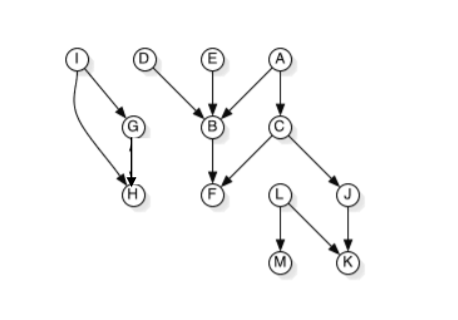
**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate my knowledge of graphs and topological sorting. This includes reachable and unreachable vertices, edges and simple paths.

****

**Answers to Questions:**

1. How many Vertices and Edges are in the graph?

There are 13 vertices and 13 edges in the graph.

vertices={I, D, E, A, G, B, C, H, F, L, J, K, M}

edges={IG, DB, EB, AB, AC, IH, GH, BF, CF, CJ, JK, LK, LM}

2. Name a vertex with the highest number of incoming edges

The vertex B has the highest number of incoming edges.

3. Name the vertices with the lowest number of incoming edges

The vertices I, D, E, A, L have the lowest number of incoming edges.

4. Are all vertices reachable from at least one other vertex? If not, specify the unreachable vertices

The vertices I, D, E, A, L are unreachable because they do not have any incoming edges.

5. Is the path A->C->F simple?

The path from A->C->F is simple because a vertex is not visited more than once.

6. What is the length of path A->C->J->K?

The length of path A->C->J->K is 3 because there are 3 edges along the path.

7. Is it possible to perform a topological sort on the graph as it is currently shown? If not, state why or what change(s) would need to be made to support the sort.

It is possible to perform a topological sort on the graph as it is currently shown. There are multiple vertices without incoming edges to start with, in order to completely sort/stack every vertex in the graph. There are no cycles.

8. What new, single edge could you add to make vertices G and H accessible from source vertex A without making vertex I accessible from source vertex A?

A single edge could be added from vertex B to vertex G so that vertices G and H are accessible from vertex A. This would not make vertex I accessible from vertex A. Or, a single edge could be added from vertex F to vertex G, so that the same conditions are met.

9. What single edge change can you make that would make destination vertices L and M accessible from source vertex A with a path going through vertex J?

A single edge could be added from vertex C to vertex L so that vertices L and M are accessible from vertex A without going through vertex J. Or, a single edge could be added from vertex F to vertex L, so that the same conditions are met.

10. What single edge change can you make that would make source vertices D and E able to traverse to J through vertex A

Vertex A to vertex B could be changed/replaced with an edge from vertex B to vertex A, so that vertices D and E could traverse to J through vertex A.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate building a topological sort. It consists of a list of 10 activities and their constraints as well as functions and definitions for the topological sort.

**Data File (startofday.in):**

10 13

1 10 2 1

2 3 4 5

6 7 2 6

2 8 9 10

3 7 2 4

7 10 5 10

8 9

1 Schedule tours for vacation

2 Look at different vacation destinations

3 Find your family's passports and tickets

4 Pick out clothes for vacation

5 Pack clothes for vacation

6 Gather your family together

7 Handout passports and tickets to family

8 Find a pet sitter for a dog

9 Give pet sitter a key to your house

10 Go to the airport

**Program Output (main.c):**

The Activity Requirements are:

1 -> 10

2 -> 1, 2 -> 3, 2 -> 4, 2 -> 6, 2 -> 8

3 -> 7

4 -> 5

5 -> 10

6 -> 7

7 -> 10

8 -> 9

9 -> 10

Topological sort of activities is:

1. Look at different vacation destinations (Activity 2)

2. Find a pet sitter for a dog (Activity 8)

3. Give pet sitter a key to your house (Activity 9)

4. Gather your family together (Activity 6)

5. Pick out clothes for vacation (Activity 4)

6. Pack clothes for vacation (Activity 5)

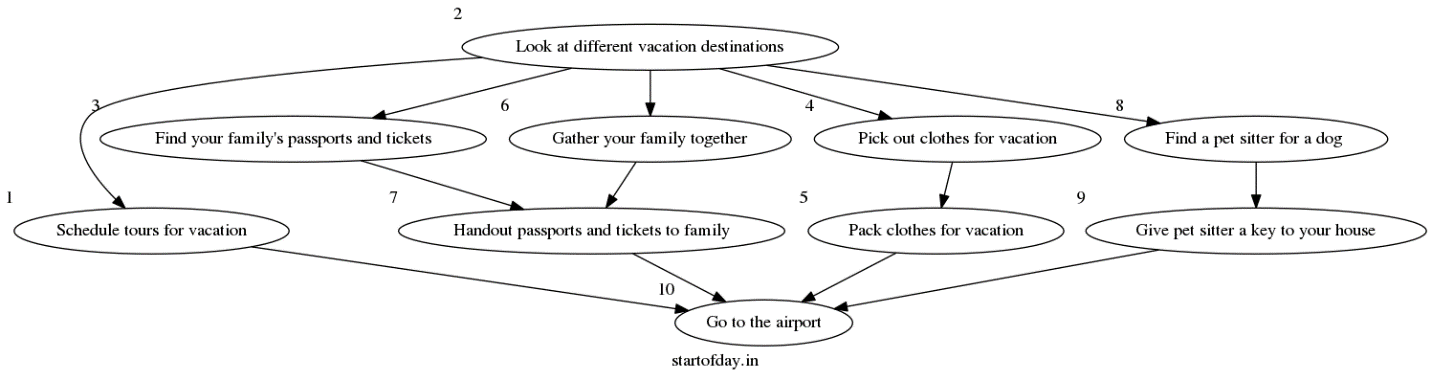
7. Find your family's passports and tickets (Activity 3)

8. Handout passports and tickets to family (Activity 7)

9. Schedule tours for vacation (Activity 1)

10. Go to the airport (Activity 10)

**Diagram:**



**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate building a topological sort that uses an adjacency list. It consists of a list of 10 activities and their constraints as well as functions and definitions for the topological sort.

**Program Code (StudentTopoDemo.c):**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include "PriorityQueue.h"

#include "Linkedlist.h"

#include "UserData.h"

// worst case number of activities that can be handled

#define MAXACTIVITIES 20

// worst case length of an activity name

#define MAXREQTEXTSIZE 79

// the file to process

#define THEFILE "DAG.in"

// AdjacencyListTopologicalSort controls the reading and topological analysis of a graph

// using an adjacency list based approach

static void AdjacencyListTopologicalSort ();

// ReadandBuildAdjacencyList

// populates the adjacency list for each edge (activity vertex),

// fills in the activity name for the activity, and

// builds the IncomingEdgeCount array to specify now many incoming

// edges there are for each activity

static int ReadandBuildAdjacencyList (LLInfoPtr Edges[MAXACTIVITIES+1],

char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[] );

// SortUsingAdjacencyList does the topological sort using the adjacency list

// and incoming edge counts for each activity for the number of activities provided

static Queue SortUsingAdjacencyList ( LLInfoPtr Edges[MAXACTIVITIES+1],

int IncomingEdgeCounts[],

int numActivities );

// AdjacencyMatrixApproach controls the reading and topological analysis of a graph

// using an adjacency matrix based approach

static void AdjacencyMatrixTopologicalSort ();

// ReadandBuildAdjacencyMatrix

// populates the AdjacencyMatrix for each edge (activity vertex),

// fills in the activity name for the activity, and

// builds the IncomingEdgeCount array to specify now many incoming

// edges there are for each activity

static int ReadandBuildAdjacencyMatrix (int AdjMatrix[MAXACTIVITIES+1][MAXACTIVITIES+1],

char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[] );

// SortUsingAdjacencyMatrix does the topological sort using the adjacency matrix

// and incoming edge counts for each activity for the number of activities provided

static Queue SortUsingAdjacencyMatrix ( int AdjMatrix[MAXACTIVITIES+1][MAXACTIVITIES+1],

int IncomingEdgeCounts[],

int numActivities );

//////////// Utilities common to the two approaches follow /////////////

// utility ReadActivityText reads the open file, filling the ActivityNames array

static void ReadActivityText( FILE\* in, char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int numActivities);

// utility ShowActivitiesAndIncomingEdgeCounts prints out all of the activity names that have

// been read in, together with the number of incoming edges that has been computed based on the

// data read in.

static void ShowActivitiesAndIncomingEdgeCounts (char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[],

int numActivities);

// utility ShowSortResults logs the message provided and the queue content holding the

// activities in order of their sort

static void ShowSortResults (char msg[], Queue SortedActivities,

char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1]);

// the test main runs a topological sort using both approaches.

