1. **Activity: Get the number of vertices of a graph and adjacency matrix. Check if the graph is a diagraph or undirected graph.**

**NOTE: Matrix for an undirected graph us symmetric.**

**Algorithm:**

1. Function to compute transpose ->
   1. For I = 0 to vertices
      1. For j = 0 to vertices
         1. Transpose[i][j] = matrix[j][i]
2. Function to check if two matrices are equal
   1. Equal = true
   2. For I = 0 to vertices
      1. For j = 0 to vertices
         1. If( mat\_1[i][j] != mat\_2[i][j]) equal = false
   3. Return equal

**Test case table:**

|  |  |  |
| --- | --- | --- |
| Matrix | Result | PASS/FAIL |
| 0 1 0 0  0 0 0 0  0 1 0 1  1 0 0 0 | Graph is directed | **PASS** |
| 0 1 1 1  1 0 1 0  1 1 0 0  1 0 0 0 | Graph is undirected | **PASS`** |

**Program:**

#include <stdio.h>

#include <stdlib.h>

int vertices;

void printMatrix(int matrix[][vertices]){

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

    {

        for(int j = 0; j < vertices; j++){

            printf("%d ", matrix[i][j]);

            }

        printf("\n");

    }

}

int checkMatrixEqual(int m1[][vertices], int m2[][vertices]){

    int equal = 1;

    for (int i = 0; i < vertices; ++i){

      for (int j = 0; j < vertices; ++j) {

            if(m1[i][j] != m2[i][j]) equal = 0;

        }

    }

    return equal;

}

void transpose(int matrix[][vertices], int trans[][vertices]){

    for (int i = 0; i < vertices; ++i){

      for (int j = 0; j < vertices; ++j) {

            trans[j][i] = matrix[i][j];

        }

    }

}

int main(){

    printf("Enter number of vertices: ");

    scanf("%d", &vertices);

    int adj[vertices][vertices];

    int trans[vertices][vertices];

    printf("Enter Adjacency matix (with space):\n");

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

    {

        for(int j = 0; j < vertices; j++){

            scanf("%d", &adj[i][j]);

            }

    }

    printf("\nAdjacency Matrix ->\n");

    printMatrix(adj);

    printf("Transpose ->\n");

    transpose(adj, trans);

    printMatrix(trans);

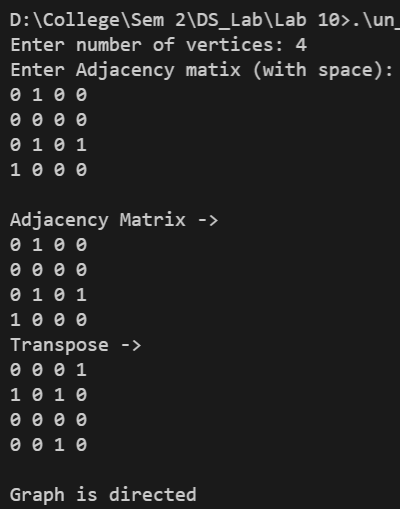
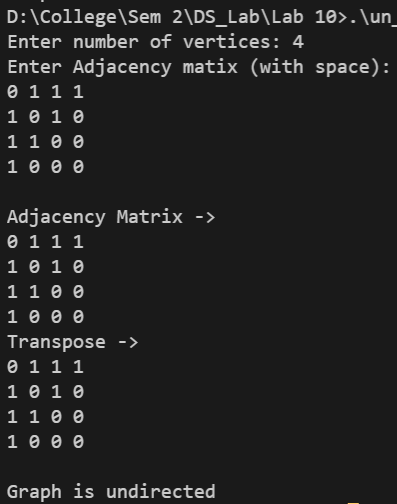
    if(checkMatrixEqual(adj, trans)){printf("\nGraph is undirected");}

    else{printf("\nGraph is directed");}

    return 0;

}

**Screenshot of compilation and execution:**

** **

1. **Activity: Write a menu driven program ….. (c) Adjacency matrices.**

**Algorithm:**

1. Function to find degree of vertices in undirected graph
   1. For i = 0 to n-1
      1. int degree = 0;
      2. For j = 0 to n-1
         1. degree += und[i][j];
      3. print “Degree of “+ vid[i] + “ : “ + degree
2. Function to find indegree & outdegree of vertices in directed graph
   1. For i = 0 to n-1
      1. indegree = 0
      2. outdegree = 0;
      3. for j = 0 to n-1
         1. outdegree += und[i][j];
         2. indegree += und[j][i];
      4. print “In-Degree of “ + vid[i] + “ -> “ + indegree
      5. print “In-Degree of “ + vid[i] + “ -> “ + outdegree

