graph* tak-berarah. Perbedaannya apabila *graph* tak-berarah terdapat *node* A dan *node* B yang terhubung, secara otomatis terbentuk panah bolak balik dari A ke B dan B ke A. Pada *Graph* berarah *node* A terhubung dengan *node* B, belum tentu *node* B terhubung dengan *node* A.

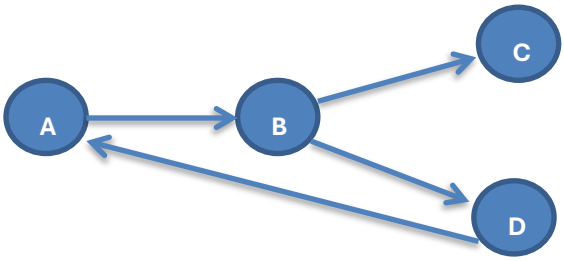
Pada *graph* berarah bisa di representasikan dalam multilist sebagai berikut,



Gambar 14-3 *Graph* Representasi *Multilist*

Dalam praktikum ini untuk merepresentasikan *graph* akan menggunakan *multilist*. Karena sifat *list* yang dinamis.

Dari *multilist* di atas apabila digambarkan dalam bentuk *graph* menjadi :



Gambar 14-4 *Graph* B. *Topological Sort*

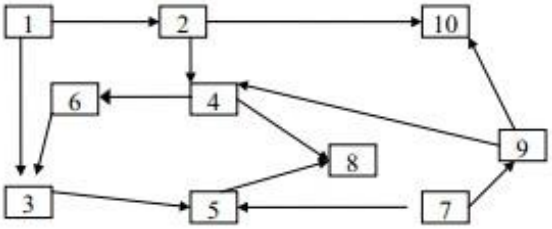
a. Pengertian

Diberikan urutan *partial* dari elemen suatu himpunan, dikehendaki agar elemen yang terurut parsial tersebut mempunyai keterurutan linier. Contoh dari keterurutan parsial banyak dijumpai dalam kehidupan sehari-hari, misalnya:

1. Dalam suatu kurikulum, suatu mata pelajaran mempunyai *prerequisite* mata pelajaran lain. Urutan linier adalah urutan untuk seluruh mata pelajaran dalamkurikulum
2. Dalam suatu proyek, suatu pekerjaan harus dikerjakan lebih dulu dari pekerjaan lain (misalnya membuat fondasi harus sebelum dinding, membuat dinding harus sebelum pintu. Namun pintu dapat dikerjakan bersamaan dengan jendela, dsb
3. Dalam sebuah program Pascal, pemanggilan prosedur harus sedemikian rupa, sehingga peletakan prosedur pada teks program harus seuai dengan urutan (*partial*) pemanggilan.

Dalam pembuatan tabel pada basis data, tabel yang di-*refer* oleh tabel lain harus dideklarasikan terlebih dulu. Jika suatu aplikasi terdiri dari banyak tabel, maka urutan pembuatan tabel harus sesuai dengan definisinya.

Jika X < Y adalah simbol untuk X "sebelum" Y, dan keterurutan *partial* digambarkan sebagai graf, maka graf sebagai berikut :

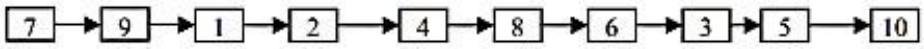


Gambar 14-5 Contoh *Graph*

akan dikatakan mempunyai keterurutan *partial*

1<2 2<4 4<6 2<10 4<8 6<3 1<3
3<5 5<8 7<5 7<9 9<4 9<10

Dan yang SALAH SATU urutan linier adalah graf sebagai berikut :



Gambar 14-6 Urutan Linier *Graph*

Kenapa disebut salah satu urutan linier ? Karena suatu urutan *partial* akan mempunyai banyak urutan linier yang mungkin dibentuk dari urutan *partial* tersebut. Elemen yang membentuk urutan linier disebut sebagai "*list*".

