CSCI 2270 – CS 2: Data Structures

University of Colorado
Boulder

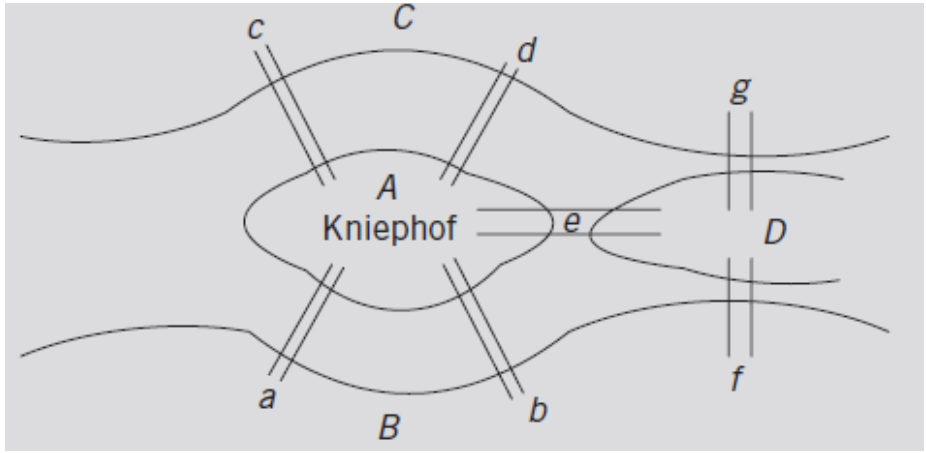
Reminders

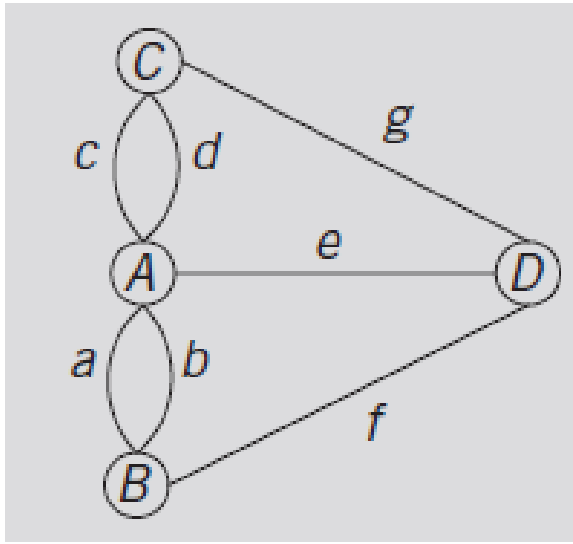
Topics

- Graphs
- Graph Traversal Algorithms
  - Breadth-First Search
  - Depth-First Search

