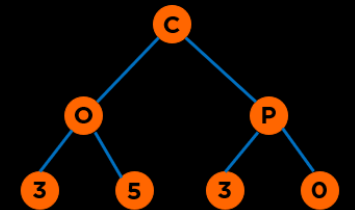


# Final Exam Review



# Categories of Data Structures

**Linear Ordered**

**Lists**

**Stacks**

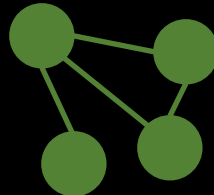
**Queues**



**Non-linear Ordered**

**Trees**

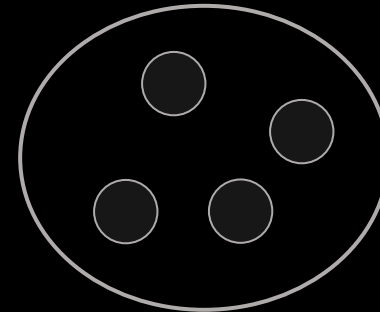
**Graphs**



**Not Ordered**

**Sets**

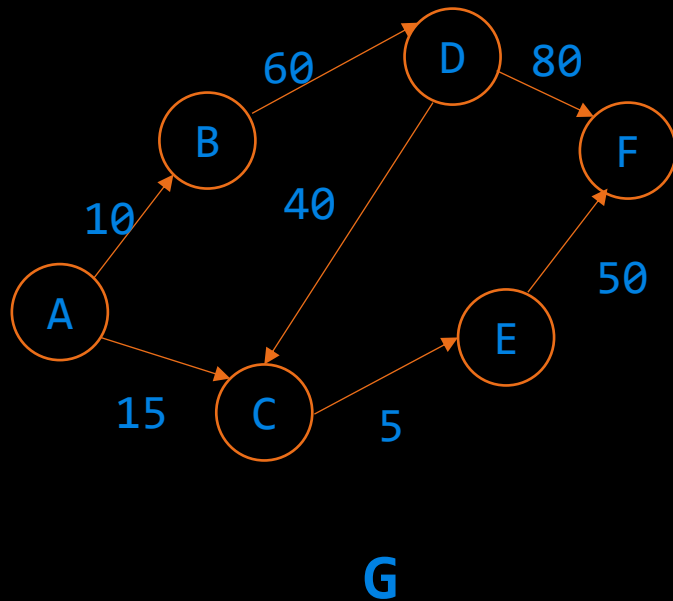
**Tables/Maps**



# Announcements

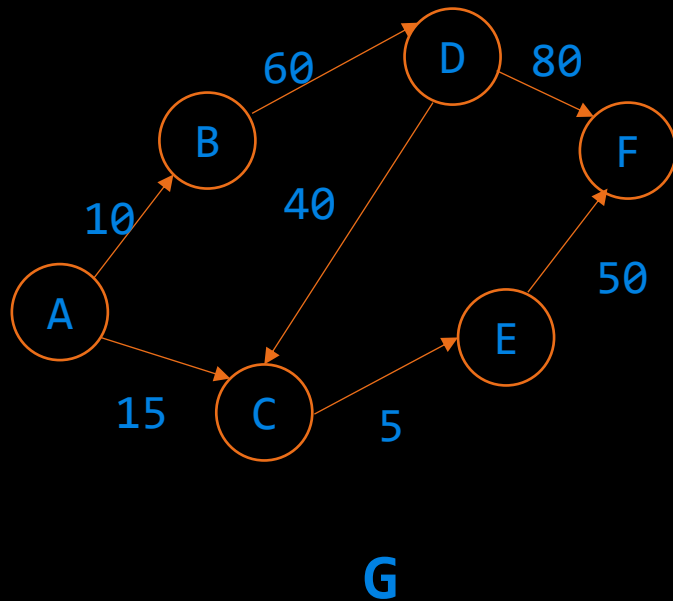
- You must take the exam between Tuesday, April 15, 7:30 am to 11:30 am EST. This means you must start by 9:30 am EST or else you will lose time.
- The exam will be over Honorlock and you are allowed one double sided handwritten sheet of notes.
- The exam duration is 2 hours.
- Exam 2 Topics and Expectations Guide: [Link](#)

