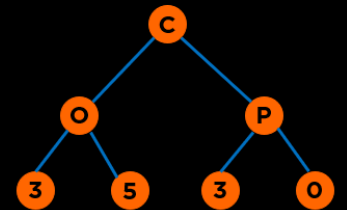


# Graphs



# Categories of Data Structures

Linear Ordered

Lists

Stacks

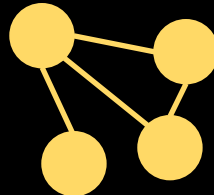
Queues



Non-linear Ordered

Trees

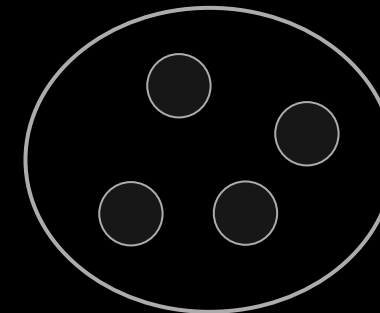
Graphs



Not Ordered

Sets

Tables/Maps



# Categories of Data Structures

Linear Ordered

Lists

Stacks

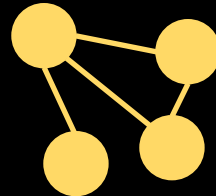
Queues



Non-linear Ordered

Trees

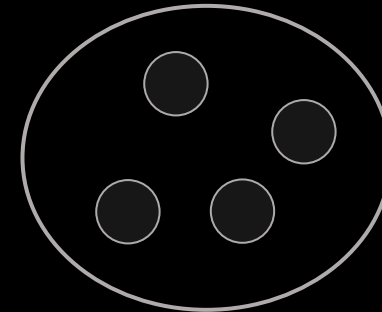
Graphs



Not Ordered

Sets

Tables/Maps



# Agenda

- **Graphs**

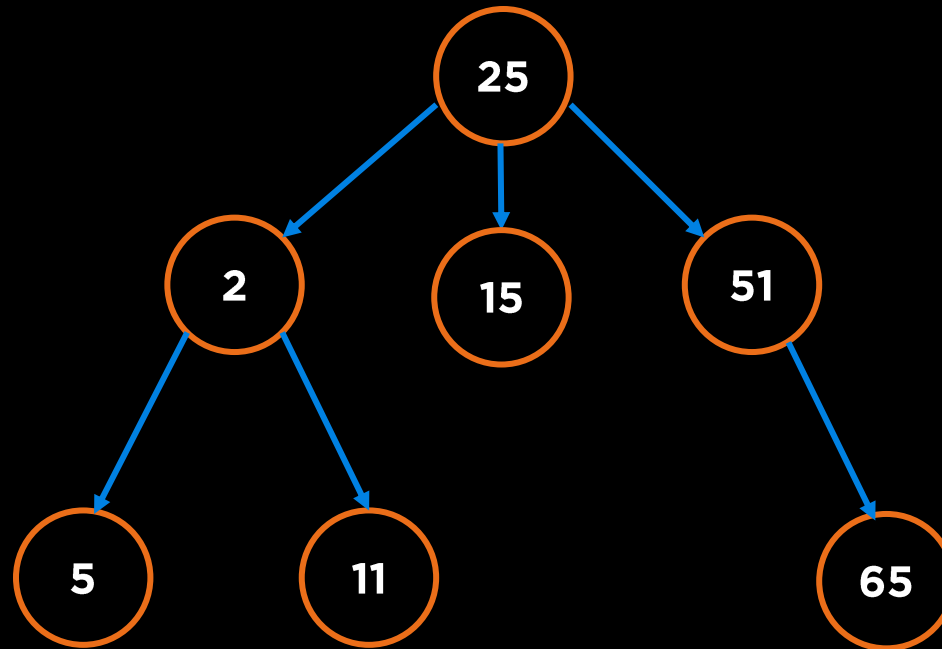
- **Terminology**
- **Types**
- **Use cases**

- **Graph Implementations**

- **Edge List**
- **Adjacency Matrix**
- **Adjacency List**

# Trees

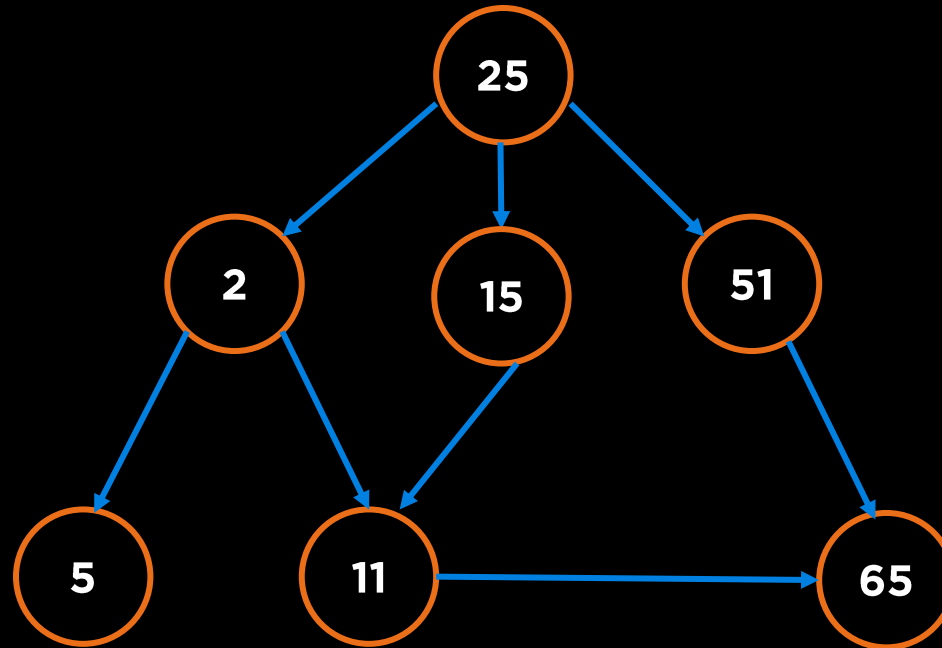
Hierarchical, Acyclic, and Exactly one path between two nodes



# Graphs

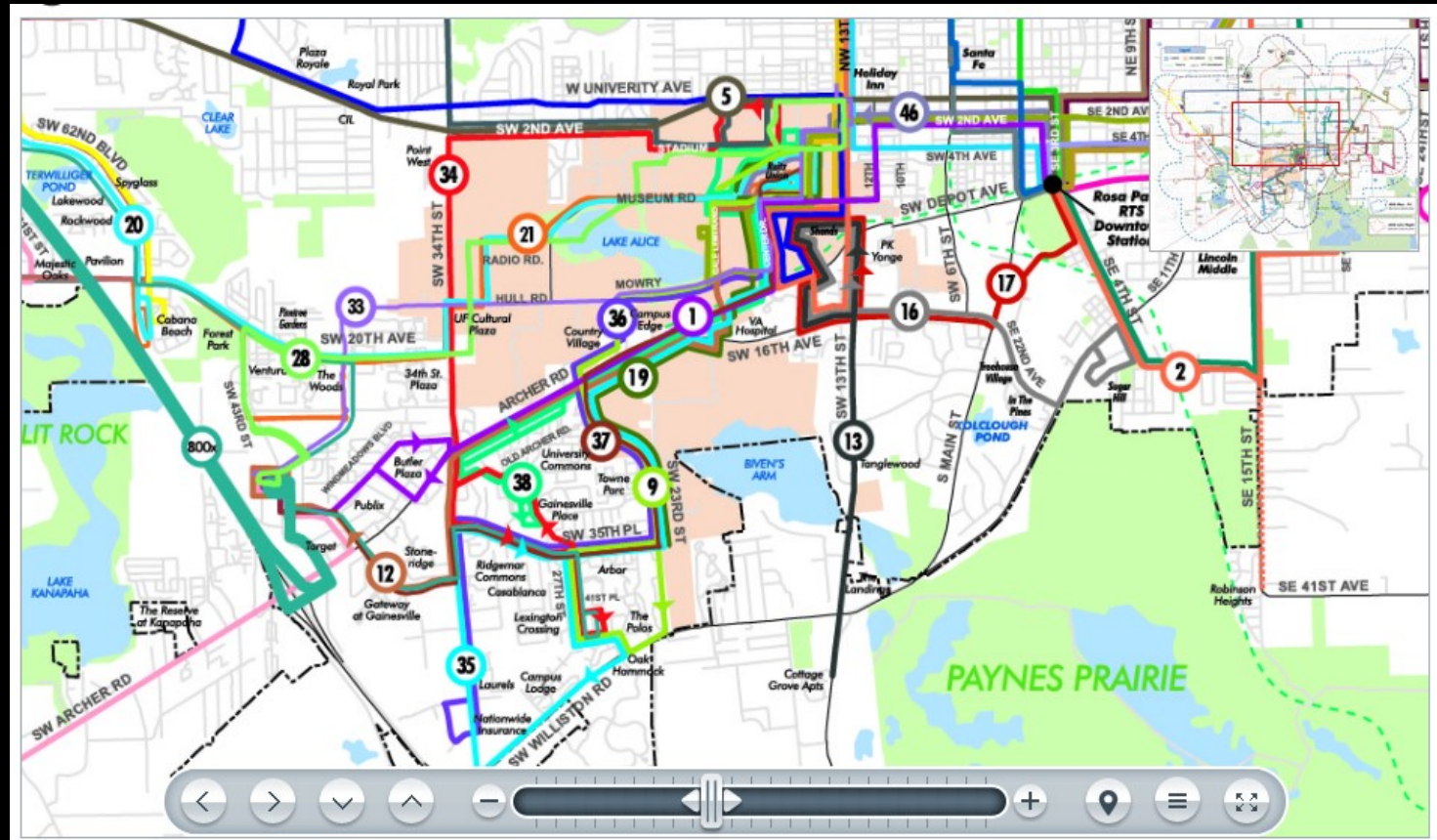
An ordered pair of a set of nodes and a set of edges.

$$G = (V, E)$$

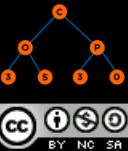


# Graphs

## Example



# Graph Terminology





# Graph: Terminology

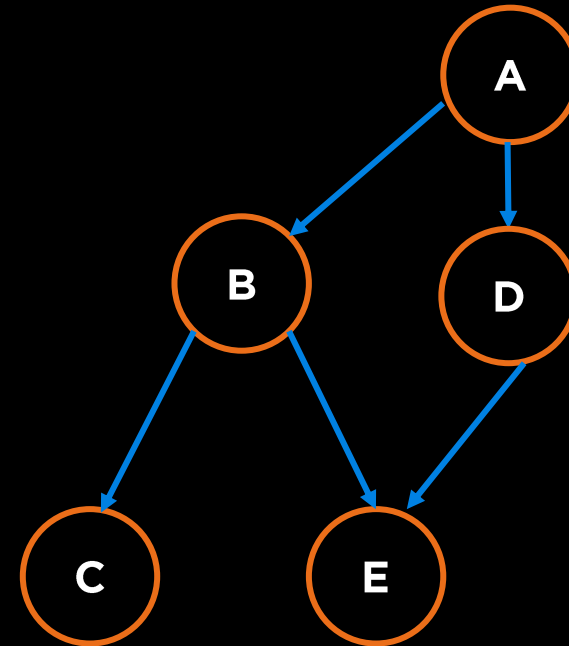
## Vertex

Each node in a Graph is called a Vertex

$V = \{A, B, C, D, E\}$

$|V|$  is the number of vertices in the graph

$|V| = 5$



# Graph: Terminology

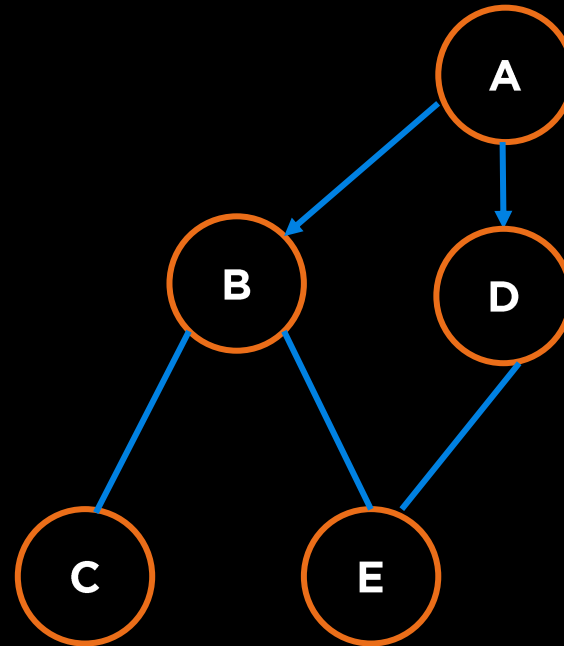
## Edge

The connections between two nodes is called an edge.

$E = \{(A,B), (A,D), \{B,C\}, \{B,E\}, \{D,E\}\}$

$|E|$  is the number of edges in the graph

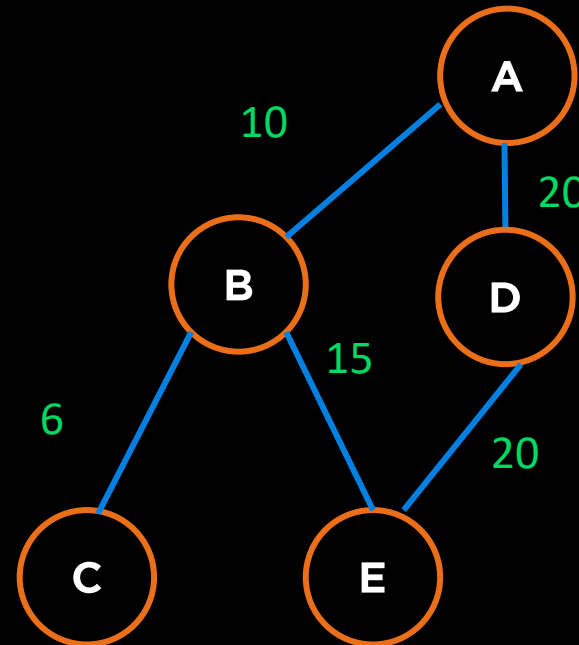
$|E| = 5$



# Graph: Terminology

## Weight

The edges in a graph may have associated values known as their weights. A weight is like a cost to travel from one vertex to the other over the edge.