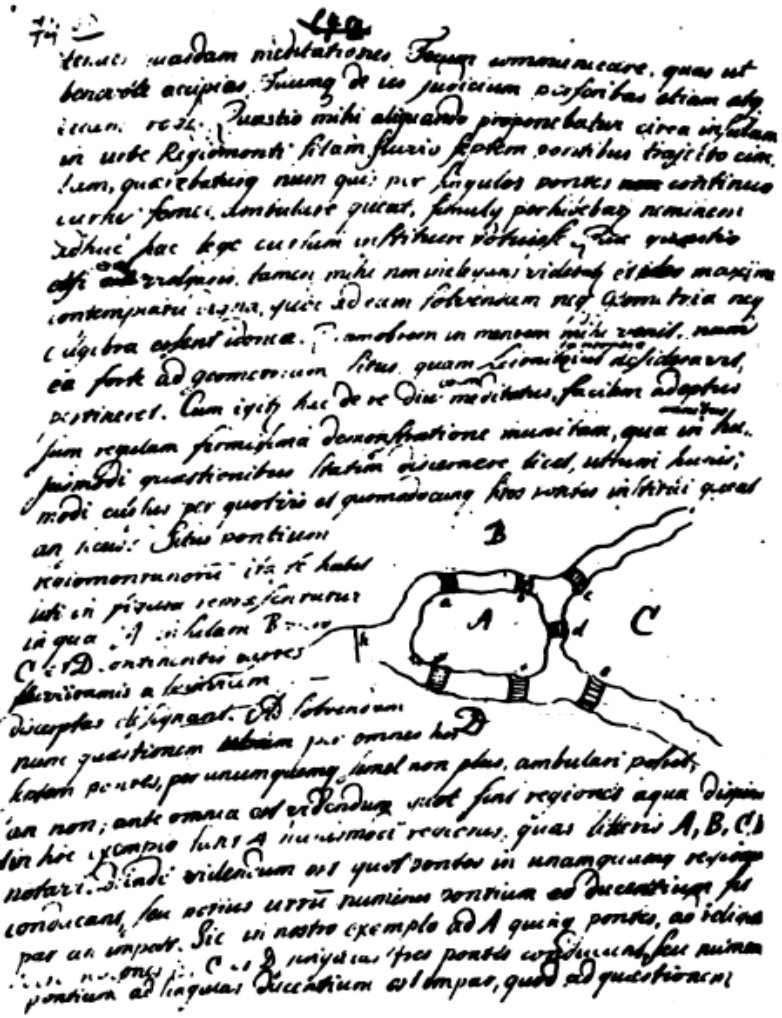
Graphs - Background

- **Königsberg bridge problem**
  - Given: river has four land areas
    - ***A, B, C, D***
  - Given: land areas connected using seven bridges
    - ***a, b, c, d, e, f, g***
  - Starting at one land area
    - ***Is it possible to walk across all the bridges exactly once and return to the starting land area?***
- **Euler represented problem as a graph**
  - Answered question in the negative
  - Marked birth of graph theory





Graphs - Background



Graphs - Applications

- Model electrical circuits
- Chemical compounds
- Highway maps
- Model flights between cities
- Social networks
- Neural networks

