Credit to: https://codeforces.com/blog/entry/67883

Introduction to Adjacency Edge Lists

This only works for directed graphs. We'll make them undirected later.

The idea of this structure is to number all the edges given.
An array will tell a single edge number, x, for a given node.

If the degree of this node is 0, you'll get -1 as the value.
If not, the value stored is the *highest edge number* where that edge's "from" is the given node.

Obviously, a node can have more than one edge (degree > 1). So, upon accessing edge x, it will give you another number to the next highest edge number whose edge also starts from the given node. This continues until you reach an edge whose "next highest edge" value is -1 (no more edges with the given node as the "from" node).

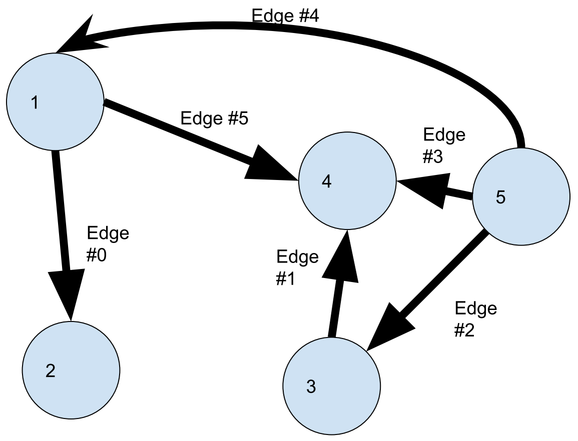
It's kind of like linked lists - each edge has its own information and then points you to the next edge, until you reach the end.

What the data structure looks like:

`int[] nodes` //the *ith* index represents node *i*, and its value is the highest edge number where that edge's starting node is *i*.

`edgeData[] edges //`if you're accessing this edge, you know the "from/start" node, and you're figuring out the "to/end" node of the edge. It'll also tell you the cost if it's weighted. And then it has a "next" value which will tell you the number of another edge involving the given "start" node.

Example of a directed graph:



Do not confuse edge number with a weighted graph (these are not the costs of these edges).

Nodes:

Node i	*Highest outgoing edge #*
1	**5**
2	**-1 //Has an out_degree of 0**
3	**1**
4	**-1**
5	**4**

Edge Data

Edge #	*To/end node of this edge*	*Next highest edge# that is outgoing from the given "from/start " node*
0	**2**	**-1 //No more edges**
1	**4**	**-1**
2	**3**	**-1**
3	**4**	**2**
4	**1**	**3**
5	**4**	**0**

Adding an Edge

When the input gives you a new directed edge, update like so:

```
addEdge(int edgenumber, node from, node to) {
      edgeData[edgenumber].to = to
      edgeData[edgenumber].next = node[from]
      node[from]  = edgenumber
}
```

Assuming you add the edges in order, the most recent added edge will be the highest edge# for the node "from". So, its # will need to go into the node array. The value replaced in the node array is now the next-highest edge (2nd highest) after the edge you are currently adding. You need to do that in the code first so that values do not get deleted forever.

Traversing a Directed Graph

Let's say we wanted to print all adjacent nodes to node 5. Pseudo code:

```
for(int i = nodes[5]; i != -1; i = edgeData[i].next) {
      print(edgeData[i].end)
}
In this case:
```

```
i = 4th edge, prints "1". i gets changed to node #3
i = 3rd edge, prints "4". i gets changes to node #2
i = 2nd node, prints "3". i gets changed to node #-1.
STOP.
```

Let's say we wanted to print the "out degree" of node 1, and then print the "out degree" of node 4. Pseudo code:

```
print(outdegree(1))
print(outdegree(4))

int outdegree (node num) {
      ans = 0
      if(nodes[num] == -1):
            return 0 //no edges at all
      for(int i = nodes[num]; i != -1; i = edgeData[i].next) {
            ans += 1
            }
      return ans
}
```

Making an Undirected Graph

To make it undirected we can add two edges, one going a → b and the other going b → a. Make sure to add both edges one immediately after the other. This will be convenient because then one undirected edge will be represented by two directed edges with edge numbers (0,1), (2,3), (4,5), etc.

```
addEdge(number, a, b)
number++
addEdge(number, b, a)
```

Putting them in pairs like this allows us to figure out the directed edge and its opposite edge quickly.
For edge "X"

- If X is an even number, the opposite edge is X+1.
- If X is an odd number, the opposite edge is X-1.