// each approach is done through a call to its sort control routine and handles

// everything from file reading to writing of the results on the console

int main()

{

printf ("Starting a topological sort using an adjacency list\n");

AdjacencyListTopologicalSort();

printf ("\n\nStarting a topological sort using an adjacency matrix\n");

AdjacencyMatrixTopologicalSort();

return 0;

}

// AdjacencyListTopologicalSort controls the reading and topological analysis of a graph

// using an adjacency list based approach

// It starts by allocating the information needed. That includes the adjacency list,

// the array to gold the incoming edge counts and the array to hold activity names.

// It then uses support functions to sort and show results of the adjacency list based

// approach.

void AdjacencyListTopologicalSort ()

{

// allocate the worst case adjacency list Table (row 0 is not used)

LLInfoPtr Edges[MAXACTIVITIES+1];

// allocate the worst case IncomingEdgeCount Table to hold

// incoming edge counts for the activities (row 0 not used)

int IncomingEdgeCounts[MAXACTIVITIES+1];

// allocate the worst case array to hold the activity names and null all entries

// row 0 is not used

char ActivityName[MAXACTIVITIES+1][MAXREQTEXTSIZE+1];

// Read in data

int numActivities = ReadandBuildAdjacencyList (Edges, ActivityName, IncomingEdgeCounts );

// show the activity names and incoming edge counts

ShowActivitiesAndIncomingEdgeCounts (ActivityName,IncomingEdgeCounts, numActivities);

// Do the sort, using the adjacency list and incoming counts

// It returns a queue ordered by the first activity to the last activity

Queue SortedActivities = SortUsingAdjacencyList (Edges, IncomingEdgeCounts, numActivities);

// Print the Results

ShowSortResults("Topological Sort using an adjacency list is:", SortedActivities, ActivityName);

// cleanup by deleting the queue returned from the sort and all of the

// activity list linked lists

deleteQueue(SortedActivities);

for (int loop=1; loop <= MAXACTIVITIES; LL\_Delete(Edges[loop++]));

printf ("After cleanup, AllocationCount is %d\n", AllocationCount);

}

// ReadandBuildAdjacencyList populates the adjacency list for each edge (activity vertex),

// fills in the activity name for the activity, and

// builds the IncomingEdgeCount array to specify now many incoming

// edges there are for each activity

//

// The file format is our enhanced Kalicharan format

int ReadandBuildAdjacencyList (LLInfoPtr Edges[MAXACTIVITIES+1],

char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[] )

{

// number of activities in the file

int numActivities;

// number of requirements in the file

int numActivityRequirements;

// each requirement is in the form of a "from" activity needed before "to" activity

int fromActivity, toActivity;

// Open the file and get the number of activities and the number of activity

// requirements. Exit if file open issue or the data is too large for

// the static allocations

FILE \* in = fopen(THEFILE, "r");

if (in == NULL) {

printf ("Unable to open the file.. exiting");

exit(0);

}

fscanf(in, "%d %d", &numActivities, &numActivityRequirements);

if (numActivities > MAXACTIVITIES) {

printf("\nToo many activities to handle! exiting!\n");

exit(1);

}

// initialize the incoming edges before updating it

for (int loop=1; loop < MAXACTIVITIES+1; IncomingEdgeCounts[loop++] = 0);

// initialize a LL for each of the adjacency list entries (index 0 not used)

for (int loop=1; loop <= MAXACTIVITIES; Edges[loop++] = LL\_Init());

// read all the requirements, placing their information

// in the adjacency list for a "fromActivity" and incrementing

// the incoming edge count for the requirement "toActivity"

for (int loop = 1; loop <= numActivityRequirements; loop++) {

//get an activity requirement (fromActivity needed to do toActivity)