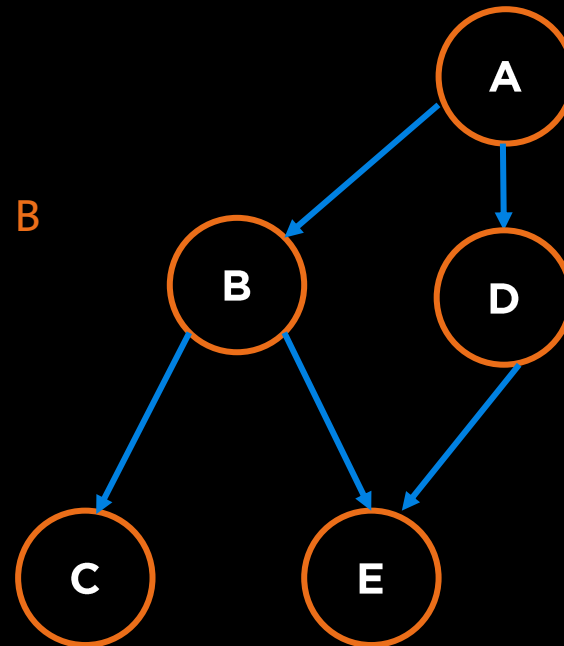


# Graph: Terminology

## Adjacent Vertices

**A vertex is adjacent to another vertex if there is an edge to it from that other vertex.**

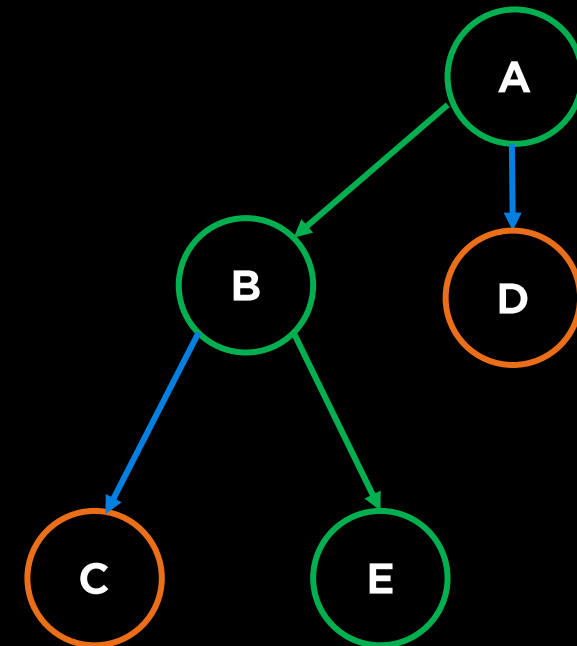
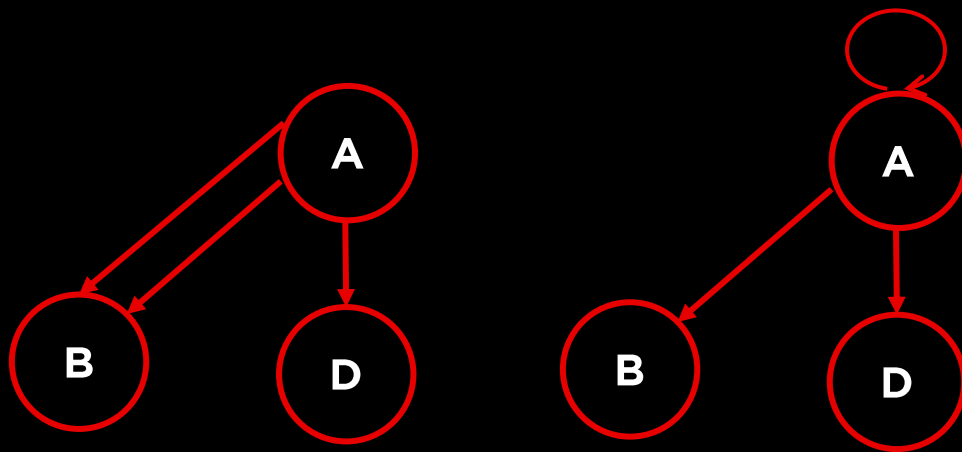
B is adjacent to A but A is not adjacent to B



# Graph: Terminology

## Simple Graph

A simple graph is a graph with no edges that connect a vertex to itself, i.e. no “loops” and no two edges that connect the same vertices, i.e. no “parallel edges”.

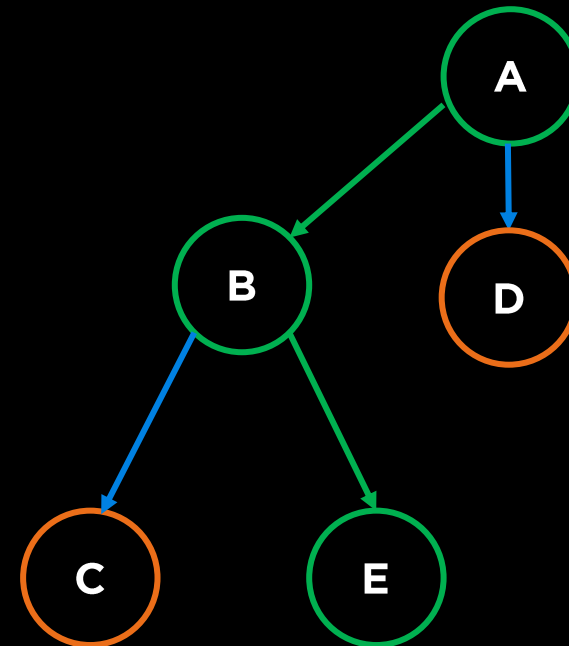


# Graph: Terminology

## Path

**A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor.**

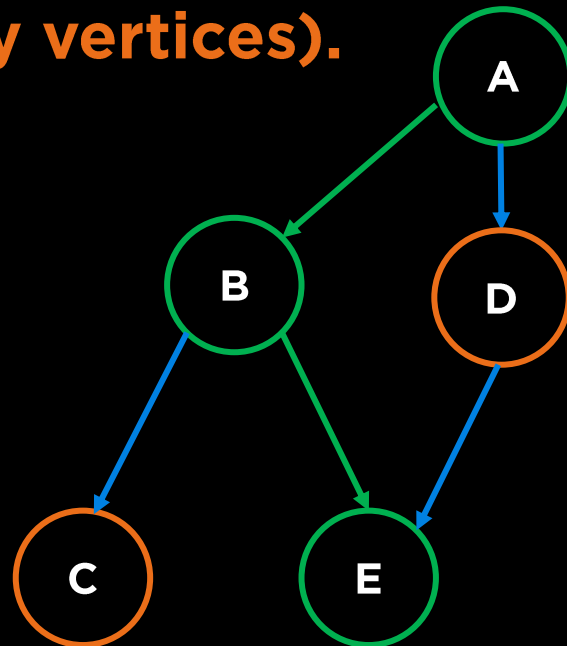
Path from A to E: A, B, E



# Graph: Terminology

## Simple Path

In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same (no repeated intermediary vertices).

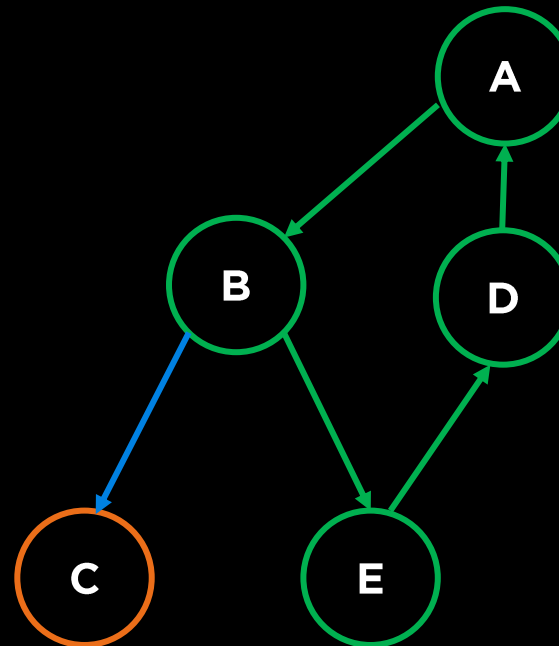


# Graph: Terminology

## Cycle

**A cycle is a simple path in which only the first and final vertices are the same.**

A - B - E - D - A is a cycle.



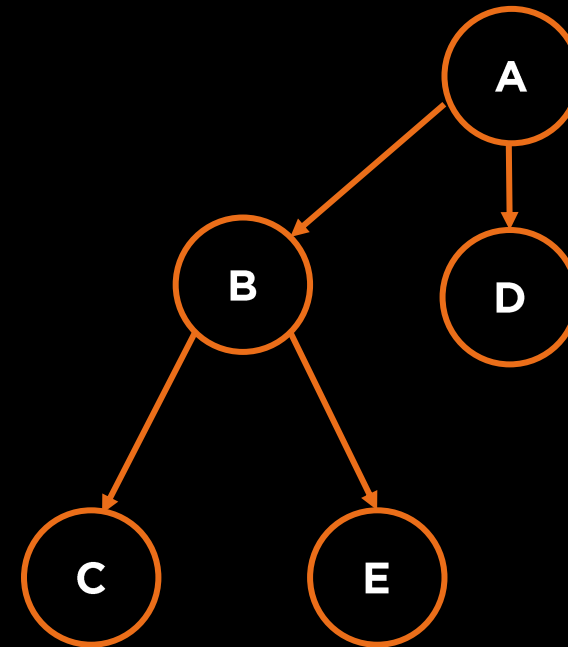


# Graph: Terminology

## Connected Vertex

**Two vertices are connected if there is a path between them.**

A and C are connected  
D and C are not connected

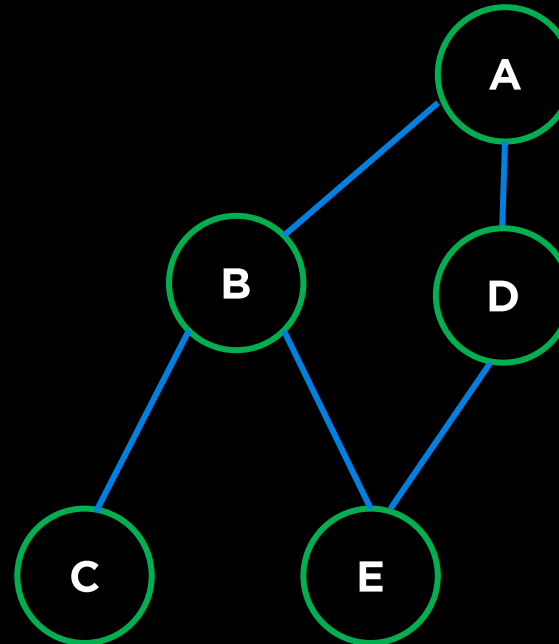


# Graph: Terminology

## Connected Graph

**An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.**

This is a connected graph



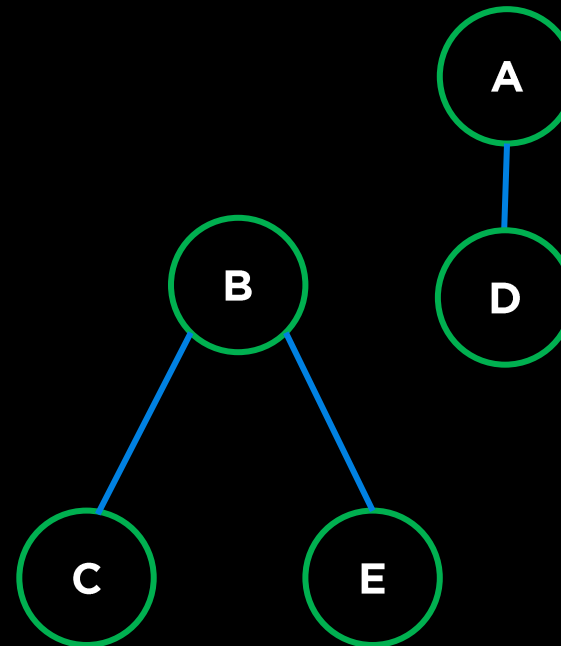
# Graph: Terminology

## Connected Graph

**An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.**

This is not a connected graph.

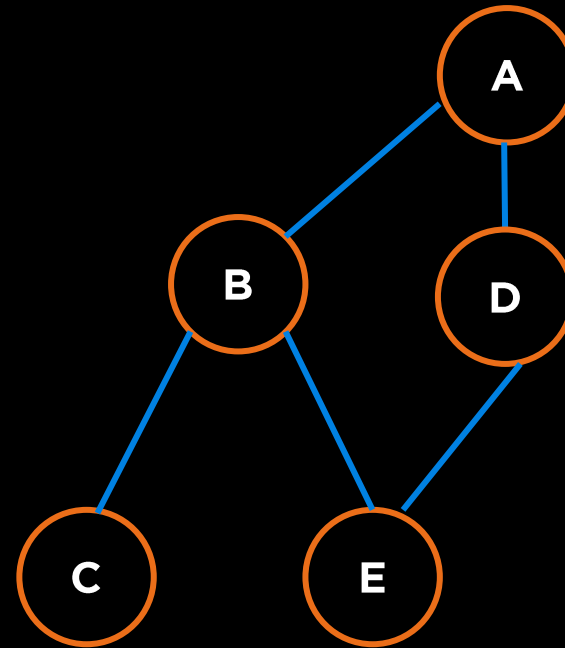
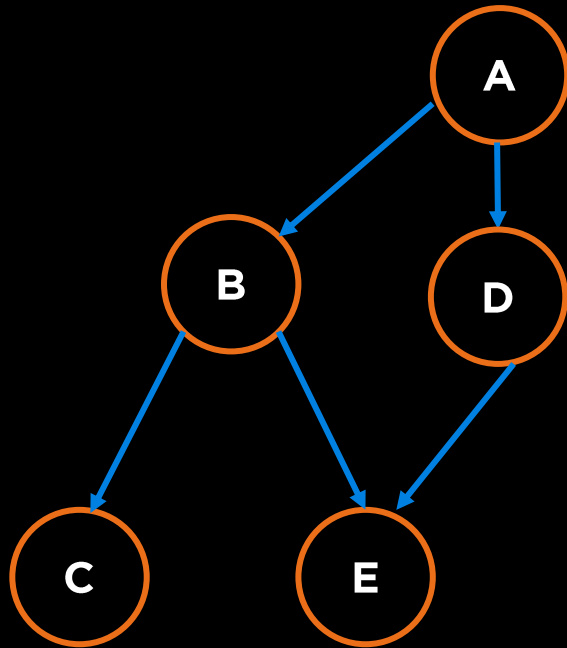
Connected components:  
 $\{A, D\}$  and  $\{B, C, E\}$



