Elementary Data Structures: Part 2: Strings, 2D Arrays, Graphs

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Strings

- What are strings, in general (independent of C)?
- Why do we care about strings?

Strings

- What are strings, in general (independent of C)?
  - Data structures that store text.
- Why do we care about strings?
  - Indispensable for text processing.
  - Ubiquitous in programming.
- Strings can be implemented in various ways.

Strings

- What are strings, in general (independent of C)?
  - Data structures that store text.
- Why do we care about strings?
  - Indispensable for text processing.
  - Ubiquitous in programming.
- Strings can be implemented in various ways.
- For the purposes of the textbook and this course, we will use a specific definition:
- A **string** is an array of characters, that contains the NULL character (ASCII code 0) at the end.
  - The NULL character can ONLY appear at the end.

Limitations of Definition

- Our definition of strings is limited.
- It only supports characters represented in ASCII.
  - Multilingual character sets are not supported.
- Strings are arrays, meaning that their maximum size has to be fixed when they are created.
- However, our definition is sufficient for the purposes of this course.
  - The basic algorithms remain the same if we extend the definition to support larger alphabets.

Strings and Arrays

- Strings are arrays. However, logically, we treat strings as different data structures.
- Why are strings different than arrays?

Strings and Arrays

- Strings are arrays. However, logically, we treat strings as different data structures.
- Why are strings different than arrays?
  – The length of an array is defined as the length that we specify when we create the array.
  – The length of a string is defined to be the position of the first occurrence of the NULL character.
- Obviously, if a string is an array, the MAXIMUM size of the string must still be declared at creation time.
- However, when we talk about the "length" of the string, we only care about the position of the first occurrence of the NULL character.

Some Strings in C

```
char * s1 = "Monday";
char * s2 = malloc(1000 * sizeof(char));
strcpy(s2, "hello");
```

```
s1[0] == ???      s2[5] = ???      s2[6] = ???
```

- What is the length of s1?
- What is the length of s2?

Some Strings in C

```
char * s1 = "Monday";
char * s2 = malloc(1000 * sizeof(char));
strcpy(s2, "hello");
```

```
s1[0] == 'M'      s2[5] = 'o'      s2[6] = '\0' = 0.
```

- What is the length of s1? 6
- What is the length of s2? 5
- The **length** of a string is the number of characters, up to and **<u>not including</u>** the first occurrence of the NULL character.

strlen: Counting String Length

Function strlen takes a string as an argument, and returns the length of the string.

How do we implement strlen?

```
int strlen(char * s)
```

strlen: Counting String Length

```
int strlen(char * s)
{
   int counter = 0;
   while (s[counter] != 0)
   {
      counter++;
   }

   return counter;
}
```

What is the time complexity?

strlen: Counting String Length

```
int strlen(char * s)
{
   int counter = 0;
   while (s[counter] != 0)
   {
      counter++;
   }

   return counter;
}
```

What is the time complexity? O(N), where N is the length of the string.

strcpy: Making a String Copy

- Function strcpy takes two arguments:
  - a string called "target" and a string called "source".
- The function copies the contents of source onto target.
  - The previous contents of target are overwritten.
- It is assumed that target has enough memory allocated, no error checking is done.

How do we implement strcpy?

```
void strcpy(char * target, char * source)
```

strcpy: Making a String Copy

```
void strcpy(char * target, char * source)
{
   int counter = 0;
   while (source[counter] != 0)
   {
      target[counter] = source[counter];
      counter++;
   }
}
```

What is the time complexity?

strcpy: Making a String Copy

```
void strcpy(char * target, char * source)
{
   int counter = 0;
   while (source[counter] != 0)
   {
      target[counter] = source[counter];
      counter++;
   }
}
```

What is the time complexity? O(N), where N is the length of the string.

copy_string: Alternative for strcpy

- Function string_copy takes as argument a string called "source".
- The function creates and returns a copy of source.
  - Memory is allocated as needed.
  - Somewhat safer than strcpy, as here we do not need to worry if we have enough memory for the result.

```
char * copy_string(char * source)
```

copy_string: Alternative for strcpy

```
char * copy_string(char * source)
{
   int length = strlen(source);
   char * result = malloc(length+1);
   strcpy1(result, source);

   return result;
}
```

strcmp: Comparing Two Strings

- Function strcmp takes two arguments: s1 and s2.
- The function returns:
  - 0 if the contents are equal, letter by letter.
    - NOT case-insensitive, case matters.
  - -1 if s1 is smaller than s2 at the first position where they differ.
  - 1 if s1 is larger than s2 at at the first position where they differ.