Proses yang dilakukan untuk mendapatkan urutan linier :

1. Andaikata item yang mempunyai keterurutan *partial* adalah anggota himpunan S.
2. Pilih salah satu item yang tidak mempunyai *predecessor*, misalnya X. Minimal adasatu elemen semacam ini. Jika tidak, maka akan looping.
3. Hapus X dari himpunan S, dan *insert* ke dalam *list*
4. Sisa himpunan S masih merupakan himpunan terurut *partial*, maka proses 2-3

dapat dilakukan lagi terhadap sisa dari S

5. Lakukan sampai S menjadi kosong, dan *list* Hasil mempunyai elemen dengan keterurutan linier

Solusi I :

Untuk melakukan hal ini, perlu ditentukan suatu representasi internal. Operasi yang penting adalah memilih elemen tanpa *predecessor* (yaitu jumlah *predecessor* elemen sama dengan nol). Maka setiap elemen mempunyai 3 karakteristik : identifikasi, *list* suksesornya, dan banyaknya *predecessor*. Karena jumlah elemen bervariasi, maka representasi yang paling cocok adalah *list* berkait dengan representasi dinamis (*pointer*). *List* dari *successor* direpresentasi pula secara berkait.

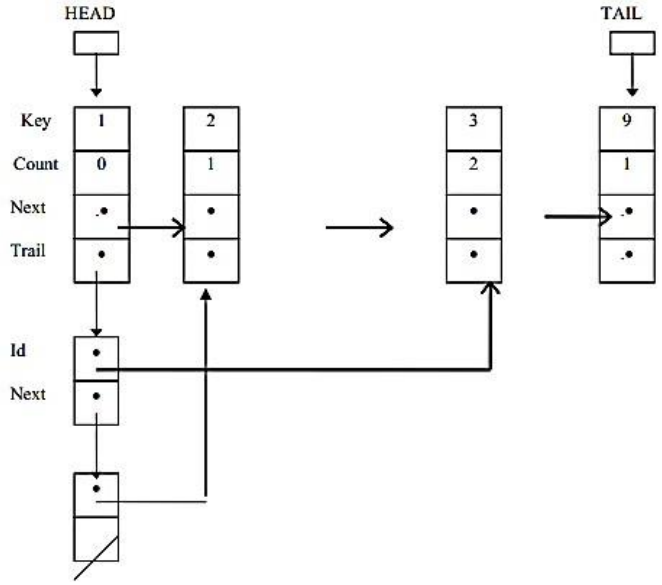
Representasi yang dipilih untuk persoalan ini adalah multilist sebagai berikut :

1. *List* yang digambarkan horisontal adalah *list* dari banyaknya *predecessor* setiap item, disebut *list* "*Leader*", yang direpresentasi sebagai *list* yang dicatat alamat elemen pertama dan terakhir (*Head-Tail*) serta elemen terurut menurut *key*. *List* ini dibentuk dari pembacaan data. Untuk setiap data keterurutan *partial* X < Y : Jika X dan/atau Y belum ada pada *list leader*, *insert* pada *Tail* dengan metoda *search* dengan sentinel.
2. *List* yang digambarkan vertikal (ke bawah) adalah *list* yang merupakan *indirect addressing* ke setiap *predecessor*, disebut sebagai "*Trailer*". Untuk setiap elemen *list Leader* X, *list* dari suksesornya disimpan sebagai elemen *list Trailer* yang setiapelemennya berisi alamat dari *successor*. Penyisipan data suatu *successor* (X < Y), dengan diketahui X, maka akan dilakukan dengan InsertFirst alamat Y sebagai elemen *list Trailer* dengan *key* X.

Algoritma secara keseluruhan terdiri dari dua pass :

1. Bentuk *list leader* dan *Trailer* dari data keterurutan *partial* : baca pasangan nilai (X < Y). Temukan alamat X dan Y (jika belum ada sisipkan), kemudian dengan mengetahui alamat X dan Y pada *list Leader*, InsertFirst alamat Y sebagai *trailer* X
2. Lakukan *topological sort* dengan melakukan *search list Leader* dengan jumlah *predecessor*=0, kemudian *insert* sebagai elemen *list* linier hasil pengurutan.

Ilustrasi umum dari *list Leader* dan *Trailer* untuk representasi internal persoalan *topological sorting* adalah sebagai berikut.



Gambar 14-7 Solusi 1

Solusi II : pendekatan “fungsional” dengan *list* linier sederhana.

Pada solusi ini, proses untuk mendapatkan urutan linier diterjemahkan secara fungsional, dengan representasi sederhana. Graf *partial* dinyatakan sebagai *list* linier dengan representasi fisik *First-Last* dengan *dummy* seperti representasi pada Solusi I. dengan elemen yang terdiri dari <Precc,Succ>.

Contoh: sebuah elemen bernilai <1,2> artinya 1 adalah *predecessor* dari 2.