# Graph Types

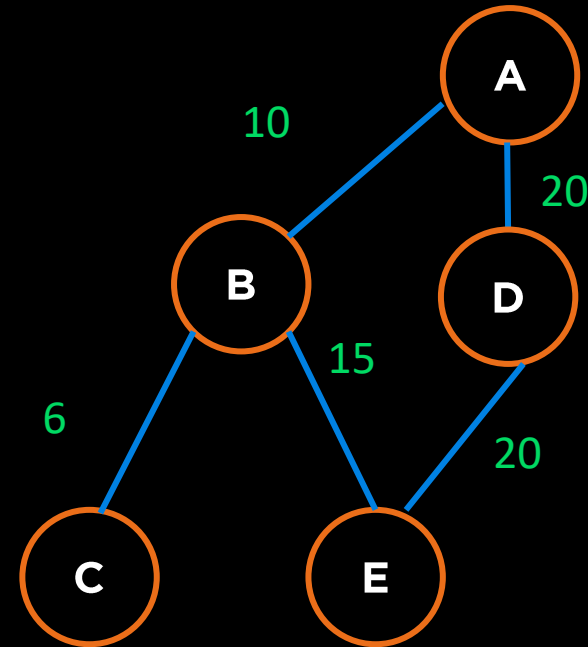
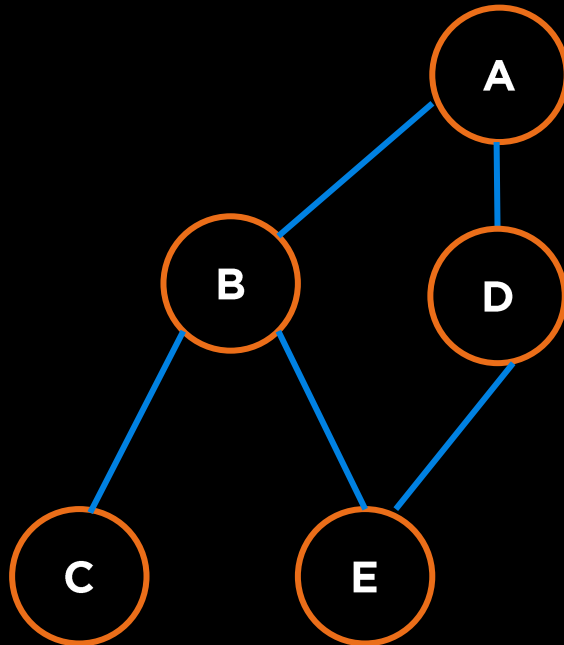
# Graph Types

## Directed (Digraph) vs Undirected



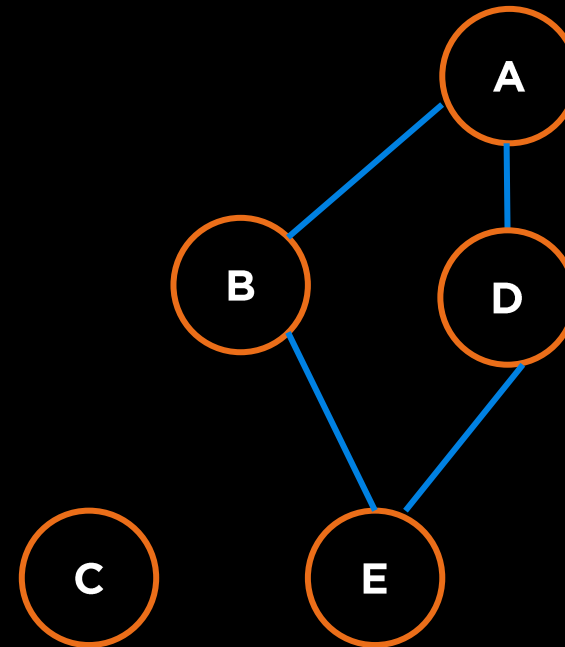
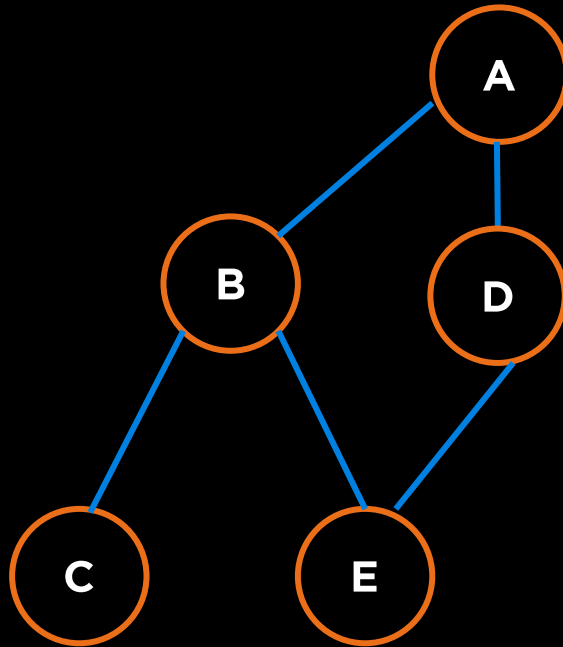
# Graph Types

## Weighted vs Unweighted



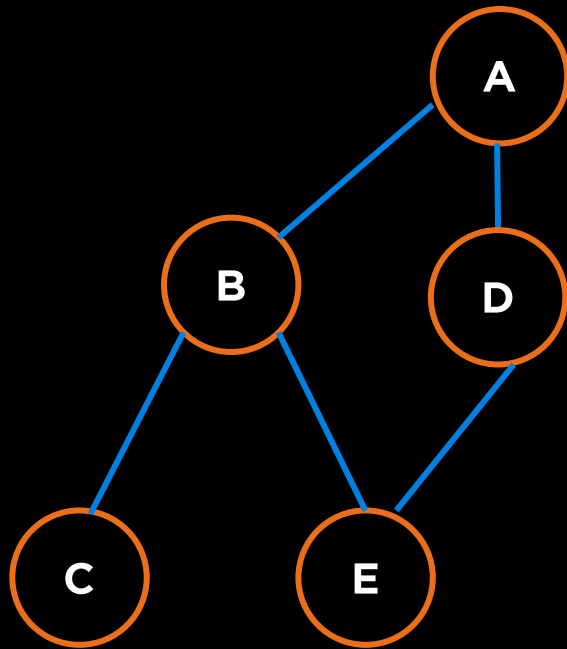
# Graph Types

## Connected vs Unconnected

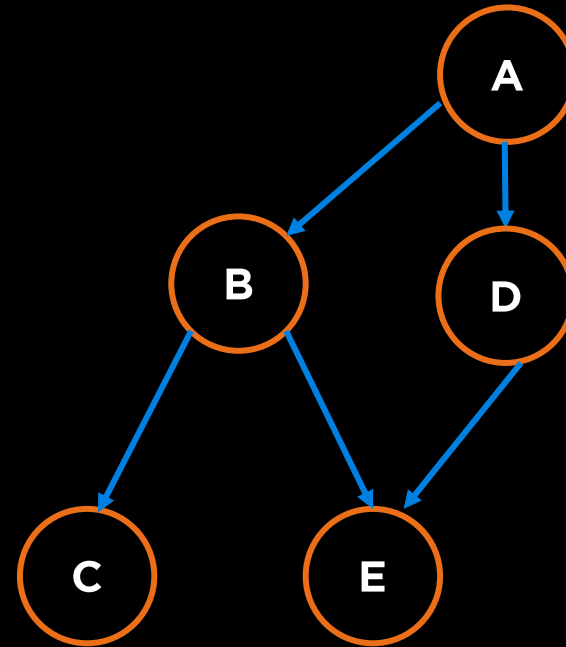


# Graph Types

## Cyclic vs Acyclic



Cyclic



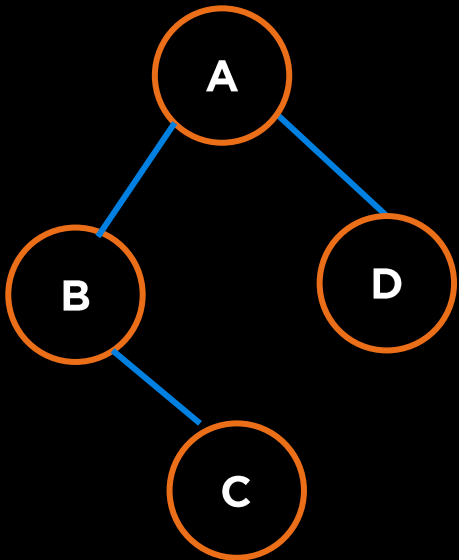
Acyclic



# Graph Types

## Dense vs Sparse

- The density of a graph is the ratio of  $|E|$  to  $|V|^2$
- We can assume that  $|E|$  is
  - $\sim |V|^2$  for a dense graph [Density  $\sim 1$ ]
  - $\sim |V|$  for a sparse graph [Density  $\sim 0$ ]



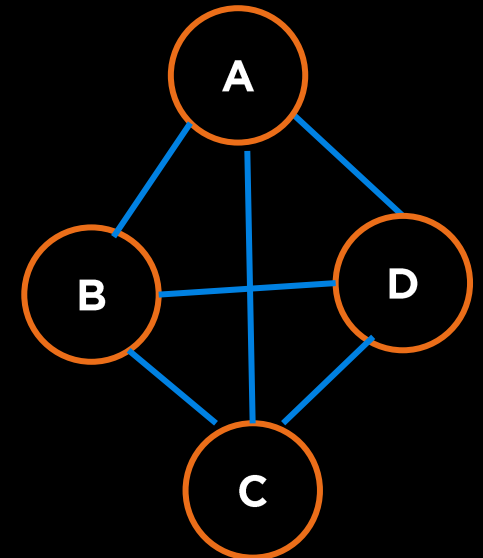
Sparse

Directed Graphs:

$$0 \leq |E| \leq |V|(|V|-1)$$

Undirected Graphs:

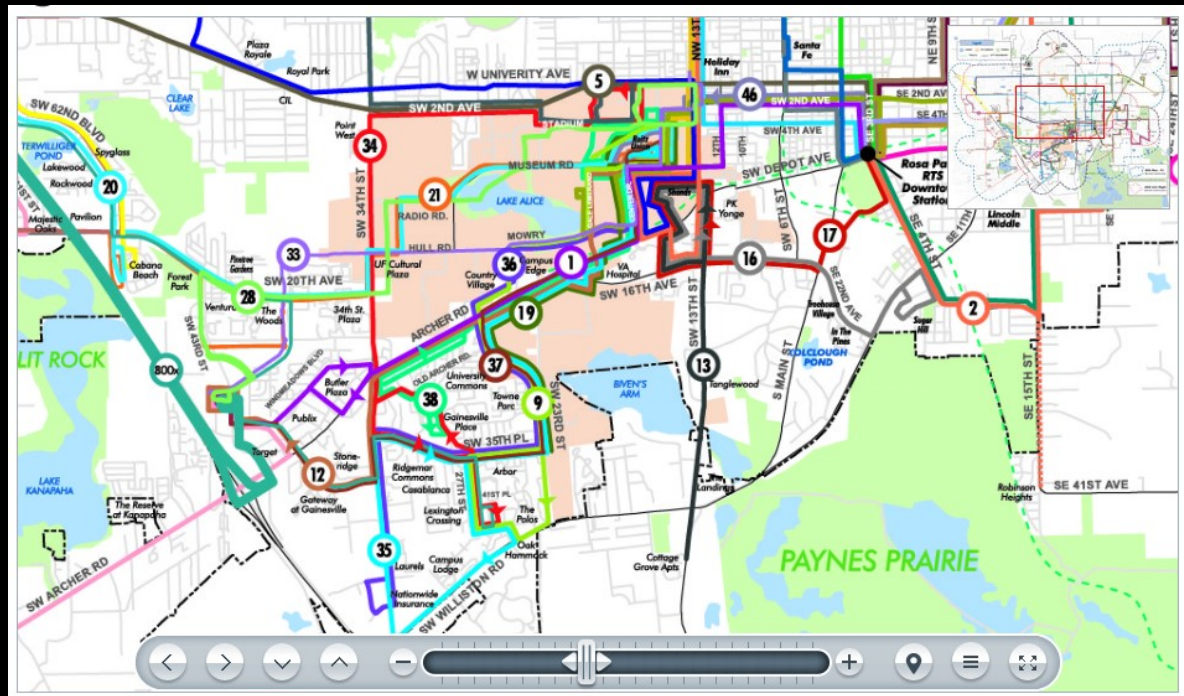
$$0 \leq |E| \leq |V|(|V|-1)/2$$



Dense

# Graphs

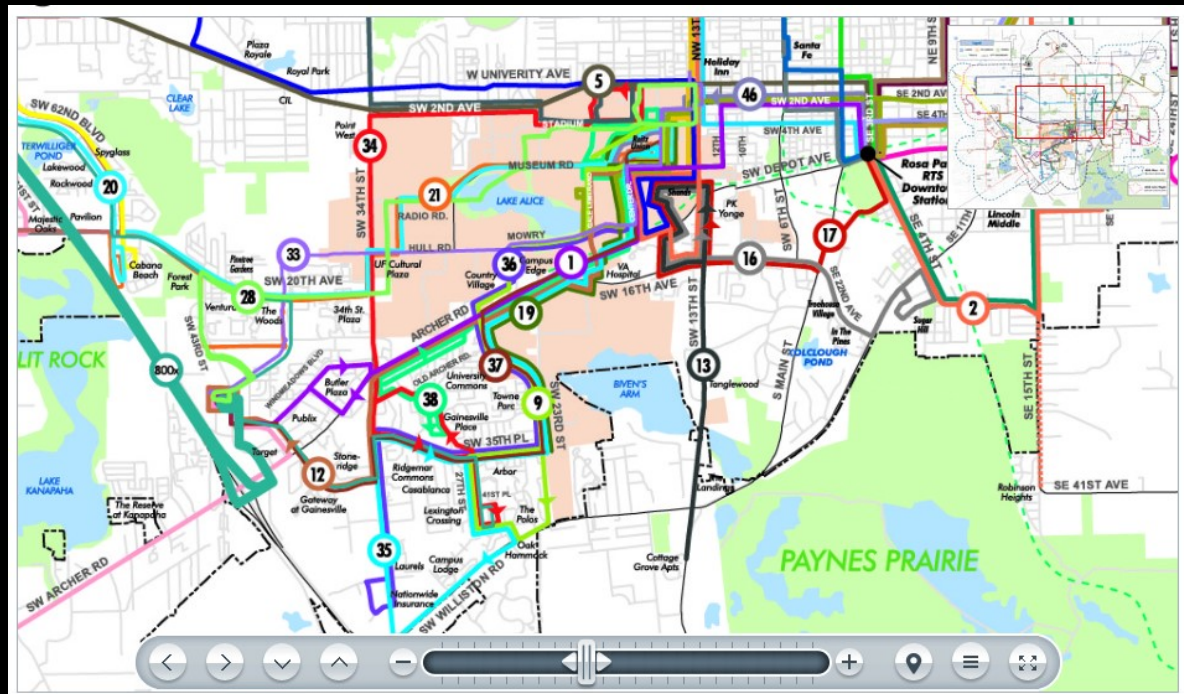
## Example



Undirected  
Directed  
Cyclic  
Connected

# Graphs

## Example



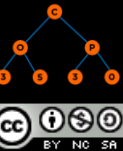
- Undirected
- Directed
- Cyclic
- Connected

# Graph Problems

Common Examples:

- Social Networks
- World Wide Web
- Maps

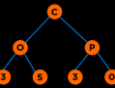
**Weighted? Directed?**



# Graph Problems

## Common Examples:

- Social Networks (Unweighted, Undirected)
- World Wide Web (Unweighted, Directed)
- Maps (Weighted, Undirected)



# Graph Problems

There are lots of interesting questions we can ask about a graph:

- What is the shortest route from S to T? What is the longest without cycles?
- Are there cycles?
- Is there a tour you can take that only uses each node (station) exactly once?
- Is there a tour that uses each edge exactly once?

