CSCE 4263/5183
Advanced Data Structures

Lecture 16

Graph Data Structures

Fall 2025

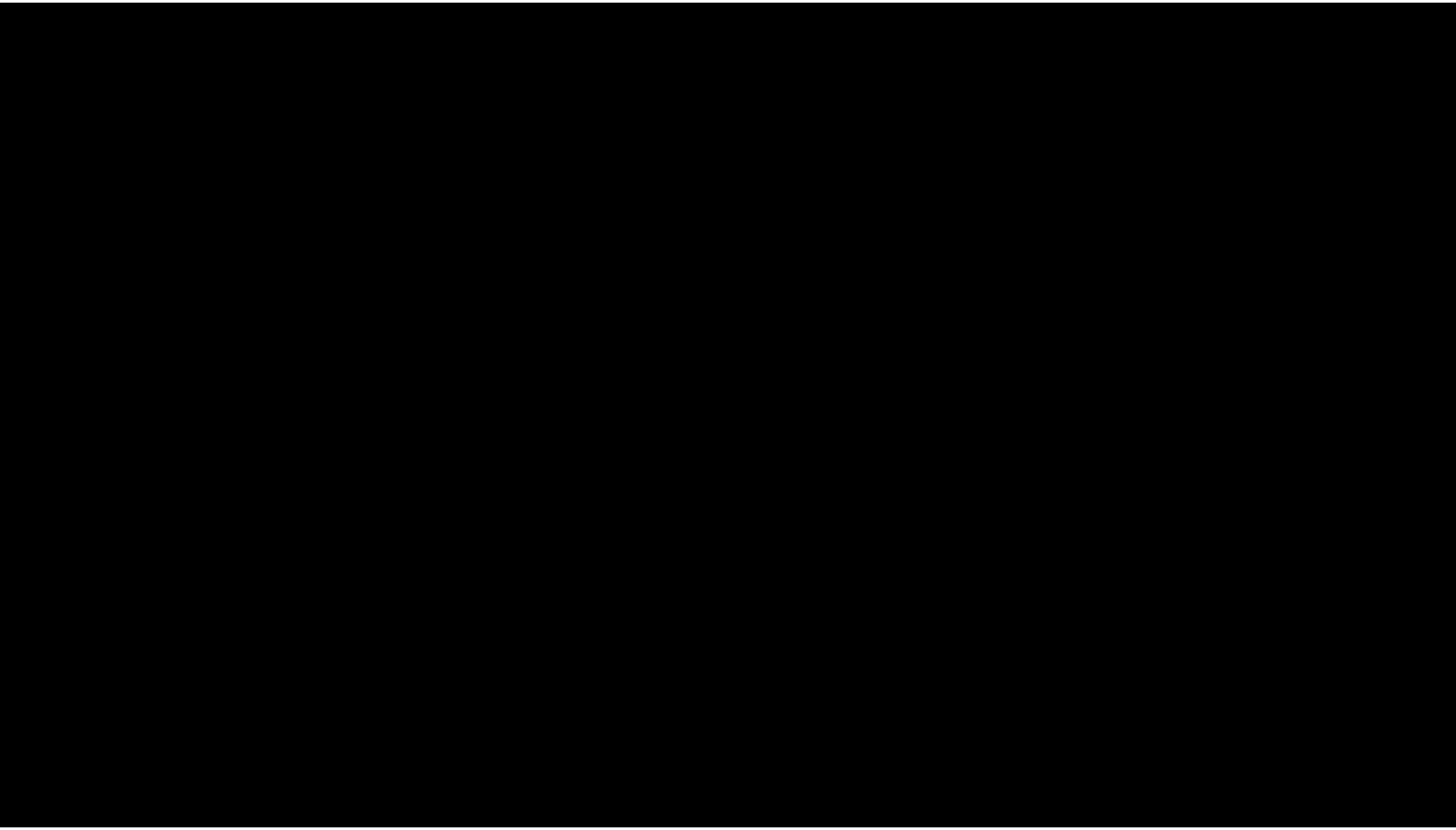
Prof. Khoa Luu
khoaluu@uark.edu

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final Project Presentations

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final Project Presentations



Outline

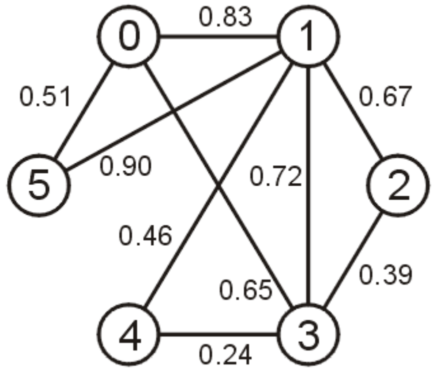
- In this topic, we will cover the <u>representation of graphs</u> on a computer

Outline

- In this topic, we will cover the representation of graphs on a computer
- We will examine:
  - an adjacency matrix representation
  - smaller representations and pointer arithmetic
  - sparse matrices and linked lists

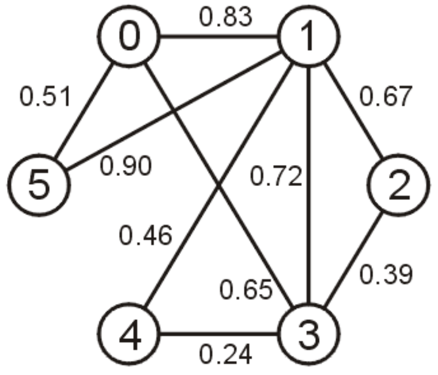
Background

- You are required to store a graph with a given number of vertices numbered 0 through $n - 1$



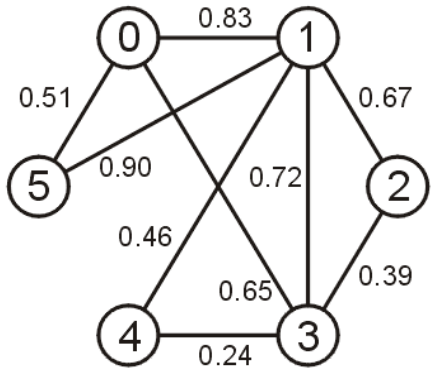
Background

- You are required to store a graph with a given number of vertices numbered 0 through $n - 1$
- Initially, there are no edges between these n vertices



Background

- You are required to store a graph with a given number of vertices numbered 0 through $n - 1$
- Initially, there are no edges between these n vertices
- The **`insert`** command adds edges to the graph while the number vertices remains unchanged



Background

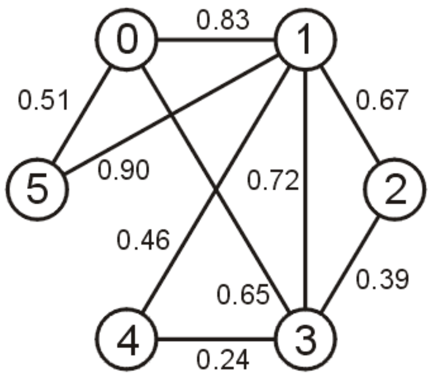
- In this lecture, we will look at techniques for storing the edges of a graph

Background

- In this lecture, we will look at techniques for storing the edges of a graph
- This lecture will focus on weighted graphs, however, for unweighted graphs, one can easily use **`bool`** in place of **`double`**

Background

- To demonstrate these techniques, we will look at storing the edges of the following graph:



Adjacency Matrix

A graph of n vertices may have up to ? edges

Adjacency Matrix

A graph of n vertices may have up to

$$\binom{n}{2} = \frac{n(n-1)}{2} = \mathbf{O}(n^2)$$

edges

The first straight-forward implementation is an adjacency matrix

Adjacency Matrix

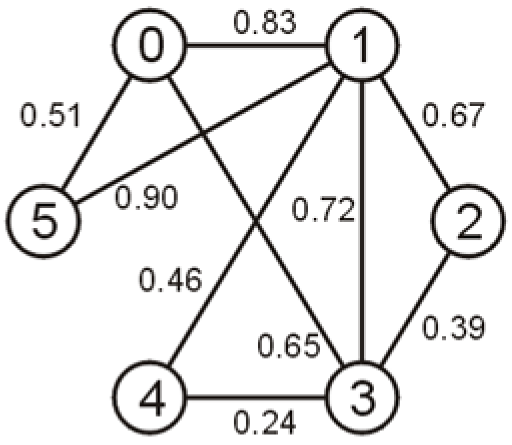
Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

