**LAPORAN**

**PRAKTIKUM ANALISIS ALGORITMA**

**TUGAS 05**



**Disusun Oleh:**

Risyad Pangestu (140810170003)

Dimas Satria Prakoso (140810170007)

Syaina Nur Fauziyah (140810170025)

Imron Madani (140810170061)

**PROGRAM STUDI TEKNIK INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS PADJADJARAN**

**5SUMEDANG**

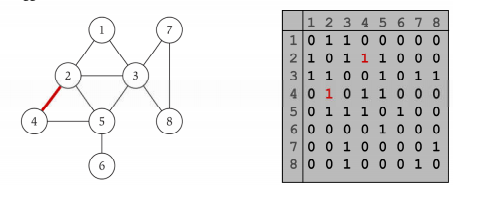
**2019**

**LAPORAN**

**PRAKTIKUM ANALISIS ALGORITMA**

**TUGAS 05**

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



**Program:**

/\*

\* C++ Program to Implement Adjacency Matrix

\*/

#include <iostream>

#include <cstdlib>

using namespace std;

#define MAX 20

/\*

\* Adjacency Matrix Class

\*/

class AdjacencyMatrix

{

private:

int n;

int \*\*adj;

bool \*visited;

public:

AdjacencyMatrix(int n)

{

this->n = n;

visited = new bool [n];

adj = new int\* [n];

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

{

adj[i] = new int [n];

for(int j = 0; j < n; j++)

{

adj[i][j] = 0;

}

}

}

/\*

\* Adding Edge to Graph

\*/

void add\_edge(int origin, int destin)

{

if( origin > n || destin > n || origin < 0 || destin < 0)

{

cout<<"Invalid edge!\n";

}

else

{

adj[origin - 1][destin - 1] = 1;

}

}

/\*

\* Print the graph

\*/

void display()

{

int i,j;

for(i = 0;i < n;i++)

{

for(j = 0; j < n; j++)

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

cout<<endl;

}

}

};

/\*

\* Main

\*/

int main()

{

int nodes, max\_edges, origin, destin;

cout<<"Enter number of nodes: ";

cin>>nodes;

AdjacencyMatrix am(nodes);

max\_edges = nodes \* (nodes - 1);

for (int i = 0; i < max\_edges; i++)

{

cout<<"Enter edge (-1 -1 to exit): ";

cin>>origin>>destin;

if((origin == -1) && (destin == -1))

break;

am.add\_edge(origin, destin);

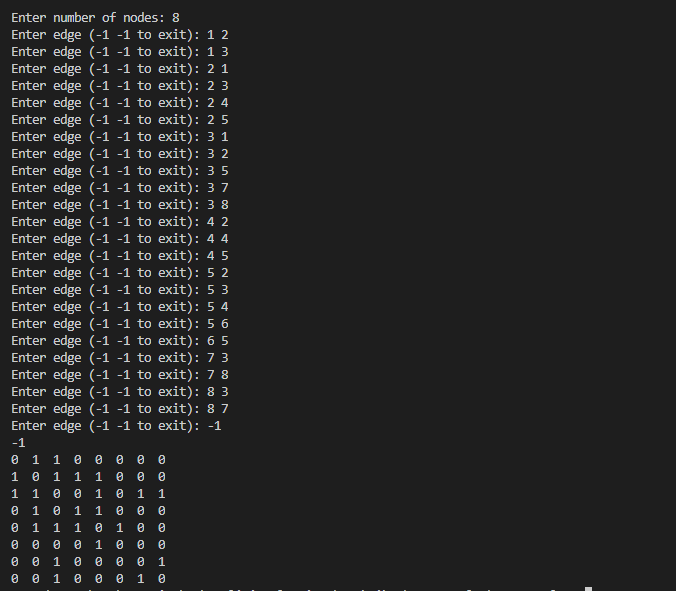
}

am.display();

return 0;

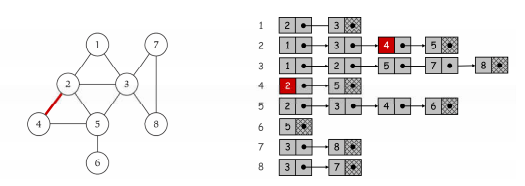
}

**Output:**





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



**Program:**

/\*

\* C++ Program to Implement Adjacency List

\*/

#include <iostream>

#include <cstdlib>

using namespace std;

/\*

\* Adjacency List Node

\*/

struct AdjListNode

{

int dest;

struct AdjListNode\* next;

};

/\*

\* Adjacency List

\*/

struct AdjList

{

struct AdjListNode \*head;

};

/\*

\* Class Graph

\*/

class Graph

{

private:

int V;

struct AdjList\* array;

public:

Graph(int V)

{

this->V = V;

array = new AdjList [V];

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

array[i].head = NULL;

}

/\*

\* Creating New Adjacency List Node

\*/

AdjListNode\* newAdjListNode(int dest)

{

AdjListNode\* newNode = new AdjListNode;

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

/\*

\* Adding Edge to Graph

\*/

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;

}

/\*

\* Print the graph

\*/

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;

}

}

};

/\*

\* Main

\*/

int main()

