>>

Graph ADT

❖ Graph ADT

Data: set of edges, set of vertices

Operations:

- building: create graph, add edge
- deleting: remove edge, drop whole graph
- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as **`int`**s, but could be arbitrary **`Item`**s

Will use this ADT as a basis for building more complex operations later.

❖ ... Graph ADT

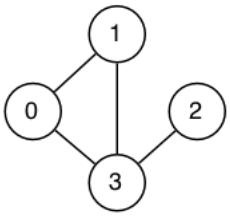
Graph ADT interface **`Graph.h`**

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);  // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex);
      // is there an edge between two vertices?
void  freeGraph(Graph);
```

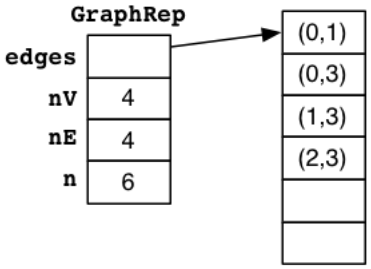
❖ Graph ADT (Array of Edges)

Implementation of **`GraphRep`** (array-of-edges representation)

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..nV-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```



Undirected graph



<< ^ >>

❖ ... Graph ADT (Array of Edges)

Implementation of graph initialisation (array-of-edges)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));
   assert(g->edges != NULL);
   return g;
}
```

How much is enough? ... No more than *V(V-1)/2* ... Much less in practice (sparse graph)

<< ^ >>

❖ ... Graph ADT (Array of Edges)

Some useful utility functions:

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
}

// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

// check if an edge is valid in a graph
bool validE(Graph g, Edge e) {
   return (g != NULL && validV(e.v) && validV(e.w));
}
```

❖ ... Graph ADT (Array of Edges)

Implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n && isValidE(g,e));
   int i = 0;  // can't define in for (...)
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE)  // edge e not found
      g->edges[g->nE++] = e;
}
```

❖ ... Graph ADT (Array of Edges)

Implementation of edge removal (array-of-edges)

```
void removeEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)  // edge e found
      g->edges[i] = g->edges[--g->nE];
}
```

❖ ... Graph ADT (Array of Edges)

Implementation of edge check (array-of-edges)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));
   Edge e;
   e.v = x; e.w = y;
   for (int i = 0; i < g->nE; i++) {
      if (eq(e,g->edges[i]))  // edge found
         return true;
   }
   return false;  // edge not found
}
```

<< ^ >>

❖ ... Graph ADT (Array of Edges)

Re-implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE) { // edge e not found
      if (g->n == g->nE) {  // array full; expand
         g->edges = realloc(g->edges, 2*g->n);
         assert(g->edges != NULL);
         g->n = 2*g->n;
      }
      g->edges[g->nE++] = e;
   }
}
```

<< ^ >>

❖ ... Graph ADT (Array of Edges)

Implementation of graph removal (array-of-edges)

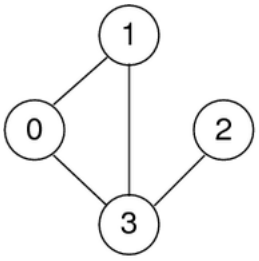
```
void freeGraph(Graph g) {
   assert(g != NULL);
   free(g->edges);  // free array of edges
   free(g);         // remove Graph object
}
```

<< ^ >>

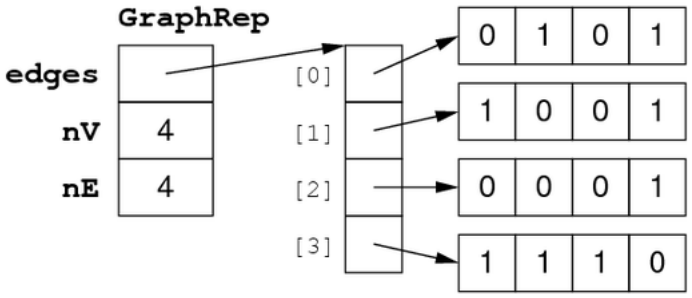
❖ Graph ADT (Adjacency Matrix)

Implementation of **`GraphRep`** (adjacency-matrix representation)

```
typedef struct GraphRep {
   int  **edges; // adjacency matrix
   int    nV;    // #vertices
   int    nE;    // #edges
} GraphRep;
```



Undirected graph



❖ ... Graph ADT (Adjacency Matrix)

Implementation of graph initialisation (adjacency-matrix)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate array of pointers to rows
   g->edges = malloc(V * sizeof(int *));
   assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (int i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int));
      assert(g->edges[i] != NULL);
   }
   return g;
}
```

Standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size **`nelems*nbytes`**
- and sets all bytes in that block to zero

<< ^ >>

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge insertion (adjacency-matrix)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}
```