//and add it to the adjacency list of the vertex

int count = fscanf(in, "%d %d", &fromActivity, &toActivity);

// abort if there was no requirement

assert (count == 2);

// place the edge in the adjacency LL for the "from" activity,

// where the data in the edge is the number of the "to" activity

UserData D;

D.Vertex = toActivity;

LL\_AddAtEnd(Edges[fromActivity], D);

// update the incoming edge count for the "to" activity to reflect the

// requirement

IncomingEdgeCounts[toActivity]++;

}

// now read in the text of the activities themselves using the utility

ReadActivityText(in, ActivityNames, numActivities);

// done, so close the file and return the actual number of activities read to the caller

fclose (in);

return numActivities;

}

// SortUsingAdjacencyList performs a topological sort that is adjacency list based.

// The strategy is to use the IncomingEdgeCounts to decide when an activity can be done.

// The process consists of initially placing the vertices of all of the activities

// with zero incoming edges in a work queue.

//

// The remaining processing dequeues an activity from the work queue and does two things:

// 1. It queues the activity in the sorted activity queue, since it is in sorted order

// 2. It goes through the adjacency list for the dequeued activity and subtracts each of its

// adjacent vertices from the corresponding IncomingEdgeCounts. If a count goes to

// zero, it means that that activity can now be done, so it queues the activity in the work

// queue.

// these steps repeat until all of the activities have been sorted.

// NOTE: If you have nothing to dequeue and have not finished all activities, it means that

// there is a cycle in the graph because there should be 1 or more remaining activities

// with zero incoming edges

static Queue SortUsingAdjacencyList (LLInfoPtr Edges[MAXACTIVITIES+1],

int IncomingEdgeCounts[],

int numActivities )

{

// create a queue to hold the sorted activities

// a queue is used because we are building the list from

// the first activity we can do to the last we can do (a FIFO behavior)

Queue ActivityOrder = initQueue(NULL);

// create a queue to hold activities with zero incoming edges

// that have not been processed yet

// they will be queued whenever an activity no longer has incoming edges

// and will be dequeued when we analyze the outgoing edges for the activity

// to update the incoming edge counts for the activities not yet sorted

Queue Q = initQueue (NULL);

// queue all vertices that have zero incoming edges (IncomingEdgeCounts == 0)

for (int in = 1; in <= numActivities; in++)

if (IncomingEdgeCounts[in] == 0) {

UserData D;

D.Vertex = in;

enqueue (Q, D);

}

// numFinished is the count of activities we have handled. When it reaches the

// number of activities loaded from the file, we are done sorting

int numFinished = 0;

// loop while we have not sorted all of the activities

while (numFinished != numActivities) {

// when we are here, one or more vertices should be in the queue for processing of

// their outgoing edges. If nothing in the queue of vertices with zero edges,

// exit with error indicating a cycle

if (empty (Q)) {

printf ("Graph has a cycle.. topological sort not possible!");

exit(0);

}

// dequeue a vertex that has a zero incoming edge count

// we need to look at the vertex outgoing edges and update the incoming edge

// counts for each of its adjacent vertices

UserData D;

D = dequeue(Q);

// place it in the queue of order of activities

enqueue(ActivityOrder, D);

// adjust the remaining incoming edge counts

// for each vertex that the finished activity had an outgoing edge to

// by going through the adjacency list and processing each outgoing edge

while (LL\_Length(Edges[D.Vertex]) != 0) {

UserData ToEdge = LL\_GetFront(Edges[D.Vertex], DELETE\_NODE);

IncomingEdgeCounts[ToEdge.Vertex]--;

// if a vertex goes to zero incoming edges, queue it for processing

// because it now can be added to the sorted activities and have

// its adjacencies processed

if (IncomingEdgeCounts[ToEdge.Vertex] == 0) {

UserData D;

D.Vertex = ToEdge.Vertex;

enqueue (Q, D);

}

} // end of processing the adjacencies for the vertex

// increment how many activities we have finished

numFinished++;

} // end while numFinished != numActivities

// delete the queue

deleteQueue(Q);

return ActivityOrder;

}

// AdjacencyMatrixTopologicalSort controls the reading and topological analysis of a graph

// using an adjacency matrix based approach

// It starts by allocating the information needed. That includes the adjacency matrix,

// the array to hold the incoming edge counts and the array to hold activity names.