Langkah :

1. Fase *input*: Bentuk *list* linier yang merepresentasi graf seperti pada solusi I.
2. Fase *output*: Ulangi langkah berikut sampai *list* “habis”, artinya semua elemen *list* selesai ditulis sesuai dengan urutan total.

- P adalah elemen pertama (*First*(L))
- *Search* pada sisa *list*, apakah X=Precc(P) mempunyai *predecessor*.

o Jika ya, maka elemen ini harus dipertahankan sampai saatnya dapat dihapus dari *list* untuk dioutputkan:

- *Delete* P, tapi jangan didealokasi
- *Insert* P sebagai Last(L) yang baru o Jika tidak mempunyai *predecessor*, maka X siap untuk di-*output*-kan, tetapi Y masih harus dipertanyakan. Maka langkah yang harus dilakukan :
- *Output*-kan X
- *Search* apakah Y masih ada pada sisa *list*, baik sebagai Precc maupun Succ o Jika ya, maka Y akan dioutputkan nanti. Hapus elemen pertama yang sedangkan diproses dari *list*

o Jika tidak muncul sama sekali, berarti Y tidak mempunyai *predecessor*. Maka *output*-kan Y, baru hapus elemen pertama dari *list*

b. Representasi *Topological Sort*

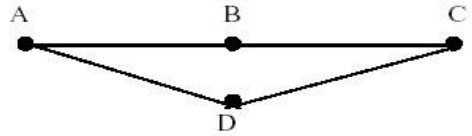
Representasi *graph* untuk *topological sort* sama dengan *graph* berarah pada umumnya.

14.2.2 Graph Tidak Berarah (Undirected Graph)


```
#ifndef GRAPH_H_INCLUDE
#define GRAPH_H_INCLUDE
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int infoGraph; typedef
struct ElmNode *adrNode;
typedef struct ElmEdge *adrEdge;

struct ElmNode{
infoGraph info;  int
Visited;  int Pred;
adrEdge firstEdge;
 adrNode Next;
};
```

```
struct ElmEdge{
adrNode Node;  adrEdge
 Next;
};
struct Graph {
 adrNode First;
};
adrNode AllocateNode (infoGraph X);
 adrEdge AllocateEdge (adrNode N); void CreateGraph (Graph &G);
void InsertNode (Graph &G, infoGraph X);
void DeleteNode (Graph &G, infoGraph X);
void ConnectNode (adrNode N1, adrNode N2);
 void DisconnectNode (adrNode N1, adrNode N2);
 adrNode FindNode (Graph G, infoGraph X);
 adrEdge FindEdge (adrNode N, adrNode NFind);
void PrintInfoGraph (Graph G);
void PrintTopologicalSort (Graph G);

#endif
```

Merupakan *graph* dimana tiap *node* memiliki edge yang dihubungkan ke *node* lain tanpa arah.



Gambar 14-8 *Graph* Tidak Berarah (Undirected *Graph*)

Selain arah, beban atau nilai sering ditambahkan pada *edge* . Misalnya nilai yang merepresentasikan panjang, atau biaya transportasi, dan lain-lain. Hal mendasar lain yang perlu diketahui adalah, suatu *node* A dikatakan bertetangga dengan *node* B jika antara *node* A dan *node* B dihubungkan langsung dengan sebuah *edge*.

Misalnya:

Dari gambar contoh *graph* pada halaman sebelumnya dapat disimpulkan bahwa: A bertetangga dengan B, B bertetangga dengan C, A tidak bertetangga dengan C, B tidak bertetangga dengan D. Masalah ketetanggaan suatu *node* dengan *node* yang lain harus benar-benar

diperhatikan dalam implementasi pada program. Ketetanggaan dapat diartikan sebagai keterhubungan antar *node* yang nantinya informasi ini dibutuhkan untuk melakukan beberapa proses seperti : mencari lintasan terpendek dari suatu *node* ke *node* yang lain, pengubahan *graph* menjadi *tree* (untuk perancangan jaringan) dan lain-lain.

Tentu anda sudah tidak asing dengan algoritma Djikstra, Kruskal, Prim dsb. Karena waktu praktikum terbatas, kita tidak membahas algoritma tersebut. Di sini anda hanya akan mencoba untuk mengimplementasikan *graph* dalam program. A. Representasi

Graph

Dari definisi *graph* dapat kita simpulkan bahwa *graph* dapat direpresentasikan dengan Matrik Ketetanggaan (*Adjacency Matrices*), yaitu matrik yang menyatakan keterhubungan antar *node* dalam *graph*. Implementasi matrik ketetanggaan dalam bahasa pemrograman dapat berupa : *Array* 2 Dimensi dan *Multi Linked List*. *Graph* dapat direpresentasikan dengan matrik n x n, dimana n merupakan jumlah *node* dalam *graph* tersebut.

