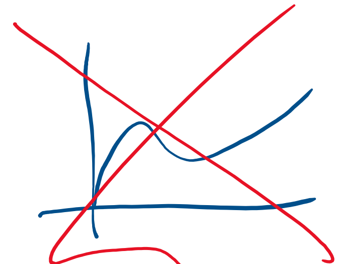
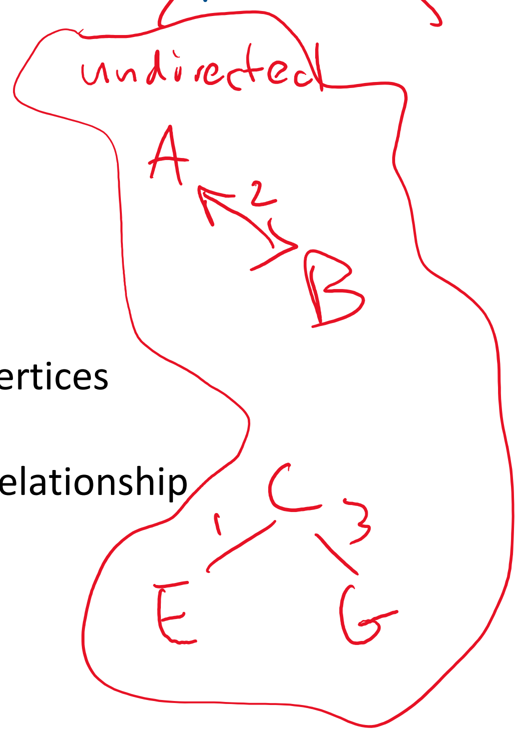
# (1) The Graph Data Structure

What is a graph?

Not a
Cartesian coordinate graph (x-y graph).



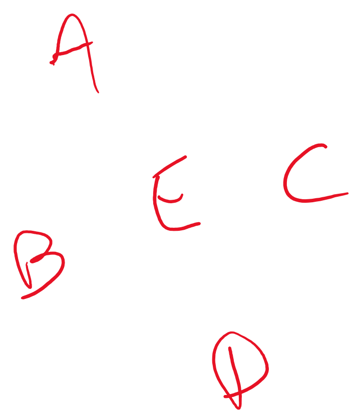
- A collection of vertices connected by edges
- Each vertex contains a key and a list of edges
- Can either undirected or directed graphs
- Can be either unweighted or weighted




How's different from a BST?

- there are no implicit relationships between vertices
  - there is no single root
  - cannot say v0>v1 based on parent child relationship
- Each edge has to be set explicitly

Applications:
- neural network
- social media
- small molecules ?
- geospacial (think google maps)
- flight routing (e.g. expedia)

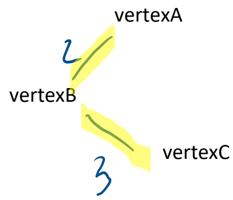


## (2) Graph ADT

- undirected
- weighted

**ADT:**
private:
- vertices
- *edges*

public:
- insertVertex(value)
- addEdge(startValue, endValue, *weight*)
- deleteVertex(value)
- deleteEdge(startValue,endValue)
- displayGraph()
- search(value) - *simple linear search*
- traversal methods... - *more on this later*



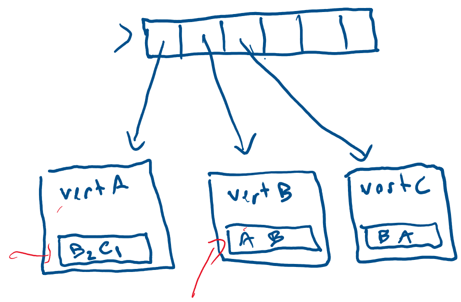

vA | vB | vC
B | A | B
 | C |

**C++ definitions WEIGHTED:**

```
private:
    std::vector<vertex*> vertices;

struct vertex{
    string key;
    vector<edge> adj;   list
};

struct edge{
    vertex* v;
    int weight; = 1
};
```

**UNWEIGHTED:**

```
private:
    std::vector<vertex*> vertices;

struct vertex{
    string key;
    vector<vertex*> adj;
};
```




Adjacency List contains the list of edges for each vertex

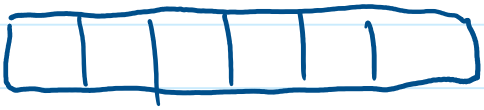
BAD:
```
std::vector<vertex> vertices;
```
don't do it

# (3) STL Vectors
## Review

Standard Template Library - very widely used set of template classes. Includes most of the common data structures (list, queue, stack, vector, etc.).