// It then uses support functions to sort and show results of the adjacency matrix based

// approach.

void AdjacencyMatrixTopologicalSort ()

{

// allocate the worst case adjacency matrix Table

// (index 0 is not used)

int AdjMatrix[MAXACTIVITIES+1][MAXACTIVITIES+1];

// allocate the worst case IncomingEdgeCount Table to hold

// incoming edge counts for the activities (index 0 not used)

int IncomingEdgeCounts[MAXACTIVITIES+1];

// allocate an array to hold the activity names

char ActivityName[MAXACTIVITIES+1][MAXREQTEXTSIZE+1];

// Read in data

int numActivities = ReadandBuildAdjacencyMatrix (AdjMatrix, ActivityName, IncomingEdgeCounts );

// show the activity names and incoming edge counts

ShowActivitiesAndIncomingEdgeCounts (ActivityName,IncomingEdgeCounts, numActivities);

// Do the sort, using the adjacency matrix and incoming counts

// It returns a queue ordered by the first activity to the last activity

Queue SortedActivities = SortUsingAdjacencyMatrix(AdjMatrix, IncomingEdgeCounts, numActivities);

// Print the Results

ShowSortResults("Topological Sort using an adjacency matrix is:", SortedActivities, ActivityName);

//cleanup by deleting the queue returned from the sort

deleteQueue(SortedActivities);

printf ("After cleanup, AllocationCount is %d\n", AllocationCount);

}

// ReadandBuildAdjacencyMatrix reads in the number of activities and number of activity

// requirements from a file. It checks that those values do not exceed defined values at

// the beginning of this file. It initializes IncomingEdgeCounts[] and AdjMatrix[][] to zero

// before it sets pairs of activities' requirements, or edges values, in the adjacency matrix.

// This increases IncomingEdgeCounts[] by 1. At the end, the names of the activites are read

// in to fill the ActivitiesNames array and the file is closed. This building of an adjacency

// matrix represents a graph. The number of activities read is returned.

int ReadandBuildAdjacencyMatrix (int AdjMatrix[MAXACTIVITIES+1][MAXACTIVITIES+1],

char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[] )

{

// number of activities in the file

int numActivities;

// number of requirements in the file

int numActivityRequirements;

// each requirement is in the form of a "from" activity needed before "to" activity

int fromActivity, toActivity;

// Open the file and get the number of activities and the number of activity

// requirements. Exit if file open issue or the data is too large for

// the static allocations

FILE \* in = fopen(THEFILE, "r");

if (in == NULL) {

printf ("Unable to open the file.. exiting");

exit(0);

}

fscanf(in, "%d %d", &numActivities, &numActivityRequirements);

if (numActivities > MAXACTIVITIES) {

printf("\nToo many activities to handle! exiting!\n");

exit(1);

}

// init the incoming edges before updating it

for (int loop=1; loop < MAXACTIVITIES+1; IncomingEdgeCounts[loop++] = 0);

// clear out the adjacency matrix

// start loop counters at 1 because we are using a file

// that starts at 1 so that vertex numbers directly map to

// the rows and columns

for (int loop1 = 1; loop1 < MAXACTIVITIES+1; loop1++) {

for (int loop2 = 1; loop2 < MAXACTIVITIES+1; loop2++) {

AdjMatrix[loop1][loop2] = 0;

}

}

// read all the requirements, placing their information

// in the adjacency list for a "fromActivity" and incrementing

// the incoming edge count for the requirement "toActivity"

for (int loop = 1; loop <= numActivityRequirements; loop++) {

//get an activity requirement (fromActivity needed to do toActivity)

//and add it to the adjacency list of the vertex

int count = fscanf(in, "%d %d", &fromActivity, &toActivity);

// abort if there was no requirement

assert (count == 2);

// set element's value to 1 inside adjacency matrix

// to signal the pair of vertices are adjacent

AdjMatrix[fromActivity][toActivity] = 1;

AdjMatrix[toActivity][fromActivity] = 1;

// add +1 to number of incoming edges

// for toActivity

IncomingEdgeCounts[toActivity]++;