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define heading "------------------ Find degrees of graphs ------------------\n\n"

int n;

// Undirected ->

// Degree = Sum of row/column

// Diagraph ->

// Indegree = Sum of column

// Outdegree = Sum of row

void printMatrix(int matrix[][n], char vid[]){

    printf("  ");

    for (int i = 0; i < n; i++)

    {

        printf("%c ", vid[i]);

    }

    printf("\n");

    for (int i = 0; i < n; i++)

    {

        printf("%c ", vid[i]);

        for(int j = 0; j < n; j++)

        {

            printf("%d ", matrix[i][j]);

            }

        printf("\n");

    }

}

void degreeUndirected(int und[][n], char vid[n]){

    for(int i = 0; i < n; i++){

        int degree = 0;

        for (int j = 0; j < n; j++)

        {

            degree += und[i][j];

        }

        printf("Degree of %c: %d\n", vid[i], degree);

    }

}

void degreeDirected(int und[][n], char vid[n]){

    for(int i = 0; i < n; i++){

        int indegree = 0, outdegree = 0;

        for (int j = 0; j < n; j++)

        {

            outdegree += und[i][j];

            indegree += und[j][i];

        }

        printf("In-Degree of %c -> %d | Out-Degree of %c -> %d\n", vid[i], indegree, vid[i], outdegree);

    }

}

int main(){

    system("cls");

    printf(heading);

    printf("Enter number of vertices: ");

    scanf("%d", &n);

    char vIDs[n];

    printf("Enter vertex IDs (each ID in one line) -> \n");

    for(int i = 0; i < n; i++){

        while((getchar()) != '\n');

        scanf("%c", &vIDs[i]);

    }

    int und[n][n], diadj[n][n];

    printf("Enter adjacency matrix for undirected->\n");

    for (int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++){

            scanf("%d", &und[i][j]);

            }

    }

    printf("Enter adjacency matrix for directed->\n");

    for (int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++){

            scanf("%d", &diadj[i][j]);

            }

    }

    setupTest(und, diadj);

    int choice;

    Main: system("cls");

    printf(heading);

    printf("Choose option ->\n");

    printf("1. View degree of vertices of undirected graph\n");

    printf("1. View indegrees and outdegrees of vertices of directed graph\n");

    printf("3. View adjacency matrices\n");

    printf("4. Exit");

    printf("\n\nOption: ");

    scanf("%d", &choice);

    switch (choice)

    {

    case 1:

        system("cls");

        printf(heading);

        degreeUndirected(und, vIDs);

        printf("\n\nPress Enter to continue..");

        while((getchar()) != '\n');

        getchar();

        goto Main;

        break;

    case 2:

        system("cls");

        printf(heading);

        degreeDirected(diadj, vIDs);

        printf("\n\nPress Enter to continue..");

        while((getchar()) != '\n');

        getchar();

        goto Main;

        break;

    case 3:

        View\_matrix: system("cls");

        printf(heading);

        printf("Choose graph ->\n");

        printf("1. Undirected graph\n");

        printf("2. Directed graph");

        printf("\n\nOption: ");

        scanf("%d", &choice);

        switch (choice)

        {

        case 1:

            system("cls");

            printf(heading);

            printf("Adjacency matrix for undirected graph ->\n");

            printMatrix(und, vIDs);

            printf("\n\nPress Enter to continue..");

            while((getchar()) != '\n');

            getchar();

            goto Main;

            break;

        case 2:

            system("cls");

            printf(heading);

            printf("Adjacency matrix for directed graph ->\n");

            printMatrix(diadj, vIDs);

            printf("\n\nPress Enter to continue..");

            while((getchar()) != '\n');

            getchar();

            goto Main;

            break;

        default:

            printf("Invalid selection. Press Enter to continue..");

            while((getchar()) != '\n');

            getchar();

            goto View\_matrix;

            break;

        }

        break;

    case 4:

        break;

    default:

        printf("Invalid selection. Press Enter to continue..");

        while((getchar()) != '\n');

        getchar();

        goto Main;

