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

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

Modified by Alexandra Stefan
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.

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


• What is the length of s1?
• What is the length of s2?
Some Strings in C

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

```

- 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 **not including** the first occurrence of the NULL character.
Some String Functions

• C functions on strings:
  – strlen
  – strcpy
  – strcat
  – strcmp
  – strncmp

• Our implementation of these functions:
  – strlen1
  – strcpy1
  – strcat1
  – strcmp1
  – strncmp1

• Search a string (a word) in a longer string (a text).
strlen1: Counting String Length

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

How do we implement strlen?

int strlen1(char * s)
strlen1: Counting String Length

```c
int strlen1(char * s)
{
    int counter = 0;
    while (s[counter] != 0)
    {
        counter++;
    }

    return counter;
}
```

What is the time complexity?
strlen1: Counting String Length

```c
int strlen1(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.
strncpy1: Making a String Copy

• Function strncpy1 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?

```c
void strncpy1(char * target, char * source)
```
strncpy1: Making a String Copy

void strncpy1(char * target, char * source)
{
    int counter = 0;
    while (source[counter] != 0)
    {
        //no error check for target[]. How can you check?
        target[counter] = source[counter];
        counter++;
    }
}

What is the time complexity?
strcpy1: Making a String Copy

```c
void strcpy1(char * target, char * source)
{
    int counter = 0;
    while (source[counter] != 0)
    {
        //no error check for target[]. How can you check? No way!
        target[counter] = source[counter];
        counter++;
    }
}
```

What is the time complexity? \(O(N)\), where \(N\) is the length of the string `source`. 
copy_string1: Alternative for strcpy

• Function copy_string 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_string1(char * source)
copy_string1: Alternative for strcpy

char * copy_string1(char * source)
{
    int length = strlen(source);
    char * result = malloc(length+1);  //note length+1
    strcpy(result, source);

    return result;
}
strcmp1: Comparing Two Strings

- Function strcmp takes two arguments: s1 and s2.
- The function returns:
  - 0 if the contents are equal, letter by letter.
    - case-sensitive, case matters.
  - A negative integer (not necessarily -1) if s1 is smaller than s2 at the first position where they differ.
  - A positive integer (not necessarily 1) if s1 is larger than s2 at the first position where they differ.
  - Easy to remember trick: think that it returns s1-s2.

How do we implement strcmp?

```c
int strcmp1(char * s1, char * s2)
```
int strcmp1(char * s1, char * s2)
{
    int i = 0;
    while ((s1[i] != 0) && (s2[i] != 0))
    {
        if (s1[i] != s2[i]) break;
        i++;
    }
    return s1[i] - s2[i]; // note the exact return value
}

What is the time complexity?
strcmp1: Comparing Two Strings

```c
int strcmp1(char * s1, char * s2)
{
    int i = 0;
    while ((s1[i] != 0) && (s2[i] != 0))
    {
        if (s1[i] != s2[i]) break;
        i++;
    }
    return s1[i] - s2[i]; // note the exact return value
}
```

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 $s_1$ and $s_2$ are equal if and only if $\text{strcmp1}(s_1, s_2)$ 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.
strncmp1: Fixed-Length Comparisons

• Function strncmp1 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 the first position where they differ.

How do we implement strncmp?

int strncmp1(char * s1, char * s2, int N)
int strncmpl1(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?
strncmp1: Fixed-Length Comparisons

```c
int strncmp1(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).
strcat1: String Concatenation

- Function strcat1 takes two 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?