# Common Representations



- Edge List
- Adjacency Matrix
- Adjacency List

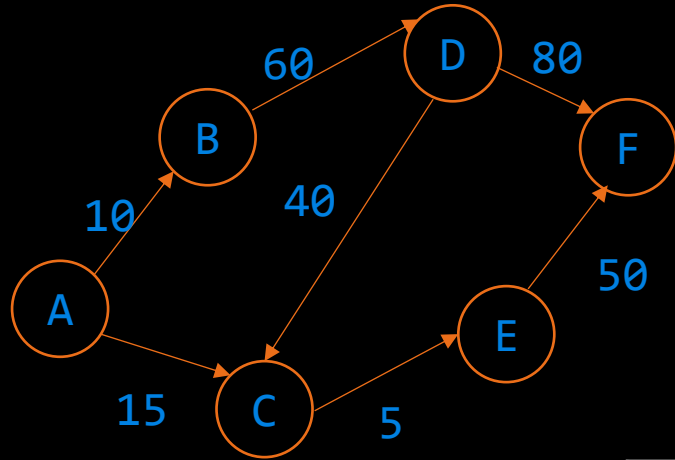
# Edge List



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?

$\sim O(E)$

2. Adjacency

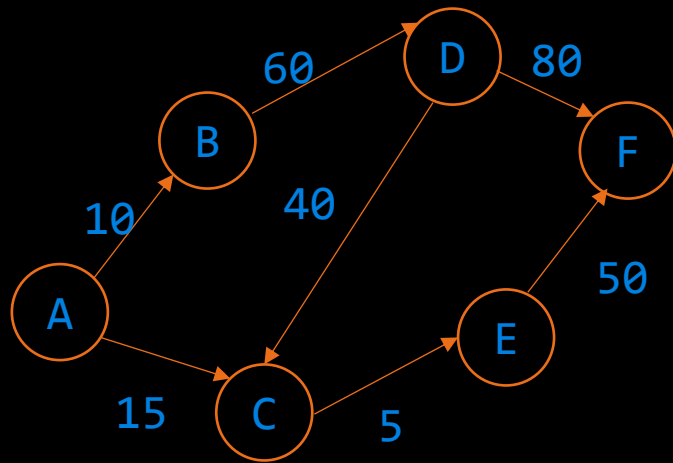
What are A's adjacent nodes?

$\sim O(E)$

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

Space:  $O(E)$

# Adjacency Matrix



**G**

**A**

**B**

**C**

**D**

**E**

**F**

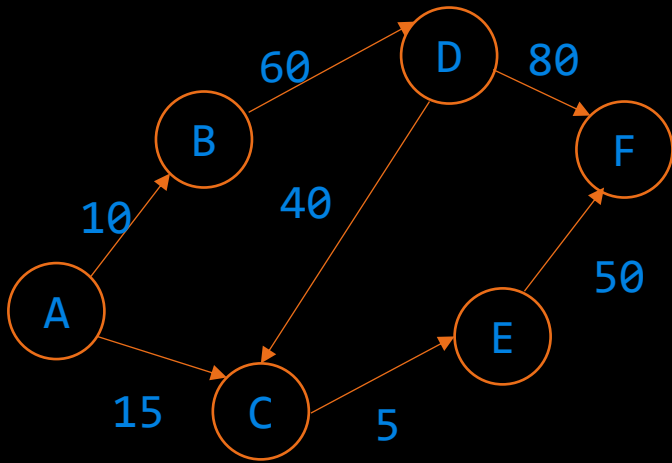
	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