Adjacency Matrix

Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

That is, the matrix is symmetric, *e.g.*,

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



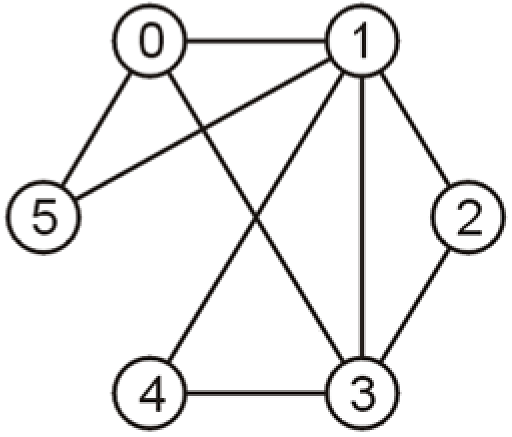
Adjacency Matrix

An unweighted graph may be saved as an array of Boolean values

– vertices v_i and v_j are connected then set

$a_{ij} = a_{ji} = true$

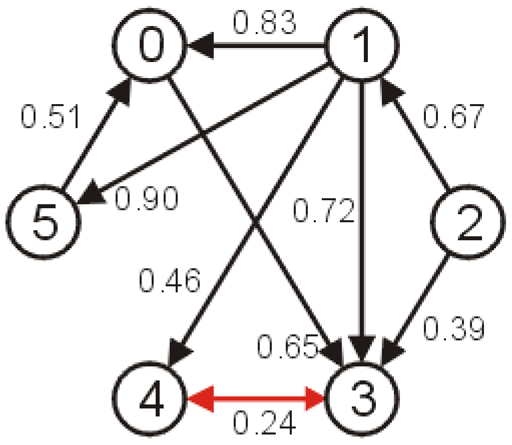
	0	1	2	3	4	5
0		T	F	T	F	T
1	T		T	T	T	T
2	F	T		T	F	F
3	T	T	T		T	F
4	F	T	F	T		F
5	T	T	F	F	F	



Adjacency Matrix

If the graph was directed, then the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



Adjacency Matrix

First we must allocate memory for a **two-dimensional array**

C++ does not have native support for anything more than one-dimensional arrays, thus how do we store a two-dimensional array?

Adjacency Matrix

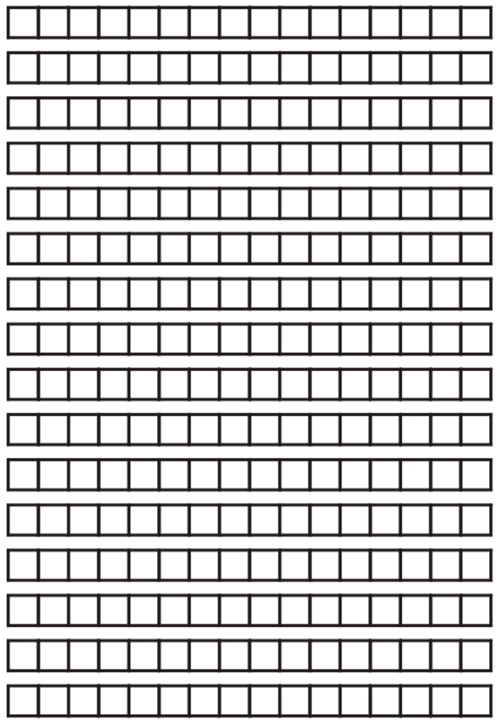
First we must allocate memory for a **two-dimensional array**

C++ does not have native support for anything more than one-dimensional arrays, thus how do we store a two-dimensional array?

– as an array of arrays

Adjacency Matrix

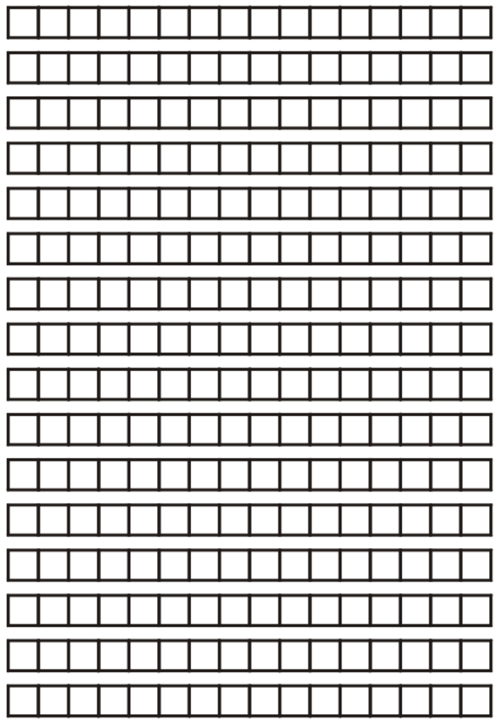
Suppose we require a 16 × 16 matrix of double-precision floating-point numbers



Adjacency Matrix

Suppose we require a 16 × 16 matrix of double-precision floating-point numbers

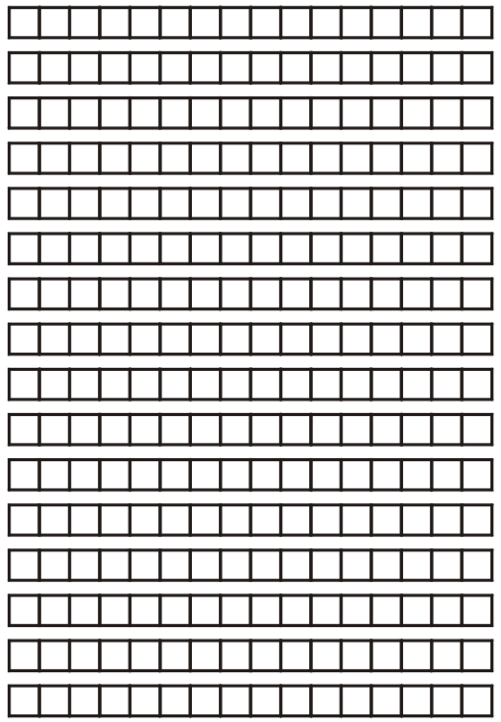
Each row of the matrix can be represented by ?



Adjacency Matrix

Suppose we require a 16 × 16 matrix of double-precision floating-point numbers

Each row of the matrix can be represented by an array

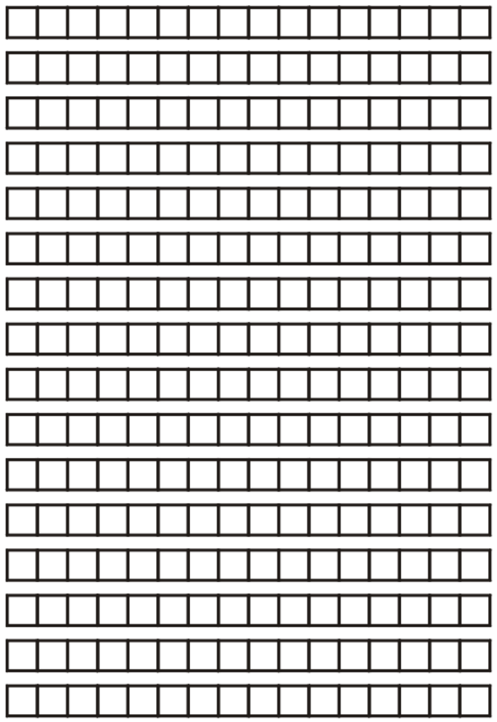


Adjacency Matrix

Suppose we require a 16 × 16 matrix of double-precision floating-point numbers

Each row of the matrix can be represented by an array

The address of the first entry must be stored in ?