# Graph Problems

Some well-known graph problems and their common names:

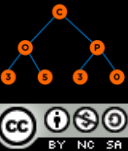
- **s-t Path.** Is there a path between vertices  $s$  and  $t$ ?
- **Connectivity.** Is the graph connected, i.e. is there a path between all vertices?
- **Biconnectivity.** Is there a vertex whose removal disconnects the graph?
- **Shortest s-t Path.** What is the shortest path between vertices  $s$  and  $t$ ?
- **Cycle Detection.** Does the graph contain any cycles?
- **Euler Tour.** Is there a cycle that uses every edge exactly once?
- **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?
- **Planarity.** Can you draw the graph on paper with no crossing edges?
- **Isomorphism.** Are two graphs isomorphic (the same graph in disguise)?

Often can't tell how difficult a graph problem is without very deep consideration.

# Questions

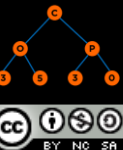


# Graph Implementations



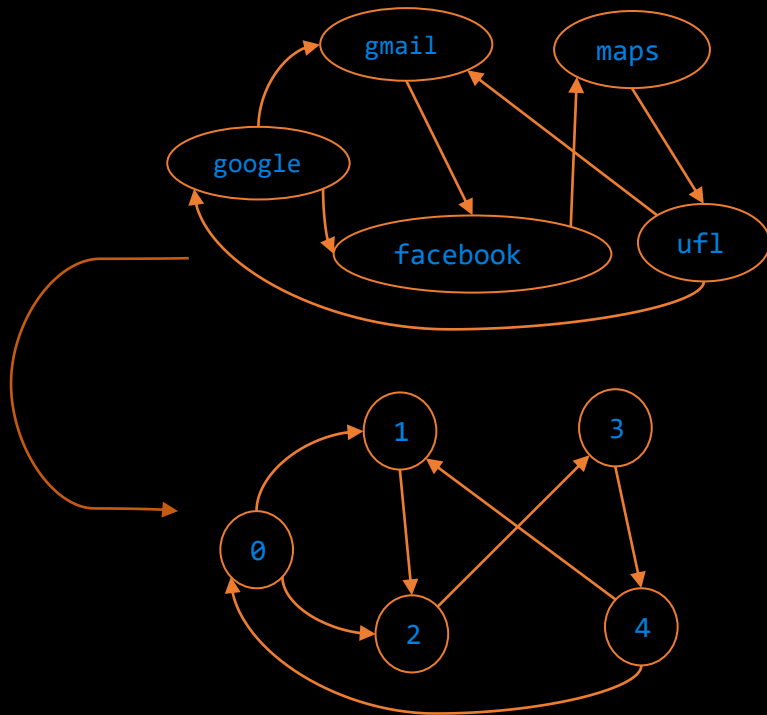
# Graph API

- No common ADT for Graphs
- Graphs were present before Object Oriented Programming
- API must include Graph methods, including their signatures and behaviors
- Defines how Graph client programmers must think.
- An underlying data structure to represent our graphs.
- Our choices can have profound implications on:
  - Runtime
  - Memory usage
  - Difficulty of implementing various graph algorithms



# Common Convention

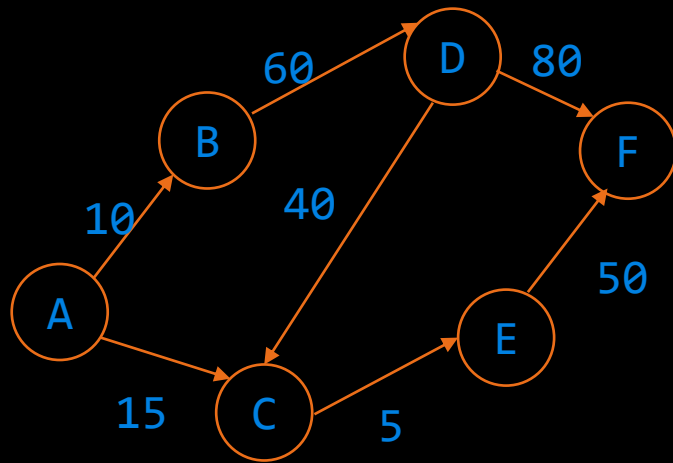
- Map labels to numbers, e.g. If node is called “google.com”, assign it a number, say 0.
- Use a map data structure to achieve this: `map<string, int>`



Label	Graph_Index
google.com	0
gmail.com	1
facebook.com	2
maps.com	3
ufl.edu	4

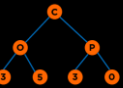
```
std::string vertex = "ufl.edu";  
int count = 0;  
  
std::map<std::string, int> graph;  
  
if(graph.find(vertex) == graph.end())  
    graph[vertex] = count++;
```

# Common Operations

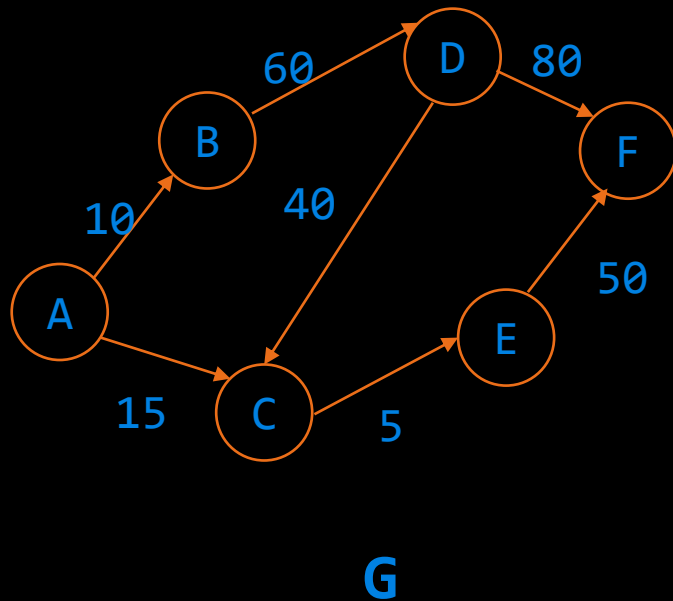


**G**

- Connectedness
- Neighborhood or Adjacency

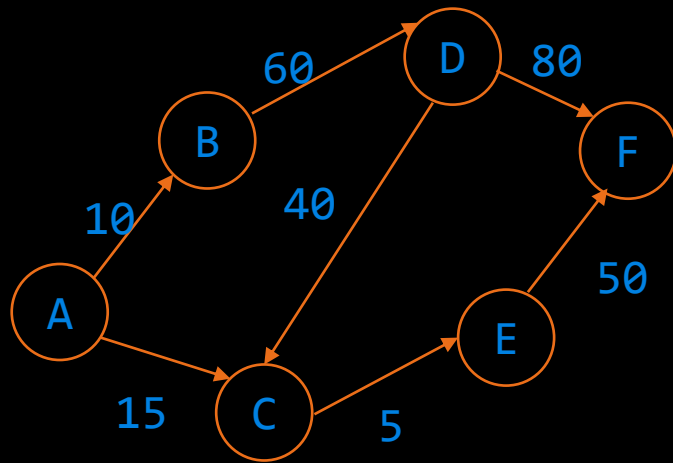


# Common Representations



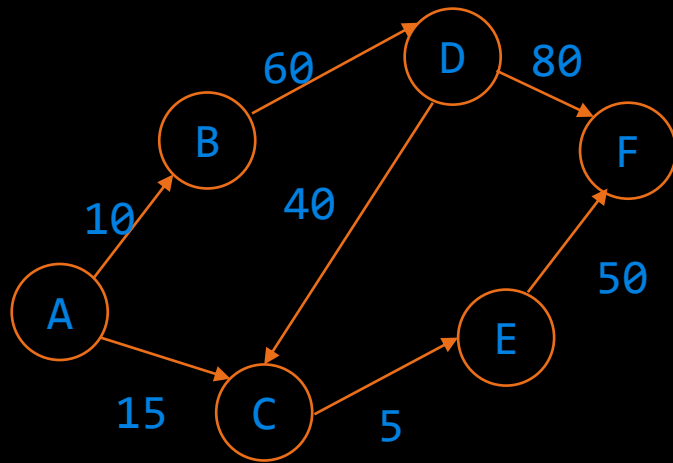
- Edge List
- Adjacency Matrix
- Adjacency List

# Edge List



$G = \{(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)\}$

# Edge List

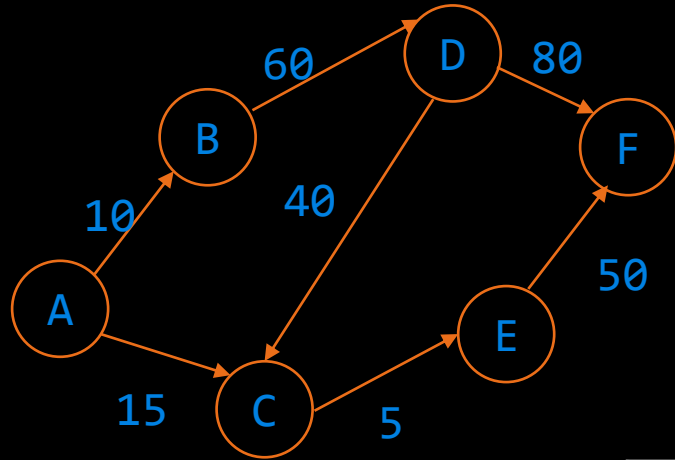


**G**

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

$G = \{(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)\}$

# Edge List



$G = \{(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)\}$

**G**

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

Common Operations:

1. Connectedness

Is A connected to B?

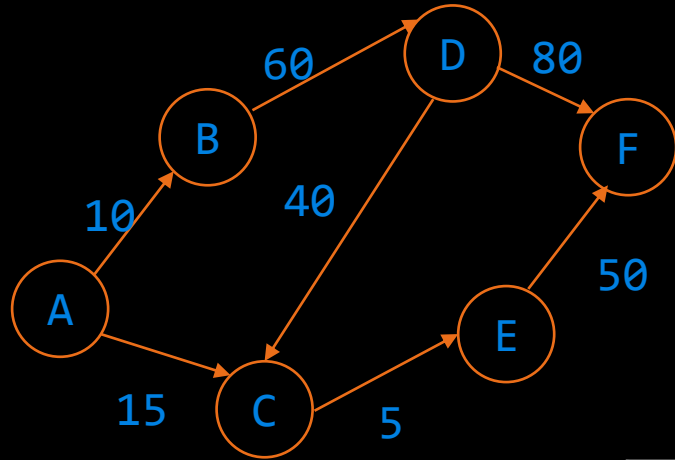
2. Adjacency

What are A's adjacent nodes?

Space: ?



# Edge List



$G = \{(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)\}$

**G**

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

Common Operations:

1. Connectedness

Is A connected to B?

$\sim O(E)$

2. Adjacency

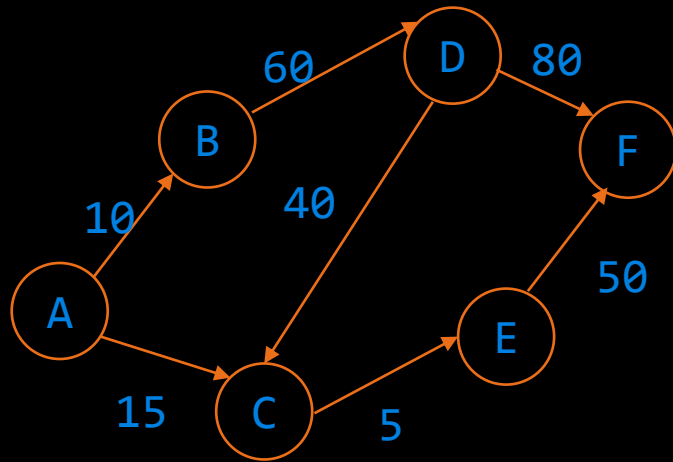
What are A's adjacent nodes?