How do we implement strcmp?

```
int strcmp(char * s1, char * s2)
```

strcmp: Comparing Two Strings

```
int strcmp(char * s1, char * s2)
{
   int i = 0;
   while ((s1[i] != 0) && (s2[i] != 0))
   {
      if (s1[i] != s2[i])  return s1[i] - s2[i];
      i++;
   }
   return s1[i] - s2[i];
}
```

What is the time complexity?

strcmp: Comparing Two Strings

```
int strcmp(char * s1, char * s2)
{
   int i = 0;
   while ((s1[i] != 0) && (s2[i] != 0))
   {
      if (s1[i] != s2[i])  return s1[i] - s2[i];
      i++;
   }
   return s1[i] - s2[i];
}
```

What is the time complexity? O(N), where N is the length of the **shortest** among the two strings.

String Equality

- People may mean several different things when they talk about two strings being "equal".
- The convention that we follow in this course is that two strings are equal if their contents are equal.
  - The two strings must have the **same length**.
  - The two strings must have the same letters (i.e., **same ASCII codes**) at all positions up to the end (the first occurrence of the NULL character).
- Equivalent definition: two strings s1 and s2 are equal if and only if strcmp(s1, s2) returns 0.
- This convention is **different** than:
  - **pointer equality**: checking if the two strings point to the same location in memory.
  - **case-insensitive equality**, where lower-case letters and upper-case letters are considered to be equal.

strncmp: Fixed-Length Comparisons

- Function strncmp takes three arguments: s1, s2, N
- The function returns:
  - 0 if the first **N letters** are equal, letter by letter.
    - Or if both strings are equal and their length is shorter than N.
  - -1 if s1 is smaller than s2 at the first position where they differ.
  - 1 if s1 is larger than s2 at at the first position where they differ.

How do we implement strncmp?

```
int strncmp(char * s1, char * s2, int N)
```

strncmp: Fixed-Length Comparisons

```
int strncmp(char * s1, char * s2, int N)
{
   int i;
   for (i = 0; i < N; i++)
   {
      if ((s1[i]==0) || (s2[i]==0) || (s1[i]!=s2[i]))
         return s1[i] - s2[i];
   }
   return 0;
}
```

What is the time complexity?

strncmp: Fixed-Length Comparisons

```
int strncmp(char * s1, char * s2, int N)
{
   int i;
   for (i = 0; i < N; i++)
   {
      if ((s1[i]==0) || (s2[i]==0) || (s1[i]!=s2[i]))
         return s1[i] - s2[i];
   }
   return 0;
}
```

What is the time complexity? O(N).

strcat: String Concatenation

- Function strcat takes three arguments: a, b.
- The function writes the contents of string b at the end of string a.
- The **new** contents of string a are the concatenation of the **old** contents of string a and the contents of string b.
- It is assumed that a has enough free memory to receive the new contents, no error checking is done.

How do we implement strcat?

```
char * strcat(char * a, char * b)
```

strcat: String Concatenation

```
char * strcat(char * a, char * b)
{
   int a_index = strlen(a);
   int b_index = 0;
   for (b_index = 0; b[b_index] != 0; b_index++)
      a[a_index+b_index] = b[b_index];

   a[a_index+b_index] = 0;
   return a;
}
```

What is the time complexity?

strcat: String Concatenation

```
char * strcat(char * a, char * b)
{
   int a_index = strlen(a);
   int b_index = 0;
   for (b_index = 0; b[b_index] != 0; b_index++)
      a[a_index+b_index] = b[b_index];

   a[a_index+b_index] = 0;
   return a;
}
```

What is the time complexity? O(N), where N is the **sum of the lengths** of the two strings.

Implementations

- The implementations of these functions are posted on the course website, as files:
  - basic_strings.h
  - basic_strings.c
- No error checking is done, the goal has been to keep the implementations simple.
- The function names have been changed to strlen1, strcpy1, and so on, because functions strlen, strcpy and so on are already defined in C.
  - Only **copy_string** is not already defined in C.

Example Function: String Search

```
void string_search(char * P, char * A)
```

- Input: two strings, P and A.
- Output: prints out the starting positions of all occurrences of P in A.
- Examples:
  - string_search("e", "Wednesday") prints: 1 4.
  - string_search("ti", "initiation") prints: 3 6.