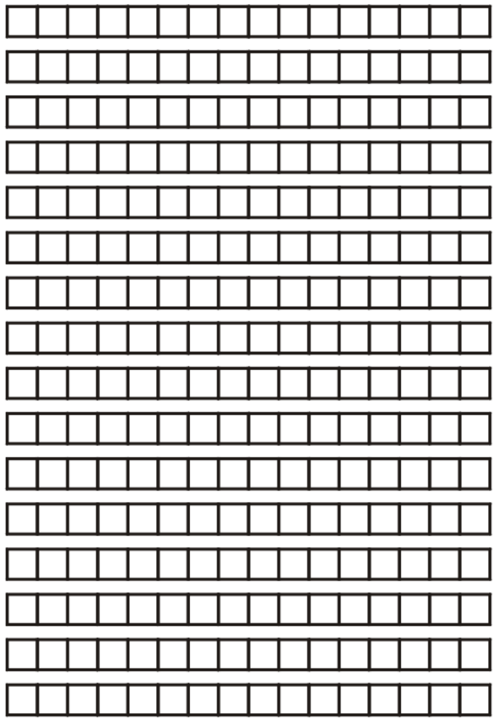
Adjacency Matrix

Suppose we require a 16 × 16 matrix of double-precision floating-point numbers

Each row of the matrix can be represented by an array

The address of the first entry must be stored in a pointer to a double:

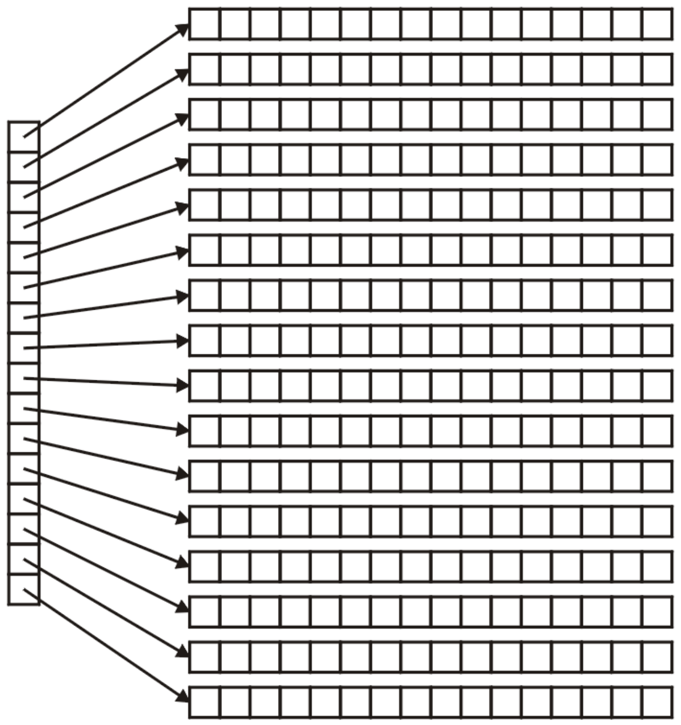
```
    double *
```



Adjacency Matrix

However, because we must store 16 of these pointers-to-doubles, it makes sense that we store these in an array

What is the declaration of this array?



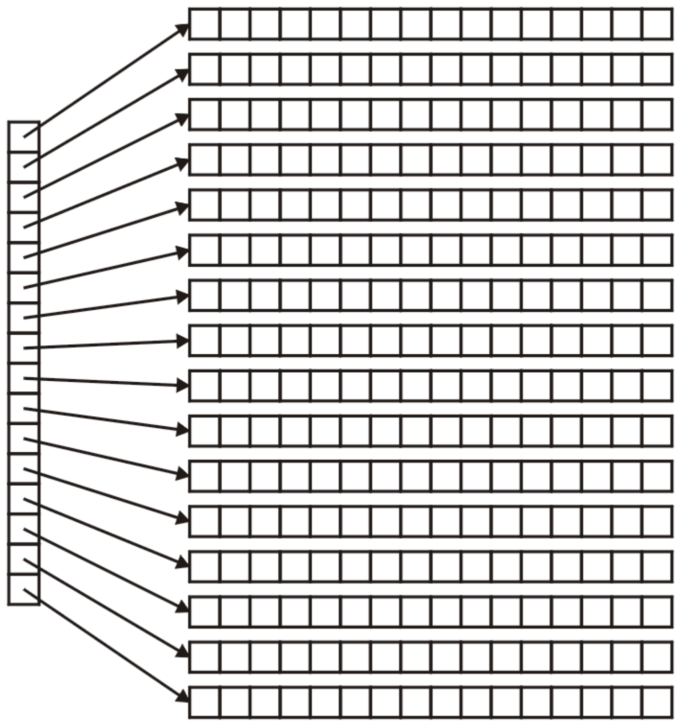
Adjacency Matrix

However, because we must store 16 of these pointers-to-doubles, it makes sense that we store these in an array

What is the declaration of this array?

Well, we must store a

pointer to a pointer to a double



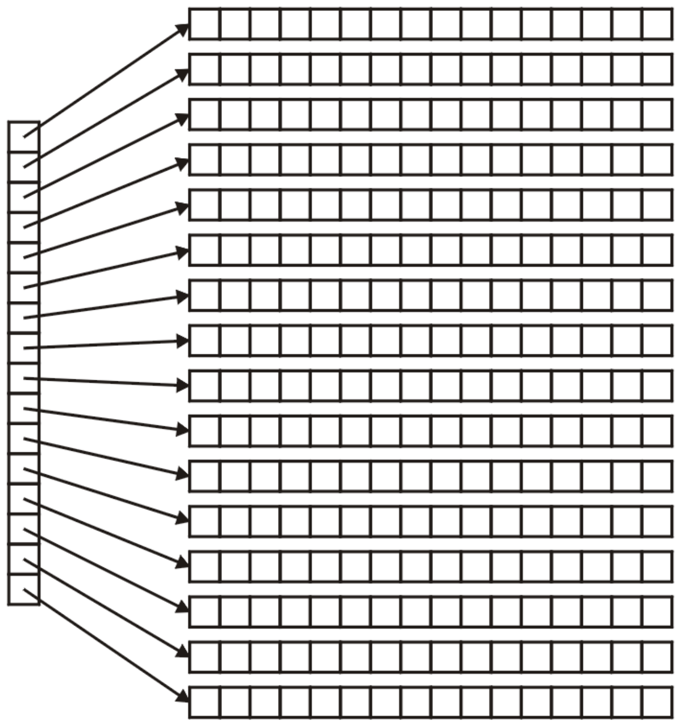
Adjacency Matrix

However, because we must store 16 of these pointers-to-doubles, it makes sense that we store these in an array

What is the declaration of this array?

Well, we must store a
pointer to a pointer to a double

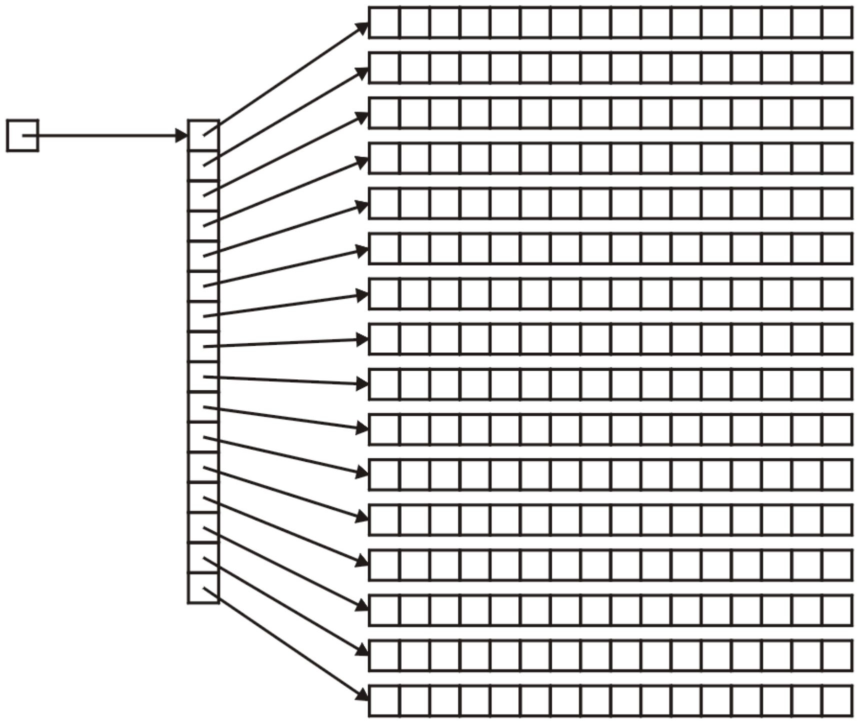
That is: `double **`



Adjacency Matrix

Thus, the address of the first array must be declared to be:

```
double **matrix;
```

