TASK 1;

#include<iostream>

#include<string>

using namespace std;

struct ListNode

{

char nodeName;

ListNode\* next = NULL;

ListNode(char des)

{

nodeName = des;

}

};

struct ListHead

{

char Data;

bool visited = 0;

ListNode\* Head = NULL;

ListNode\* Tail = NULL;

};

struct node

{

int data;

node\* next = NULL;

};

class Queue

{

node\* head = NULL, \* Tail = NULL;

public:

void EnQueue(int X)

{

this->ADDnode(X);

}

void ADDnode(int x)

{

node\* newnode = NULL;

newnode = new node;

if (this->Overflow(newnode))

{

return;

}

newnode->data = x;

if (head == NULL)

{

head = Tail = newnode;

return;

}

Tail->next = newnode;

Tail = newnode;

return;

}

void Deletenode()

{

node\* temp = head;

head = head->next;

delete temp;

}

int DeQueue()

{

if (this->UnderFlow())

{

return 0;

}

int x = head->data;

Deletenode();

return x;

}

bool UnderFlow()

{

if (head == NULL)

{

cout << "Queue is underflowed\n";

return 1;

}

return 0;

}

bool Overflow(node\* temp)

{

if (temp == NULL)

{

cout << "Queue is Overflowed\n";

return 1;

}

return 0;

}

void Dispaly()

{

if (this->UnderFlow())

{

cout << "There is no Element in the Queue\n";

return;

}

node\* temp = head;

while (temp != NULL)

{

cout << temp->data << "\t";

temp = temp->next;

}

cout << endl;

}

};

class Graph

{

ListHead\* G = NULL;

int NoOfVertices = 0;

int\* Parent = NULL;

bool findornot = 0;

public:

Graph(int NoOfVertices = 0)

{

this->NoOfVertices = NoOfVertices;

G = new ListHead[this->NoOfVertices];

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

{

G[i].Data = i + 65;

}

Parent = new int[NoOfVertices];

}

void FindIndex(char node)

{

int i = static\_cast<int>(node) - 65;

}

void Insert(char src, char Des)

{

ListNode\* t = new ListNode(Des);

if (G[src - 65].Head == NULL)

{

G[src - 65].Head = G[src - 65].Tail = t;

}

else

{

G[src - 65].Tail->next = t;

G[src - 65].Tail = t;

}

t = new ListNode(src);

if (G[Des - 65].Head == NULL)

{

G[Des - 65].Head = G[Des - 65].Tail = t;

return;

}

G[Des - 65].Tail->next = t;

G[Des - 65].Tail = t;

return;

}

void Print()

{

ListNode\* temp = NULL;

for (int i = 0; i < this->NoOfVertices; i++)

{

temp = G[i].Head;

cout << "Vertex = " << G[i].Data << " =>";

while (temp != NULL)

{

cout << temp->nodeName << " -->";

temp = temp->next;

}

cout << "NULL" << endl;

}

}

void DFS(int vertex, int des)

{

ListNode\* adjList = G[vertex].Head;

ListNode\* temp = adjList;

G[vertex].visited = 1;

if (vertex == des)

{

findornot = 1;

return;

}

while (temp != NULL)

{

int connectedVertex = temp->nodeName - 65;

if (G[connectedVertex].visited == 0)

{

Parent[connectedVertex] = vertex;

DFS(connectedVertex, des);

}

if (findornot == 0)

{

temp = temp->next;

}

else

{

break;

}

}

}

void DisplayParent(int src, int des)

{

if (findornot == 0)

{

return;

}

cout << "End\t" << static\_cast<char>(des + 65) << "<-";

while (1)

{

if (des == src)

{

break;

}

cout << static\_cast<char>(Parent[des] + 65) << "<-";

des = Parent[des];

}

cout << "\tStart" << endl;

for (int i = 0; i < this->NoOfVertices; i++)

{

G[i].visited = 0;

}

}

};

int main()

{

Graph G(13);

char ArrEdes[12][2] = { { 'A','B' },{ 'A','C' },{ 'B','D' },{ 'B','E' },{ 'C','F' },{ 'D','G' },{ 'D','H' },{ 'E','I' },{ 'F','J' },{ 'G','K' },{ 'G','L' },{ 'I','M' } };

for (int i = 0; i < 12; i++)

{

G.Insert(ArrEdes[i][0], ArrEdes[i][1]);

}

G.Print();

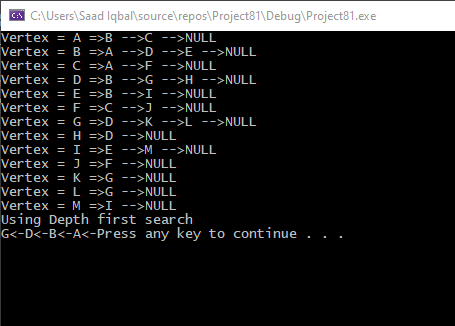
cout << "\n\nUsing DFS\n\n";