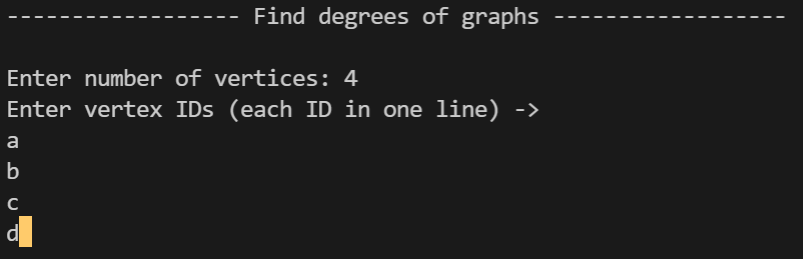
        break;

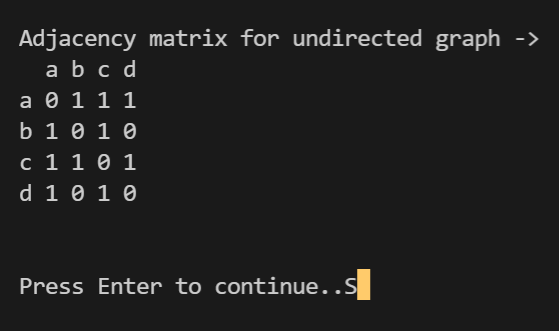
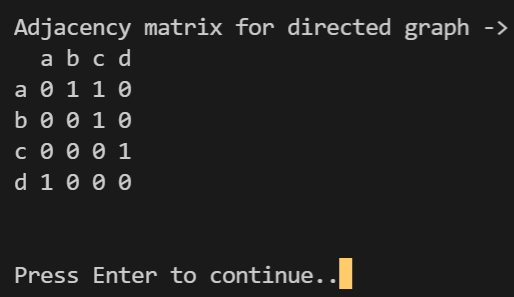
    }

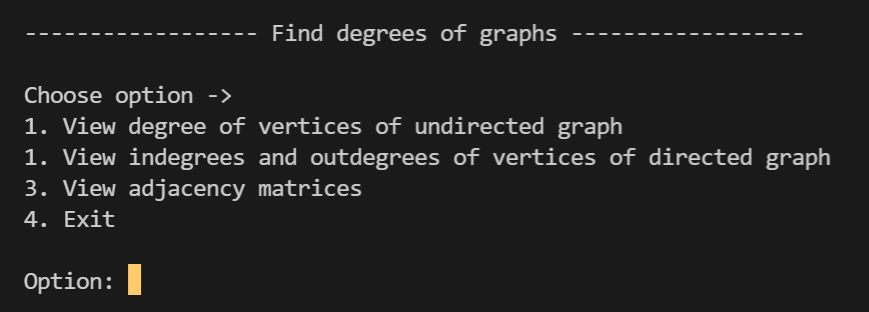
    return 0;

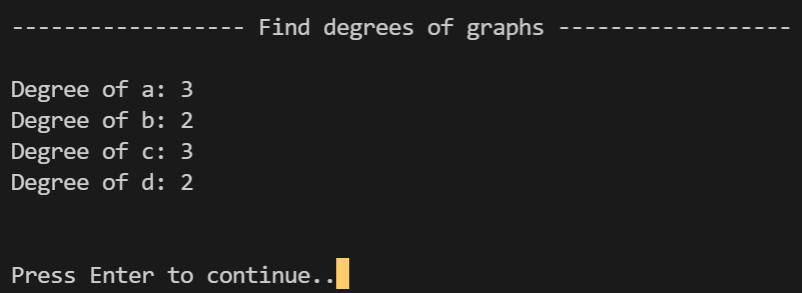
}

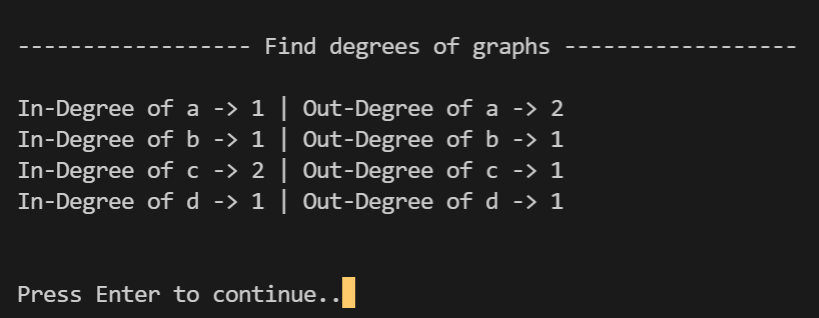
**Screenshot of compilation and execution:**