Example Function: String Search

```
void string_search(char * P, char * A)
{
   int p_length = strlen1(P);
   int i;
   for (i = 0; A[i] != 0; i++)
      if (strncmp1(P, &(A[i]), p_length) == 0)
         printf("position %d\n", i);
}
```

- What is the time complexity of this function?

Example Function: String Search

```
void string_search(char * P, char * A)
{
   int p_length = strlen1(P);
   int i;
   for (i = 0; A[i] != 0; i++)
      if (strncmp1(P, &(A[i]), p_length) == 0)
         printf("position %d\n", i);
}
```

- What is the time complexity of this function? O(length(P) * length(A)).

Example of Uncessesarily Bad Performance

```
void string_search(char * P, char * A)
{   int p_length = strlen1(P);
    int i;
    for (i = 0; A[i] != 0; i++)
        if (strncmp1(P, &(A[i]), p_length) == 0)
            printf("position %d\n", i);
}
```

previous version

```
void string_search_slow(char * P, char * A)
{   int i;
    for (i = 0; i < strlen(A); i++)
        if (strncmp1(P, &(A[i]), strlen(P)) == 0)
            printf("position %d\n", i);
}
```

new version: what is wrong with it?

Example of Uncessesarily Bad Performance

- Let M be the length of string A, and N be the length of string B.
- The first version of string search has running time $O(MN)$.
- The second version of string search has running time $O(M^2N)$.
  - That is a huge difference.
- If M = 1 million (size of a book), N = 10 (size of a word):
  - The second version is 1 million times slower.
  - If the first version takes 0.1 seconds to run, the second version takes 100,000 seconds, which is about 28 hours.

The Need for 2D Arrays

- Arrays, lists, and strings are data types appropriate for storing *sequences* of values.
- Some times, the data is more naturally organized in two dimensions, and want to access each value by specifying the row and column.
- For example:
  - Mathematical matrices of M rows and N columns..
  - A course gradebook may have one column per assignment and one row per student.
  - A black-and-white (also called grayscale) image is specified as a 2D array of numbers between 0 and 255. Each number specifies the brightness at a specific image location (pixel).

Allocating Memory for a 2D Array in C

- We want to write a function **malloc2d** that is the equivalent of **malloc** for 2D arrays.

- What should the function take as input, what should it return as result?

Allocating Memory for a 2D Array in C

- We want to write a function **malloc2d** that is the equivalent of **malloc** for 2D arrays.

- What should the function take as input, what should it return as result?

```
int ** malloc2d(int rows, int columns)
```

Allocating Memory for a 2D Array in C

```
int ** malloc2d(int rows, int columns)
{
   int row;
   int ** result = malloc(rows * sizeof(int *));
   for (row = 0; row < rows; row++)
      result[row] = malloc(columns * sizeof(int));

   return result;
}
```

- What is the time complexity of this?

Allocating Memory for a 2D Array in C

```
int ** malloc2d(int rows, int columns)
{
   int row;
   int ** result = malloc(rows * sizeof(int *));
   for (row = 0; row < rows; row++)
      result[row] = malloc(columns * sizeof(int));

   return result;
}
```

- What is the time complexity of this?
  - Linear to the number of rows. In other word, O(rows).

Deallocating Memory for a 2D Array

- We want to write a function **free2d** that is the equivalent of **free** for 2D arrays.

- What should the function take as input, what should it return as result?

Deallocating Memory for a 2D Array

- We want to write a function **free2d** that is the equivalent of **free** for 2D arrays.

- What should the function take as input, what should it return as result?

```
void free2d(int ** array, int rows, int columns)
```

Deallocating Memory for a 2D Array

```
void free2d(int ** array, int rows, int columns)
{
   int row;
   for (row = 0; row < rows; row++)
      free(array[row]);

   free(array);
}
```

- Note: the **columns** argument is not used. Why pass it as an argument then?

- What is the time complexity of this?

Deallocating Memory for a 2D Array

```
void free2d(int ** array, int rows, int columns)
{
    int row;
    for (row = 0; row < rows; row++)
        free(array[row]);

    free(array);
}
```

- Note: the **columns** argument is not used. However, by passing it as an argument we allow different implementations later (e.g., indexing first by column and second by row).
- What is the time complexity of this? O(rows) again.