G.DFS('A' - 65, 'G' - 65);

G.DisplayParent('A' - 65, 'G' - 65);

system("pause");

}



TASK 2:

#include<iostream>

#include<string>

using namespace std;

struct ListNode

{

char nodeName;

ListNode\* next = NULL;

ListNode(char des)

{

nodeName = des;

}

};

struct ListHead

{

char Data;

bool visited = 0;

ListNode\* Head = NULL;

ListNode\* Tail = NULL;

};

struct node

{

int data;

node\* next = NULL;

};

class Queue

{

node\* head = NULL, \* Tail = NULL;

public:

void EnQueue(int X)

{

this->ADDnode(X);

}

void ADDnode(int x)

{

node\* newnode = NULL;

newnode = new node;

if (this->Overflow(newnode))

{

return;

}

newnode->data = x;

if (head == NULL)

{

head = Tail = newnode;

return;

}

Tail->next = newnode;

Tail = newnode;

return;

}

void Deletenode()

{

node\* temp = head;

head = head->next;

delete temp;

}

int DeQueue()

{

if (this->UnderFlow())

{

return 0;

}

int x = head->data;

Deletenode();

return x;

}

bool UnderFlow()

{

if (head == NULL)

{

cout << "Queue is underflowed\n";

return 1;

}

return 0;

}

bool Overflow(node\* temp)

{

if (temp == NULL)

{

cout << "Queue is Overflowed\n";

return 1;

}

return 0;

}

void Dispaly()

{

if (this->UnderFlow())

{

cout << "There is no Element in the Queue\n";

return;

}

node\* temp = head;

while (temp != NULL)

{

cout << temp->data << "\t";

temp = temp->next;

}

cout << endl;

}

};

class Graph

{

ListHead\* G = NULL;

int NoOfVertices = 0;

public:

Graph(int NoOfVertices = 0)

{

this->NoOfVertices = NoOfVertices;

G = new ListHead[this->NoOfVertices];

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

{

G[i].Data = i + 65;

}

}

void Insert(int src, int Des)

{

ListNode\* t = new ListNode(Des + 65);

if (G[src].Head == NULL)

{

G[src].Head = G[src].Tail = t;

}

else

{

G[src].Tail->next = t;

G[src].Tail = t;

}

t = new ListNode(src + 65);

if (G[Des].Head == NULL)

{

G[Des].Head = G[Des].Tail = t;

return;

}

G[Des].Tail->next = t;

G[Des].Tail = t;

return;

}

void Print()

{

ListNode\* temp = NULL;

for (int i = 0; i < this->NoOfVertices; i++)

{

temp = G[i].Head;

cout << "Vertex = " << G[i].Data << " =>";

while (temp != NULL)

{

cout << temp->nodeName << " -->";

temp = temp->next;

}

cout << "NULL" << endl;

}

}

void BFS(int startVertex, int key)

{

Queue q;

int\* Parent = new int[NoOfVertices];

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

{

Parent[i] = -1;

}

this->G[startVertex].visited = 1;

q.EnQueue(startVertex);

while (!q.UnderFlow())

{

int currentVertex = q.DeQueue();

if (Parent[currentVertex] == -1)

{

Parent[currentVertex] = currentVertex;

}

if (key == currentVertex)

{

break;

}

ListNode\* temp = this->G[currentVertex].Head;

while (temp)

{

int adjVertex = temp->nodeName - 65;

if (G[adjVertex].visited == 0)

{

G[adjVertex].visited = 1;

q.EnQueue(adjVertex);

Parent[adjVertex] = currentVertex;

}

temp = temp->next;

}

}

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

{

G[i].visited = 0;

}

int i = key;

cout << static\_cast<char>(i + 65) << "<-";

while (1)

{

if (i == 0)

{

break;

}

cout << static\_cast<char>(Parent[i] + 65) << "<-";

i = Parent[i];

}

}

};

int main()

{

Graph G(13);

char ArrEdes[12][2] = { { 'A','B' },{ 'A','C' },{ 'B','D' },{ 'B','E' },{ 'C','F' },{ 'D','G' },{ 'D','H' },{ 'E','I' },{ 'F','J' },{ 'G','K' },{ 'G','L' },{ 'I','M' } };

for (int i = 0; i < 12; i++)

{

G.Insert(ArrEdes[i][0] - 65, ArrEdes[i][1] - 65);

}

G.Print();

cout << "\n\nUsing Breadth first search\n";

G.BFS('A' - 65, 'G' - 65);

system("pause");

}