{

Graph gh(8);

gh.addEdge(1, 2);

gh.addEdge(1, 3);

gh.addEdge(2, 4);

gh.addEdge(2, 5);

gh.addEdge(2, 3);

gh.addEdge(3, 7);

gh.addEdge(3, 8);

gh.addEdge(4, 5);

gh.addEdge(5, 3);

gh.addEdge(5, 6);

gh.addEdge(7, 8);

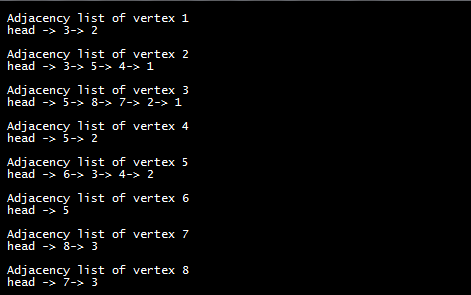
// print the adjacency list representation of the above graph

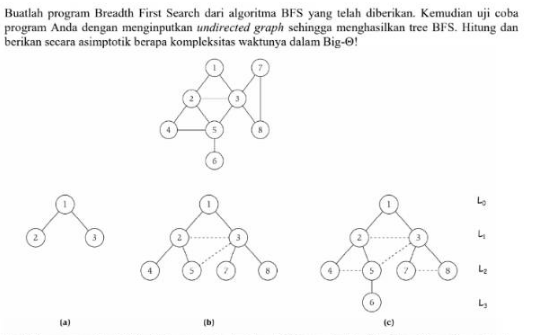
gh.printGraph();

return 0;

}

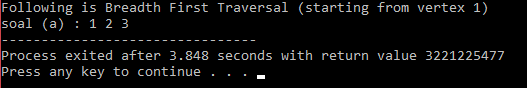
Output :

****



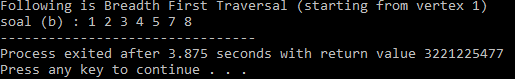
(a)

|  |
| --- |
| // 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(3);  g.addEdge(1,2);  g.addEdge(1,3);    cout << "Following is Breadth First Traversal "  << "(starting from vertex 2) \n"  << "soal (a) : ";  g.BFS(1);    return 0;  } |



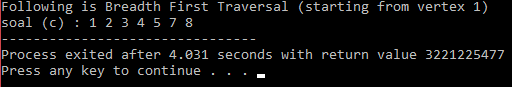
(b)

|  |
| --- |
| // 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 f(8);  f.addEdge(1,2);  f.addEdge(1,3);  f.addEdge(2,3);  f.addEdge(2,4);  f.addEdge(2,5);  f.addEdge(3,5);  f.addEdge(3,7);  f.addEdge(3,8);    cout << "Following is Breadth First Traversal "  << "(starting from vertex 2) \n"  << "soal (b) : ";  f.BFS(1);    return 0;  } |



(c)

|  |
| --- |
| // 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 e(8);  e.addEdge(1,2);  e.addEdge(1,3);  e.addEdge(2,3);  e.addEdge(2,4);  e.addEdge(2,5);  e.addEdge(3,7);  e.addEdge(3,8);  e.addEdge(4,5);  e.addEdge(5,6);  e.addEdge(7,8);    cout << "Following is Breadth First Traversal "  << "(starting from vertex 2) \n"  << "soal (c) : ";  e.BFS(1);    return 0;  } |



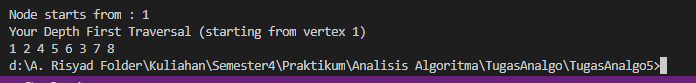
Big O dari algoritma BFS O(N) dengan N=V+E dimana V=jumlah Vertex ,E=jumlah Edges



Code

|  |
| --- |
| #include<iostream>  #include<list>  using namespace std;  class Graph  {  int V;  list<int> \*adj;  void DFSUtil(int v, bool visited[]);  public:  Graph(int V);  void addEdge(int v, int w);  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);  }    void Graph::DFSUtil(int v, bool visited[])  {  visited[v] = true;  cout << v << " ";  list<int>::iterator i;  for (i = adj[v].begin(); i != adj[v].end(); ++i)  if (!visited[\*i])  DFSUtil(\*i, visited);  }    void Graph::DFS(int v)  {  bool \*visited = new bool[V];  for (int i = 0; i < V; i++)  visited[i] = false;  DFSUtil(v, visited);  }    int main()  {  int node,start;  cout<<"Input the amount of your nodes : ";cin>>node;  Graph g(node);  cout<<"Instructions :"<<endl;  cout<<"1. Enter the number of nodes from 0 to n-1"<<endl;  cout<<"2. Enter negative numbers (such as -1) on either node input to to exit the program"<<endl;  for(;;){  int node1,node2;  cout<<"Enter number between "<<0<<" to "<<node-1<<endl;  cout<<"Input node 1 : ";cin>>node1;  cout<<"Input node 2 : ";cin>>node2;  if(node1>=0&&node2>=0&&node1<node&&node2<node){  g.addEdge(node1,node2);  cout<<endl;  }  else if(node1<0||node2<0)  break;  else  cout<<"Wrong input. Please enter again"<<endl;  }  back:  cout<<"\nNode starts from : ";cin>>start;  if(start<0||start>node-1){  cout<<"Wrong input. Please enter again"<<endl;  goto back;  }  cout<<"Your Depth First Traversal (starting from vertex "<<start<<")"<<endl;  g.DFS(start);  return 0;  } |

Hasil Program :



Kompleksitas Bio O dari DFS 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.