```c
char * strcat1(char * a, char * b)
```
strcat1: String Concatenation

char * strcat1(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; // mark end of string
    return a;
}

What is the time complexity?
strcat: String Concatenation

char * strcat1(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; // mark end of string
    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_string1 is not already defined in C.
void string_search(char * word, char * text)

• Input: two strings, \texttt{word} and \texttt{text}.
• Output: prints out the starting positions of all occurrences of \texttt{word} in \texttt{text}.
• Examples:
  – \texttt{string\_search("e", "Wednesday")} prints: 1 4.
  – \texttt{string\_search("ti", "initiation")} prints: 3 6.
Example Function: String Search

```c
void string_search(char * word, char * text)
{
    int word_len = strlen1(word);
    int i;
    for (i = 0; text[i] != 0; i++)
        if (strncmpl(word, &(text[i]), word_len) == 0)
            printf("position %d\n", i);
}
```

• What is the time complexity of this function?
Example Function: String Search

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

• What is the time complexity of this function? O( length(word) * length(text) ).
Example of Unnecessarily Bad Performance

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

void string_search_slow(char * word, char * text)
{
    int i;
    for (i = 0; i < strlen(text); i++)
        if (strncmp1(word, &(text[i]), strlen1(word)) == 0)
            printf("position %d\n", i);
}
```
Example of Unnecessarily Bad Performance

Let \( M \) be the length of string text, and \( N \) be the length of string word.

- First version of string search: \( \Theta(MN) \).
- Second version of string search: \( \Theta(M^2(M+N)) \).
  - Assuming \( M > N \), this is \( \Theta(M^2) \).
  - That is a huge difference over \( \Theta(MN) \), when \( M \gg N \).

- If \( M = 1 \) million (size of a book), \( N = 10 \) (size of a word):
  - If the first version takes 0.1 seconds to run, the second version takes 100,000 seconds, which is about 28 hours.
  - The second version is 1 million times slower.
2D Arrays
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).
Check Out Posted Code

• The code discussed in the following slides is provided on the web. See files:
  – twoD_arrays.h – abstract interface for basic functions for 2D arrays
  – twoD_arrays.c – implementation of the functions in twoD_arrays.h
  – matrices.c - client program. See compilation instructions as comments in the file.

• Both the slides and the code (including comments in there) give useful information in understanding this topic.
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?

```c
int ** malloc2d(int rows, int columns)
```
Allocating Memory for a 2D Array in C

```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?
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?

```c
void free2d(int ** array, int rows, int columns)
```
Deallocating Memory for a 2D Array

```c
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

```c
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? $O(\text{rows})$ again.
Using 2D Arrays: Print

```c
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

```c
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++)
        {
        }
    }
    return result;
}
```
More Complicated Data Structures

• Using 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
Graphs

• A graph is a fundamental data type.

• Graphs are at the core of many algorithms we will cover in this course.

• We have seen 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 $V$.

• How many graphs do we have here?

• Vertices?
  $V = $

• Edges?
  $E = $
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 \( V \).

How many graphs do we have here? One.

Vertices?
\[ V = \{0, 1, 2, 3, 4, 5, 6, 7\} \]

Edges?
\[ E = \{ (0, 2), (0, 6), (0, 7), (3, 4), (3, 5), (4, 5), (6, 7) \}. \]
Paths, Connected Components

• Are 2 and 7 connected?
  Yes: via the paths 2-0-7 or 2-0-6-7.

• Are 1 and 3 connected?
  No.

• Connected components?
  3: \{0,2,6,7\}, \{1\}, \{3,4,5\}.

• Degree of a vertex: number of edges incident to the vertex:
  Examples: degree(0) = 3, degree(1) = 0, degree(5) = 2.
Directed vs Undirected

- 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 will 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.
  - We have already seen an example.
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 as Integers

• Vertices are integers from 0 to V - 1 (where V is the number of vertices in the graph)
  – Simple and common approach.
  – More complicated approaches have their own advantages and disadvantages.
  – This way, the graph object just needs to know how many vertices it contains.
  – Example: If graph G has 10 vertices, we know that those vertices are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Graph representation:

Adjacency Matrix
Adjacency Matrix

Assume V vertices: 0, 1, \ldots, V-1.

