Nama : Hafidh Akhdan N
Kelas : A
NPM : 140810180061

Analgo 6

1. Matriks Adjacency

```
/*
        Nama    : Hafidh Akhdan N
        Kelas   : A
        NPM             : 140810180061
        Matriks Adjacency
*/

#include <iostream>
using namespace std;

int vertArr[20][20];
int count = 0;

void displayMatrix(int v){
   int i, j;
   for (i = 1; i <= v; i++){
      for (j = 1; j <= v; j++)
      {
         cout << vertArr[i][j] << " ";
      }
      cout << endl;
   }
}

void add_edge(int u, int v){
   vertArr[u][v] = 1;
   vertArr[v][u] = 1;
}

int main(int argc, char *argv[]){
   int v;
   cout << "Masukkan jumlah matrix : "; cin >> v;

   int pilihan,a,b;
   while(true){
      cout << "1. Tambah edge " << endl;
      cout << "2. Print " << endl;
      cout << "3. Exit " << endl;
      cout << "Masukan pilihan : "; cin >> pilihan;
      if (pilihan==1){
           cout << "Masukkan node A : "; cin >> a;
            cout << "Masukkan node B : "; cin >> b;
```

```
            add_edge(a,b);
            cout << "Edge telah ditambahkan\n";
            system("Pause");
            system("CLS");
                    } else if(pilihan==2){
                            displayMatrix(v);
            system("Pause");
            system("CLS");
                    } else{
                            return 0;
                    }
    }
}
```

2. List Adjacency

```
/*
        Nama    : Hafidh Akhdan N
        Kelas   : A
        NPM             : 140810180061
        List Adjacency
*/

#include <iostream>
#include <cstdlib>
using namespace std;

struct AdjListNode{
   int dest;
   struct AdjListNode* next;
};

struct AdjList{
   struct AdjListNode *head;
};

class Graph{
   private:
      int V;
      struct AdjList* array;
   public:
      Graph(int V){
         this->V = V;
         array = new AdjList [V];
         for (int i = 1; i <= V; ++i)
            array[i].head = NULL;
      }

      AdjListNode* newAdjListNode(int dest){
```

```
        AdjListNode* newNode = new AdjListNode;
        newNode->dest = dest;
        newNode->next = NULL;
        return newNode;
    }

    void addEdge(int src, int dest){
        AdjListNode* newNode = newAdjListNode(dest);
        newNode->next = array[src].head;
        array[src].head = newNode;
        newNode = newAdjListNode(src);
        newNode->next = array[dest].head;
        array[dest].head = newNode;
    }

    void printGraph(){
        int v;
        for (v = 1; v <= V; ++v){
            AdjListNode* pCrawl = array[v].head;
            cout << "\n Adjacency list of vertex " << v << "\n head ";
            while (pCrawl){
                cout<<"-> "<<pCrawl->dest;
                pCrawl = pCrawl->next;
            }
            cout<<endl;
        }
    }
};


int main(){
    int pilihan,a,b,n;
    cout << "Banyak node : "; cin >> n;
    Graph gh(n);
    for(; ;){
        cout << "\nMenu\n";
        cout << "1. Tambah edge\n";
        cout << "2. Print Edge\n";
        cout << "3. Exit\n\n";
        cout << "Pilihan : "; cin >> pilihan;

        switch (pilihan){
            case 1:
                cout << "\nedge(a,b)\n";
               cout << "Input a : "; cin >> a;
                cout << "Input b : "; cin >> b;
                gh.addEdge(a,b);
                continue;
```

```
            case 2:
              gh.printGraph();
              continue;
            case 3:
              return 0;
              break;
            default:
              continue;
          }
        }

        return 0;
    }
```

3. BFS

```
    /*
            Nama    : Hafidh Akhdan N
            Kelas   : A
            NPM             : 140810180061
            BFS
    */
    #include<iostream>
    #include <list>

    using namespace std;

    class Graph{
            int V; // No. of vertices

            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 << "Breadth First Traversal ";
        cout << "(starting from vertex 1) \n";
        g.BFS(1);

        return 0;
}
```

• BFS merupakan metode pencarian secara melebar sehingga mengunjungi node dari kiri ke kanan di level yang sama. Apabila semua node pada suatu level sudah dikunjungi semua, maka akan berpindah ke level selanjutnya. Dalam worst case BFS harus mempertimbangkan semua jalur (path) untuk semua node yang mungkin, maka nilai kompleksitas waktu dari BFS adalah O(|V| + |E|).
• Karena Big-O dari BFS adalah O(V+E) dimana V itu jumlah vertex dan E itu adalah jumlah edges maka Big-O = O(n) dimana n = v+e
• Maka dari itu Big-Θ nya adalah Θ(n).

4. DFS

```
/*
        Nama    : Hafidh Akhdan N
        Kelas   : A
        NPM             : 140810180061
        DFS
*/

#include<iostream>
#include<list>
using namespace std;

class Graph{
  int V; // No. of vertices

  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(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 << "Depth First Traversal";
        cout << " (starting from vertex 1) \n";
    g.DFS(1);

    return 0;
}
```

- DFS merupakan metode pencarian mendalam, yang mengunjungi semua node dari yang terkiri lalu geser ke kanan hingga semua node dikunjungi. Kompleksitas ruang algoritma DFS adalah O(bm), karena kita hanya hanya perlu menyimpan satu buah lintasan tunggal dari akar sampai daun, ditambah dengan simpul-simpul saudara kandungnya yang belum dikembangkan.
- Big O Kompleksitas total DFS () adalah (V+E). O(n) dengan V = Jumlah Verteks dan E = Jumlah Edges