$\sim O(E)$

$O(|E|) \sim O(|V| * |V|)$

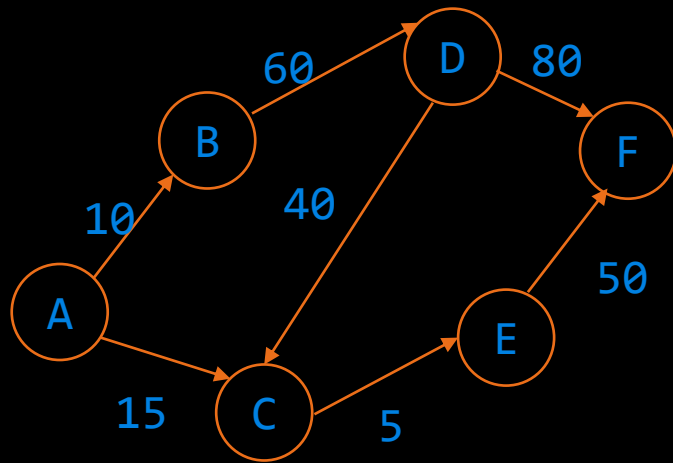
Space:  $O(E)$

# Adjacency Matrix



G

# Adjacency Matrix



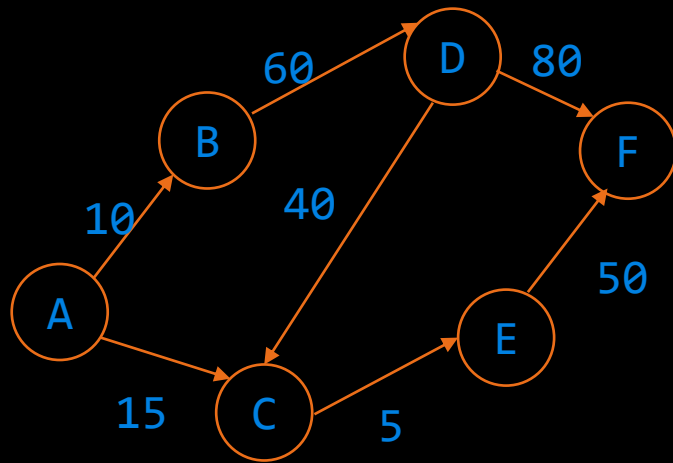
**G**

**Insertion:**

$G[\text{from}][\text{to}] = \text{weight};$  (if there is an edge, “from”  $\rightarrow$  “to”)

$G[\text{from}][\text{to}] = 0;$  (otherwise)

# Adjacency Matrix



**G**

**A**

**B**

**C**

**D**

**E**

**F**

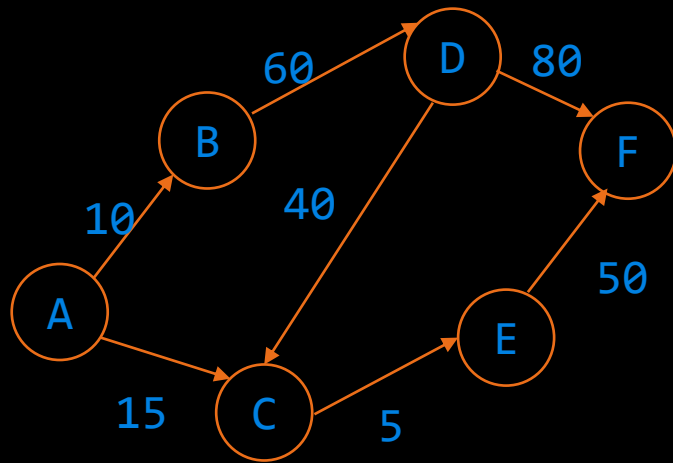
	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

**Insertion:**

$G[\text{from}][\text{to}] = \text{weight};$  (if there is an edge, “from”  $\rightarrow$  “to”)

$G[\text{from}][\text{to}] = 0;$  (otherwise)

# Adjacency Matrix



**G**

**A**

**B**

**C**

**D**

**E**

**F**

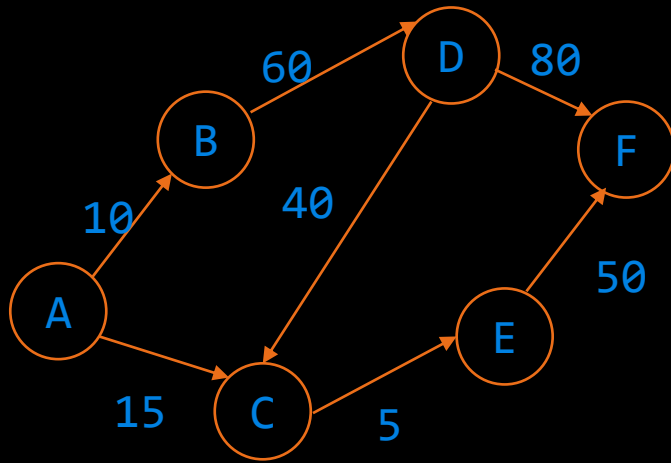
	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

**Insertion:**

$G[\text{from}][\text{to}] = \text{weight};$  (if there is an edge, “from”  $\rightarrow$  “to”)

$G[\text{from}][\text{to}] = 0;$  (otherwise)

# Adjacency Matrix Implementation



Insertion:

$G[\text{from}][\text{to}] = \text{weight}$ ; (if there is an edge, “from” -> “to”)

$G[\text{from}][\text{to}] = 0$ ; (otherwise)

Input

```
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50
```

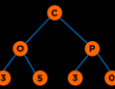
**G**

Map

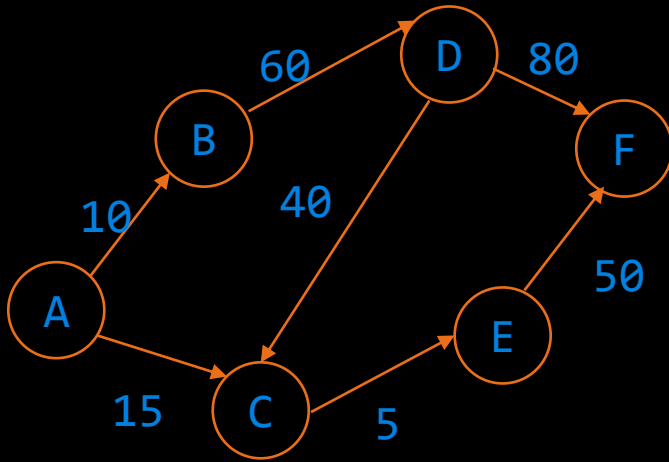
```
A 0
B 1
C 2
D 3
E 4
F 5
```

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

<https://www.onlinegdb.com/Hy8M0CnsS>



# Adjacency Matrix Implementation



Input

```
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50
```

**G**

Map

```
A 0
B 1
C 2
D 3
E 4
F 5
```

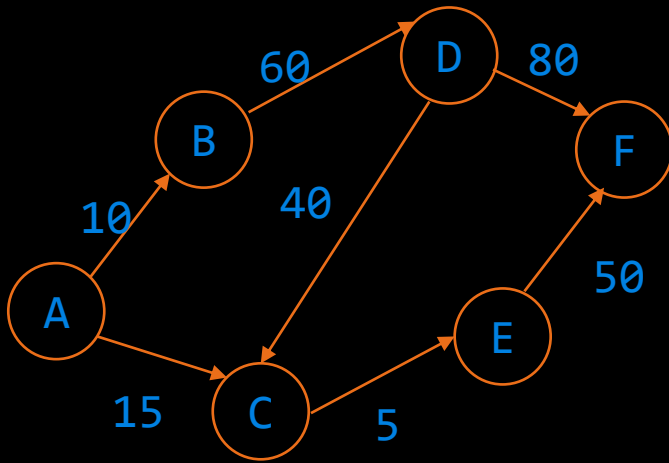
	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Insertion:

```
G[from][to] = weight; (if there is an edge, "from" -> "to")
G[from][to] = 0;      (otherwise)
```

```
01 #include <iostream>
02 #include<map>
03 #define VERTICES 6
04 using namespace std;
05 int main()
06 {
07     int no_lines, wt, j=0;
08     string from, to;
09     int graph [VERTICES][VERTICES] = {0};
10     map<string, int> mapper;
11     cin >> no_lines;
12
13
14
15
16
17
18
19
20
21     return 0;
22 }
```

# Adjacency Matrix Implementation



Input

```

7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50
  
```

**G**

Map

```

A 0
B 1
C 2
D 3
E 4
F 5
  
```

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Insertion:

```

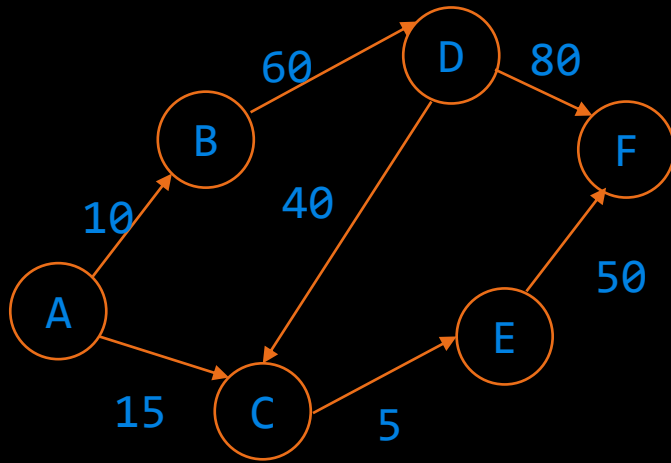
G[from][to] = weight; (if there is an edge, "from" -> "to")
G[from][to] = 0;      (otherwise)
  
```

```

01 #include <iostream>
02 #include<map>
03 #define VERTICES 6
04 using namespace std;
05 int main()
06 {
07     int no_lines, wt, j=0;
08     string from, to;
09     int graph [VERTICES][VERTICES] = {0};
10     map<string, int> mapper;
11     cin >> no_lines;
12     for(int i = 0; i < no_lines; i++)
13     {
14         cin >> from >> to >> wt;
15         if (mapper.find(from) == mapper.end())
16             mapper[from] = j++;
17         if (mapper.find(to) == mapper.end())
18             mapper[to] = j++;
19         graph[mapper[from]][mapper[to]] = wt;
20     }
21     return 0;
22 }
  
```



# Adjacency Matrix



Common Operations:

1. Connectedness

Is A connected to B?

2. Adjacency

What are A's adjacent nodes?

G

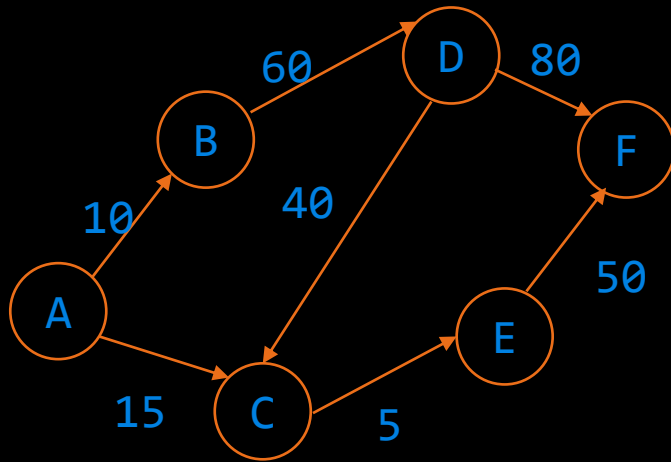
Map

A 0  
B 1  
C 2  
D 3  
E 4  
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Space: ?

# Adjacency Matrix



**G**

Map  
A 0  
B 1  
C 2  
D 3  
E 4  
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Common Operations:

1. Connectedness

Is A connected to B?

$G["A"]["B"] \sim O(1)$

2. Adjacency

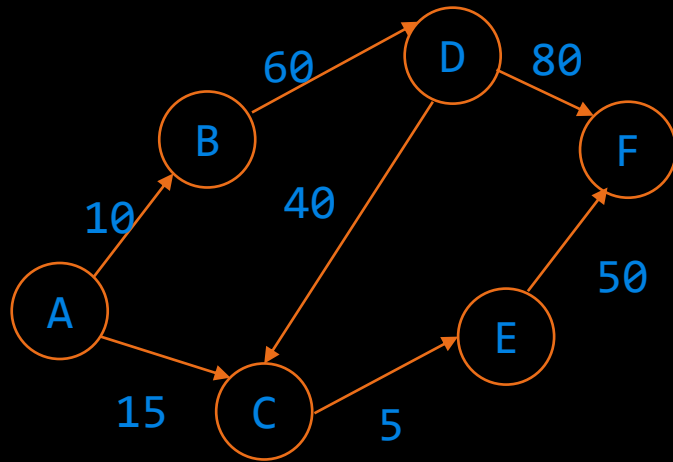
What are A's adjacent nodes?

for each element  $x$  in  $G["A"]$   
if  $x \neq 0$

$\sim O(|V|)$

Space:  $O(|V| * |V|)$

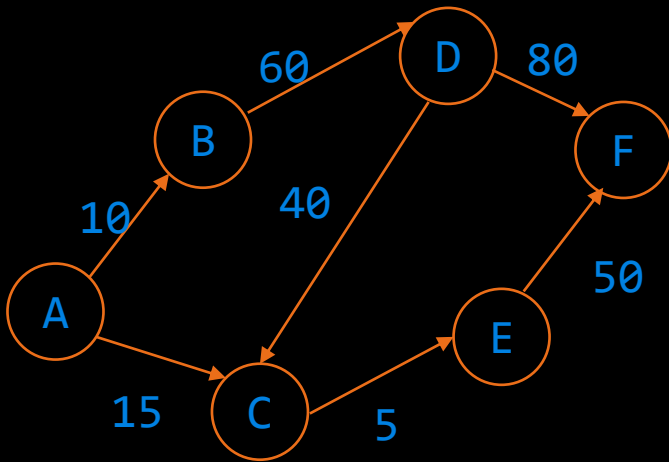
# Adjacency Matrix Problem



G

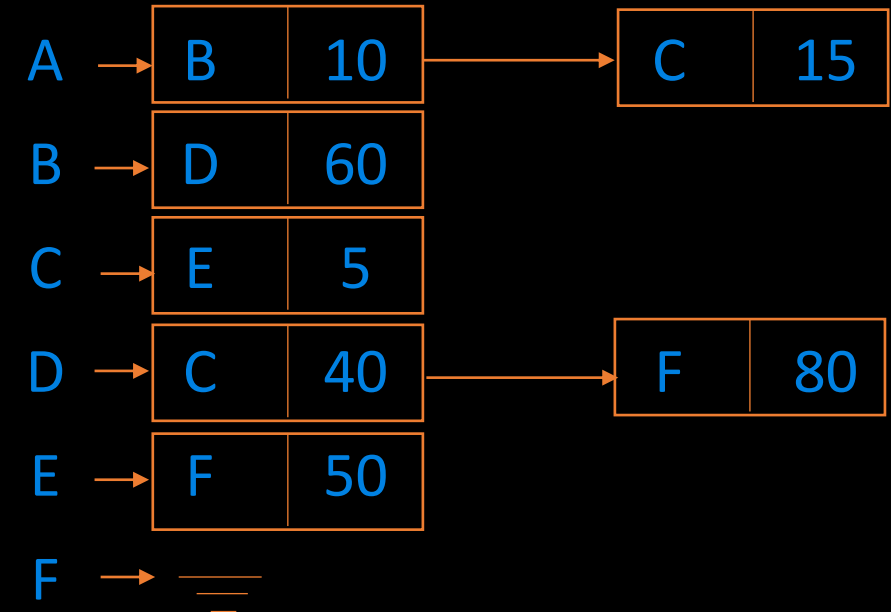
	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