Represent adjacencies using a 2D binary matrix $A$, of size $V \times V$.

\[ A[V_1][V_2] = 1 \text{ if and only if there is an edge from } V_1 \text{ to } V_2. \]
\[ A[V_1][V_2] = 0 \text{ otherwise (there is no edge from } V_1 \text{ to } V_2). \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: the adjacency is symmetric of non-directed graphs is symmetric.
Defining a Graph

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

```c
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?

```c
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?

```c
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;
}
```
New graph representation:

Adjacency Lists
Adjacency Lists

- Represent the edges of the graph by an array of lists.
  - Let’s name that array A
  - $A[V_1]$ is a list containing the neighbors of vertex $V_1$. 

```plaintext
0: 0
1: 0
2: 5 → 4
3: 5 → 3 → 7 → 6
4: 0 → 3 → 4
5: 0 → 4
6: 0
7: 0 → 4
```
Adjacency Lists

- **Space**

- **Time** to check if an edge exists or not
  - Worst case:

- **Time** to remove an edge?

- **Time** to add an edge?
Adjacency Lists

**Space**
- \( O(E) \), where \( E \) is the number of edges.
- If the graph is relatively sparse, and \( E << V^2 \), this can be a significant advantage.

**Time 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.

**Time to remove an edge?**
- Same as for checking if an edge exists.

**Time 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?

• Note: the same interface as for adjacency matrix.

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

• Defining the object type itself:

```c
typedef struct struct_graph * graph;

struct struct_graph
{
    int number_of_vertices;
    list * adjacencies;
};
```
Defining a Graph

• **Check if an edge exists:**

```c
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: add an edge

• **Add a new edge:**

```c
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: remove edge

• Removing an edge: see posted file graph_list.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.

• We’ll see next how to compute/verify such answers.
Steps for Solving This Problem

• 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.

• Find ‘keywords’, understand numbers:
  – 10 million vertices. => 10 * 10^6
  – Trillion = 10^{12}
  – 1 TB (terra bytes) = 10^{12} bytes
  – 1GB = 10^9
  – 100 Trillion bits vs 12.5 TB
Solving: Adjacency Matrix

• Suppose we have a graph with:
  – 10 million vertices. => \( V = 10 \times 10^6 \)
  – Each vertex has at most 20 neighbors.

• Adjacency matrix representation for the graph:
  – The smallest possible matrix: a 2D array of bits =>
  – The matrix size will be: \( V \times V \times 1\text{bit} \) =>
    \((10 \times 10^6) \times (10 \times 10^6) \times 1\text{bit} = 100 \times 10^{12} \text{bits} = 100 \text{ trillion bits} \)
  – Bits => bytes:
    \[ 1\text{byte} = 8\text{bits} \Rightarrow 100 \times 10^{12} \text{bits} = 100/8 \times 10^{12} \text{bytes} = 12.5 \times 10^{12} \text{bytes} \]
  – \( 12.5 \times 10^{12} \text{bytes} = 12.5 \text{ TB (final result)} \)
Solving: Adjacency List

• Suppose we have a graph with:
  – 10 million vertices. => \( V = 10 \times 10^6 \)
  – Each vertex has **at most 20 neighbors**.

• “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.”

• **Adjacency lists** representation of graphs:
  – For each vertex, keep a list of edges (a list of neighboring vertexes)
  – Space required for all the lists for all the vertices:
    \[ \leq 10 \text{ million vertices} \times 20 \text{ neighbors/vertex} = 200 \text{million items} = 200 \times 10^6 \text{ items} \]
    – Here an ‘item’ is a node in the adjacency lists

    Assume 8 bytes per item (we need to store both the next address and the vertex index in the node):

    \[ 200 \text{ million items} \times 8 \text{byte/item} = 1600 \times 10^6 \text{ bytes} = 1.6 \times 10^9 \text{ bytes} = \textbf{1.6GB} \]
    ( \( 10^9 \text{ bytes} = 1 \text{GB (GigaByte)} \) )
Check Out Posted Code

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