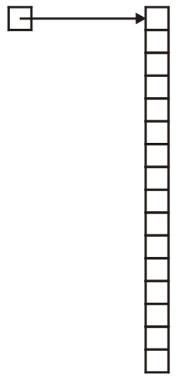


Adjacency Matrix

The next question is memory allocation

First, we must allocate the memory for the array of pointers to doubles:

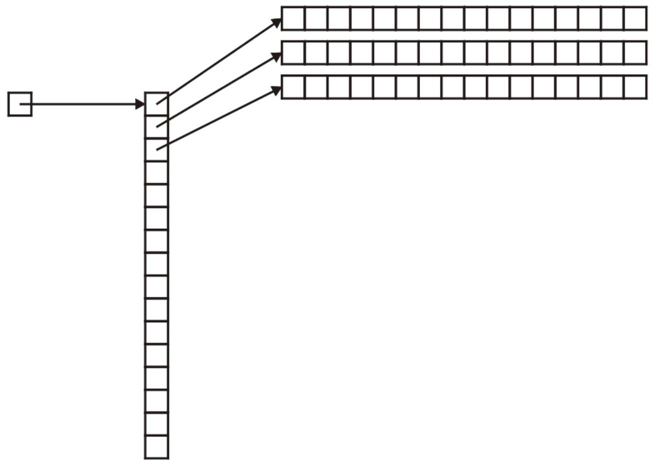
```
  matrix = new double * [16];
```



Adjacency Matrix

Next, to each entry of this matrix, we must assign the memory allocated for an array of doubles

```
  for ( int i = 0; i < 16; ++i ) {
     matrix[i] = new double[16];
 }
```



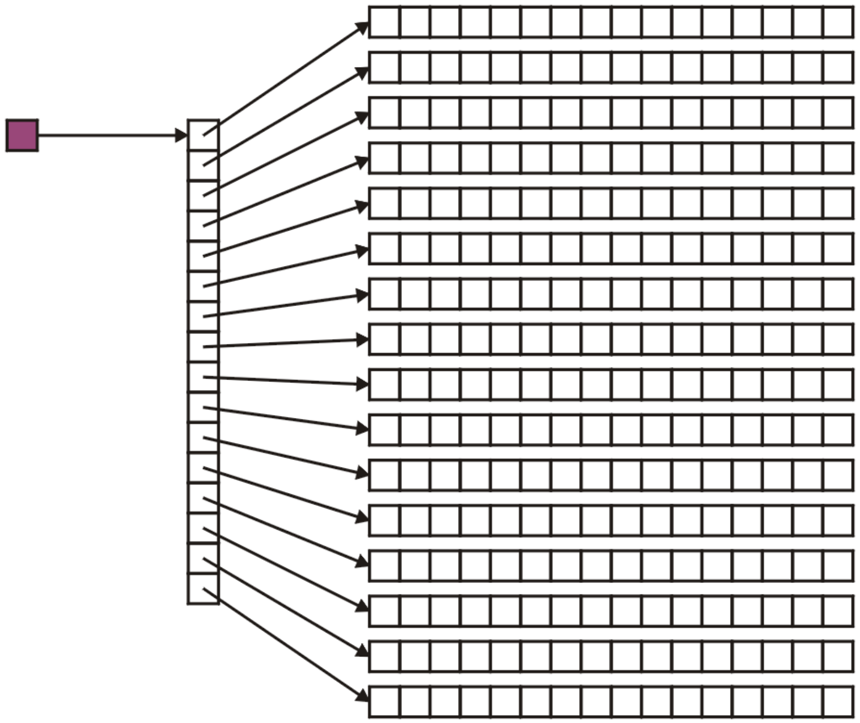
Adjacency Matrix

Accessing a matrix is done through a double index, *e.g.*, **`matrix[3][4]`**

You can interpret this as **`(matrix[3])[4]`**

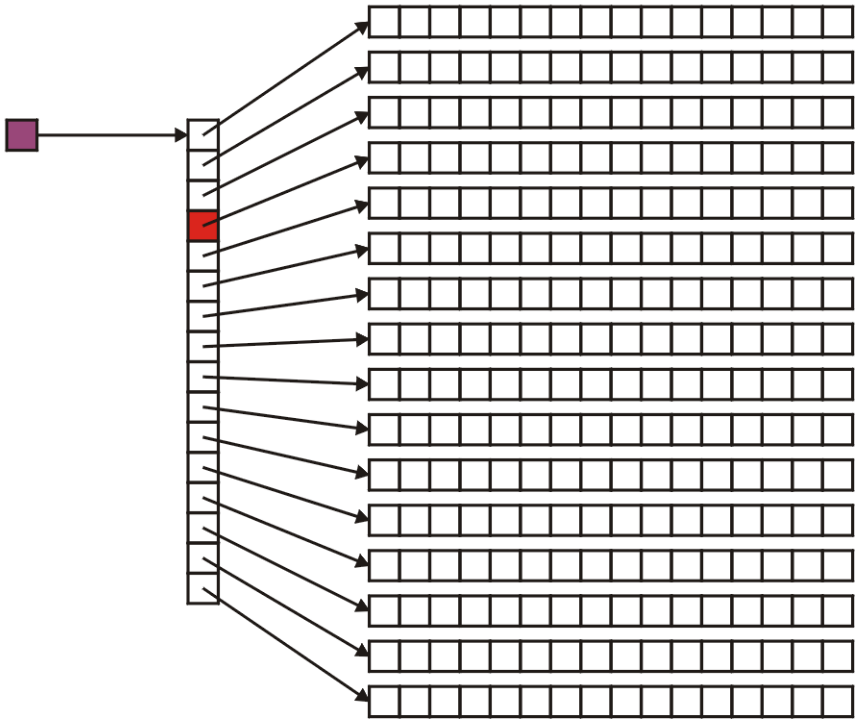
Adjacency Matrix

Recall that in **`matrix[3][4]`**, the variable `matrix` is a pointer-to-a-pointer-to-a-double:



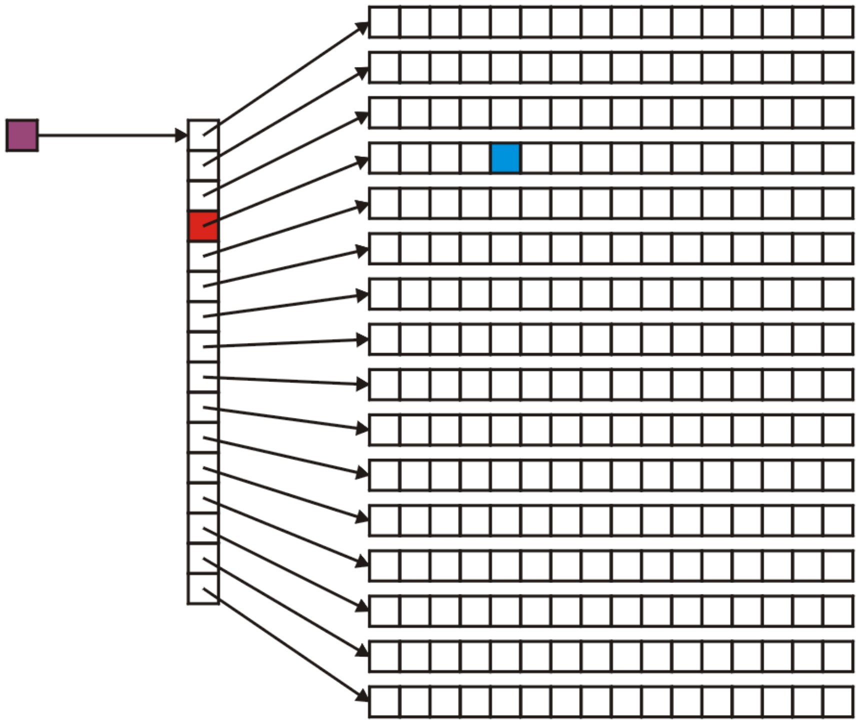
Adjacency Matrix

Therefore, `matrix[3]` is a pointer-to-a-double:



Adjacency Matrix

And consequently, `matrix[3][4]` is a double:



C++ Notation Warning

Do not use **`matrix[3, 4]`** because:

- in C++, the comma operator evaluates the operands in order from left-to-right
- the *value* is the last one

C++ Notation Warning

Do not use **`matrix[3, 4]`** because:

- in C++, the comma operator evaluates the operands in order from left-to-right
- the *value* is the last one

Therefore, **`matrix[3, 4]`** is equivalent to calling **`matrix[4]`**

Try it:

```
    int i = (3, 4);
    cout << i << endl;
```

C++ Notation Warning

Many things will compile if you try to use this notation:

```
    matrix = new double[N, N];
```

will allocate an array of N doubles, just like:

```
    matrix = new double[N];
```

However, this is likely not to do what you really expect...

Adjacency Matrix

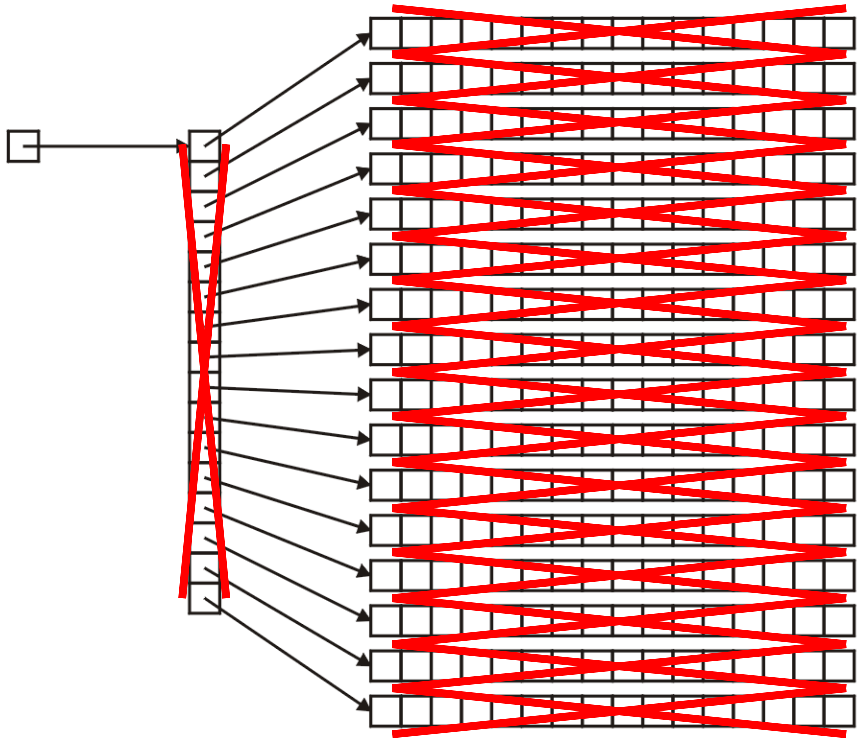
Now, once you’ve used the matrix, you must also delete it...

Adjacency Matrix

Recall that for each call to `new[]`,you must have a corresponding call to `delete[]`

Therefore, we must use a for-loop to delete the arrays

– implementation up to you



Default Values

Question: what do we do about vertices which are not connected?

- the value 0
- a negative number, *e.g.*, -1
- positive infinity: ∞

The last is the most logical, in that it makes sense that two vertices which are not connected have an infinite distance between them

Default Values

To use infinity, you may declare a constant static member variable `INF`:

```
#include <limits>

class Weighted_graph {
    private:
        static const double INF;
        // ...
    // ...
};

const double Weighted_graph::INF =
    std::numeric_limits<double>::infinity();
```

Default Values

In this case, you can initialize your array as follows:

```
for ( int i = 0; i < N; ++i ) {
    for ( int j = 0; j < N; ++j ) {
        matrix[i][j] = INF;
    }

    matrix[i][i] = 0;
}
```

It makes intuitive sense that the distance from a node to itself is `0`

Default Values

If we are representing an unweighted graph, use Boolean values:

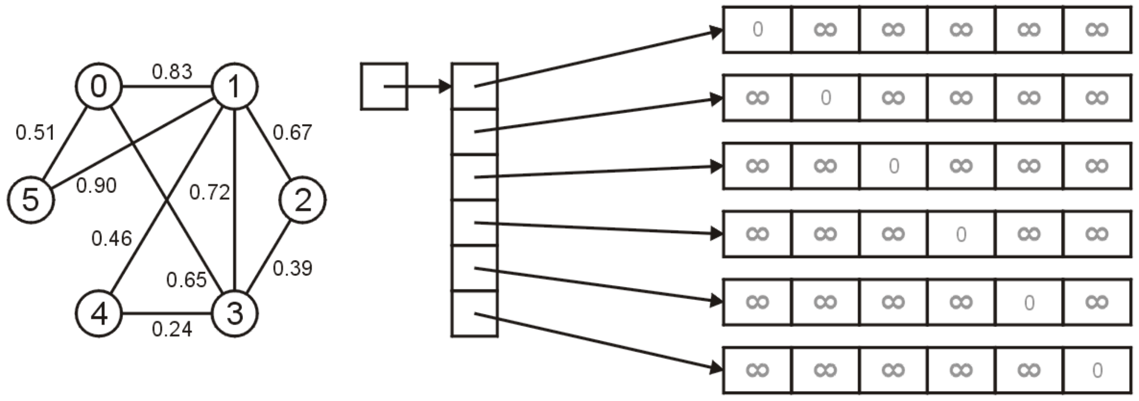
```
for ( int i = 0; i < N; ++i ) {
    for ( int j = 0; j < N; ++j ) {
        matrix[i][j] = false;
    }

    matrix[i][i] = true;
}
```

It makes intuitive sense that a vertex is connected to itself

Adjacency Matrix

Let us look at the representation of our example graph

Initially none of the edges are recorded:

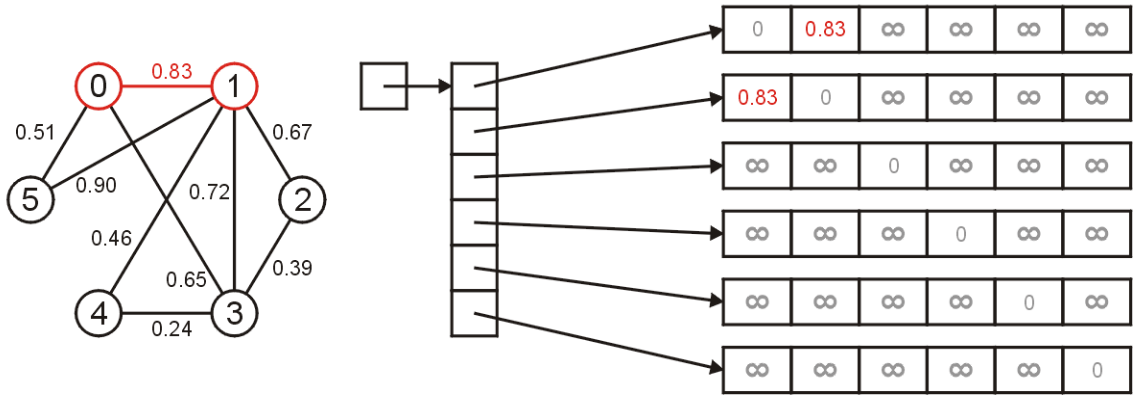


0	∞	∞	∞	∞	∞
∞	0	∞	∞	∞	∞
∞	∞	0	∞	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	0