Template class - *A CLASS THAT WORKS generically on any type (primitive or user-defined)*

Primitive type example:



```
vector<int> v0;
```

User-defined type example:

```
struct myStruct{
   int numbers;
   string words;
};
```

```
vector<myStruct> vectorOfStructs;
```

*see vectorSTLdemo.cpp*

vectorSTLdemo

# (4) Insert Vertex

insertVertex(key) - inserts a vertex into graph with no edges (empty adjacency list)

Alogorithm:

1. search to ensure no duplicates
2. create a vertex with key value

Example
Given the following graph, insert new vertex with key = "fairbanks"

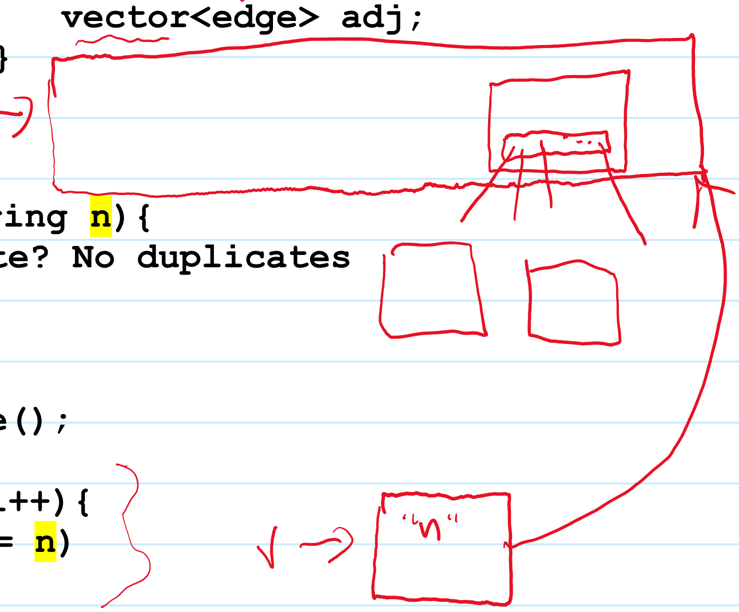
*fairbanks*

**denver**

**boulder**

**new orleans**

```
class Graph{
private:
    std::vector<vertex*> vertices;

    ...
}

struct vertex{
    string key;
    vector<edge> adj;
}
```

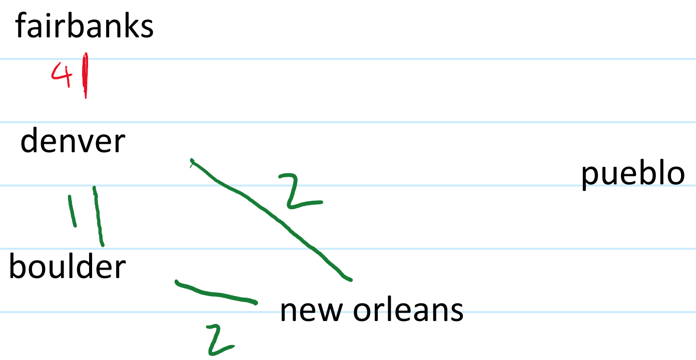



```
void Graph::insertVertex(string n){
    // 1.: check for duplicate? No duplicates
    // allowed

    bool found = false;
    int vSize = vertices.size();

    for(int i = 0; i<vSize; i++){
        if(vertices[i]->key == n)
            found = true;
    }
    // 2.: insert if no duplicate
    if(!found){
        vertex *v = new vertex;
        v->key = n;
        vertices.push_back(v);
    }
} else {
```

# (5) Add Edge

addEdge(key0, key1, weight) - add a connection between two keys with a specified weight

Example:
Given the following graph, add an edge between fairbanks and denver. Set weight = 4.




Approach:

1. Locate key0 in the graph - call this v0
2. Locate key 1 in the graph - call this v1
3. Create a new edge (e0)
    a. set e0 to point to v1
    b. set weight to 4
    c. append e0 to v0's adjacency list
4. Create a new edge (e1)
    a. set e1 to point to v0
    b. set weight to 4
    c. append e1 to v1's adjacency list

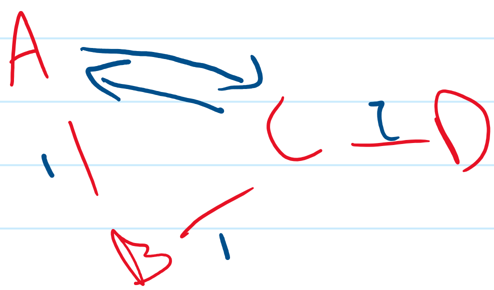
# (6) Add Edge C++

```
struct edge{
    vertex* v;
    int weight;
};
```