****

** **

****

****

****

1. **Activity: Implement a graph using vertex and edge list**

**Algorithm:**

1. Function addVertex(int id)
   1. Check if vertex id already exist. If yes, return.
   2. struct vertex \*temp = NULL;
   3. temp = (struct vertex \*)malloc(sizeof(struct vertex));
   4. if(temp == NULL)
      1. print "Out of memory. Returning.."
      2. return;
   5. temp->id = id;
   6. temp->next = vertexHead;
   7. vertexHead = temp;
   8. print "Vertex entered successfully."
2. Function addEdge(int start, int end)
   1. int startinlist = 0, endinlist = 0;
   2. for(vertexPtr = vertexHead; vertexPtr != NULL; vertexPtr = vertexPtr->next)
      1. if (vertexPtr->id == start)
         1. startinlist = 1;
      2. if (vertexPtr->id == end)
         1. endinlist = 1;
   3. if(!(startinlist))
      1. print "Start node doesn't exist."
      2. return;
   4. if(!(endinlist))
      1. print "End node doesn't exist.”
      2. return;
   5. for(edgePtr = edgeHead; edgePtr; edgePtr = edgePtr->next)
      1. if(edgePtr->from == start && edgePtr->to == end){
         1. print "Edge already exist."
         2. return;
   6. struct Edge \*temp = NULL;
   7. temp = (struct Edge \*)malloc(sizeof(struct Edge));
   8. if(temp == NULL){
      1. print "Out of memory. Returning.."
      2. return;
   9. temp->from = start;
   10. temp->to = end;
   11. temp->next = edgeHead;
   12. edgeHead = temp;
   13. print "Edge entered successfully."

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

struct vertex{

    int id;

    struct vertex \*next;

} \*vertexHead = NULL, \*vertexPtr = NULL;

struct Edge{

    int from;

    int to;

    struct Edge \*next;

} \*edgeHead = NULL, \*edgePtr = NULL;

void addVertex(int id){

    for(vertexPtr = vertexHead; vertexPtr != NULL; vertexPtr = vertexPtr->next){

        if (vertexPtr->id == id)

        {

            printf("Vertex already exist.\n");

            return;

        }

    }

    struct vertex \*temp = NULL;

    temp = (struct vertex \*)malloc(sizeof(struct vertex));

    if(temp == NULL){

        printf("Out of memory. Returning..\n");

        return;

    }

    temp->id = id;

    temp->next = vertexHead;

    vertexHead = temp;

    printf("Vertex entered successfully.\n");

}

void displayVertexList(){

    if (vertexHead == NULL)

    {

        printf("No vertices found.\n");

        return;

    }

    for(vertexPtr = vertexHead; vertexPtr != NULL; vertexPtr = vertexPtr->next){

        printf("%d -> ", vertexPtr->id);

    }

    printf("END\n");

}

void addEdge(int start, int end){

    //Check if terminals of edge are in the vertex list

    int startinlist = 0, endinlist = 0;

    for(vertexPtr = vertexHead; vertexPtr != NULL; vertexPtr = vertexPtr->next){

        if (vertexPtr->id == start)

        {

            startinlist = 1;

        }

        if (vertexPtr->id == end)

        {

            endinlist = 1;

        }

    }

    if(!(startinlist)){

        printf("Start node doesn't exist.\n");

        return;

    }

    if(!(endinlist)){

        printf("End node doesn't exist.\n");

        return;

    }

    for(edgePtr = edgeHead; edgePtr; edgePtr = edgePtr->next){

        if(edgePtr->from == start && edgePtr->to == end){

            printf("Edge already exist.\n");

            return;

        }

    }

    struct Edge \*temp = NULL;

    temp = (struct Edge \*)malloc(sizeof(struct Edge));

    if(temp == NULL){

        printf("Out of memory. Returning..");

        return;

    }

    temp->from = start;

    temp->to = end;

    temp->next = edgeHead;

    edgeHead = temp;

    printf("Edge entered successfully.\n");

}

void displayEdgeList(){

    if (edgeHead == NULL)

    {

        printf("No edges found.");

        return;