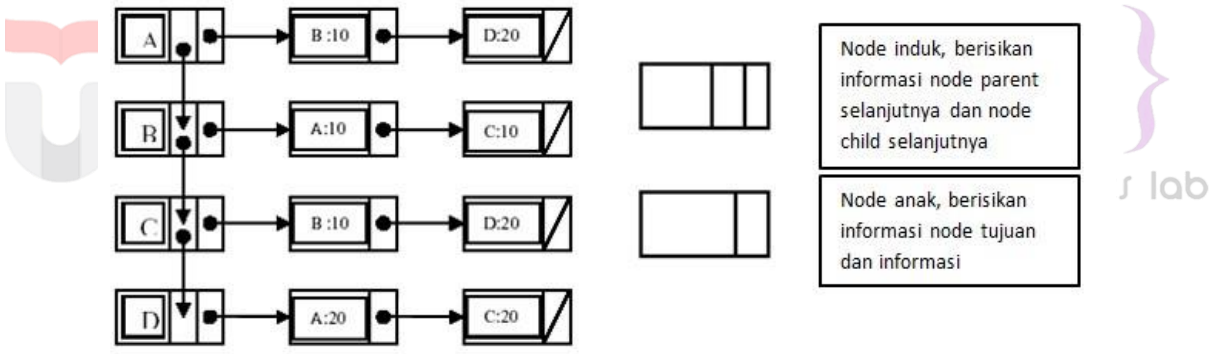
a. *Array* 2 Dimensi

	A	B	C	D
A	-	1	0	1
B	1	-	1	0
C	0	1	-	1
D	1	0	1	-

Keterangan : 1 bertetangga
0 tidak bertetangga

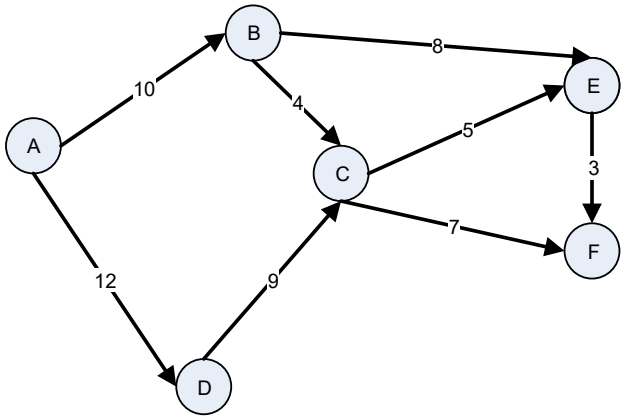
Gambar 14-9 Representasi *Graph Array* 2 Dimensi

b. Multi Linked List



Gambar 14-10 Representasi *Graph Multi Linked list*

Dalam praktikum ini untuk merepresentasikan *graph* akan menggunakan *multi list*. Karena sifat *list* yang dinamis, sehinga data yang bisa ditangani bersifat dinamis. Contoh ada sebuah *graph* yang menggambarkan jarak antar kota:



Gambar 14-11 *Graph* Jarak Antar kota

Gambar *multilist*-nya sama dengan gambar di atas.

Representasi struktur data *graph* pada multilist:

```
        #ifndef GRAPH_H_INCLUDE
        #define GRAPH_H_INCLUDE
        #include <stdio.h>
        #include <stdlib.h>
        #include <conio.h>

        typedef int infoGraph; typedef struct ElmNode *adrNode; typedef struct
        ElmEdge *adrEdge;

        struct ElmNode{  infoGraph info;  int Visited;  adrEdge firstEdge;
         adrNode Next;
        };
        struct ElmEdge{  adrNode Node;  adrEdge Next;
        };
        struct Graph {  adrNode First;
};
```

Berikut adalah contoh fungsi tambah *node* (*addNode*) dan prosedur tambah *edge* (*addEdge*):

