

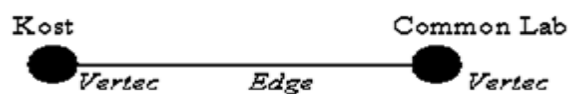
Modul 14 GRAPH

TUJUAN PRAKTIKUM

1. Memahami konsep *graph*
2. Mengimplementasikan *graph* dengan menggunakan *pointer*.

14.1 Pengertian

Graph merupakan himpunan tidak kosong dari *node* (*vertex*) dan garis penghubung (*edge*). Contoh sederhana tentang *graph*, yaitu antara Tempat Kost Anda dengan *Common Lab*. Tempat Kost Anda dan *Common Lab* merupakan *node* (*vertex*). Jalan yang menghubungkan tempat Kost dan *Common Lab* merupakan garis penghubung antara keduanya (*edge*).

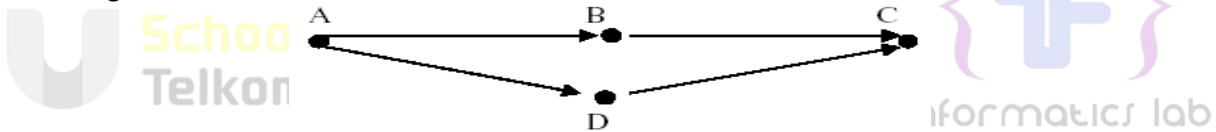


Gambar 14-1 *Graph* Kost dan *Common Lab*

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.

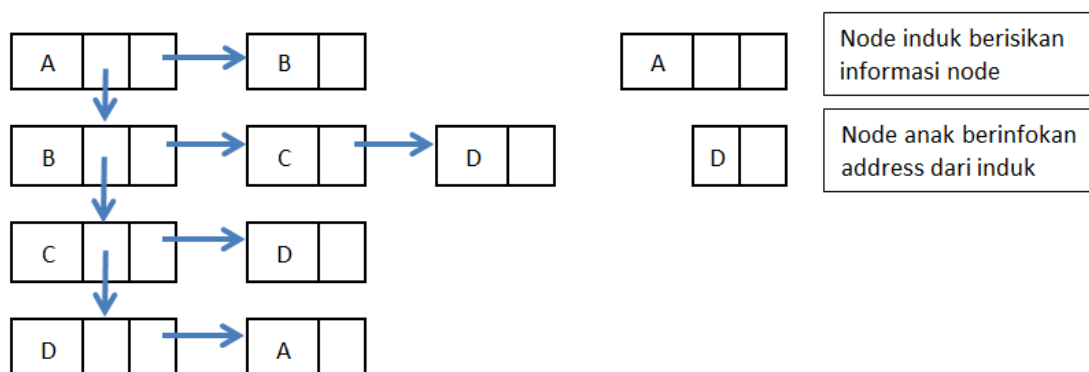


Gambar 14-2 *Graph* Berarah (*Directed Graph*)

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.

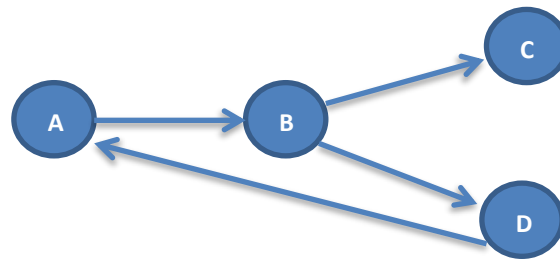
Pada *graph* berarah bisa di representasikan dalam multilist sebagai berikut,



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 dalam kurikulum
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 sesuai 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 :

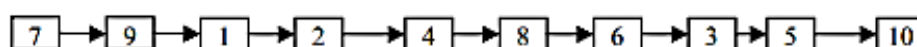


Gambar 14-5 Contoh *Graph*

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* berkaitan dengan representasi dinamis (*pointer*). *List* dari *successor* direpresentasi pula secara berkait.

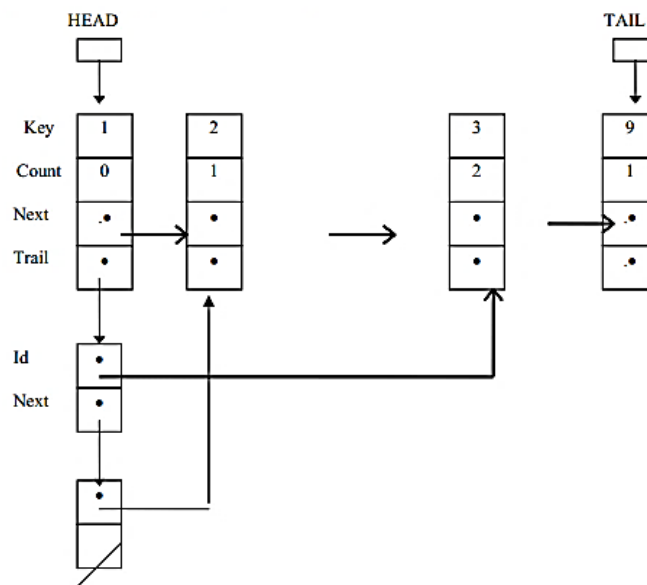
Representasi yang dipilih untuk persoalan ini adalah multilist sebagai berikut :