    }

    for(edgePtr = edgeHead; edgePtr != NULL; edgePtr = edgePtr->next){

        printf("%d->%d\n", edgePtr->from, edgePtr->to);

    }

}

int main(){

    int inf = 1;

    char choice;

    while(inf){

        printf("Do you want to enter vertex? (y/n): ");

        scanf("%c", &choice);

        switch (tolower(choice))

        {

        case 'y':

            printf("Enter vertex ID: ");

            int vid;

            scanf("%d", &vid);

            addVertex(vid);

            break;

        case 'n':

            inf = 0;

            break;

        default:

            printf("Invalid selection.\n");

            break;

        }

        while((getchar()) != '\n');

    }

    printf("Vertex List: ");

    displayVertexList();

    printf("\n");

    inf = 1;

    while(inf){

        printf("Do you want to enter edge? (y/n): ");

        scanf("%c", &choice);

        switch (tolower(choice))

        {

        case 'y':

            printf("Enter start vertex: ");

            int start, end;

            scanf("%d", &start);

            printf("Enter end vertex: ");

            scanf("%d", &end);

            addEdge(start, end);

            break;

        case 'n':

            inf = 0;

            break;

        default:

            printf("Invalid selection.\n");

            break;

        }

        while((getchar()) != '\n');

    }

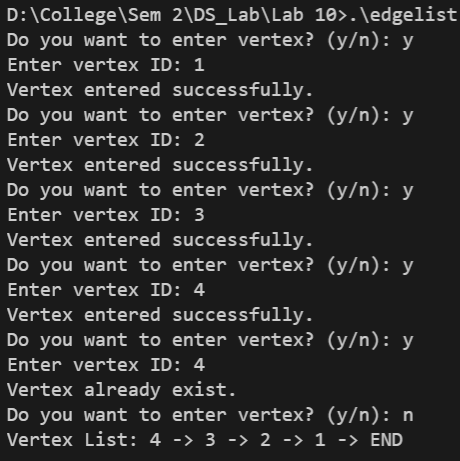
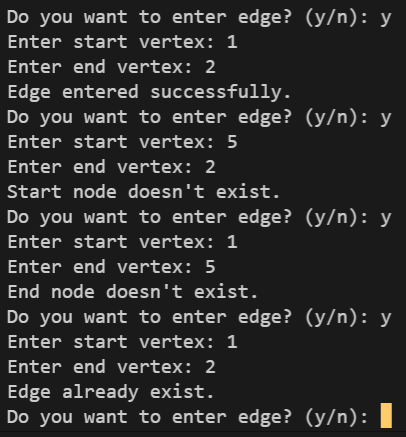
    printf("\nEdge List (start -> end) ->\n");

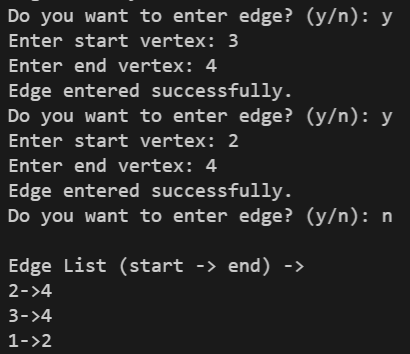
    displayEdgeList();

    return 0;

}

**Screenshot of compilation and execution:**

** **

****

1. **Activity: Write a menu driven program …….. of a given graph.**

**NOTE: Used adjacency list as graph implementation**

**Data Structures used:**

1. struct node {

int vertex;

bool visited;

struct node\* next;

};

1. struct Queue{

struct node \*vertexNode;

struct Queue \*next;

} \*queueHead = NULL, \*queuePtr = NULL;

1. struct Stack{

struct node \*vertexNode;

struct Stack \*next;

} \*stackHead = NULL, \*stackPtr = NULL;

1. struct Graph {

int numVertices;

struct node\* vertexList;

struct node\*\* adjLists;

};

**Algorithm:**

1. Function bfsVisit(struct Graph\* graph, int id)
   1. struct node\* temp = graph->adjLists[id], \*vertex;
   2. while (temp)
      1. vertex = searchVertextList(graph->vertexList, temp->vertex);
      2. if(vertex->visited != 1)
         1. enqueue(graph, vertex);
         2. vertex->visited = 1;
      3. temp = temp->next;

