KIT205 Data Structures and Algorithms

Week 8 Tutorial

In this tutorial you will be implementing a graph data structure for use in later tutorials and the second assignment.

1. Create a new empty project
2. Add a file called *graph.h* with the following type definitions

```
typedef struct edge{
    int to_vertex;
    int weight;
} Edge;

typedef struct edgeNode{
    Edge edge;
    struct edgeNode *next;
} *EdgeNodePtr;

typedef struct edgeList{
    EdgeNodePtr head;
} EdgeList;

typedef struct graph{
    int V;
    EdgeList *edges;
} Graph;
```

The first definition defines an edge. It contains only a to-vertex and weight, the from-vertex will be implied by the position in the data structure – we are implementing an adjacency list representation.

The second two definitions are typical linked list types: one for the node, and a wrapper for the head. This time it will be a linked list of edges.

The final definition is for the graph itself and consists of the number of vertices, V, and an array of edge lists (very similar to the HashTable structure we used earlier.

1. Add a file called *main.c* with a *main* function and *#include* the graph definitions
2. Declare a variable called G in the main function that is a *Graph*

We are going to skip over keyboard input for the graph since this would be tedious. But we will again use redirected input from a file so that keyboard input is an option for debugging.

The following text defines a graph:

```
7
2
6,1 3,2
1
4,4
3
4,2 3,1 0,3
0
1
3,2
4
6,2 3,4 2,1 0,3
3
5,2 4,1 3,4
```

The first number (7) is the number of vertices in the graph.

The next number (2) is the number of edges for vertex 0.

The next row is a list of pairs with each pair representing one edge from vertex 0. i.e. 6,1 indicates an edge from vertex 0 to vertex 6 with weight 1.

The next line contains a single number (1), which is the number of edges from vertex 1.

The next row is the edges from vertex 1, and so on.

Note that the edges are listed in decreasing order of to-vertex. This will make it easier to insert them into the adjacency list in ascending order (we will just insert them at the front to reverse the order).

3. Draw a diagram of the graph and ask your tutor to check that you have understood the format correctly
4. Add a file called *input.txt* to resources and copy the above text into the file. Modify the project properties to redirect input from this file
5. Add code to *scanf* the first number and store it in G, and initialise the edge list array of G
6. Now add a for loop that loops through each vertex and:
    a. scans the number of edges from the vertex
    b. initialises the edge list for the vertex to *NULL*
7. Now add an inner for loop that loops through each edge and:
    a. creates a new edge node
    b. scans the edge to-vertex and weight (you can do this by using the format string “%d,%d”)
    c. inserts the new node at the front of the current list (refer to your previous insert_at_front function for lists)

You should now be able to run your code without errors, but there will be no output.

We could just print the graph somehow to check if it is correct, but this is messy. Instead we will calculate the in-degrees of all vertices and print those instead.

8. Add some code to initialise an array of in-degrees and set them all to 0
9. Add some code to loop through all of the edges in the graph and increment the in-degree for the to-vertex of each edge
10. Add some code to print the in-degrees and check that you got the right answer
11. Finally, if you have time, add code to free all dynamically allocated memory