**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



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

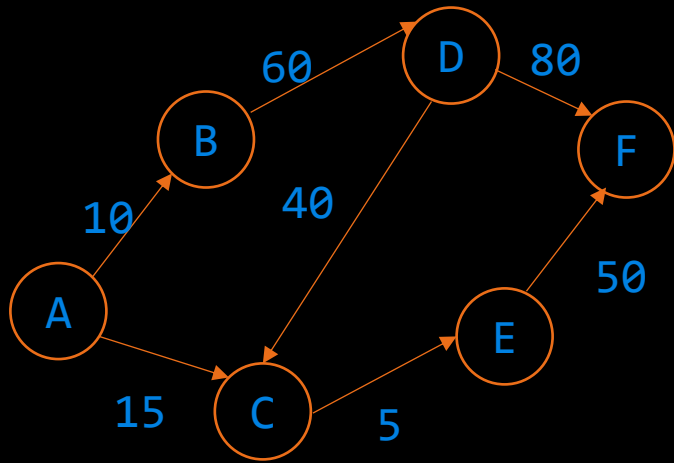
Insertion:

```
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



**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

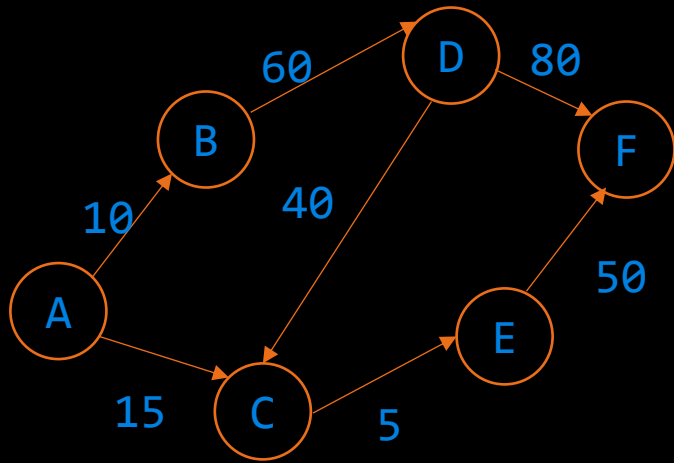
What are A's adjacent nodes?

for each element x in G["A"]  
if x != 0

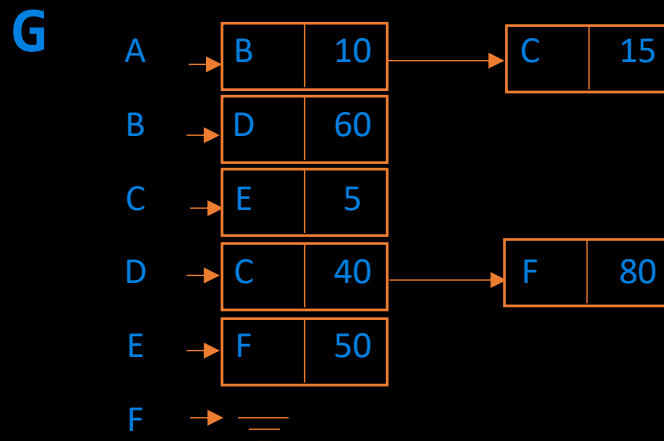
$\sim O(|V|)$

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

# 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'**  
**~ O(outdegree|V|)**

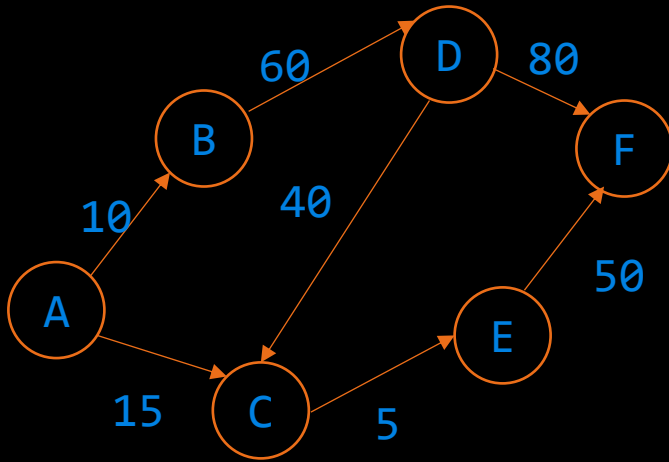
2. Adjacency

What are A's adjacent nodes?

**G["A"] ~ O(outdegree|V|)**

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