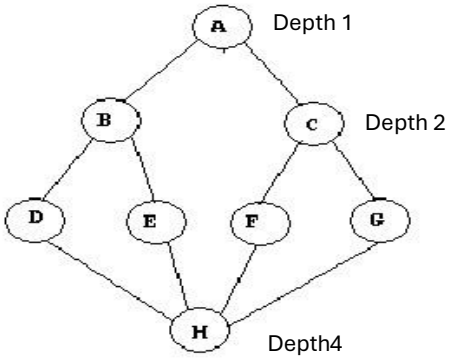
```
      // Adds Node
      ElmNode addNode (infoGraph a, int b, adrEdge c, adrNode d){
            ElmNode newNode;
            newNode.Info = a ;
            newNode.Visited = b ;
            newNode.firstEdge = c ;
            newNode.Next = d ;
            return newNode;
}

// Adds an edge to a graph  void addEdge(ElmNode newNode){  ElmEdge newEdge
;  newEdge.Node = newNode.Next;  newEdge.Next = newNode.firstEdge; }
```

Program 3 *Add newNode* dan *newEdge*

Karena representasinya menggunakan *multilist* maka primitif-primitif *graph* sama dengan primitif primitif pada *multilist*. Jadi untuk membuat ADT *graph* bisa memanfaatkan ADT yang sudah dibuat pada *multilist*.

B. Metode-Metode Penulusuran *Graph* a. Breadth First Search (BFS)

Cara kerja algoritma ini adalah dengan mengunjungi *root* (depth 0) kemudian ke *depth* 1, 2, dan seterusnya. Kunjungan pada masing-masing *level* dimulai dari kiri ke kanan. Secara umum, Algoritma BFS pada *graph* adalah sebagai berikut:




```
Prosedur BFS ( g : graph, start : node )
Kamus
 Q : Queue  x, w : node
Algoritma
 enqueue ( Q, start )  while ( not isEmpty( Q ) ) do   x ß dequeue ( Q )
  if ( isVisited( x ) = false ) then    isVisited( x ) ß true    output ( x )    for each
node w Є Vx
        if ( isVisited( w ) = false ) then                  enqueue( Q, w )
```

Perhatikan *graph* berikut :
Depth 3

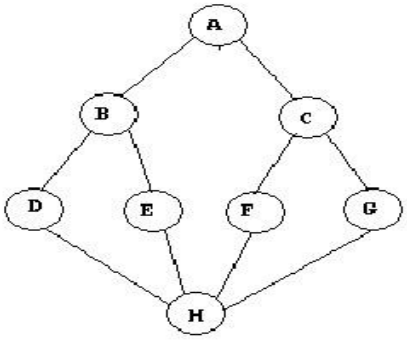
Gambar 14-12 Graph Breadth First Search (BFS)
Urutannya hasil penelusuran BFS : A B C D E F G H

b. Depth First Search (DFS)

Cara kerja algoritma ini adalah dengan mengunjungi *root*, kemudian rekursif ke *subtree node* tersebut.
Secara umum, Algoritma DFS pada *graph* adalah sebagai berikut:

```
Prosedur BFS ( g : graph, start : node )
Kamus
 S : Stack  x, w : node Algoritma  push ( S, start )
 while ( not isEmpty( S ) ) do
  x ß pop ( S )
  if ( isVisited( x ) = false ) then    isVisited( x ) ß true    output ( x )    for each
node w Є Vx
        if ( isVisited( w ) = false ) then                  push ( S, w )
```

Perhatikan *graph* berikut :



Gambar 14-13 Graph Depth First Search (DFS)

Urutannya : A B D H E F C G

4. Guided

1.Kode Progam:

main.cpp

```
#include <iostream>
#include <queue>

using namespace std;

struct ElmNode;

struct ElmEdge {
    ElmNode *Node;
    ElmEdge *Next;
};

struct ElmNode {
    char info;
    bool visited;
    ElmEdge *firstEdge;
    ElmNode *Next;
};

struct Graph {
    ElmNode *first;
};

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

void InsertNode(Graph &G, char X) {
    ElmNode *newNode = new ElmNode;
    newNode->info = X;
    newNode->visited = false;
    newNode->firstEdge = NULL;
    newNode->Next = NULL;

    if (G.first == NULL) {
        G.first = newNode;
    } else {
        ElmNode *temp = G.first;
        while (temp->Next != NULL) {
```

main.cpp

```
            temp = temp->Next;
        }
        temp->Next = newNode;
    }
}

void ConnectNode(ElmNode *N1, ElmNode *N2) {
    ElmEdge *newEdge = new ElmEdge;
    newEdge->Node = N2;
    newEdge->
    N1->firstEdge = newEdge;
}

void PrintInfoGraph(Graph G) {
    ElmNode *temp = G.first;
    while (temp != NULL) {
        cout << temp->info << " ";
        temp = temp->Next;
    }
    cout << endl;
}

void ResetVisited(Graph &G) {
    ElmNode *temp = G.first;
    while (temp != NULL) {
        temp->visited = false;
        temp = temp->Next;
    }
}

void PrintDFS(Graph G, ElmNode *N) {
    if (N == NULL) {
        return;
    }
    N->visited = true;
    cout << N->info << " ";
    ElmEdge *edge = N->firstEdge;
    while (edge != NULL) {
        if (!edge->Node->visited) {
```