Using 2D Arrays: Print

```
void printMatrix(int ** array, int rows, int cols)
{
   int row, col;
   for (row = 0; row < rows; row++)
   {
      for (col = 0; col < cols; col++)
      {
         printf("%5d", array[row][col]);
      }
      printf("\n");
   }
   printf("\n");
}
```

Using 2D Arrays: Adding Matrices

```
int ** addMatrices(int ** A, int ** B, int rows, int cols)
{
   int ** result = malloc2d(rows, cols);
   int row, col;
   for (row = 0; row < rows; row++)
   {
      for (col = 0; col < columns; col++)
      {
         result[row][col] = A[row][col] + B[row][col];
      }
   }

   return result;
}
```

More Complicated Data Structures

- Usign arrays, lists and strings, we can build an infinite variety of more complicated data structures.
- Examples:
  - N-dimensional arrays (for any integer N > 1).
  - arrays of strings.
  - arrays of lists.
  - lists of lists of lists of lists of strings.
  - lists of arrays.
  - …

Graphs

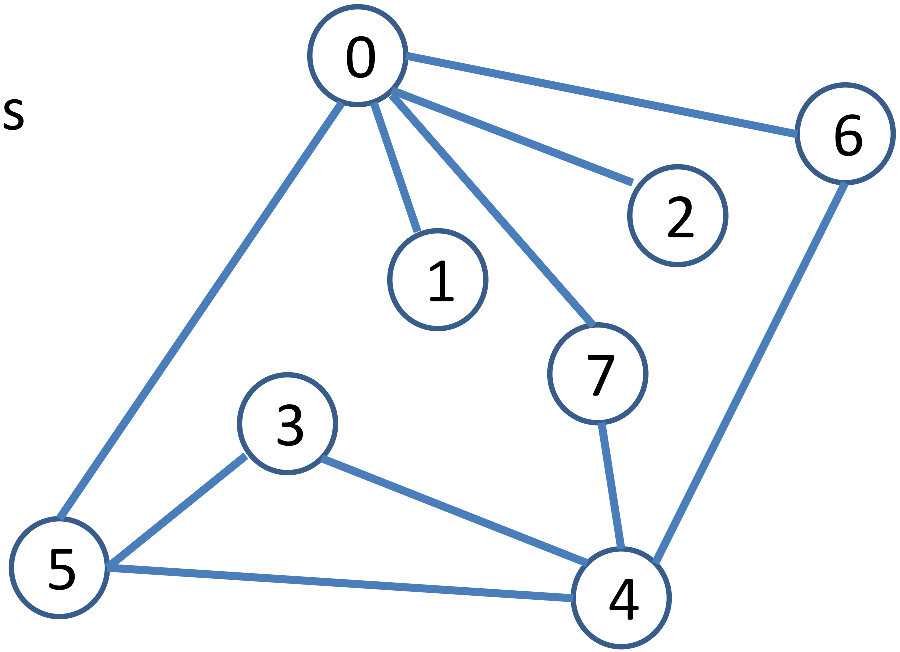
- A graph is a fundamental data type.
- Graphs are at the core of many algorithms we will cover in this course.
- We already saw an example with the Union-Find program.
- Other examples:
  - road networks
  - computer networks
  - social networks
  - game-playing algorithms (e.g., for chess).
  - problem-solving algorithms (e.g., for automated proofs).

Graphs

- A graph is formally defined as:
  - A set V of vertices.
  - A set E of edges. Each edge is a pair of two vertices in V.
- Graphs can be directed or undirected.
- In a directed graph, edge (A, B) means that we can go (using that edge) from A to B, but **not** from B to A.
  - We can have both edge (A, B) and edge (B, A) if we want to show that A and B are linked in both directions.
- In an undirected graph, edge (A, B) means that we can go (using that edge) from both A to B and B to A.