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 setiap elemennya 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 $X = \text{Precc}(P)$ mempunyai *predecessor*.
 - 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
 - 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
 - Jika ya, maka Y akan dioutputkan nanti. Hapus elemen pertama yang sedangkan diproses dari *list*
 - 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.

```
1 #ifndef GRAPH_H_INCLUDE
2 #define GRAPH_H_INCLUDE
3 #include <stdio.h>
```

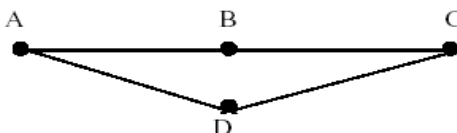
```

4  #include <stdlib.h>
5  #include <conio.h>
6
7  typedef int infoGraph;
8  typedef struct ElmNode *adrNode;
9  typedef struct ElmEdge *adrEdge;
10
11 struct ElmNode{
12     infoGraph info;
13     int Visited;
14     int Pred;
15     adrEdge firstEdge;
16     adrNode Next;
17 };
18 struct ElmEdge{
19     adrNode Node;
20     adrEdge Next;
21 };
22 struct Graph {
23     adrNode First;
24 };
25
26 adrNode AllocateNode (infoGraph X);
27 adrEdge AllocateEdge (adrNode N);
28 void CreateGraph (Graph &G);
29 void InsertNode (Graph &G, infoGraph X);
30 void DeleteNode (Graph &G, infoGraph X);
31 void ConnectNode (adrNode N1, adrNode N2);
32 void DisconnectNode (adrNode N1, adrNode N2);
33 adrNode FindNode (Graph G, infoGraph X);
34 adrEdge FindEdge (adrNode N, adrNode NFind);
35 void PrintInfoGraph (Graph G);
36 void PrintTopologicalSort (Graph G);
37
38 #endif

```

14.2.2 Graph Tidak Berarah (Undirected Graph)

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, perubahan *graph* menjadi *tree* (untuk perancangan jaringan) dan lain-lain.

Tentu anda sudah tidak asing dengan algoritma Dijkstra, 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 $n \times n$, dimana n merupakan jumlah *node* dalam *graph* tersebut.

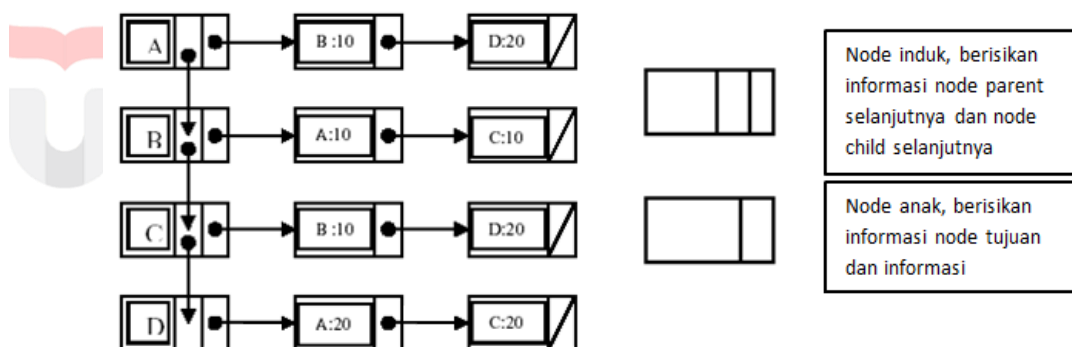
a. Array 2 Dimensi

	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

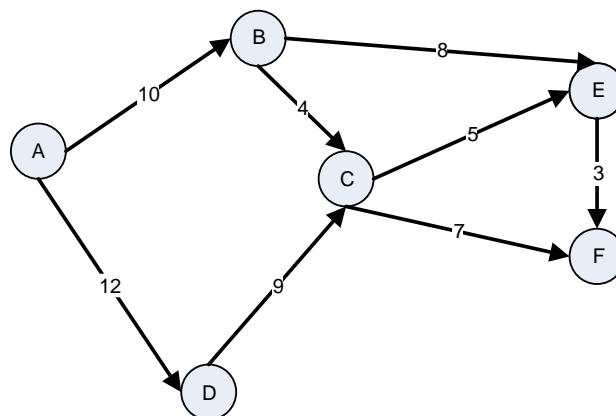
Gambar 14-9 Representasi Graph Array 2 Dimensi