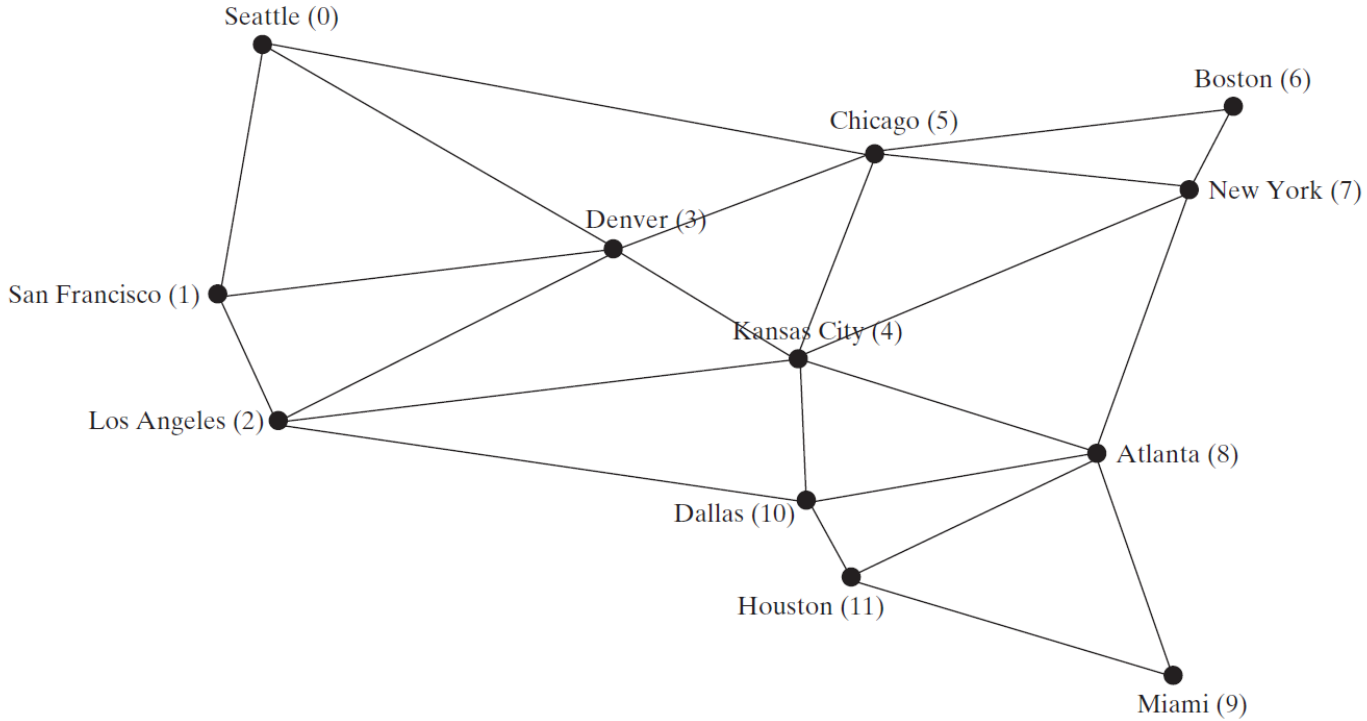
Graphs Definitions/Notations

- Borrow definitions, terminology from set theory
- **Subset**
  - Set *Y* is a subset of *X*: $Y \subseteq X$
    - ***If every element of Y is also an element of X***
- **Intersection** of sets *A* and *B*: $A \cap B$
  - Set of all elements that are in *A* and *B*
- **Union** of sets *A* and *B*: $A \cup B$
  - Set of all elements in *A* or in *B*
- **Cartesian product**: $A \times B$
  - Set of all ordered pairs of elements of *A* and *B*

Graphs Definitions/Notations

- **Graph *G* pair**
  - $G = (V, E)$, where V is a finite nonempty set
    - ***V* is called the set of vertices of *G***
    - $E \subseteq V \times V$
      - Elements of *E* are pairs of elements of V
- ***E*: set of edges of *G***
  - *G* called *trivial* if it has only one vertex

Graphs Definitions/Notations

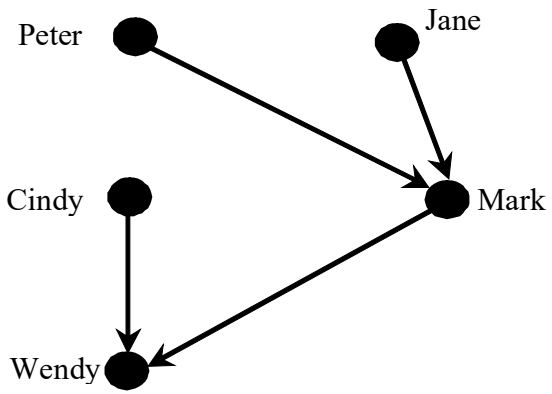


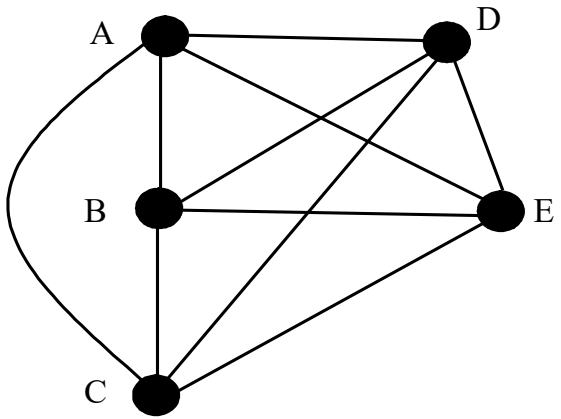
V = {“Seattle”, “San Francisco”, “Los Angeles”, “Denver”, “Kansas City”, “Chicago”, “Boston”, “New York”, “Atlanta”, “Miami”, “Dallas”, “Houston”};

E = {{“Seattle”, “San Francisco”}, {“Seattle”, “Chicago”}, {“Seattle”, “Denver”}, {“San Francisco”, “Denver”}, {“Chicago”, “Denver”}, … };