Example: of an Undirected Graph

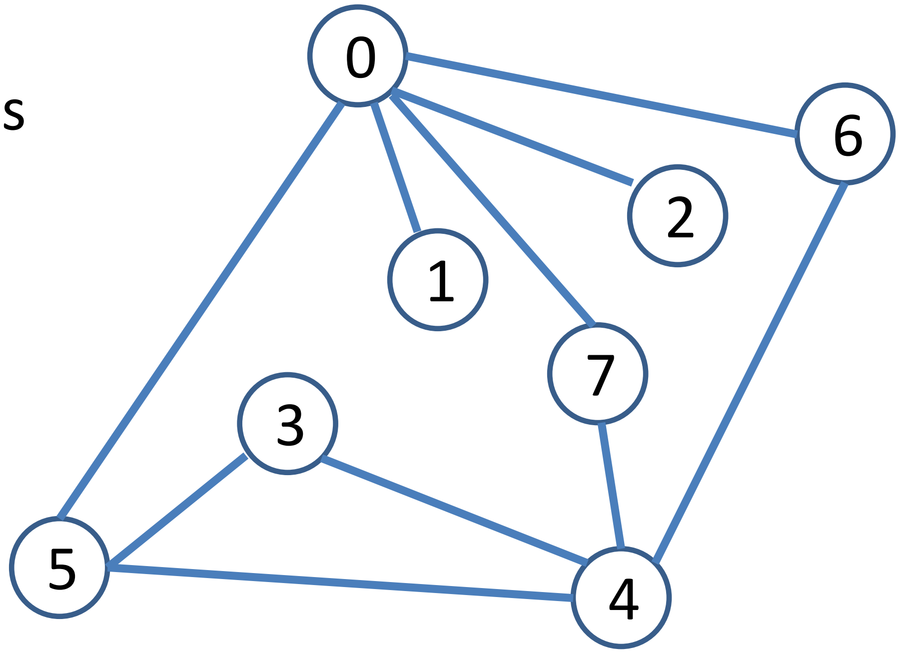
- A graph is formally defined as:
  - A set V of vertices.
  - A set E of edges. Each edge is a pair of two vertices in V.
- What is the set of vertices on the graph shown here?



- What is the set of edges?

Example: of an Undirected Graph

- A graph is formally defined as:
  - A set V of vertices.
  - A set E of edges. Each edge is a pair of two vertices in V.
- What is the set of vertices on the graph shown here?
  - {0, 1, 2, 3, 4, 5, 6, 7}
- What is the set of edges?
  - {(0,1), (0,2), (0,5), (0,6), (0, 7), (3, 4), (3, 5), (4, 5), (4, 6), (4,7)}.



Designing a Data Type for Graphs

- If we want to design a data type for graphs, the key questions are:
  - How do we represent vertices?
  - How do we represent edges?
- There are multiple ways to answer these questions.
- Can you think of some ways to represent vertices and edges?

Representing Vertices

- In the most general solution, we could make a new data type for vertices.
- Each vertex would be a struct (object), containing fields such as:
  - ID (a description of the vertex that can be an int, string, etc.).
  - A list of neighboring vertices.
- Then, each vertex would be represented as an object of that type.
- The graph would need store the list of vertices that it contains.

Representing Vertices as Integers

- We can also use a much more simple approach, that is sufficient in many cases:
- Vertices are integers from 0 to V - 1 (where V is the number of vertices in the graph).
  - More complicated approaches have their own advantages and disadvantages.
- This way, the graph object just needs to know how many vertices it contains.
  - If graph G has 10 vertices, we know that those vertices are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Representing Edges

- Two vertices at opposite ends of an edge are called **neighbors**.
- Knowing the edges of a graph is the same thing as knowing, for each vertex V of the graph, who the neighbors of V are.
- The list of neighbors of vertex V is called the **adjacency list** of V.
- How can we represent adjacency lists?
  - Assume that we represent vertices as integers from 1 to V-1.

Adjacency Matrix

- Suppose we have V vertices, represented as integers from 0 to V-1.
- We can represent adjacencies using a 2D binary matrix A, of size V*V.
- $A[V_1][V_2] = 1$ if and only if there is an edge connecting vertices V_1 and V_2.
- $A[V_1][V_2] = 0$ otherwise (if V_1 and V_2 are not connected by an edge).
- How much memory does that take?
- How much time does it take to add, remove, or check the status of an edge?

Adjacency Matrix

- Suppose we have V vertices, represented as integers from 0 to V-1.
- We can represent adjacencies using a 2D binary matrix A, of size V*V.
- $A[V_1][V_2] = 1$ if and only if there is an edge connecting vertices V_1 and V_2.
- $A[V_1][V_2] = 0$ otherwise (if V_1 and V_2 are not connected by an edge).
- How much memory does that take? $O(V^2)$.
- How much time does it take to add, remove, or check the status of an edge? O(1).

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency matrix representation?

```
typedef struct struct_graph * graph;
struct struct_graph
{
...
};

int edgeExists(graph g, int v1, int v2) ...

void addEdge(graph g, int v1, int v2) ...

void removeEdge(graph g, int v1, int v2) ...
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency matrix representation?