}

// now read in the text of the activities themselves

ReadActivityText(in, ActivityNames, numActivities);

fclose (in);

// return the actual number of activities read to the caller

return numActivities;

}

// SortUsingAdjacencyMatrix creates two FIFO queues, one to hold activities with zero incoming

// edges and another to hold the final ActivityOrder. It starts sorting using a given adjacency

// matrix by queueing activities with zero incoming edges. Then it checks that there are no cycles,

// before it dequeues an activity, which then adds it to the final ActivityOrder queue and updates

// incoming edges for the finished activity's adjacent activities (outgoing edges). Once finished, the

// queue is deleted and the final sorted order of activities gets returned.

Queue SortUsingAdjacencyMatrix (

int Adjacencies[MAXACTIVITIES+1][MAXACTIVITIES+1],

int IncomingEdgeCounts[],

int numActivities)

{

// create a queue to hold the sorted activities

// a queue is used because we are building the list from

// the first activity we can do to the last we can do (a FIFO behavior)

Queue ActivityOrder = initQueue(NULL);

// create a queue to hold activities with zero incoming edges

// that have not been processed yet

// they will be queued whenever an activity no longer has incoming edges

// and will be dequeued when we analyze the outgoing edges for the activity

// to update the incoming edge counts for the activities not yet sorted

Queue Q = initQueue (NULL);

// queue all vertices that have zero incoming edges (IncomingEdgeCounts == 0)

for (int in = 1; in <= numActivities; in++)

if (IncomingEdgeCounts[in] == 0) {

UserData D;

D.Vertex = in;

enqueue (Q, D);

}

// numFinished is the count of activities we have handled. When it reaches the

// number of activities loaded from the file, we are done sorting

int numFinished = 0;

// loop while we have not sorted all of the activities

while (numFinished != numActivities) {

// when we are here, one or more vertices should be in the queue for processing of

// their outgoing edges. If nothing in the queue of vertices with zero edges,

// exit with error indicating a cycle

if (empty (Q)) {

printf ("Graph has a cycle.. topological sort not possible!");

exit(0);

}

// dequeue a vertex that has a zero incoming edge count

UserData D;

D = dequeue(Q);

// place it in the queue of order of activities

enqueue(ActivityOrder, D);

// we need to look at the finished activity outgoing edges and update the incoming edge

// counts for each of its adjacent activities

/\*

// get edge

// ?

while (D.Vertex != 0) {

// update by removing incoming edge

IncomingEdges[Adjacencies[AdjacentToD.Vertex][D.Vertex]]--;

// if incoming edge is zero, queue it

if (IncomingEdgeCounts[AdjacentToD.Vertex] == 0) {

D.Vertex = AdjacentToD.Vertex;

enqueue (Q, D);

}

}

\*/

numFinished++;

// end while numFinished != numActivities

// delete the queue

}

deleteQueue(Q);

return ActivityOrder;

}

// utility ReadActivityText fills the ActivityNames array with the activity names

// It will clear the ActivityNames array and then populate only those

// names provided in the open filoe "in"

void ReadActivityText( FILE\* in, char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int numActivities)

{

// clear all ActivityNames entries

for (int loop=1; loop <= numActivities; strcpy(ActivityNames[loop++], "") );

// now read in the text of the activities themselves

for (int loop = 0; loop < numActivities; loop++) {

int ActivityNumber;

// read the activity number. It must be provided, else assert

int count = fscanf(in, "%d", &ActivityNumber);

assert (count == 1);

// read in the activity name a character at a time

char ch;

// skip leading blanks

do

ch = getc(in);

while (ch == ' ');

// in the loop, read a character at a time until the end of line or too

// many characters have been read

count = 0;

while ((ch != '\n') && (ch != EOF)) {

// exit if the activity description is too large

if (count >= MAXREQTEXTSIZE) {

printf ("Activity description too long!! exiting..");

exit(0);

}

// otherwise, move the character into the description

// and make sure the string ends with a NULL

ActivityNames[ActivityNumber][count++] = ch;

ActivityNames[ActivityNumber][count] = 0;

// get the next character (or the indicator that an end of the line has been reached)

ch = getc(in);