<< ^ >>

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge removal (adjacency-matrix)

```
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge check (adjacency matrix)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return (g->edges[x][y] != 0);
}
```

Note: all operations, except creation, are *O(1)*

<< ^ >>

❖ ... Graph ADT (Adjacency Matrix)

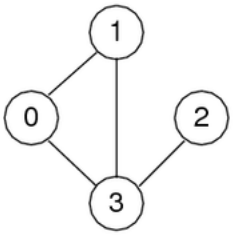
Implementation of graph removal (adjacency matrix)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      // free one row of matrix
      free(g->edges[i]);
   free(g->edges);  // free array of row pointers
   free(g);         // remove Graph object
}
```

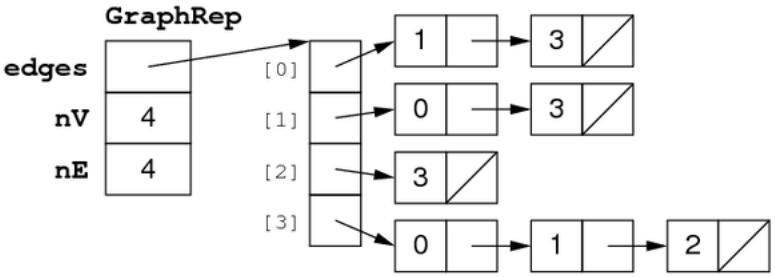
❖ Graph ADT (Adjacency Lists)

Implementation of **`GraphRep`** (adjacency-lists representation)

```
typedef struct GraphRep {
   Node **edges;  // array of lists
   int    nV;     // #vertices
   int    nE;     // #edges
} GraphRep;
```



Undirected graph



<< ^ >>

❖ ... Graph ADT (Adjacency Lists)

Assume that we have a linked list implementation

```
typedef struct Node {
   Vertex v;
   struct Node *next;
} Node;
```

with operations like **`inLL`**, **`insertLL`**, **`deleteLL`**, **`freeLL`**, e.g.

```
bool inLL(Node *L, Vertex v) {
   while (L != NULL) {
      if (L->v == v) return true;
      L = L->next;
   }
   return false;
}
```

❖ ... Graph ADT (Adjacency Lists)

Implementation of graph initialisation (adjacency lists)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate memory for array of lists
   g->edges = malloc(V * sizeof(Node *));
   assert(g->edges != NULL);
   for (int i = 0; i < V; i++)
      g->edges[i] = NULL;
   return g;
}
```

<< ^ >>

❖ ... Graph ADT (Adjacency Lists)

Implementation of edge insertion/removal (adjacency lists)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (!inLL(g->edges[e.v], e.w)) {  // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (inLL(g->edges[e.v], e.w)) {  // edge e in graph
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

❖ ... Graph ADT (Adjacency Lists)

Implementation of edge check (adjacency lists)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return inLL(g->edges[x], y);
}
```

Note: all operations, except creation, are *O(E)*

❖ ... Graph ADT (Adjacency Lists)

Implementation of graph removal (adjacency lists)

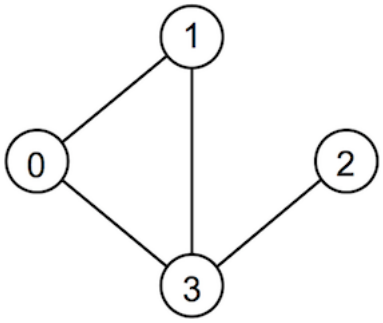
```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      freeLL(g->edges[i]); // free one list
   free(g->edges);  // free array of list pointers
   free(g);         // remove Graph object
}
```

<< ^ >>

❖ Example: Graph ADT Client

A program that uses the graph ADT to

- build the graph depicted below
- print all the nodes that are incident to vertex **1** in ascending order



<< ^

❖ ... Example: Graph ADT Client

```
#include <stdio.h>
#include "Graph.h"

#define NODES 4
#define NODE_OF_INTEREST 1

int main(void) {
   Graph g = newGraph(NODES);
   Edge e;

   while (scanf("%d %d", &(e.v), &(e.w)) == 2)
      insertEdge(g,e);

   for (Vertex v = 0; v < NODES; v++) {
      if (adjacent(g, v, NODE_OF_INTEREST))
         printf("%d\n", v);
   }

   freeGraph(g);
   return 0;
}
```

Produced: 22 Jun 2020