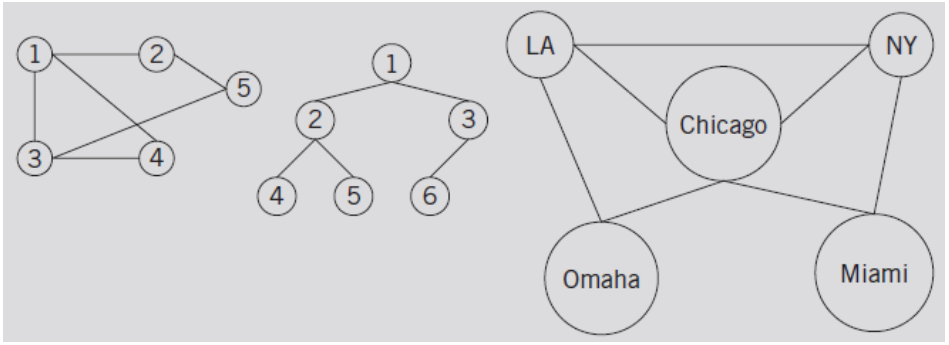
Graphs Definitions/Notations

- **A graph may be directed or undirected**
  - Directed graph (digraph)
    - ***Elements in set of edges of graph G: ordered***
    - ***Each edge has a direction so you can move from one vertex to the other through the edge.***
  - Undirected graph: not ordered
    - ***Move in both directions between vertices (see previous slide)***

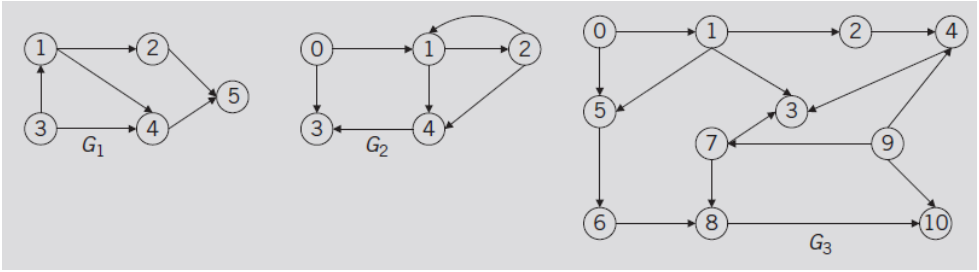




Graphs Definitions/Notations



Various undirected graphs



Various directed graphs

$V(G_1) = \{1, 2, 3, 4, 5\}$
$V(G_2) = \{0, 1, 2, 3, 4\}$
$V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}$
$E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}$
$E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3),$
$(5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4),$
$(9, 7), (9, 10)\}$

Graphs Definitions/Notations

- **Weighted graph**
  - In the examples so far, the edges represent a connection between two places, but do not contain any other information, such as the distance between the places. In a weighted graph, the edge has a weight that provides information about the connection, such as the distance, the cost of travel between vertices, or the flow of goods between two vertices. Using a weighted graph, questions such as, “What is the shortest distance between all vertices?” or “What is the cheapest path between two or more cities?” can be answered.

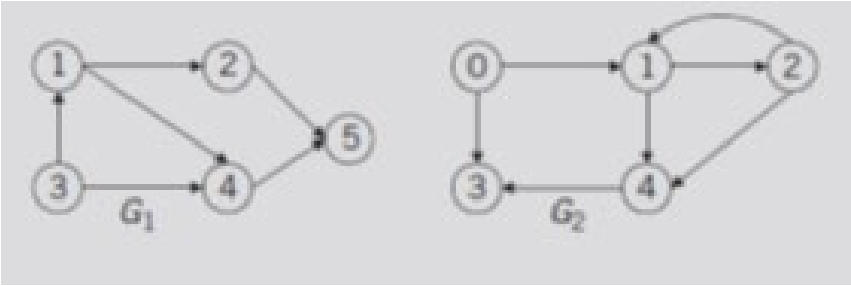
Graphs Representation

- To write programs that process and manipulate graphs, the graphs must be stored – that is, represented – in computer memory. A graph can be represented (in computer memory) in several ways. Two common ways are:
  - Adjacency matrix
  - Adjacency list

Graphs Representation

- Adjacency Matrix
  - Let *G* be a graph with *n* vertices where *n* > zero
  - Let $V(G) = \{v_1, v_2, \ldots, v_n\}$