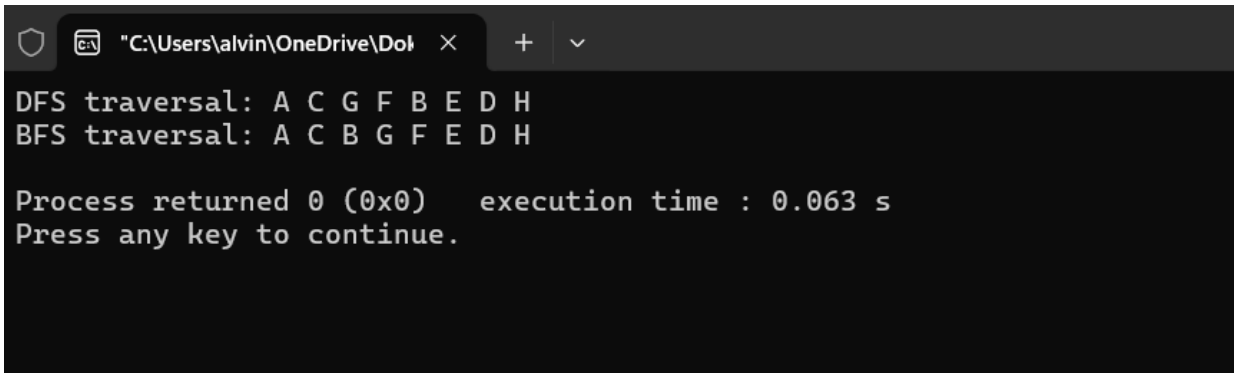
ElmNode* ElmEdge::Node

main.cpp

```
            PrintDFS(G, edge->Node);
        }
        edge = edge->Next;
    }
}

void PrintBFS(Graph G, ElmNode *N) {
    queue<ElmNode*> q;
    q.push(N);
    N->visited = true;

    while (!q.empty()) {
        ElmNode *current = q.front();
        q.pop();
        cout << current->info << " ";

        ElmEdge *edge = current->firstEdge;
        while (edge != NULL) {
            if (!edge->Node->visited) {
                edge->Node->visited = true;
                q.push(edge->Node);
            }
            edge = edge->Next;
        }
    }
}

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

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

    ElmNode *A = G.first;
    ElmNode *B = A->Next;
    ElmNode *C = B->Next;
    ElmNode *D = C->Next;
    ElmNode *E = D->Next;
    ElmNode *F = E->Next;
    ElmNode *G1 = F->Next;
    ElmNode *H = G1->Next;

    ConnectNode(A, B);
    ConnectNode(A, C);
    ConnectNode(B, D);
    ConnectNode(B,
    ConnectNode(C, F);
    ConnectNode(C, G1);
    ConnectNode(D, H);

    cout << "DFS traversal: ";
    ResetVisited(G);
    PrintDFS(G, A);
    cout << endl;

    cout << "BFS traversal: ";
    ResetVisited(G);
    PrintBFS(G, A);
    cout << endl;

    return 0;
}
```

