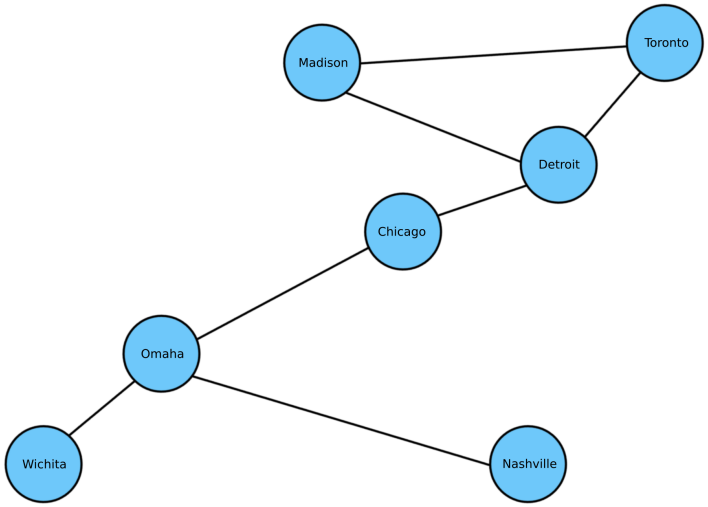
Program 6: Graphs

1 Graph Basics

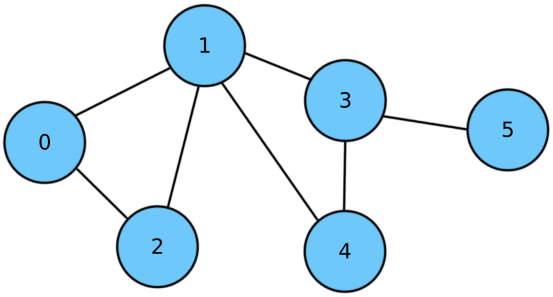
In this program, we will perform calculations on data that be represented as a *graph*. A graph is a useful tool that describes relationships between objects, and can represent a wide variety of structured data. For example you can use them to describe the links between web pages, airports and the flight routes that connect them, or computer networks.



Graphs have *nodes*, which in the example above are the blue circles, and *edges*, which are the links between nodes. Given an arbitrary node in the graph, we might want to know the number of edges that leave the node. This is commonly called the *degree* of the node. In the example above, the node Omaha has a degree of 3.

2 Input File

Your program will take a single file as input, which holds information about the graph edges. To the right, we have an example of the file format. The graph edges will each occupy a single line. Below is an example of an input file and the graph it represents.



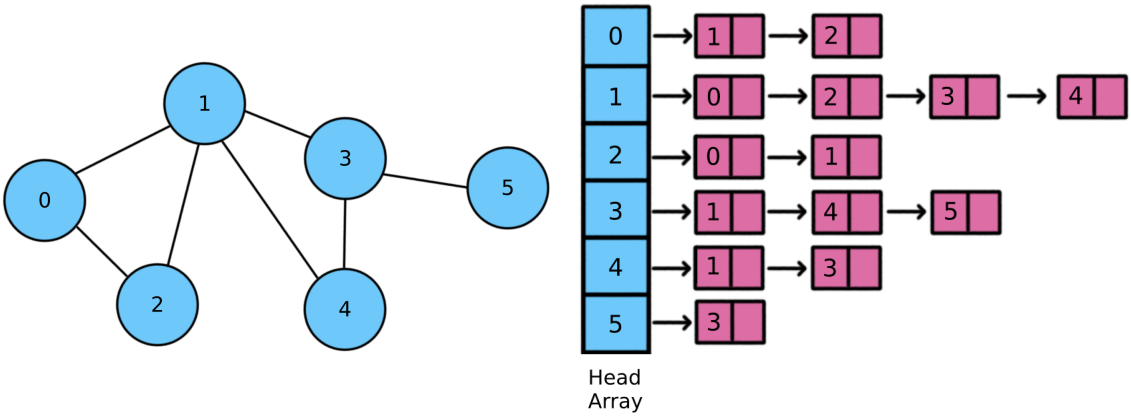

```
0 1
0 2
1 2
1 3
1 4
3 4
3 5
```

Each line of the file describes one edge in the graph, as a pair of Node *id*'s. Nodes are identified by their id, and above you can see some of those edges. For example, Node 3 shares an edge with Node 4 and another edge with Node 5. The order does not matter for id's in a line. If we had seen `1 0` in the file instead of `0 1` then there would be no difference, in terms of the graph.