$$A_G(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

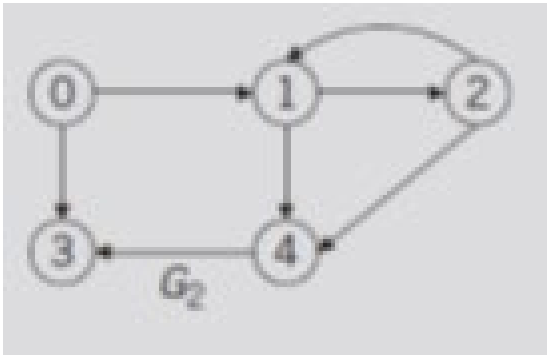


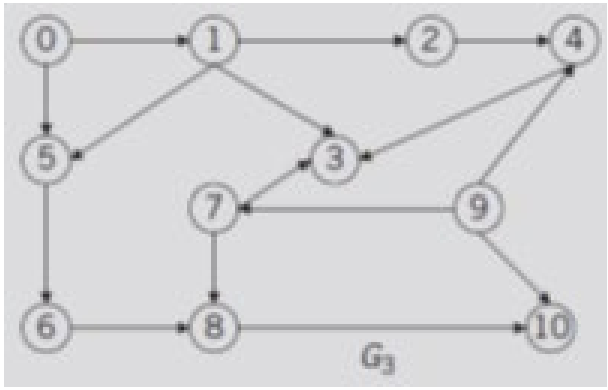
$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

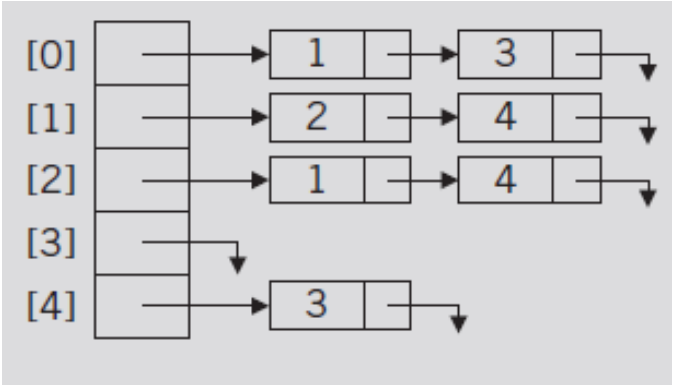
Graphs Representation

- Adjacency List
  - Given:
    - ***Graph G with n vertices, where n > zero***
    - ***V(G) = {v_1, v_2, ..., v_n}***
  - For each vertex *v*: linked list exists
    - ***Linked list node contains vertex u: (v, u) $\in$ E(G)***
  - Use array *A*, of size *n*, such that *A*[*i*] is a:
    - ***Reference variable pointing to first linked list node containing vertices to which v_i adjacent***
  - Each node has two components: vertex, link
    - ***Component vertex***
      - Contains index of vertex adjacent to vertex i

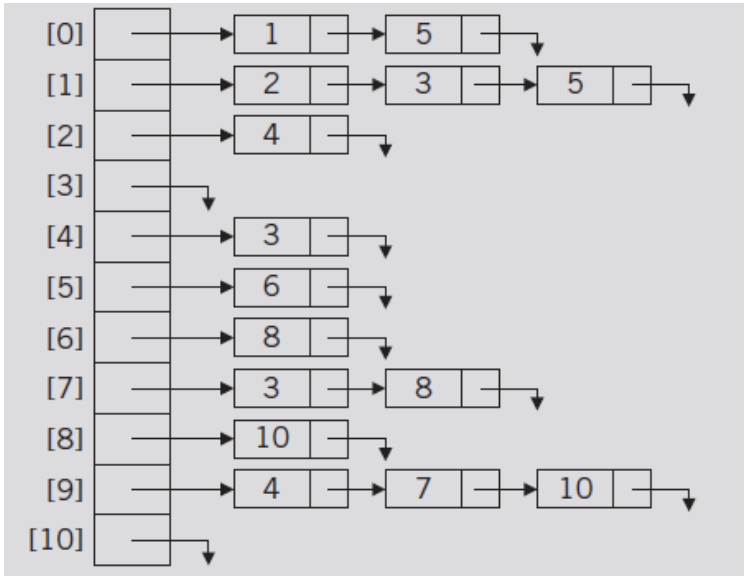
Graphs Representation







Adjacency list of graph *G2*



Adjacency list of graph *G3*

Graph ADT

- In the graph ADT, the vertices in the graph are stored as a private variable. There are public methods to initialize the graph, insert and delete edges and vertices, print the graph, and search the graph. The edges are stored in an adjacency list for each vertex in the vertices variable, and therefore, don’t need to be represented separately as private variables in the graph ADT.

