15 Digraphs

A *directed graph* or *digraph* is a pair (V, E) where V is a finite set, elements called *vertices*, and E is a subset of

$$\{(u, v) \in V \times V : \quad u \neq v\}$$

whose elements are called *directed edges.* (u, v) is an edge from u to v, and is an *out-edge* from u (and an in-edge into v).

For our purposes we use a simple representation.

```
typedef struct EDGE { int from, to; struct EDGE * next, * prev; } EDGE;
```

So it is an edge *from* `from` *to* `to`.

The pointers `next, prev` link together, in circular fashion, all the edges emanating from the same vertex (i.e., all edges with the same value of `from`).

The list of edges from u is called (in this module, anyway) the *forward adjacency list* from u.

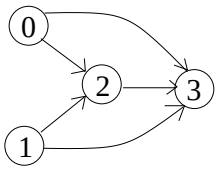
```
typedef struct { int n, m; int * out_deg; EDGE ** an_edge; } DIGRAPH;
```

The digraph has n vertices and m edges (the value of m is updated when edges are added). The vertices coincide with the numbers $0 \ldots n-1$.

For $0 \leq u < n$, `an_edge[u]` is either null or points to some out-edge from u. Since the out-edges are linked in a circular list, it does not matter much which of these edges is assigned to `an_edge[u]`.

Also, the size of the list of out-edges from u is in `outdeg[u]`; this is called the out-degree at u. This makes traversing the out-edges easier and safer.

The usual way of depicting digraphs is rather like that used for trees.



input:

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

For input purposes, a digraph will be presented as

```
n m          (actually, the number of edges begins at zero)
0 size_0 t_0 ... t_size_0   t_j: the j-th 'to' vertex.
1 size_1 u_0 ... u_size_1
... etcetera
n-1 etcetera
```

For example

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

Next is a complete `build-digraph` program. It may be faulty, having only been tested on the data shown.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct EDGE { int from, to; struct EDGE * next, * prev; } EDGE;
typedef struct { int n, m; int * out_deg; EDGE ** an_edge; } DIGRAPH;

DIGRAPH * make_digraph ( int n )
{
  DIGRAPH* digraph = (DIGRAPH*) calloc(1,sizeof(DIGRAPH));

        // Creates a 'graph' object.  Calloc
        // initialises every bit to zero.

  digraph->n = n;
        // The vertices 0...n-1

  digraph->an_edge = (EDGE**) calloc(n, sizeof(EDGE*));
        // The array of pointers to out-edge lists

  digraph->out_deg = (int*) calloc ( n, sizeof(int) );
        // The out-degrees

  return digraph;
}

EDGE * add_edge ( int from, int to, DIGRAPH * digraph )
{
  EDGE * edge = (EDGE*) calloc(1,sizeof(EDGE));
  edge->from = from; edge->to = to;

        // an edge from 'from' to 'to.'

  EDGE * insertion_point = digraph -> an_edge[ from ];

        // edge will be inserted just before this
        // 'insertion point,' and that insertion
```

```
        // point remains unchanged.

  if ( insertion_point == NULL )
        // First edge out of 'from' inserted.
        // In a circular list, it points back
        // to itself.
  { digraph->an_edge[from] = edge;
    edge->next = edge->prev = edge;
  }
  else
  {
    EDGE * lastadded = insertion_point->prev;
        // this was the last edge added --- possibly
        // the first as well.
    lastadded->next = edge; edge->prev = lastadded;
    insertion_point -> prev = edge; edge->next = insertion_point;
  }

  ++ digraph->m;
        // increment the edge count

  ++ (digraph->out_deg[from]);
        // increment the outdegree count

  return edge;
}

void print_digraph ( DIGRAPH * digraph )
{
  printf ( "%d %d\n", digraph->n, digraph->m );
  int i;
  for (i=0; i<digraph->n; ++i)
  {
    int size = digraph->out_deg[i];
    printf("%d %d", i, size);
    int j;
    EDGE * edge = digraph->an_edge[i];
    for (j=0; j<size; ++j)
    {  printf(" %d", edge->to); edge = edge->next; }
    printf("\n");
  }
}

int main()
{
```

```
  int n, m;

  scanf("%d %d", &n, &m );

  DIGRAPH * digraph = make_digraph ( n );

  // now add the edges

  int i;
  for (i=0; i<n; ++i)
  {
    int size;
    int ignore;  // skip first int
    scanf("%d %d", &ignore, & size);
    int j;
    for ( j = 0; j<size; ++j )
    {
      int k;
      scanf("%d", &k);
      add_edge ( i, k, digraph);
    }
  }
  print_digraph ( digraph );
}
```

One test

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

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