A i C 3 j

```
void Graph::addEdge(string v1, string v2, int _weight){
    int vSize = vertices.size();
    for(int i = 0; i <vSize; i++){
        if(vertices[i]->name == v1)
            for( int j = 0; j<vSize; j++ )
              if(vertices[j]->name == v2 && i != j ){
                  edge e0;
                  e0.v = verticec[j];
                  e0.weight = _weight;
                  vertices[i]->adj.push_back(e0);

                  edge e1;
                  e1.v = vertices[i];
                  e1.weight = _weight;
                  vertices[j]->adj.push_back(e1);
            }


    }

}
```

```
struct vertex{
    string key;
    vector<edge> adj;
}
```

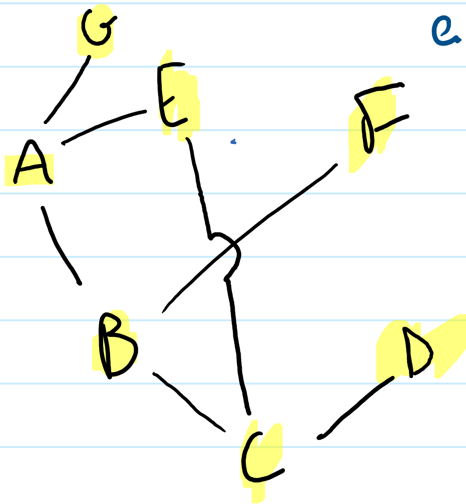


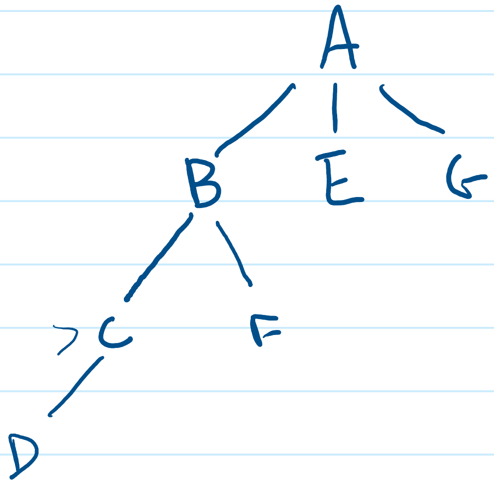
# (7) Breadth First Traversal

Say we are given a graph (unweighted, undirected). We are asked to come up with a traversal algorithm, such that:

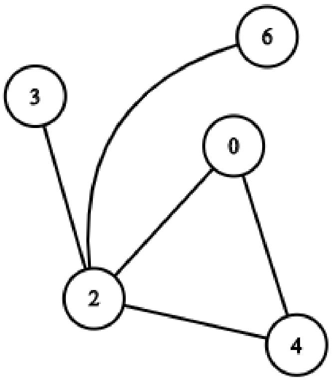
- Given a starting vertex, we visit all neighboring vertices (depth = 0). Then, we visit all the vertices in the next depth level (depth = 1). Then the next depth level, and so on.

e.g.: traverse (A)



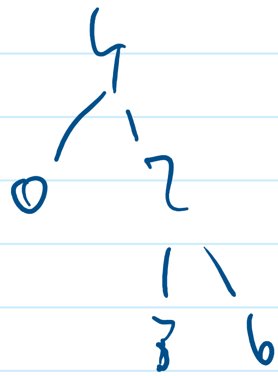



# (8) BFT Examples

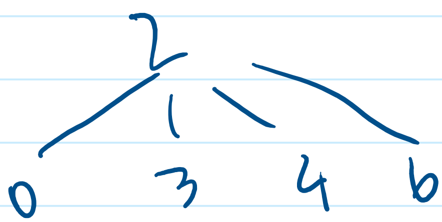



| Node | Adj List |
|---|---|
| 0 | |
| 2 | |
| 3 | |
| 4 | |
| 6 | |

BFT(4)




BFT(2)




How do we do this in code?

# extra - spot the problem?

```
UNWEIGHTED:
private:
   std::vector<vertex> vertices;

struct vertex{
   string key;
   vector<vertex*> adj; // adjacency list
};
```

*If in **insert** we had:*

```
if(!found){
   vertex v;
   v.key = n;
   vertices.push_back(v);
}
```

*And in addEdge we had:*

```
e0.v = &vertices[j];

vertices[i]->adj.push_back(e0);
```

Friday, November 1, 2024 1:56 PM

recPtrs

Monday, November 4, 2024 1:57 PM

```
class Graph{
private:
    std::vector<vertex*> vertices;

    ...
}
```