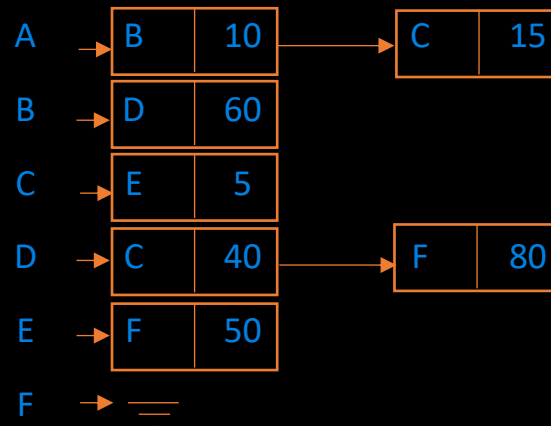
# Adjacency List Implementation



## 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         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	$O(E)$	$O(V*V)$	$O(V+E)$

# Alt Text for the Graph on Next Slide

Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: B, E, F

B: A, C, F, H

C: B, D, F, G, H, I

D: C, E, F, G

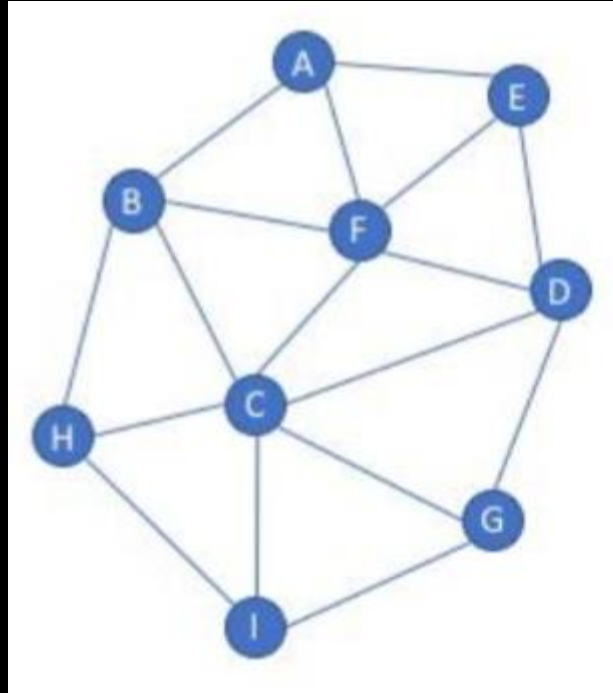
E: A, D, F

F: A, B, C, D, E

G: C, D, I

H: B, C, I

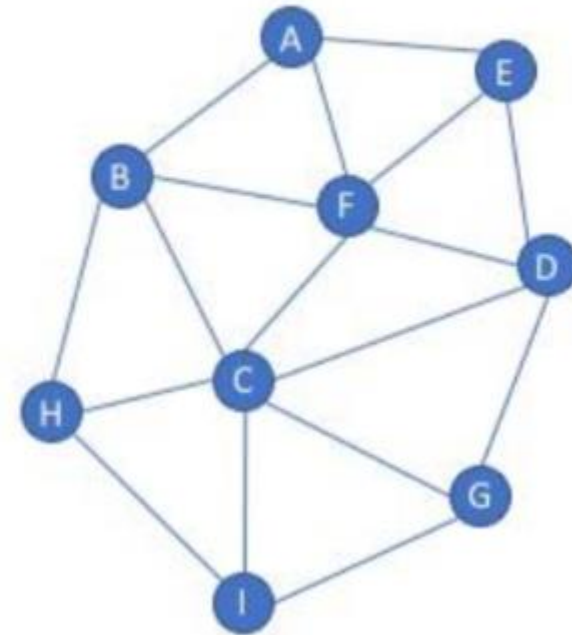
I: C, G, H



# Graph - BFS

- Which of the following are valid breadth first search traversals for this graph?

- a) AFBEDCHGI
- b) ICHGBFDAE
- c) DCFEGHIBA
- d) EAFDBHCIG
- e) FAEDCBGHIH