1. void bfs(struct Graph\* graph, int vertexID)
   1. struct node\* temp = graph->adjLists[vertexID],
   2. \*vListPtr = searchVertextList(graph->vertexList, vertexID);
   3. vListPtr->visited = true;
   4. enqueue(graph, vListPtr);
   5. while(queueHead)
      1. int visitIndex = dequeue();
      2. printf("%d, ", visitIndex);
      3. bfsVisit(graph, visitIndex);
   6. printQueue();
2. void dfsVisit(struct Graph\* graph, int id)
   1. struct node\* temp = graph->adjLists[id], \*vertex;
   2. while (temp)
      1. vertex = searchVertexList(graph->vertexList, temp->vertex);
      2. if(vertex->visited != 1)
         1. push(graph, vertex);
         2. vertex->visited = 1;
      3. temp = temp->next;

1. void dfs(struct Graph\* graph, int vertexID)
   1. struct node\* temp = graph->adjLists[vertexID],
   2. \*vListPtr = searchVertexList(graph->vertexList, vertexID);
   3. vListPtr->visited = true;
   4. push(graph, vListPtr);
   5. while(stackHead)
      1. int visitIndex = pop();
      2. printf("%d, ", visitIndex);
      3. dfsVisit(graph, visitIndex);

**Test case:**

Sample graph shape

    0

   /  \

 1 - 2

 |   |

 3   4

 \

   5

BFS: 0, 2, 1, 4, 3, 5

DFS: 0, 1, 3, 5, 2, 4

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

  int vertex;

  bool visited;

  struct node\* next;

};

struct Queue{

    struct node \*vertexNode;

    struct Queue \*next;

} \*queueHead = NULL, \*queuePtr = NULL;

struct Stack{

    struct node \*vertexNode;

    struct Stack \*next;

} \*stackHead = NULL, \*stackPtr = NULL;

struct Graph {

  int numVertices;

  struct node\* vertexList;

  struct node\*\* adjLists;

};

void enqueue(struct Graph\* graph, struct node \*vertexNode){

    struct Queue \*temp = NULL;

    temp = (struct Queue \*)malloc(sizeof(struct Queue));

    temp->vertexNode = vertexNode;

    temp->next = queueHead;

    queueHead = temp;

}

void printQueue(){

    for (queuePtr = queueHead; queuePtr; queuePtr = queuePtr->next)

    {

        printf("%d -> ", queuePtr->vertexNode->vertex);

    }

}

int dequeue(){

    struct Queue \*temp = NULL;

    if (queueHead == NULL)

    {

      return -1;

    }

    if (queueHead->next == NULL)

    {

      int id = queueHead->vertexNode->vertex;

      queueHead = NULL;

      return id;

    }

    for(temp = queueHead; temp->next->next; temp = temp->next);

    int id = temp->next->vertexNode->vertex;

    free(temp->next);

    temp->next = NULL;

    return id;

}

void push(struct Graph\* graph, struct node \*vertexNode){

    struct Stack \*temp = NULL;

    temp = (struct Stack \*)malloc(sizeof(struct Stack));

    temp->vertexNode = vertexNode;

    temp->next = stackHead;

    stackHead = temp;

}

void printStack(){

    for (stackPtr = stackHead; stackPtr; stackPtr = stackPtr->next)

    {

        printf("%d -> ", stackPtr->vertexNode->vertex);

    }

}

int pop(){

    if(stackHead == NULL){

      return -1;

    }

    if (stackHead->next == NULL)

    {

      int id = stackHead->vertexNode->vertex;

      stackHead = NULL;

      return id;

    }

    struct Stack \*temp = NULL;

    temp = stackHead;

    int id = temp->vertexNode->vertex;

    stackHead = temp->next;

    free(temp);

    return id;

}

struct node \*searchVList(struct node \*vlist, int id){

    struct node\* vListptr = vlist;

    for(; vListptr; vListptr = vListptr->next){

      if (vListptr->vertex == id)

      {

        return vListptr;

      }

    }

    return vListptr;

}

void bfsVisit(struct Graph\* graph, int id){

    struct node\* temp = graph->adjLists[id], \*vertex;

    while (temp) {

      vertex = searchVList(graph->vertexList, temp->vertex);

      if(vertex->visited != 1){

        enqueue(graph, vertex);

        vertex->visited = 1;

      }

      temp = temp->next;