Graph ADT

Graph:

1. private:
2. vertices
3. public:
4. Init()
5. insertVertex(value)
6. insertEdge(startValue, endValue, weight)
7. deleteVertex(value)
8. deleteEdge(startValue, endValue)
9. printGraph()
10. search(value)

Vectors

- With arrays, we specified the size of the array, but we don't always know how many values we'll have.
- Let's use a vector, which can change in size.
  - A vector stores a sequence of values whose size can change.
  - Add to a vector until your computer runs out of RAM.
- Include the <vector> header

Vectors

- When you declare a vector, you specify the type of the elements like you would with an array, but the type must be preceded by the word vector.

```
vector<double> data;
```

- The element type must be in angle brackets. Other examples:

```
vector<int> counts;
vector<string> team_names;
vector<double> values = {32, 54, 67.5, 29, 34.5};
```

- By default, a vector is empty when created.

Vectors

`vector<int> numbers(10);`	A vector of ten integers.
`vector<string> names(3);`	A vector of three strings.
`vector<double> values;`	A vector of size 0.
`vector<double> values();`	**Error:** Does not define a vector.

`vector<int> numbers;`
`for (int i = 1; i <=10; i++)`
`{ numbers.push_back(i);}`
	A vector of ten integers, filled with 1, 2, 3, ..., 10.

`vector<int> numbers(10);`
`for (int i = 0; i < numbers.size(); i++)`
`{ numbers[i] = i + 1;}`
	Another way of defining a vector of ten integers 1, 2, 3, ..., 10.

`vector<int> numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`	This syntax is supported with C++ 11 and above.

Vectors

- You can access the elements in a vector the same way as in an array, using an [index].

```
vector<double> values(4);
//display the forth element
cout << values[3] << end;
```

- It is an error to access an element that is not in a vector.

```
vector<double> values(4);
//display the fifth element
cout << values[4] << end;  //ERROR
```

Vectors – push_back and pop_back

- The function *push_back* puts a value into a vector:

```
values.push_back(32); // 32 added to end of vector
```

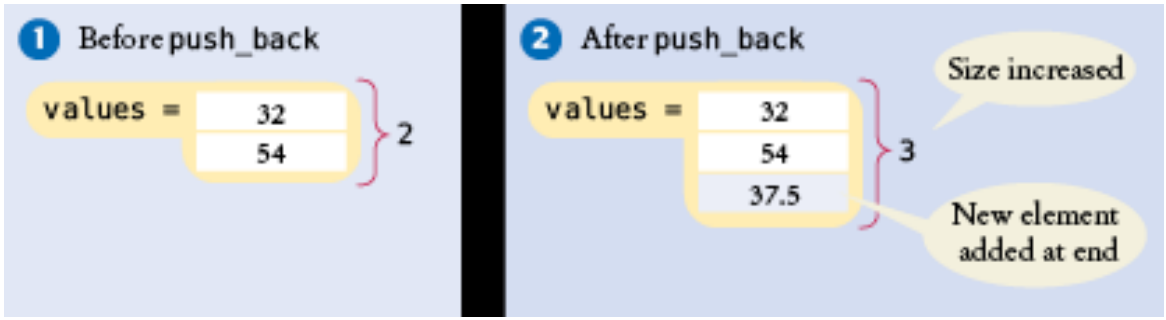
- The vector increases its size by 1.

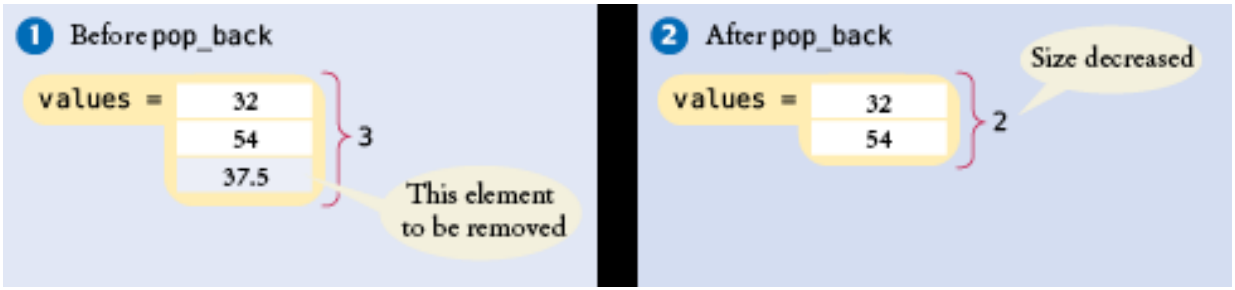
- *pop_back* removes the last value placed into the vector, and the size decreases by 1:

```
values.pop_back();
```