}

}

}

void ShowActivitiesAndIncomingEdgeCounts (char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1],

int IncomingEdgeCounts[],

int numActivities)

{

// show the activity names and incoming edge counts

printf ("After reading in data, the activities and incoming edges are:\n");

for (int loop = 1; loop <= numActivities; loop++)

printf ("\t%s (Activity # %d) has %d incoming edge(s)\n",

ActivityNames[loop], loop, IncomingEdgeCounts[loop]);

}

void ShowSortResults (char msg[], Queue SortedActivities, char ActivityNames[MAXACTIVITIES+1][MAXREQTEXTSIZE+1])

{

printf ("\n%s\n", msg);

while (!empty(SortedActivities)) {

int ActivityNumber = dequeue(SortedActivities).Vertex;

printf ("\t%s (Activity # %d)\n",

ActivityNames[ActivityNumber],

ActivityNumber);

}

}

**Program Output:**

Starting a topological sort using an adjacency list

After reading in data, the activities and incoming edges are:

A (Activity # 1) has 2 incoming edge(s)

B (Activity # 2) has 1 incoming edge(s)

C (Activity # 3) has 1 incoming edge(s)

D (Activity # 4) has 0 incoming edge(s)

E (Activity # 5) has 3 incoming edge(s)

F (Activity # 6) has 3 incoming edge(s)

G (Activity # 7) has 1 incoming edge(s)

Topological Sort using an adjacency list is:

D (Activity # 4)

B (Activity # 2)

G (Activity # 7)

C (Activity # 3)

E (Activity # 5)

A (Activity # 1)

F (Activity # 6)

After cleanup, AllocationCount is 0

Starting a topological sort using an adjacency matrix

After reading in data, the activities and incoming edges are:

A (Activity # 1) has 2 incoming edge(s)

B (Activity # 2) has 1 incoming edge(s)

C (Activity # 3) has 1 incoming edge(s)

D (Activity # 4) has 0 incoming edge(s)

E (Activity # 5) has 3 incoming edge(s)

F (Activity # 6) has 3 incoming edge(s)

G (Activity # 7) has 1 incoming edge(s)

Graph has a cycle.. topological sort not possible!

**Problem 4 (Extra Credit) follows…**  
**Description:** Problem 4 is designed to demonstrate building a topological sort. It consists of a list of 13 vertices and their constraints as well as functions and definitions for the topological sort.

**Data File:**

13 13

9 7 7 8 9 8

4 2 5 2 2 1

2 6 3 6 1 3

3 10 10 11

12 13 12 11

1 A

2 B

3 C

4 D

5 E

6 F

7 G

8 H

9 I

10 J

11 K

12 L

13 M

**Program Output:**

The Activity Requirements are:

1 -> 3

2 -> 1, 2 -> 6

3 -> 6, 3 -> 10

4 -> 2

5 -> 2

7 -> 8

9 -> 7, 9 -> 8

10 -> 11

12 -> 11, 12 -> 13

Topological sort of activities is:

1. L (Activity 12)

2. M (Activity 13)

3. I (Activity 9)

4. G (Activity 7)

5. H (Activity 8)

6. E (Activity 5)

7. D (Activity 4)

8. B (Activity 2)

9. A (Activity 1)

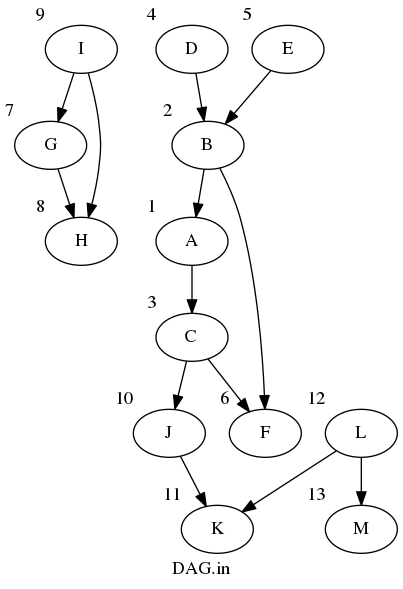
10. C (Activity 3)

11. J (Activity 10)

12. K (Activity 11)

13. F (Activity 6)

**Diagram:**

****