مقرر خوارزميات و بنى المعطيات2 - جلسة العملي الأولى

خوارزميات البيان (graph algorithm)

خوارزمية تمثيل البيان:

يتم تمثيل البيان باستخدام قوائم التجاور ، وبنية المعطيات التي تستخدم لتمثيل قوائم التجاور برمجياً هي السلاسل المترابطة linked list .

```
#include <iostream>
using namespace std;
// stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};
class DiaGraph{
    // insert new nodes into adjacency list from given
graph
    adjNode* getAdjListNode(int value, int weight,
adjNode* head)   {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;
        newNode->next = head;   // point new node to
current head
        return newNode;
    }

public:
    adjNode **head;
    int N;          // number of nodes in the graph
     //adjacency list as array of pointers
    // Constructor
    DiaGraph(graphEdge edges[], int n, int N)  {
        // allocate new node
        head = new adjNode*[N]();
        this->N = N;
        // initialize head pointer for all vertices
```

```
        for (int i = 0; i < N; ++i)
            head[i] = NULL;
        // construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++)  {
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;
            // insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver,
weight, head[start_ver]);

             // point head pointer to new node
            head[start_ver] = newNode;
             }
    }
      // Destructor
     ~DiaGraph() {
    for (int i = 0; i < N; i++)
        delete[] head[i];
        delete[] head;
     }
};
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != NULL) {
        cout << "(" << i << ", " << ptr->val
            << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}
// graph implementation
int main()
{
    // graph edges array.
   // (x, y, w) -> edge from x to y with weight w

    graphEdge edges[] = {
        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
    };
    int N = 5;      // Number of vertices in the graph
    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);
```

```
    // construct graph
    DiaGraph diagraph(edges, n, N);
    // print adjacency list representation of graph
    cout<<"Graph adjacency list "<<endl<<"(start_vertex,
end_vertex, weight):"<<endl;
    for (int i = 0; i < N; i++)
    {
        // display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }
    return 0;
}
```

خوارزمية البحث في العمق (Depth First Search) :

يتم تطبيق الخوارزمية برمجياً عن طريق الاستدعاء العودي .

```
#include <iostream>
#include <list>
using namespace std;
//graph class for DFS travesal
class DFSGraph
{
int V;    // No. of vertices
list<int> *adjList;    // adjacency list
// A function used by DFS
void apply(int v,int pred, bool visited[]);
public:
// class Constructor
DFSGraph(int V)
    {
 this->V = V;
 adjList = new list<int>[V];
    }
// function to add an edge to graph
void addEdge(int v, int w){
adjList[v].push_back(w); // Add w to v's list
    }

void DFS();    // DFS traversal function
};
void DFSGraph::apply(int v,int pred, bool visited[])
{
// current node v is visited
visited[v] = true;
if(pred==v)
    cout<<"\n new tree root="<<v<<" edeges : ";
    else
cout << pred<<v<< " ";

// recursively process all the adjacent vertices of the
node
list<int>::iterator i;
for(i = adjList[v].begin(); i != adjList[v].end(); ++i)
if(!visited[*i])
```

```
apply(*i,v, visited);
}

// DFS traversal
void DFSGraph::DFS()
{
    // initially none of the vertices are visited
bool *visited = new bool[V];
for (int i = 0; i < V; i++){
visited[i] = false;

}
// explore the vertices one by one by recursively calling
apply
for (int i = 0; i < V; i++)
if (visited[i] == false)
apply(i,i, visited);
}

int main()
{
// Create a graph
DFSGraph gdfs(7);
gdfs.addEdge(0,1);
gdfs.addEdge(0,6);
gdfs.addEdge(1,2);
gdfs.addEdge(1,5);
gdfs.addEdge(2,0);
gdfs.addEdge(3,2);
gdfs.addEdge(4,2);
gdfs.addEdge(4,3);
gdfs.addEdge(5,4);
gdfs.addEdge(6,3);
cout << "Depth-first traversal for the given
graph:"<<endl;
gdfs.DFS();

return 0;
}
```

خوارزمية البحث في العرض (Breadth First Search) :

يتم تطبيق الخوارزمية برمجياً باستخدام الرتل .

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

// This class represents a directed graph using adjacency
list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing
adjacency lists
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // function to add an
edge to graph
    void BFS(int s);  // prints BFS traversal from a
given source 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(7);
g.addEdge(0,1);
g.addEdge(0,6);
g.addEdge(1,2);
g.addEdge(1,5);
g.addEdge(2,0);
g.addEdge(3,2);
```

```
g.addEdge(4,2);
g.addEdge(4,3);
g.addEdge(5,4);
g.addEdge(6,3);
    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(0);

    return 0;
}
```