# Adjacency List

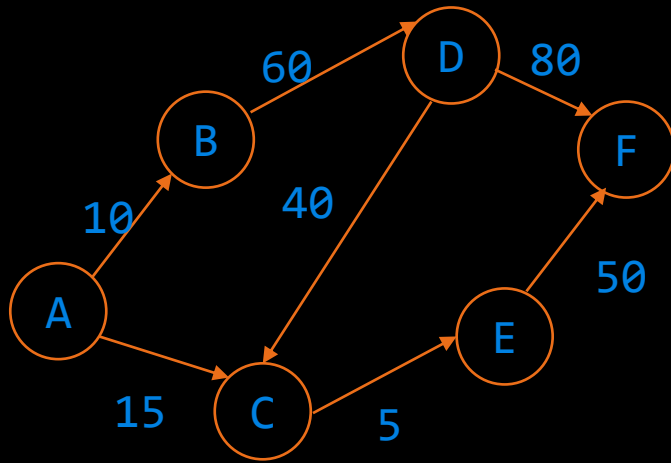


**G**

	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0



# Adjacency List



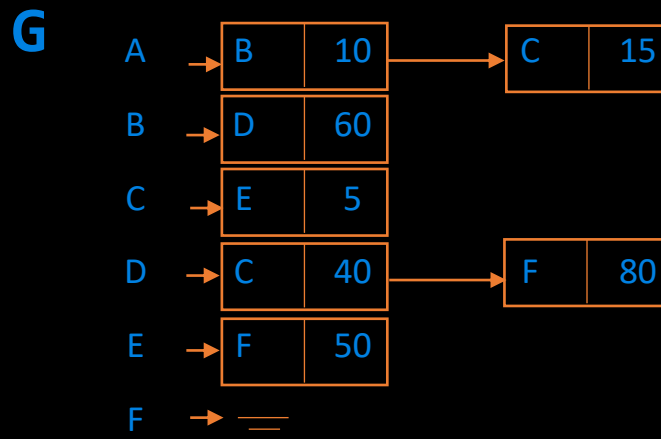
Common Operations:

1. Connectedness

Is A connected to B?

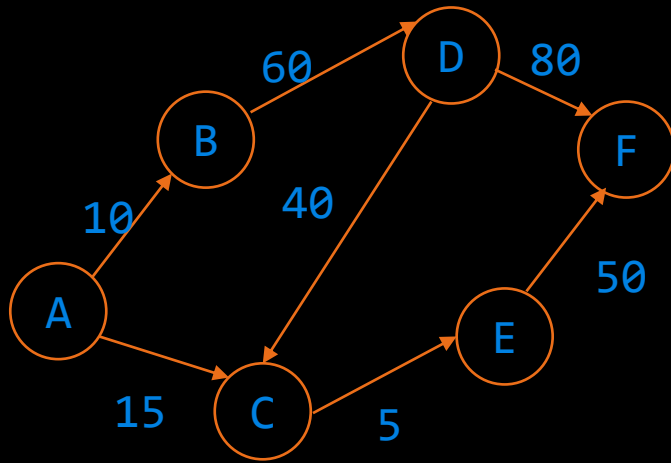
2. Adjacency

What are A's adjacent nodes?

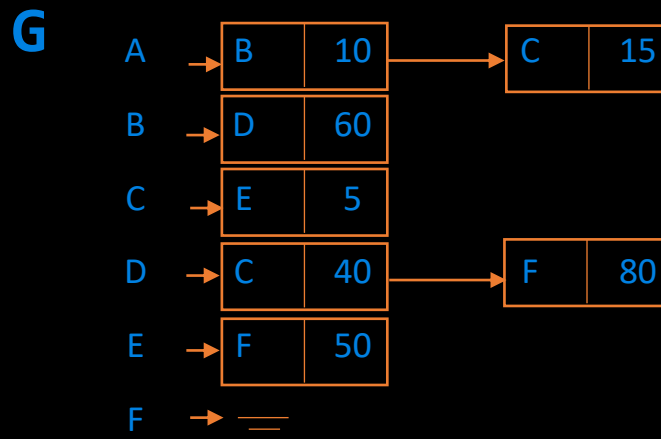


Space: ?

# Adjacency List



**Sparse Graph:**  
Edges  $\sim$  Vertices



Common Operations:

1. Connectedness

Is A connected to B?  
for each element x in G["A"]  
if x != 'B'  
 $\sim O(\text{outdegree}|V|)$

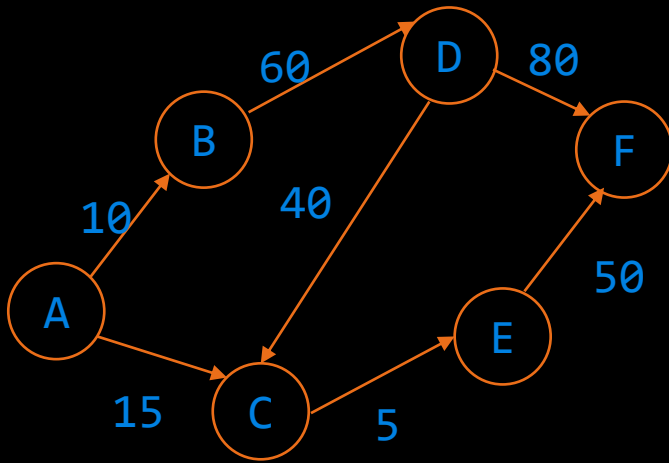
2. Adjacency

What are A's adjacent nodes?

$G["A"] \sim O(\text{outdegree}|V|)$

Space:  $O(|V| + |E|)$

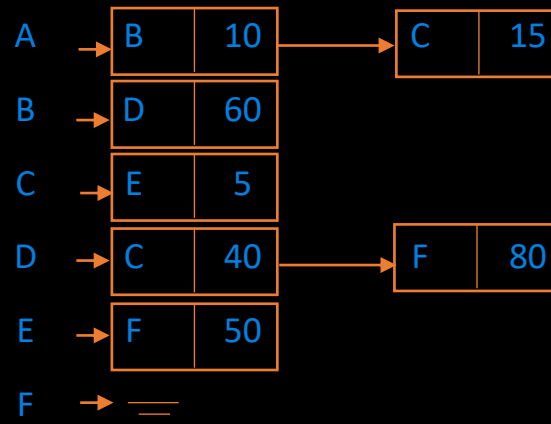
# Adjacency List Implementation



## Input

```
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50
```

G



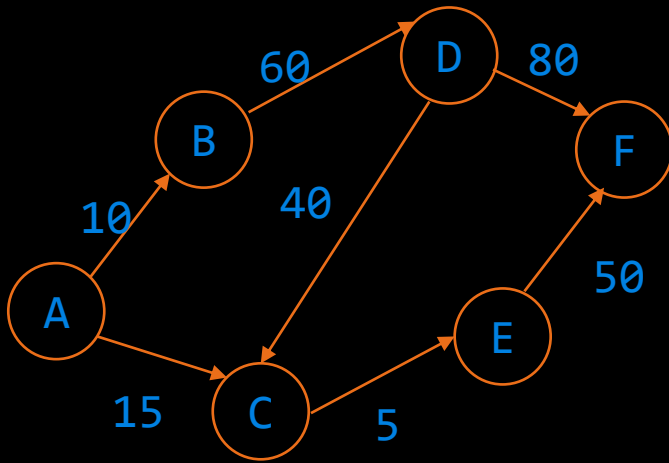
## Insertion:

If to or from vertex not present add vertex

Otherwise add edge at the end of the list

```
01 #include <iostream>
02 #include<map>
03 #include<vector>
04 #include<iterator>
05 using namespace std;
06
07 int main()
08 {
09     int no_lines;
10     string from, to, wt;
11     map<string, vector<pair<string,int>>> graph;
12     cin >> no_lines;
13     for(int i = 0; i < no_lines; i++)
14     {
15
16
17
18
19     }
20 }
```

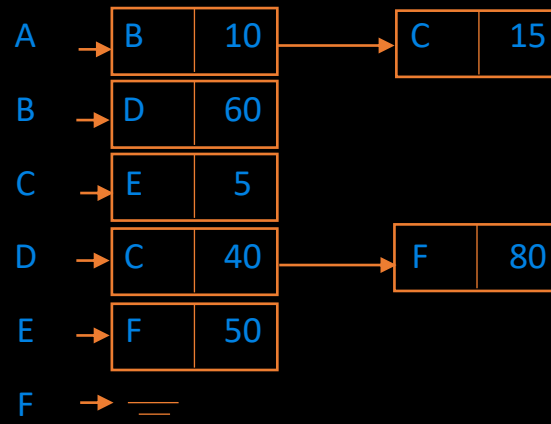
# Adjacency List Implementation



## Input

```
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50
```

G

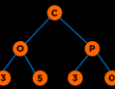


## Insertion:

If to or from vertex not present add vertex

Otherwise add edge at the end of the list

```
01 #include <iostream>
02 #include<map>
03 #include<vector>
04 #include<iterator>
05 using namespace std;
06
07 int main()
08 {
09     int no_lines;
10     string from, to, wt;
11     map<string, vector<pair<string,int>>> graph;
12     cin >> no_lines;
13     for(int i = 0; i < no_lines; i++)
14     {
15         cin >> from >> to >> wt;
16         graph[from].push_back(make_pair(to, stoi(wt)));
17         if (graph.find(to)==graph.end())
18             graph[to] = {};
19     }
20 }
```





# Graph Implementation

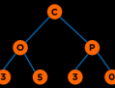
	Edge List	Adjacency Matrix	Adjacency List
Time Complexity: Connectedness	$O(E)$	$O(1)$	$O(\text{outdegree}(V))$
Time Complexity: Adjacency	$O(E)$	$O(V)$	$O(\text{outdegree}(V))$
Space Complexity	$O(E)$	$O(V*V)$	$O(V+E)$

# One Graph API

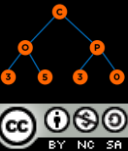
```
class Graph
{
    private:
        //Graph Data Structure

    public:
        Graph();
        Graph(int V); //Creates graph with v vertices
        int V();      //Returns number of vertices
        int E();      //Returns number of edges

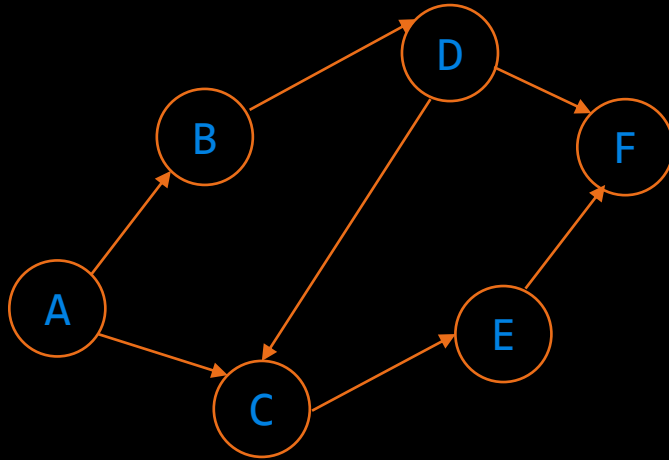
        void insertEdge(int from, int to, int weight);
        bool isEdge(int from, int to);
        int getWeight(int from, int to);
        vector<int> getAdjacent(int vertex);
        void printGraph();
};
```



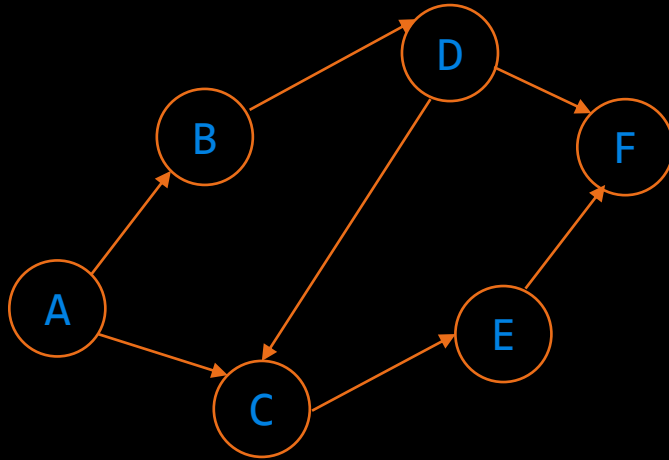
# Graph Traversal



# Breadth First Search



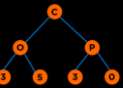
# Breadth First Search



Valid BFS:

A, B, C, D, E, F

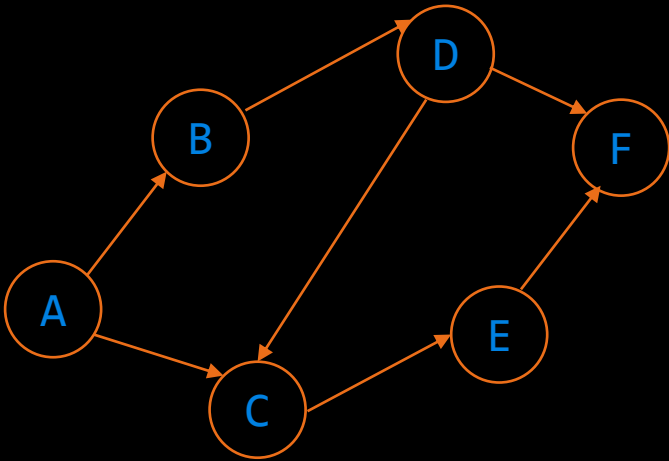
A, C, B, E, D, F



# Breadth First Search

## Algorithm for Breadth-First Search

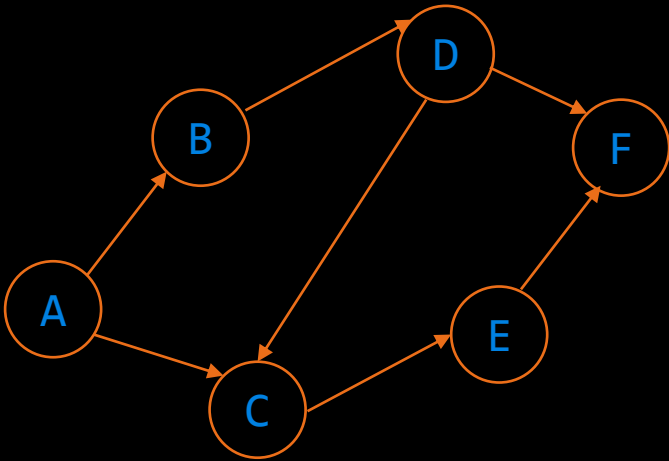
1. Take an arbitrary start vertex, mark it identified, and place it in a queue.
2. while the queue is not empty
3.     Take a vertex,  $u$ , out of the queue and visit  $u$ .
4.     for all vertices,  $v$ , adjacent to this vertex,  $u$
5.         if  $v$  has not been identified or visited
6.             Mark it identified
7.             Insert vertex  $v$  into the queue.
8.     We are now finished visiting  $u$ .