    }

}

void bfs(struct Graph\* graph, int vertexID){

  struct node\* temp = graph->adjLists[vertexID],

  \*vListPtr = searchVList(graph->vertexList, vertexID);

  vListPtr->visited = true;

  enqueue(graph, vListPtr);

  while(queueHead){

    int visitIndex = dequeue();

    printf("%d, ", visitIndex);

    bfsVisit(graph, visitIndex);

  }

  printQueue();

}

void dfsVisit(struct Graph\* graph, int id){

    struct node\* temp = graph->adjLists[id], \*vertex;

    while (temp) {

      vertex = searchVList(graph->vertexList, temp->vertex);

      if(vertex->visited != 1){

        push(graph, vertex);

        vertex->visited = 1;

      }

      temp = temp->next;

    }

}

void dfs(struct Graph\* graph, int vertexID){

  struct node\* temp = graph->adjLists[vertexID],

  \*vListPtr = searchVList(graph->vertexList, vertexID);

  vListPtr->visited = true;

  push(graph, vListPtr);

  while(stackHead){

    int visitIndex = pop();

    printf("%d, ", visitIndex);

    dfsVisit(graph, visitIndex);

  }

  // printStack();

}

void resetVisited(struct node \*vlist){

  struct node\* vListptr = vlist;

    for(; vListptr; vListptr = vListptr->next){

      vListptr->visited = false;

    }

}

// Create a node

struct node\* createNode(int v) {

  struct node\* newNode = malloc(sizeof(struct node));

  newNode->vertex = v;

  newNode->visited = false;

  newNode->next = NULL;

  return newNode;

}

// Create a graph

struct Graph\* createAGraph(int vertices) {

  struct Graph\* graph = malloc(sizeof(struct Graph));

  graph->numVertices = vertices;

  graph->vertexList = NULL;

  graph->adjLists = malloc(vertices \* sizeof(struct node\*));

  int i;

  for (i = 0; i < vertices; i++){

    struct node \*temp, \*vListptr;

    temp = (struct node \*)malloc(vertices \* sizeof(struct node));

    temp->vertex = i;

    temp->visited = false;

    temp->next = NULL;

    if (graph->vertexList == NULL)

    {

      graph->vertexList = temp;

    }

    else{

    for(vListptr = graph->vertexList; vListptr->next != NULL; vListptr = vListptr->next);

    vListptr->next = temp;

    }

    graph->adjLists[i] = NULL;

  }

  return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int source, int dest) {

  // Add edge from s to d

  struct node\* newNode = createNode(dest);

  newNode->next = graph->adjLists[source];

  graph->adjLists[source] = newNode;

  // Add edge from d to s

  newNode = createNode(source);

  newNode->next = graph->adjLists[dest];

  graph->adjLists[dest] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph) {

  int v;

  struct node\* vListptr = graph->vertexList;

  for (v = 0; v < graph->numVertices; v++) {

    struct node\* adListPtr = graph->adjLists[v];

    printf("\nVertex %d: ", vListptr->vertex);

    while (adListPtr) {

      printf("%d -> ", adListPtr->vertex);

      adListPtr = adListPtr->next;

    }

    vListptr = vListptr->next;

    printf("\n");

  }

}

int main() {

    struct Graph\* graph = createAGraph(6);

    addEdge(graph, 0, 1);

    addEdge(graph, 0, 2);

    addEdge(graph, 1, 2);

    addEdge(graph, 1, 3);

    addEdge(graph, 2, 4);

    addEdge(graph, 3, 5);

    printGraph(graph);

    printf("\nBFS Traversal: ");

    bfs(graph, 0);

    resetVisited(graph->vertexList);

    printf("\nDFS Traversal: ");

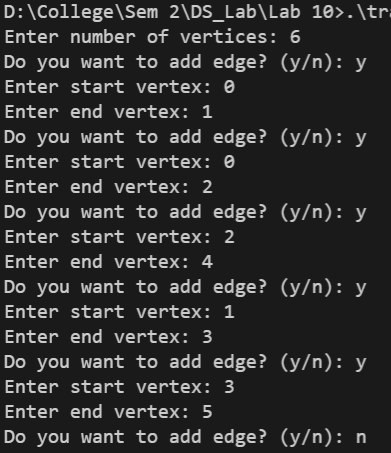
    dfs(graph, 0);

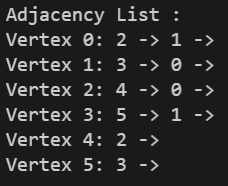
    return 0;

}

\

**Screenshot of compilation and execution:**

****

****

****