```
typedef struct struct_graph * graph;
struct struct_graph
{
   int number_of_vertices;
   int ** adjacencies;
};

int edgeExists(graph g, int v1, int v2)
{
   return g->adjacencies[v1][v2];
}
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency matrix representation?

```
void addEdge(graph g, int v1, int v2)
{
   g->adjacencies[v1][v2] = 1;
   g->adjacencies[v2][v1] = 1;
}

void removeEdge(graph g, int v1, int v2)
{
   g->adjacencies[v1][v2] = 0;
   g->adjacencies[v2][v1] = 0;
}
```

Adjacency Lists

- An alternative to representing adjacencies using a 2D array is to save adjacencies as an array A of lists.
- $A[V_1]$ is a list containing the neighbors of vertex V_1.
- How much space does this take?

Adjacency Lists

- An alternative to representing adjacencies using a 2D array is to save adjacencies as an array A of lists.
- $A[V_1]$ is a list containing the neighbors of vertex V_1.
- How much space does this take?
  - O(E), where E is the number of edges.
- If the graph is relatively sparse, and $E << V^2$, this can be a significant advantage.

Adjacency Lists

- An alternative to representing adjacencies using a 2D array is to save adjacencies as an array A of lists.
- $A[V_1]$ is a list containing the neighbors of vertex V_1.
- How much time does it take to check if an edge exists or not?
  - Worst case: O(V). Each vertex can have up to V-1 neighbors, and we may need to go through all of them to see if an edge exists.
  - For sparse graphs, the behavior can be much better. If let's say each vertex has at most 10 neighbors, then we can check if an edge exists much faster.
  - Either way, slower than using adjacency matrices.

Adjacency Lists

- An alternative to representing adjacencies using a 2D array is to save adjacencies as an array A of lists.
- A[V_1] is a list containing the neighbors of vertex V_1.
- How much time does it take to remove an edge?

- How much time does it take to add an edge?

Adjacency Lists

- An alternative to representing adjacencies using a 2D array is to save adjacencies as an array A of lists.
- A[V_1] is a list containing the neighbors of vertex V_1.
- How much time does it take to remove an edge?
  - Same as for checking if an edge exists.
- How much time does it take to add an edge?
  - Same as for checking if an edge exists.
  - Why? Because if the edge already exists, we should not duplicate it.

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency list representation?

```
typedef struct struct_graph * graph;
struct struct_graph
{
...
};

int edgeExists(graph g, int v1, int v2) ...

void addEdge(graph g, int v1, int v2) ...

void removeEdge(graph g, int v1, int v2) ...
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency list representation?
- Defining the object type itself:

```
typedef struct struct_graph * graph;

struct struct_graph
{
   int number_of_vertices;
   list * adjacencies;
};
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency list representation?
- Checking if an edge exists:

```
int edgeExists(graph g, int v1, int v2)
{
   link n;
   for (n = g->adjacencies[v1]->first);
        n != NULL; n = linkNext(n))
   {
      if (linkItem(n) == v2) return 1;
   }
   return 0;
}
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency list representation?
- Adding a new edge:

```
void addEdge(graph g, int v1, int v2)
{
   if !(edgeExists(g, v1, v2))
   {
      insertAtBeginning(g->adjacencies[v1], newLink(v2));
      insertAtBeginning(g->adjacencies[v2], newLink(v1));
   }
}
```

Defining a Graph

- How do we define in C a data type for a graph, using the adjacency list representation?
- Removing an edge: see posted file graph_lists.c
- Pseudocode: removeEdge(V1, V2)
  - Go through adjacency list of V1, remove link corresponding to V2
  - Go through adjacency list of V2, remove link corresponding to V1.

Adjacency Matrices vs. Adjacency Lists

- Suppose we have a graph with:
  - 10 million vertices.
  - Each vertex has at most 20 neighbors.
- Which of the two graph representations would you choose?

Adjacency Matrices vs. Adjacency Lists

- Suppose we have a graph with:
  - 10 million vertices.
  - Each vertex has at most 20 neighbors.
- Adjacency matrices: we need at least 100 trillion bits of memory, so at least 12.5TB of memory.
- Adjacency lists: in total, they would store at most 200 million items. With 8 bytes per item (as an example), this takes 1.6 Gigabytes.

Check Out Posted Code

- **graphs.h**: defines an abstract interface for basic graph functions.
- **graphs_matrix.c**: implements the abstract interface of graphs.h, using an adjacency matrix.
- **graphs_list.c**: also implements the abstract interface of graphs.h, using adjacency lists.
- **graphs_main:** a test program, that can be compiled with **either** graphs_matrix.c or graphs_list.c.