Adjacency Matrix

To insert the edge between 0 and 1 with weight 0.83, we set

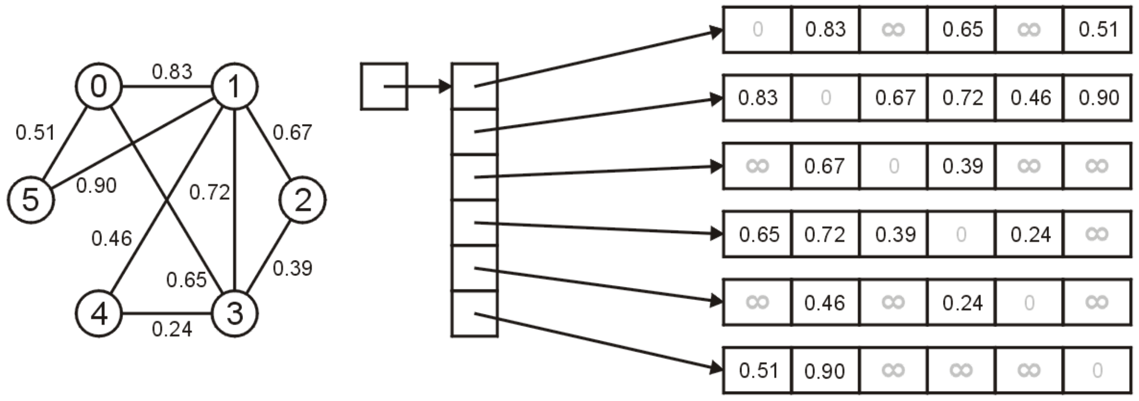
```
matrix[0][1] = matrix[1][0] = 0.83;
```



Adjacency Matrix

The final result is shown as follows

Note, however, that these six arrays could be anywhere in memory...



Adjacency Matrix

We have now looked at how we can store an adjacency graph in C++

Next, we will look at:

- Two improvements for the array-of-arrays implementations, including:
  - allocating the memory for the matrix in a single contiguous block of code, and
  - a lower-triangular representation; and
- A sparse linked-list implementation

Adjacency Matrix Improvement

To begin, we will look at the first improvement:

- allocating all of the memory of the arrays in a single array with n^2 entries

Adjacency Matrix Improvement

For those of you who would like to reduce the number of calls to **`new`**, consider the following idea:

- allocate an array of 16 pointers to doubles
- allocate an array of $16^2 = 256$ doubles

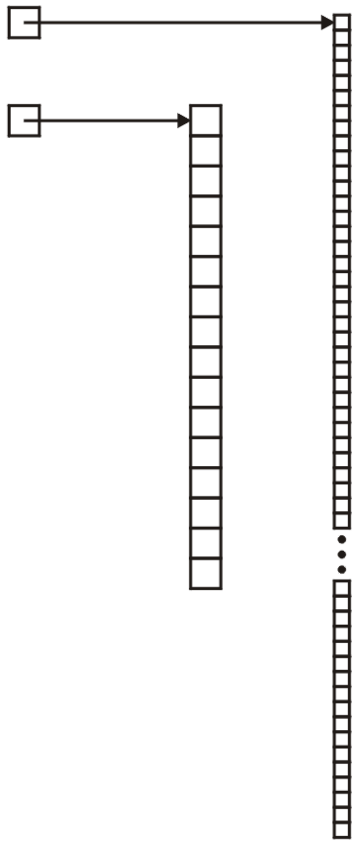
Then, assign to the 16 pointers in the first array the addresses of entries

0, 16, 32, 48, 64, ..., 240

Adjacency Matrix Improvement

First, we allocate memory:

```
matrix = new double * [16];
double * tmp = new double[256];
```



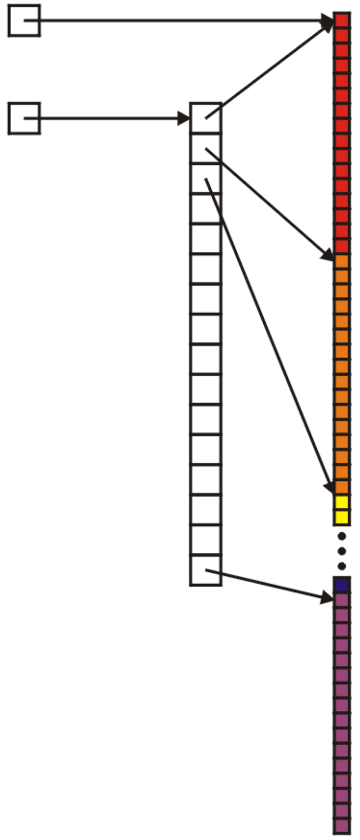
Adjacency Matrix Improvement

Next, we allocate the addresses:

```
matrix = new double * [16];
double * tmp = new double[256];

for ( int i = 0; i < 16; ++i ) {
   matrix[i] = &( tmp[16*i] );
}
```

This assigns:

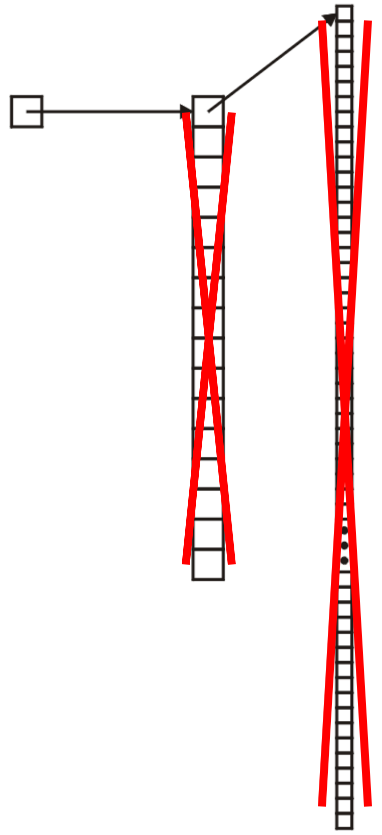
```
matrix[ 0] = &( tmp[  0] );
matrix[ 1] = &( tmp[ 16] );
matrix[ 2] = &( tmp[ 32] );
             .
             .
             .
matrix[15] = &( tmp[240] );
```



Adjacency Matrix Improvement

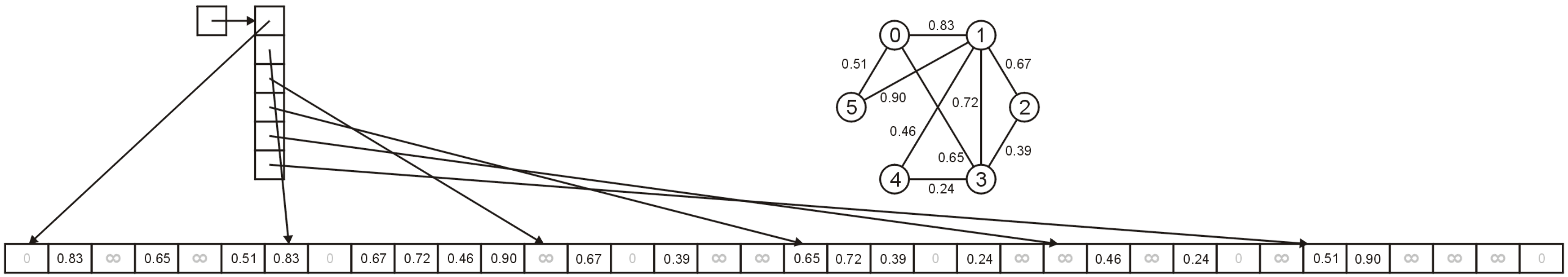
Deleting this array is easier:

```
delete [] matrix[0];
delete [] matrix;
```



Adjacency Matrix Improvement

Our sample graph would be represented as follows:



Lower-triangular adjacency matrix

Next we will look at another improvement which can be used for undirected graphs

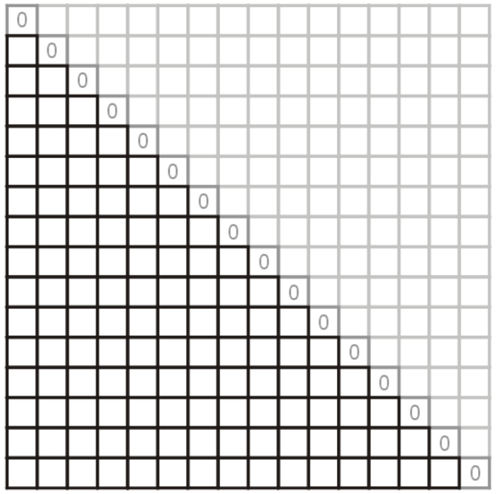
We will store only half of the entries

– To do this, we must also learn about pointer arithmetic

Lower-triangular adjacency matrix

Note also that we are not storing a directed graph: therefore, we really need only store half of the matrix

Thus, instead of 256 entries, we really only require 120 entries



Lower-triangular adjacency matrix

The memory allocation for this would be straight-forward, too:

```
matrix = new double * [16];
matrix[0] = 0;
matrix[1] = new double[120];

for( int i = 2; i < 16; ++i ) {
    matrix[i] = matrix[i – 1] + i - 1;
}
```

Lower-triangular adjacency matrix

- What we are using here is pointer arithmetic:
  - in C/C++, you can add values to a pointer
  - the question is, what does it mean to set:

```
ptr = ptr + 1;
```

  or

```
ptr = ptr + 2;
```

Lower-triangular adjacency matrix

- Suppose we have a pointer-to-a-double:

  `double * ptr = new double( 3.14 );`

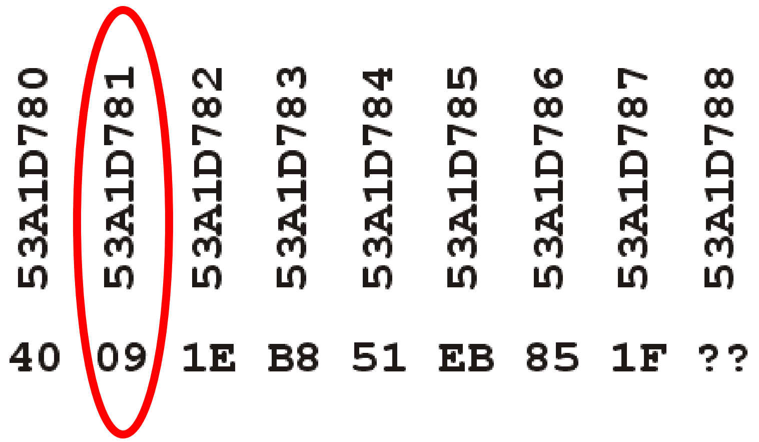
  where:

  - the pointer has a value of **`0x53A1D780`**, and
  - the representation of 3.14 is **`0x40091Eb851EB851F`**

53A1D780	53A1D781	53A1D782	53A1D783	53A1D784	53A1D785	53A1D786	53A1D787	53A1D788
40	09	1E	B8	51	EB	85	1F	??

Lower-triangular adjacency matrix

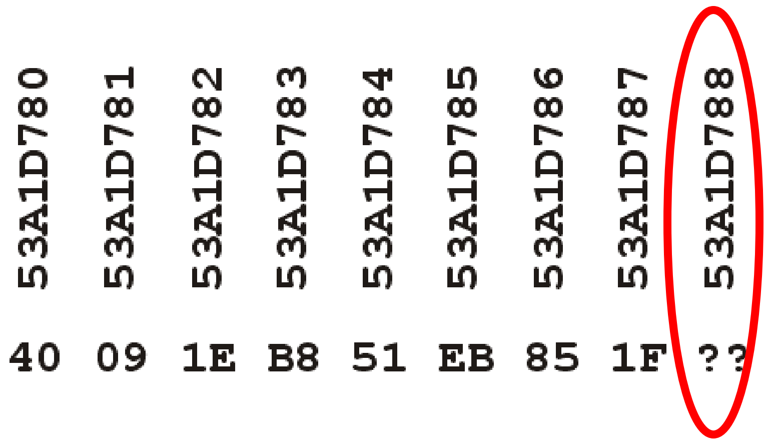
- If we just added one to the address, then this would give us the value **`0x53A1D781`**, but this contains no useful information...



53A1D780	53A1D781	53A1D782	53A1D783	53A1D784	53A1D785	53A1D786	53A1D787	53A1D788
40	09	1E	B8	51	EB	85	1F	??

Lower-triangular adjacency matrix

- The only logical interpretation of **`ptr + 1`** is to go to the *next* location a different double could exist, *i.e.*, **`0x53A1D788`**



Lower-triangular adjacency matrix

Therefore, if we define:

double * array = new double[4];

then the following are all equivalent:

```
array[0]                *array
array[1]                *(array + 1)
array[2]                *(array + 2)
array[3]                *(array + 3)
```

Lower-triangular adjacency matrix

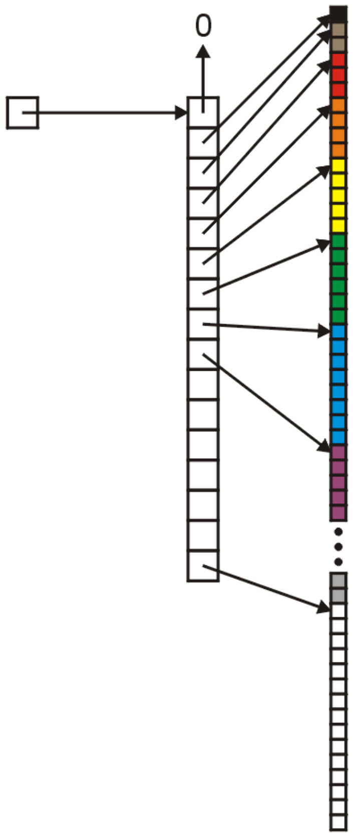
- Thus, the following code simply adds appropriate amounts to the pointer:

```
matrix = new double *[N];
matrix[0] = nullptr;
matrix[1] = new double[N*(N – 1)/2];

for( int i = 2; i < N; ++i ) {
    matrix[i] = matrix[i – 1] + i - 1;
}
```

Lower-triangular adjacency matrix

Visually, we have, for $N = 16$, the following:

```
matrix[0] = nullptr;
matrix[1] = &( tmp[0] );
matrix[2] = &( tmp[1] );
matrix[3] = &( tmp[3] );
matrix[4] = &( tmp[6] );
matrix[5] = &( tmp[10] );
matrix[6] = &( tmp[15] );
matrix[7] = &( tmp[21] );
matrix[7] = &( tmp[28] );
            .
            .
            .
matrix[15] = &( tmp[105] );
```



Lower-triangular adjacency matrix

The only thing that we would have to do is ensure that we always put the larger number first:

```
void insert( int i, int j, double w ) {
    if ( j < i ) {
        matrix[i][j] = w;
    } else {
        matrix[j][i] = w;
    }
}
```

Lower-triangular adjacency matrix

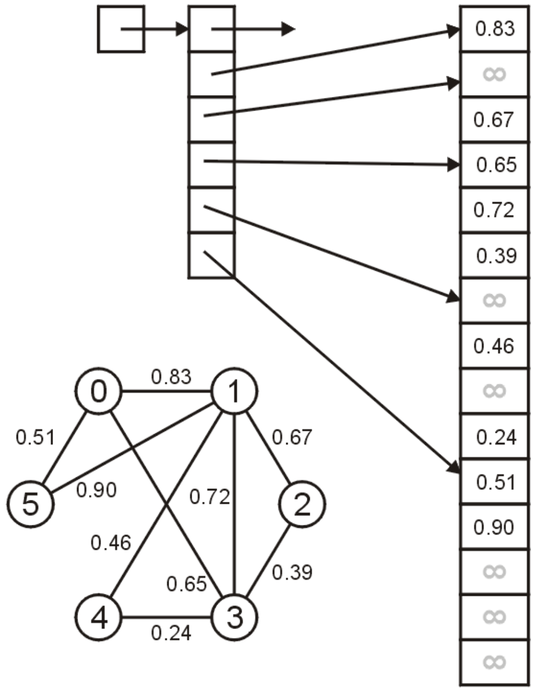
- A slightly less efficient way of writing this would be:

```
void insert( int i, int j, double w ) {
    matrix[max(i,j)][min(i,j)] = w;
}
```

- The benefits (from the point-of-view of clarity) are much more significant...