Vectors – push_back and pop_back

```
// an empty vector
vector<double> values;

values.push_back(32)
// values.size() now is 1

values.push_back(54);

values.push_back(37.5);
// values.size() now is 3

values.pop_back();
//removes the 37.5
//values.size()==2
```





Vectors – size()

- Vectors have the size member function which returns the current size of a vector.

  The vector always knows how many elements are in it and you can always ask it to give you that quantity by calling the size method:

```
for (int i = 0; i < values.size(); i++)
{
   cout << values[i] << endl;
}
```

Vectors – Parameters to Functions

The following function computes the sum of a vector of floating-point numbers:

```
double sum(vector<double> values)
{
   double total = 0;
   for (int i = 0; i < values.size(); i++)
   {
      total = total + values[i];
   }
   return total;
}
```

This function *visits* the vector elements, but does *not change* them.

Unlike an array, a vector is *passed by value* (copied) to a function, not passed by reference.

Vectors – Parameters to Functions

- If the function *should change* the values stored in the vector, then a vector reference must be passed, just like with int and double reference parameters. The & goes after the <type>:

```
void multiply(vector<double>& values, double factor)
{
      for (int i = 0; i < values.size(); i++)
      {
         values[i] = values[i] * factor;
      }
}
```

Vectors – Returning from Function

- Sometimes the function should ***return*** a **vector.**
- Vectors are no different from any other data types in this regard.
- Simply declare and build up the result in the function and return it:

```
vector<int> squares(int n)
{
   vector<int> result;
   for (int i = 0; i < n; i++)
   {
      result.push_back(i * i);
   }
   return result;
}
```

- The function returns the squares from 0^2 up to $(n - 1)^2$ as a vector.

Vectors – Copying

- Vectors do not suffer the limitations of arrays, where we had to include a code loop to copy each element.

```
vector<int> squares;

for (int i = 0; i < 5; i++)
{
   squares.push_back(i * i);
}

vector<int> lucky_numbers;
    // Initially empty

lucky_numbers = squares; //vector copy

    // Now lucky_numbers contains
    // the same elements as squares
```

Graph Implementation

In code, the graph can be represented in a Graph class:

```
class Graph{
  private:
    //vertices and edges definition goes here
  public:
    //methods for accessing the graph go here
}
```

Each vertex in the graph is defined by a **struct** with two members: a *key* that serves as the key value for the vertex, and a vector (named *adjacent* below) to store the adjacency list for the vertex.

A vertex is defined as:

```
struct vertex{
    string key;
    vector<adjVertex> adjacent;
}
```

Graph Implementation

```
struct vertex{
    string key;
    vector<adjVertex> adjacent;
}

struct adjVertex{
   vertex *v;
   int weight; // optional: weighted vs unweighted
               // stores the edge weight between 2 vertices
};
```

The *adjVertex* data type is also defined by a **struct** with two members: contains a pointer to the adjacent vertex *v*, and an integer *weight* that stores the edge weight between the two vertices.

Graph Implementation

An empty vector of *vertex* can be created using the statement:

```
vector <vertex> vertices;
```

The *adjVertex* **struct** only stores the destination vertex in *v* because the origin vertex is stored in the *vertices* vector. In this design, each vertex in the graph has a vector of adjacent vertices that contains the vertex at the other end of the edge. The number of adjacent vertices can vary for each vertex in *vertices*. The size of the *adjacent* vector is also dynamic for each vertex.

Graph Implementation

- Graph sample - Graph.cpp

Graph Traversals

- Traversing a graph is similar to binary tree traversal
- Graph traversal can have cycles, whereas binary trees do not
  - May not be able to traverse the entire graph from a single vertex
  - Therefore, we must keep track of the vertices that have been visited
  - We must traverse the graph from each vertex (that hasn't been visited) of the graph. This ensures that the entire graph is traversed.
- Two common graph traversal algorithms:
  - Depth-first traversal
  - Breadth-first traversal

Questions?