# Breadth First Search

## Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified, and place it in a queue.
2. while the queue is not empty
3.     Take a vertex,  $u$ , out of the queue and visit  $u$ .
4.     for all vertices,  $v$ , adjacent to this vertex,  $u$
5.         if  $v$  has not been identified or visited
6.             Mark it identified
7.             Insert vertex  $v$  into the queue.
8.     We are now finished visiting  $u$ .



```
01 string source = "A";
02 std::set<string> visited;
03 std::queue<string> q;
04
05 visited.insert(source);
06 q.push(source);
07 cout<<"BFS: ";
08
09 while(!q.empty())
10 {
11     string u = q.front();
12     cout << u;
13     q.pop();
14     vector<string> neighbors = graph[u];
15     std::sort(neighbors.begin(), neighbors.begin() + neighbors.size());
16     for(string v: neighbors)
17     {
18         if(visited.count(v) == 0)
19         {
20             visited.insert(v);
21             q.push(v);
22         }
23     }
24 }
```

# Breadth First Search: Alternate way (7.2.2)

## Algorithm for Breadth-First Search

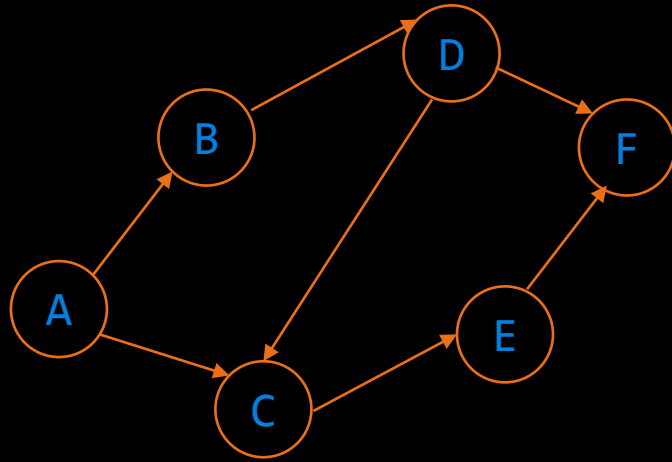
1. Take an arbitrary start vertex, mark it identified, and place it in a queue.
2. while the queue is not empty
3.     Take a vertex,  $u$ , out of the queue and visit  $u$ .
4.     for all vertices,  $v$ , adjacent to this vertex,  $u$
5.         if  $v$  has not been identified or visited
6.             Mark it identified
7.             Insert vertex  $v$  into the queue.
8.     We are now finished visiting  $u$ .

```
// Visited Vertices Alternate
set<string> visited;
visited.insert(source);
if(visited.count(v)==0)
    visited.insert(v);
```

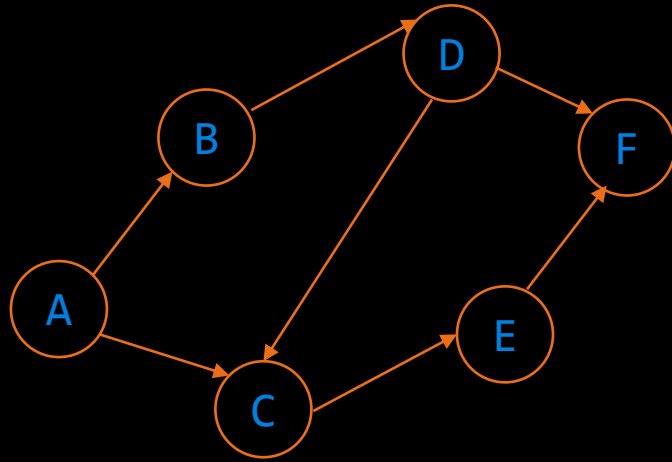
```
01 void bfs(const Graph& graph, int src)
02 {
03     vector<bool> visited(graph.numVertices);
04     queue<int> q;
05
06     visited[src] = true;
07     q.push(src);
08
09     while (!q.empty())
10     {
11         int u = q.front();
12         cout << u << " ";
13         q.pop();
14
15         for (int v : graph.adjList[u])
16         {
17             if (!visited[v])
18             {
19                 visited[v] = true;
20                 q.push(v);
21             }
22         }
23     }
24 }
```



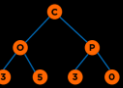
# Depth First Search



# Depth First Search



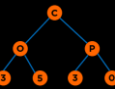
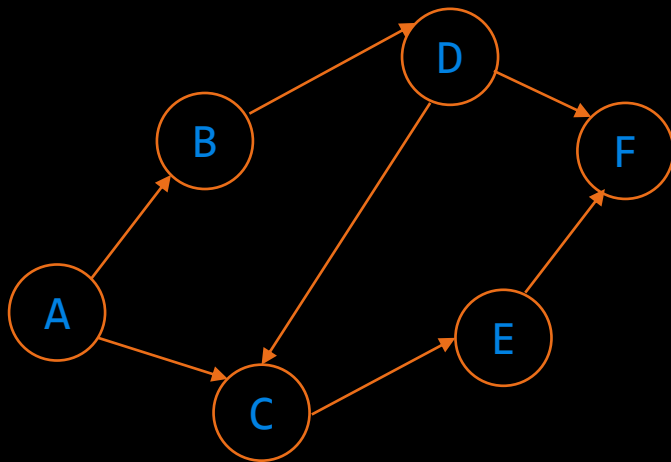
Valid DFS: A, B, D, C, E, F



# Depth First Search

## Algorithm for Depth-First Search

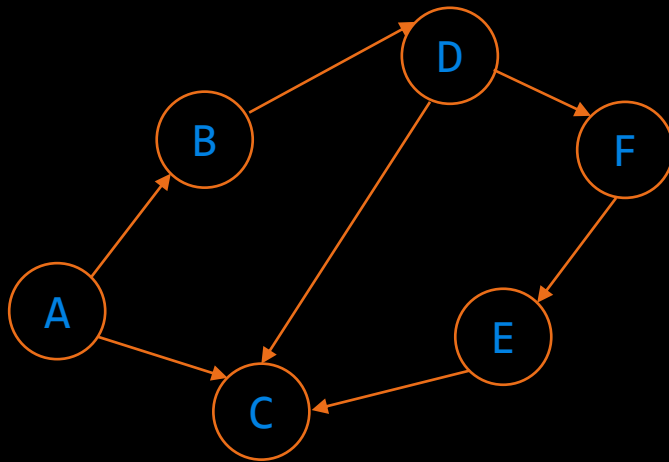
1. Take an arbitrary start vertex, mark it visited, and place it in a stack.
2. while the stack is not empty
3.     the item on top of the stack is  $u$
4.     if there is a vertex,  $v$ , adjacent to this vertex,  $u$ , that has not been visited
5.         Mark  $v$  visited
6.         Push vertex  $v$  onto the top of the stack
7.     else
8.         pop stack



# Depth First Search – Modified BFS

## Algorithm for Depth-First Search

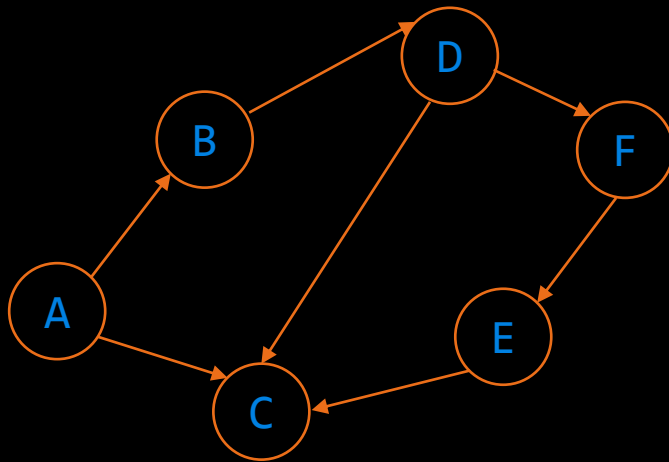
1. Take an arbitrary start vertex, mark it identified, and place it in a stack.
2. while the stack is not empty
3.     Take a vertex,  $u$ , out of the stack and visit  $u$ .
4.     for all vertices,  $v$ , adjacent to this vertex,  $u$
5.         if  $v$  has not been identified or visited
6.             Mark it identified
7.             Insert vertex  $v$  into the stack.
8.     We are now finished visiting  $u$ .



# Depth First Search – Modified BFS

## Algorithm for Depth-First Search

1. Take an arbitrary start vertex, mark it identified, and place it in a stack.
2. while the stack is not empty
3.     Take a vertex,  $u$ , out of the stack and visit  $u$ .
4.     for all vertices,  $v$ , adjacent to this vertex,  $u$
5.         if  $v$  has not been identified or visited
6.             Mark it identified
7.             Insert vertex  $v$  into the stack.
8.     We are now finished visiting  $u$ .



```
01 string source = "A";
02 std::set<string> visited;
03 std::stack<string> s;
04
05 visited.insert(source);
06 s.push(source);
07 cout<<"DFS: ";
08
09 while(!s.empty())
10 {
11     string u = s.top();
12     cout<<u;
13     s.pop();
14     vector<string> neighbors = graph[u];
15     for(string v: neighbors)
16     {
17         if(visited.count(v)==0)
18         {
19             visited.insert(v);
20             s.push(v);
21         }
22     }
23 }
```

# BFS

# vs

# DFS

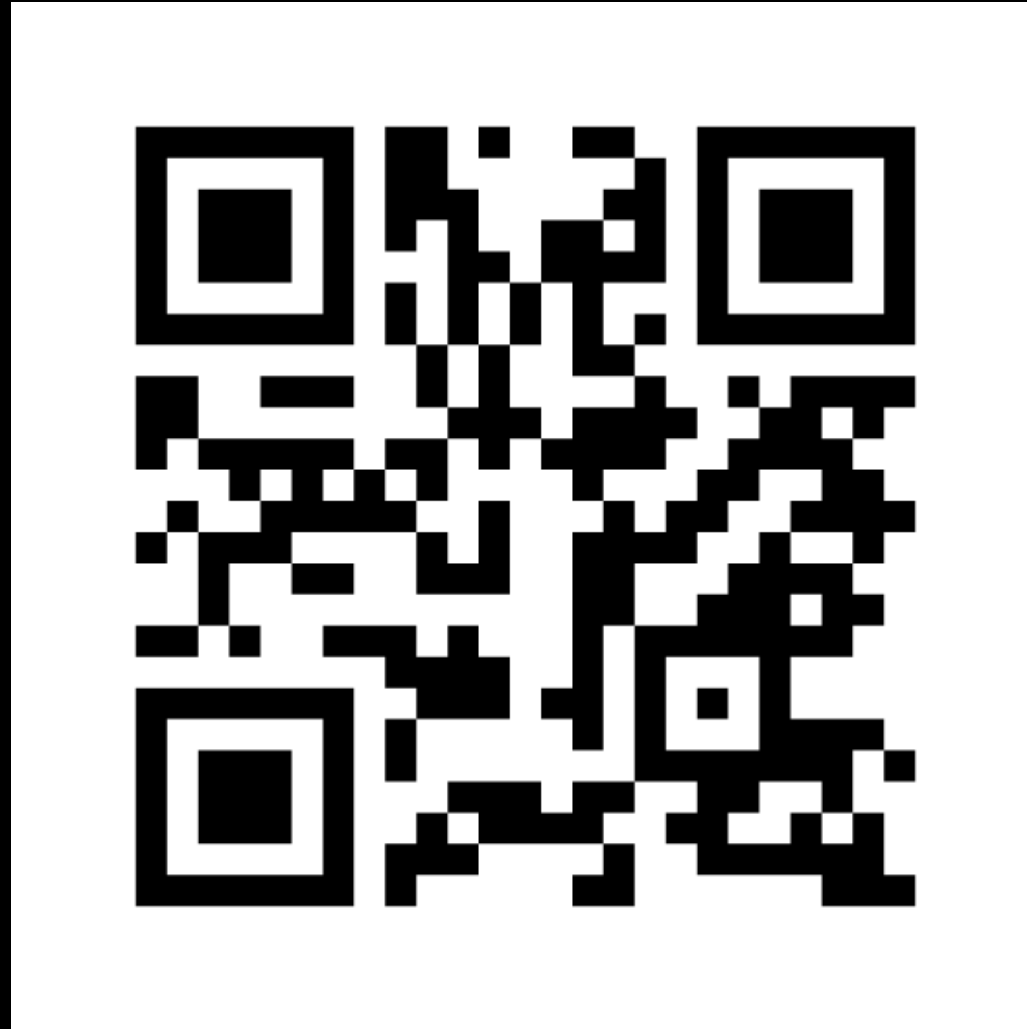
```
01 string source = "A";
02 std::set<string> visited;
03 std::queue<string> q;
04
05 visited.insert(source);
06 q.push(source);
07 cout<<"BFS: ";
08
09 while(!q.empty())
10 {
11     string u = q.front();
12     cout<<u;
13     q.pop();
14     vector<string> neighbors = graph[u];
15     for(string v: neighbors)
16     {
17         if(visited.count(v)==0)
18         {
19             visited.insert(v);
20             q.push(v);
21         }
22     }
23 }
```

```
01 string source = "A";
02 std::set<string> visited;
03 std::stack<string> s;
04
05 visited.insert(source);
06 s.push(source);
07 cout<<"DFS: ";
08
09 while(!s.empty())
10 {
11     string u = s.top();
12     cout<<u;
13     s.pop();
14     vector<string> neighbors = graph[u];
15     for(string v: neighbors)
16     {
17         if(visited.count(v)==0)
18         {
19             visited.insert(v);
20             s.push(v);
21         }
22     }
23 }
```

Theoretical Complexity:  $O(V+E)$

# Mentimeter

**3176 5158**

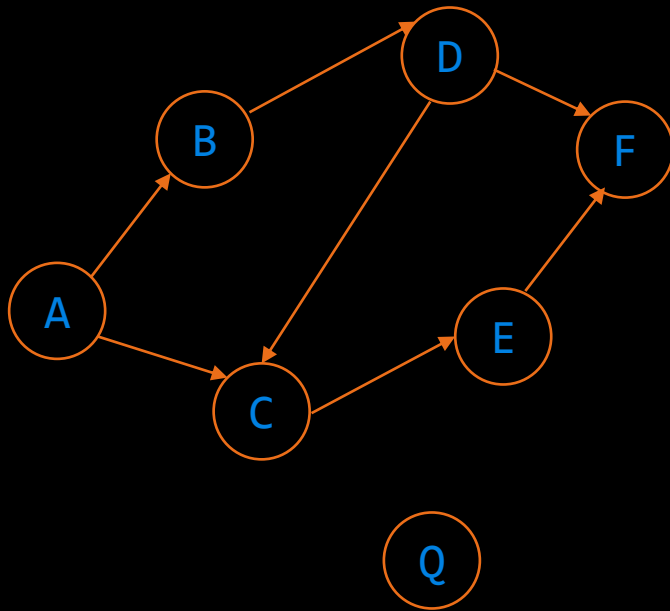


# Questions



# s-t Path

Is there a path between vertices s and t?