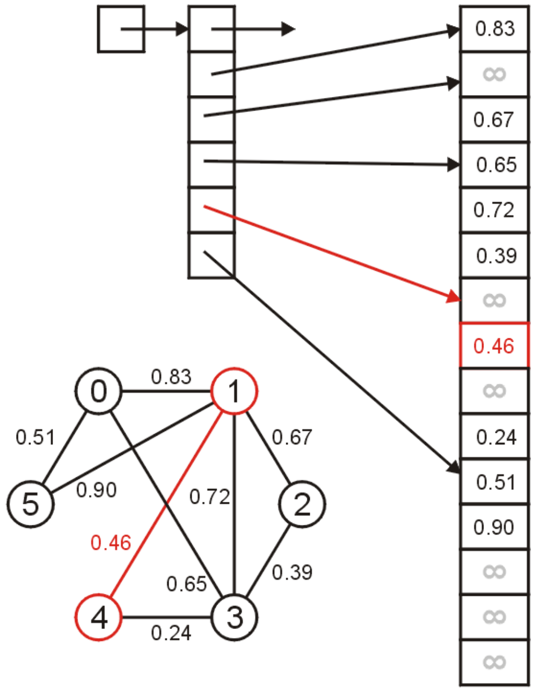
Lower-triangular adjacency matrix

- Our example graph is stored using this representation as shown here
- Notice that we do not store any 0's, nor do we store any duplicate entries
- The second array has only 15 entries, versus 36



Lower-triangular adjacency matrix

- To determine the weight of the edge connecting vertices 1 and 4, we must look up the entry
  `matrix[4][1]`



Sparse Matrices

- Finally we will consider the problem with sparse matrices and we will look at one implementation using linked lists

Sparse Matrices

- The memory required for creating an $n \times n$ matrix using an array-of-arrays is:

  $$4 \text{ bytes} + 4n \text{ bytes} + 8n^2 \text{ bytes} = \Theta(n^2) \text{ bytes}$$

Sparse Matrices

- The memory required for creating an $n \times n$ matrix using an array-of-arrays is:

  $$4 \text{ bytes} + 4n \text{ bytes} + 8n^2 \text{ bytes} = \Theta(n^2) \text{ bytes}$$

- This could potentially waste a significant amount of memory:
  - consider all intersections in US as vertices and streets as edges
  - how could we estimate the number of intersections in US?

Sparse Matrices

- Matrices where less than 5% of the entries are not the default value (either infinity or 0, or perhaps some other default value) are said to be *sparse*

Sparse Matrices

- Matrices where less than 5% of the entries are not the default value (either infinity or 0, or perhaps some other default value) are said to be *sparse*
- Matrices where most entries (25% or more) are not the default value are said to be *dense*

Sparse Matrices

- Matrices where less than 5% of the entries are not the default value (either infinity or 0, or perhaps some other default value) are said to be *sparse*
- Matrices where most entries (25% or more) are not the default value are said to be *dense*
- Clearly, these are not hard limits

Sparse Matrices

- Assume that each intersection connects, on average, four other intersections
- Therefore, less than 0.0005% of the entries of the matrix are used to store connections
  - the rest are storing the value *infinity*

Sparse Matrices

- We will look at a very efficient sparse-matrix implementation with the last topic
- Here, we will consider a simpler implementation:
  - use an array of linked lists to store edges
- Note, however, that each node in a linked list must store two items of information:
  - the connecting vertex and the weight

Sparse Matrices

- One possible solution:
  - modify the **`SingleNode`** data structure to store both an integer and a double:

```
class SingleNode {
    private:
        int adacent_vertex;
        double edge_weight;
        SingleNode * next_node;
    public:
        SingleNode( int, double SingleNode = 0 );
        double weight() const;
        int vertex() const;
        SingleNode * next() const;
};
```

  - exceptionally inefficient

Sparse Matrices

A better solution is to create a new class which stores a vertex-edge pair

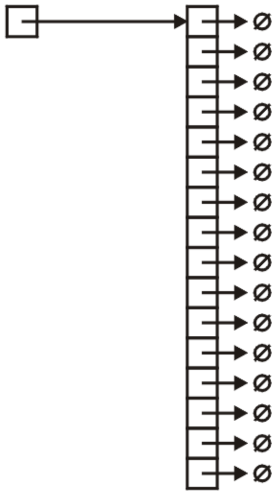
```
class Pair {
    private:
        double edge_weight;
        int adacent_vertex;
    public:
        Pair( int, double );
        double weight() const;
        int vertex() const;
};
```

Now create an array of linked-lists storing these pairs

Sparse Matrices

Thus, we define and create the array:

```
SingleList<Pair> * array;

array = new SingleList<Pair>[16];
```



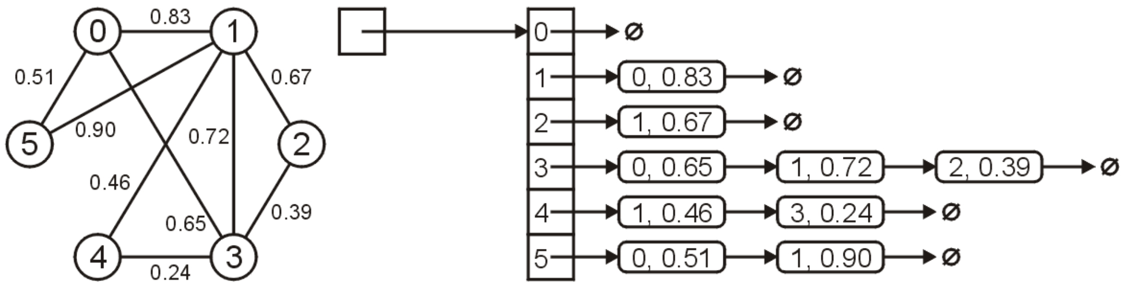
Sparse Matrices

As before, to reduce redundancy, we would only insert the entry into the entry corresponding with the larger vertex

```
void insert( int i, int j, double w ) {
    if ( i < j ) {
        array[j].push_front( Pair(i, w) );
    } else {
        array[i].push_front( Pair(j, w) );
    }
}
```

Sparse Matrices

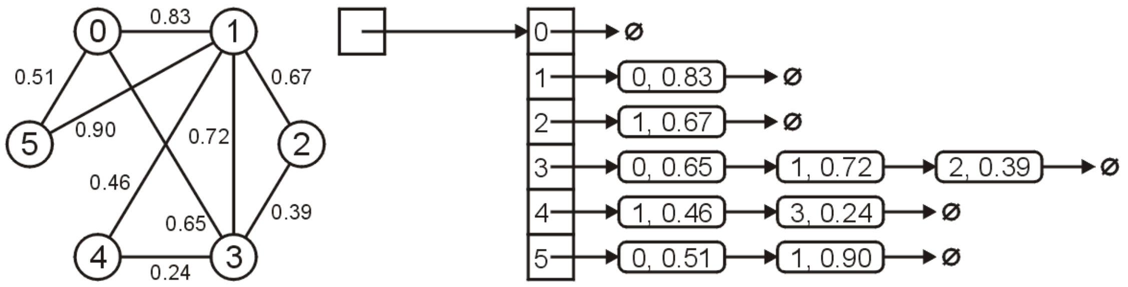
For example, the graph shown below would be stored as



Sparse Matrices

Later, we will see an even more efficient implementation

– The old an new Yale sparse matrix formats



Summary

- In this lecture, we have looked at a number of graph representations
- C++ lacks a *matrix* data structure
  - must use array of arrays
- The possible factors affecting your choice of data structure are:
  - weighted or unweighted graphs
  - directed or undirected graphs
  - dense or sparse graphs