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, sehingga 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*:

```
1  #ifndef GRAPH_H INCLUDE
2  #define GRAPH_H INCLUDE
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <conio.h>
6
7  typedef int infoGraph;
8  typedef struct ElmNode *adrNode;
9  typedef struct ElmEdge *adrEdge;
10
11 struct ElmNode{
12     infoGraph info;
13     int Visited;
14     adrEdge firstEdge;
15     adrNode Next;
16 };
17 struct ElmEdge{
18     adrNode Node;
19     adrEdge Next;
20 };
21 struct Graph {
22     adrNode First;
23 };
```

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

```
1  // Adds Node
2  ElmNode addNode (infoGraph a, int b, adrEdge c, adrNode d){
3      ElmNode newNode;
4      newNode.Info = a ;
5      newNode.Visited = b ;
6      newNode.firstEdge = c ;
7      newNode.Next = d ;
8      return newNode;
9  }
10
11 // Adds an edge to a graph
12 void addEdge(ElmNode newNode){
13     ElmEdge newEdge ;
14     newEdge.Node = newNode.Next;
15     newEdge.Next = newNode.firstEdge;
16 }
```

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 Penelusuran *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:

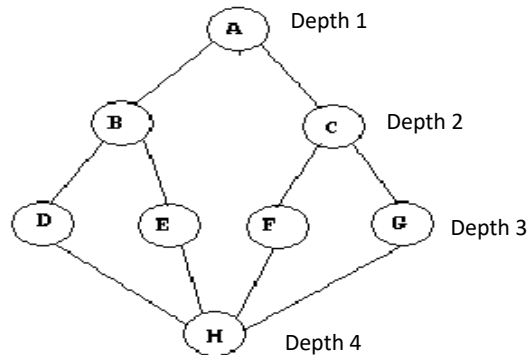
```
Prosedur BFS ( g : graph, start : node )
Kamus
    Q : Queue
    x, w : node
Algoritma
```

```

enqueue ( Q, start )
while ( not isEmpty( Q ) ) do
  x ← dequeue ( Q )
  if ( isVisited( x ) = false ) then
    isvisited( x ) ← true
    output ( x )
    for each node w ∈ vx
      if ( isVisited( w ) = false ) then
        enqueue( Q, w )

```

Perhatikan *graph* berikut :



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.

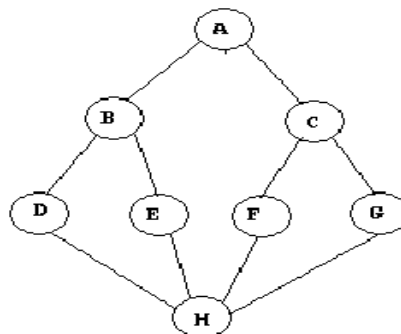
Secara umum, Algoritma DFS pada *graph* adalah sebagai berikut:

```

Prosedur DFS ( g : graph, start : node )
Kamus
  S : Stack
  x, w : node
Algoritma
  push ( S, start )
  while ( not isEmpty( S ) ) do
    x ← pop ( S )
    if ( isvisited( x ) = false ) then
      isvisited( x ) ← true
      output ( x )
      for each node w ∈ 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

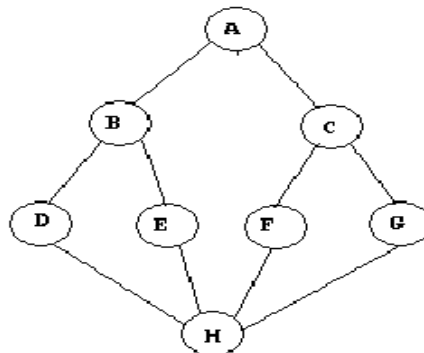
14.3 Latihan

1. Buatlah ADT *Graph* tidak berarah file “**graph.h**”:

```
Type infoGraph: char
Type adrNode : pointer to ElmNode
Type adrEdge : pointer to ElmNode
Type ElmNode <
    info : infoGraph
    visited : integer
    firstEdge : adrEdge
    Next : adrNode
>
Type ElmEdge <
    Node : adrNode
    Next : adrEdge
>
Type Graph <
    first : adrNode
>
prosedur CreateGraph (Graph &G);
prosedur InsertNode (Graph &G, infotype X);
prosedur ConnectNode (adrNode N1, adrNode N2);
prosedur PrintInfoGraph (Graph G);
```

Program 4 *Graph.h*

Buatlah implementasi ADT *Graph* pada file “**graph.cpp**” dan cobalah hasil implementasi ADT pada file “**main.cpp**”.



Gambar 14-14 Ilustrasi *Graph*

2. Buatlah prosedur untuk menampilkan hasil penelusuran DFS.
prosedur PrintDFS (*Graph* G, adrNode N);
3. Buatlah prosedur untuk menampilkan hasil penelusuran DFS.
prosedur PrintBFS (*Graph* G, adrNode N);