3 Adjacency List

Often we want to extract important information about the graph, an example being to determine how many nodes are in the graph. It would not be very efficient to read the entire file for each query, because the graph could have millions of nodes and edges. Therefore, we will implement a well known data structure for graphs called an *Adjacency List*.

The Adjacency List, that we will develop, uses an array of pointers to linked lists. In our case we will use a vector instead of an array. Below is an example of what it would look like for the example in Section 2.



Each node in the graph is associated with one linked list, and that list is accessed through an index into the *Head Array* that matches the id of the node. For example, you can see above that the Node 1 has a linked list of size 4 (one for each of the nodes connected by edges). There are two important points to make, when constructing the Adjacency List for this assignment.

1. Each edge will introduce two nodes into the Adjacency List. For example, if there is an edge `0 1` that means the linked list for Node 0 includes Node 1 and vice versa.

2. The nodes in each list are in *ascending order* based on their id.

4 Adjacency List File Format

After constructing the Adjacency List, the next step is to store the list to a file. The file format will be to store the node id, followed by all of the nodes connected to it by an edge. To the right is the Adjacency File for the example graph in Section 3.

```
0: 1 2
1: 0 2 3 4
2: 0 1
3: 1 4 5
4: 1 3
5: 3
```

5 List Of All Tasks

1. Read the edge list into an adjacency list data structure. Instead of using an array for the head, use a vector.

2. Save the adjacency list data structure to a file

3. Print number of nodes, number of edges, and maximum degree.

For the last item, you can compute the number of edges by taking the sum of all the degrees and dividing by two. Remember that the degree of a node is the number of edges it has, therefore the maximum degree is the degree of the node with the most number of edges. Below is an example of how to print the information for the example graph in Section 3.

```
Number of nodes: 6
Number of edges: 7
Maximum degree: 4
```

6 Requirements

Failure to follow these rules can lead to a bad score.

1. The Adjacency List must be a vector of linked lists (the vector's element type should be a pointer to a Node, a Linked List object[1], or a pointer to a Linked List object). Failure to do so will result in a zero

[1] In other words, a class you define that holds a linked list inside of it

2. The edge list may have duplicates but not allowed in adjacency (ignore any duplicates after first setup)

3. The edge lists must be sorted in ascending order by id.

4. Assume node id starts at 0

5. The autograder needs an easy entry point for your program, so you will write a function, described in Section 7.

7 Required Function Definition for Autograder

You need to define a function with the following signature

```
void run(const string& src_path, const string& dst_path)
```

The `src_path` is the path to the edge list file, that your program will read to find the information about the graph. You simply need to use this path to read the data file. The `dst_path` is the path to the file that you will create that stores the Adjacency List in the file format described in Section 4.

8 Hints

1. Assume that Node id's start at 0, you can use it as the index into the vector.

2. We will use a vector to represent the "head array" of the Adjacency List, with the index into the vector corresponding to the id of a node. The elements in the vector will be pointers to the start of the linked list for that node. See the diagram in Section 3 for an example.

3. You will not know the number of nodes in the graph in advance. While you are reading the file, you might encounter a Node id which would correspond to an index outside the bounds of your vector. In that scenario, you will have to `push_back NULL` values to increase the vector size until that id will be in range.

4. The total number of nodes in the graph is the number of entries in the vector that are not `NULL`

9 Writing To A File

Writing to a file is very similar to using `cin/cout` because we can set them up as streams to read and write. Below is an example of how to open a file as a stream, write a single line `HELLO\n` to the file, then closing the stream.

```
1 #include <fstream>
2
3 int main()
4 {
5     ofstream fstr{"file.txt"};
6     fstr << "HELLO" << endl;
7     fstr.close();
8     return 0;
9 }
```

Line 5 creates an `ofstream` object that will accept the stream operator `<<` and allow writing to the file. The name of the file [2] is passed as an argument to the constructor, and it is located in the same directory as the executable.

[2]The ofstream can open a file described by a relative path. The relative path to the file is the location of the file in your machine's active filesystem relative to the program executable. If the file you are trying to open is in the same directory as the executable, which was produced by the compiler, then the relative path is simply the file name. This is why the line constructing the ofstream opens a file called file.txt in the same directory as the executable.