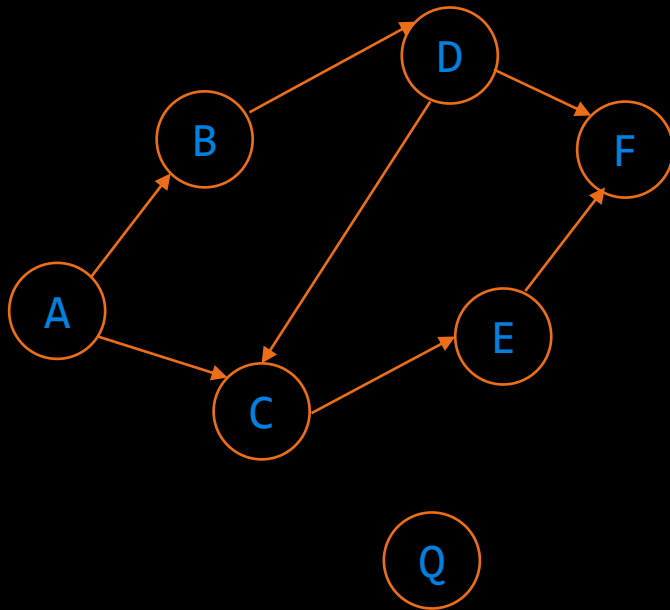


Is there a path between vertices A and C? - Yes

Is there a path between vertices A and Q? - No

# s-t Path

Is there a path between vertices s and t?



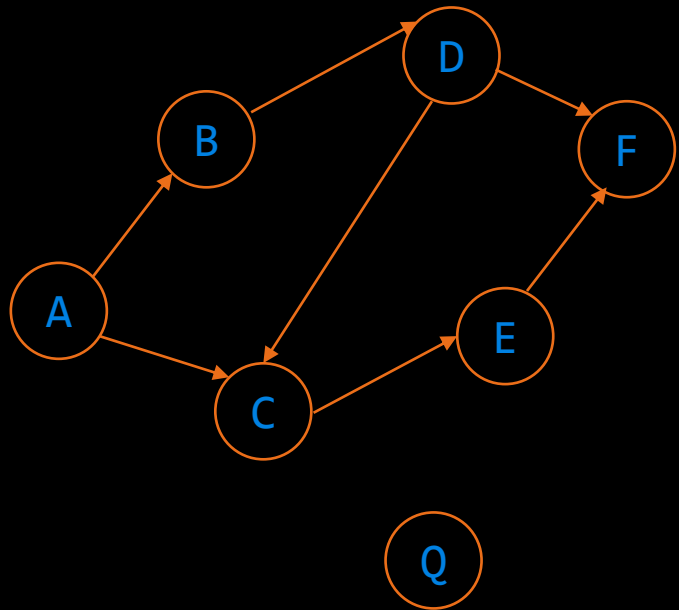
Is there a path between vertices A and C? - Yes

Is there a path between vertices A and Q? - No

## Solution

Perform **DFS** or **BFS** with source “s” and if we encounter “t” in the path/traversal, then return True otherwise False

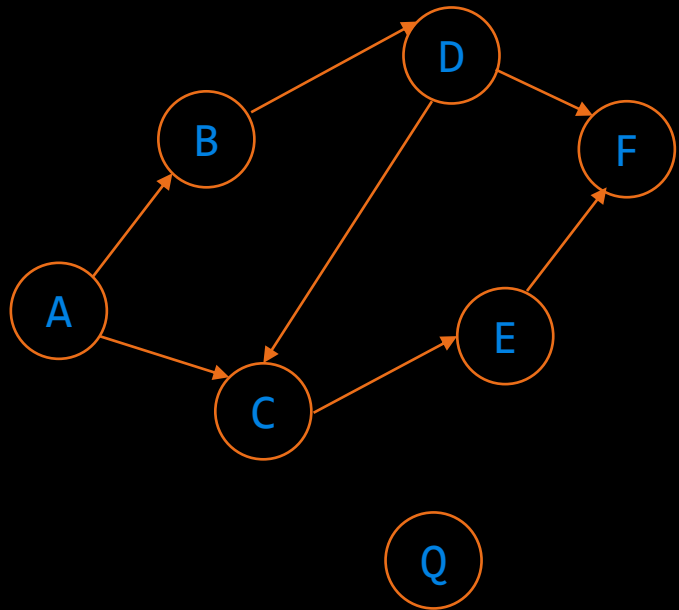
## 7.2.1 DFS to Find Whether a Given Vertex is Reachable (Iterative)



s-t Path

```
1.  bool dfs(const Graph& graph, int src, int dest)
2.  {
3.      set<int> visited;
4.      stack<int> s;
5.      visited.insert(src);
6.      s.push(src);
7.      while(!s.empty())
8.      {
9.          int u = s.top();
10.         s.pop();
11.         for(auto v: graph.adjList[u])
12.         {
13.             if(v == dest)
14.                 return true;
15.             if ((visited.find(v) == visited.end()))
16.             {
17.                 visited.insert(v);
18.                 s.push(v);
19.             }
20.         }
21.     }
22.     return false;
23. }
```

## 7.2.1 DFS to Find Whether a Given Vertex is Reachable (Recursive)



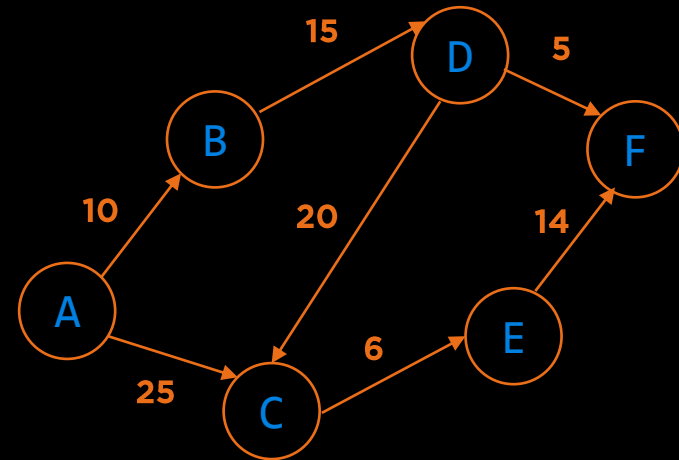
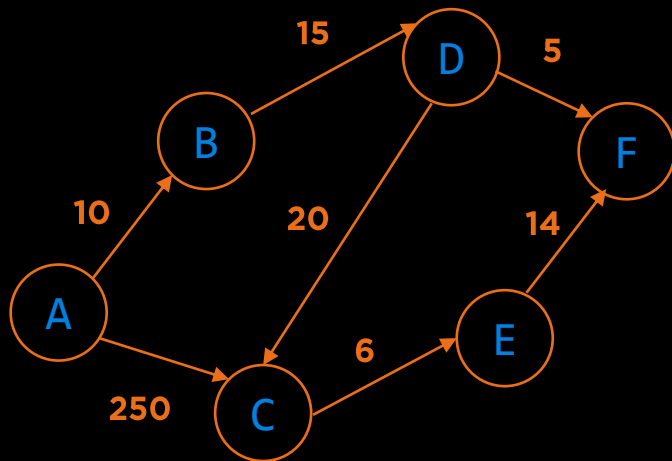
s-t Path: Recursive

```
1. bool dfs_helper(const Graph& graph, int src, int dest, vector<bool>& visited)
2. {
3.     visited[src] = true;
4.
5.     if (src == dest)
6.         return true;
7.
8.     for (int neighbor : graph.adjList[src]) {
9.         if (!visited[neighbor]) {
10.            if (dfs_helper(graph, neighbor, dest, visited))
11.                return true;
12.        }
13.    }
14.    return false;
15. }
16.
17. bool dfs(const Graph& graph, int src, int dest)
18. {
19.     vector<bool> visited(graph.numVertices);
20.     return dfs_helper(graph, src, dest, visited);
21. }
```

# Problem with s-t Path

What if the edges are weighted?

The algorithms do not consider the weights.

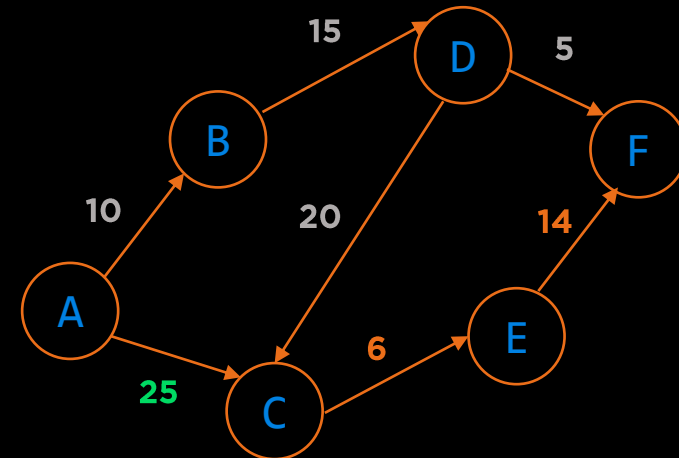


# Problem with s-t Path

What if the edges are weighted?

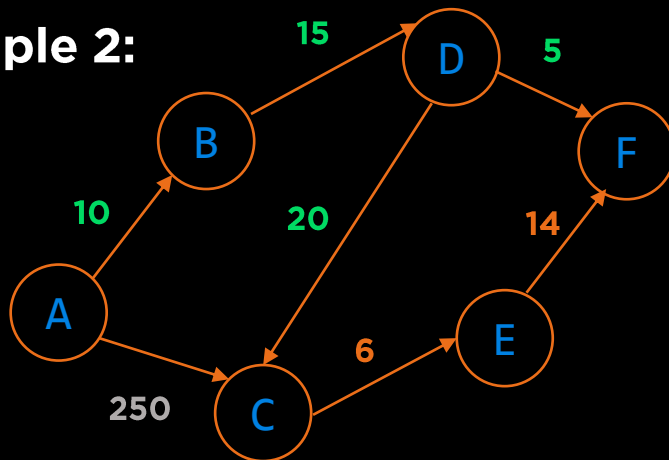
The algorithms do not consider the weights.

Example 1: Path for A to C will be **A-B-D-C** for a **DFS traversal** which will have a total cost of **45** against **25** for the path directly from **A-C**.



Example 1:

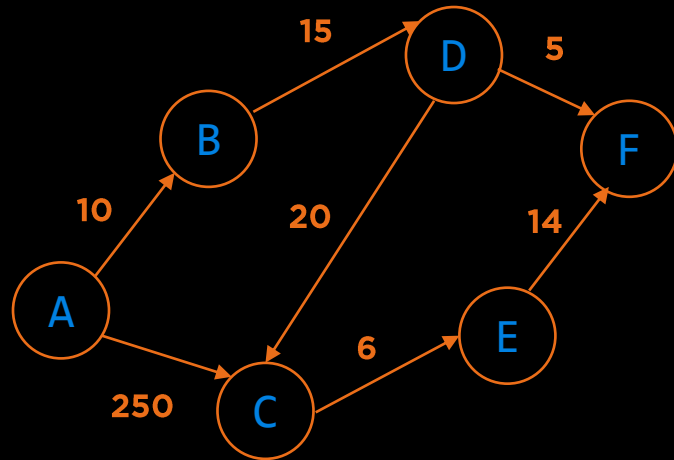
Example 2:



Example 2: Path for A to C will be **A-C** for a **BFS traversal** which might have a total cost of **250** against **45** for the path directly from **A-B-D-C**.

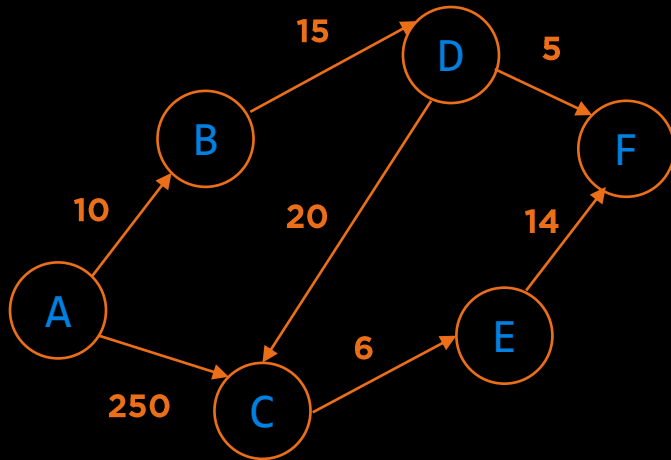
# Shortest Weighted s-t Path

What is the shortest weighted path between vertices **s** and **t**?



# Shortest Weighted s-t Path

What is the shortest weighted path between vertices **s** and **t**?



- **Dijkstra's Algorithm**
  - Single Source: Path to all vertices
  - Directed Graphs
  - No negative weights allowed
  - No negative weight cycles allowed
- **Bellman Ford**
  - Single Source: Path to all vertices
  - Negative Weights allowed
  - No negative weight cycles allowed
- **Floyd-Warshall**
  - All pair shortest paths
- **A\* Search**