ElmNode* main::D

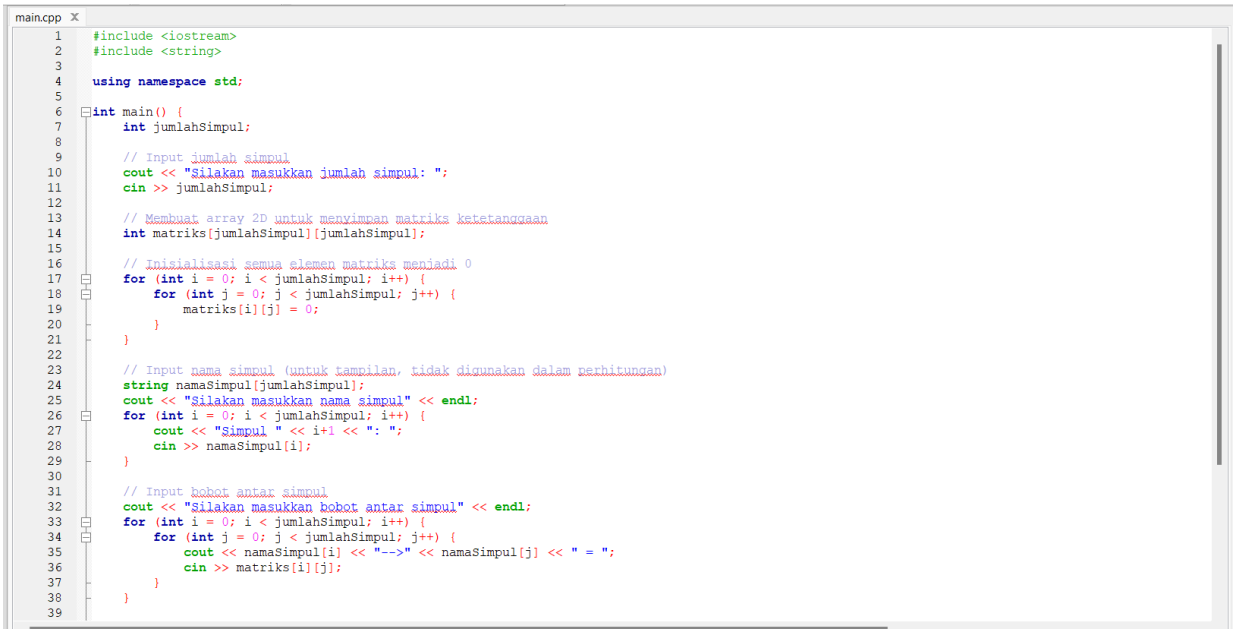
Hasil Outputnya:



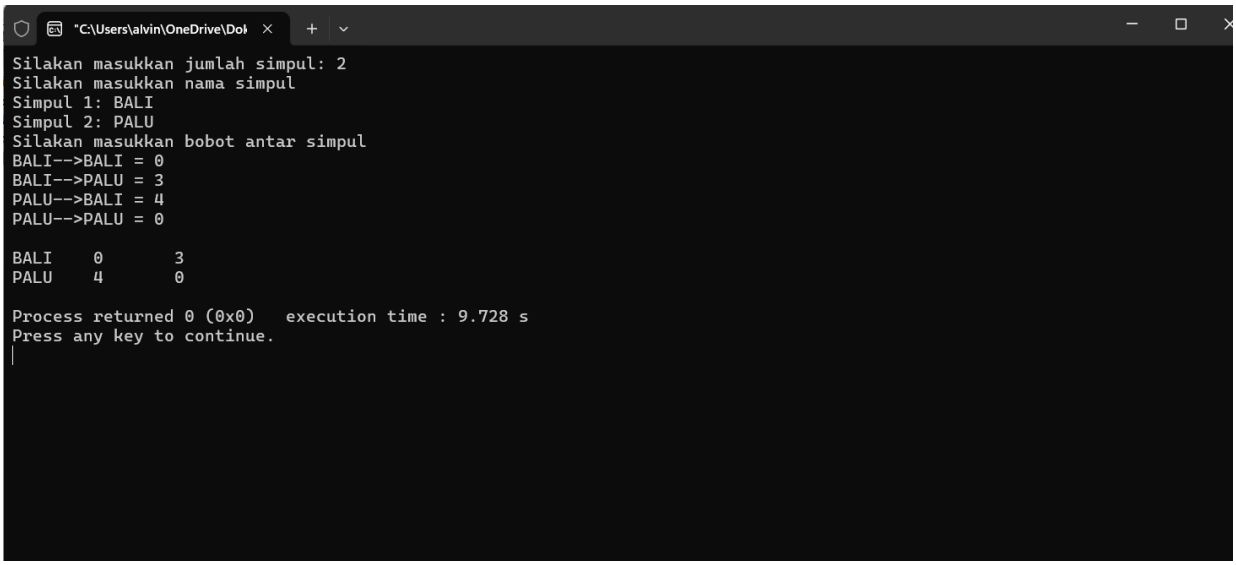
5. Unguided

1. Program Graph untuk Menghitung Jarak antar Kota (Dengan Input dari User)

   Kode Progam:



   Hasil Outputnya:



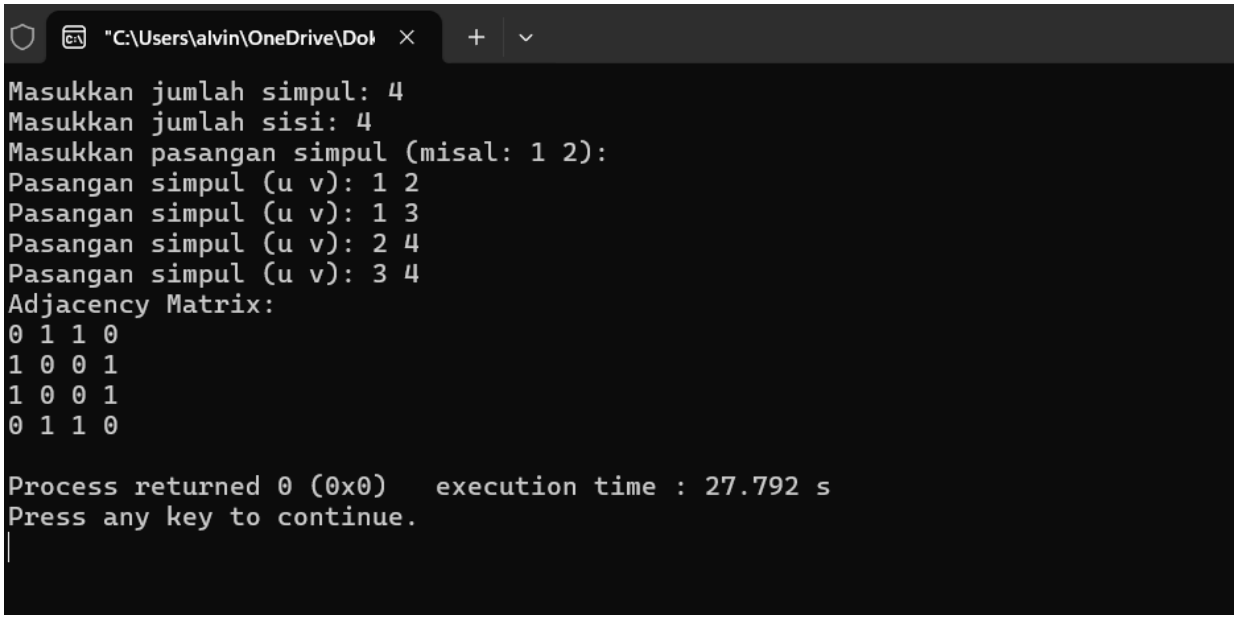
2. Program C++ untuk Merepresentasikan Graf Tidak Berarah Menggunakan Adjacency Matrix

Kode Progam:

main.cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n, m;
    cout << "Masukkan jumlah simpul: ";
    cin >> n;

    cout << "Masukkan jumlah sisi: ";
    cin >> m;

    // Matriks adjacency untuk graf tidak berarah
    vector<vector<int>> adjMatrix(n, vector<int>(n, 0));

    cout << "Masukkan pasangan simpul (misal: 1 2):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cout << "Pasangan simpul (u v): ";
        cin >> u >> v;

        // Karena simpul yang dimasukkan adalah 1-indexed, kita kurangi 1 untuk indexing 0
        adjMatrix[u - 1][v - 1] = 1;
        adjMatrix[v - 1][u - 1] = 1;  // Karena graf tidak berarah
    }

    cout << "Adjacency Matrix:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```


Hasil Outputnya:



6. **Kesimpulan**

   Dalam praktikum ini, telah dipelajari mengenai konsep dan implementasi graph, termasuk berbagai jenis graph seperti graph berarah dan tidak berarah, serta algoritma-algoritma yang digunakan untuk memanipulasi graph. Representasi graph dapat dilakukan dengan berbagai cara, termasuk menggunakan matriks ketetanggaan atau multilist, yang masing-masing memiliki kelebihan dan kekurangannya. Selain itu, algoritma penelusuran graph seperti Breadth First Search (BFS) dan Depth First Search (DFS) juga dipelajari untuk mengakses dan mencari elemen-elemen dalam graph. Dengan implementasi menggunakan bahasa pemrograman seperti C atau C++, praktikum ini memberikan pemahaman yang lebih dalam tentang penerapan graph dalam pemecahan masalah nyata, seperti perhitungan jarak antar kota dan penataan urutan dalam topological sort.