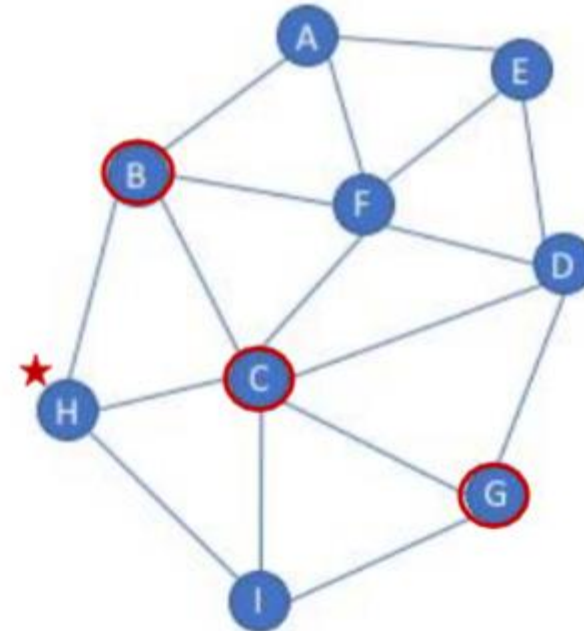
# Graph - BFS

- Which of the following are valid breadth first search traversals for this graph?

- a) AFBEDCHGI
- b) ICHGBFDAE
- c) DCFEGHIBA
- d) EAFDBHCIG
- e) FAEDCBGHI

All the options except for d  
Why not d?

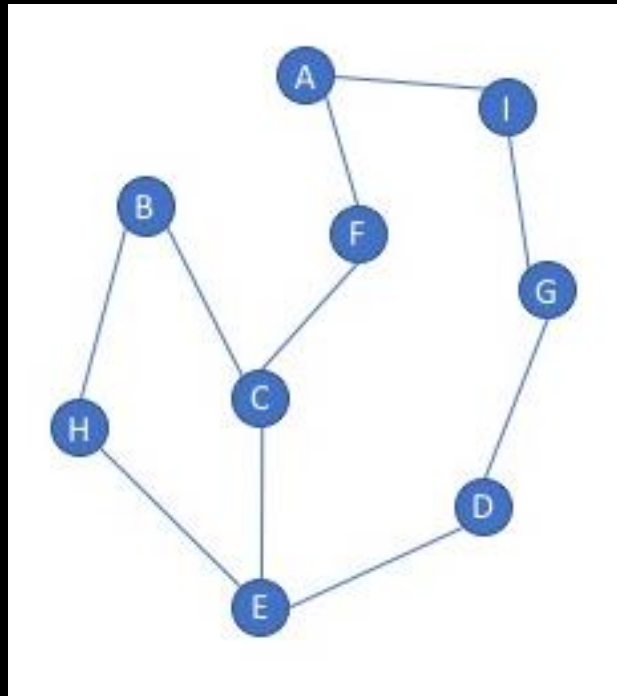
\*\* H is visited before C and G



# Alt Text for the Graph on Next Slide

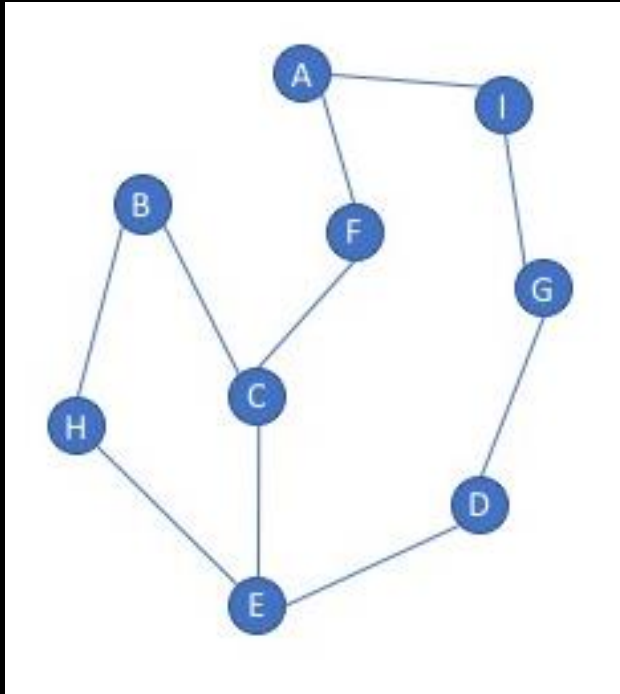
Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: F, I  
B: C, H  
C: B, E, F  
D: E, G  
E: C, D, H  
F: A, C  
G: D, I  
H: B, E  
I: A, G



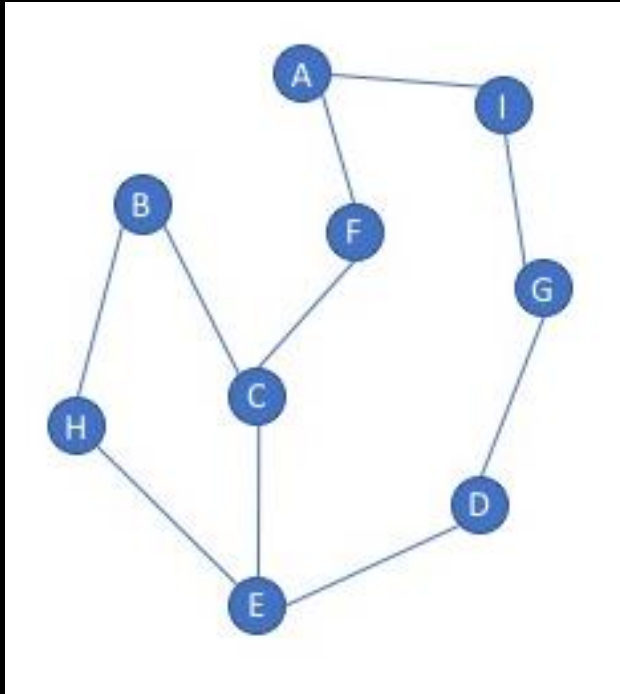


# Valid DFS: Which DFS are valid?



- HECBDGIAF
- CEHBDGIAF
- AFCEHBIGD
- DECBHFAIG

# Valid DFS: Which DFS are valid?



- HECBDGIAF
- **CEHBDGIAF**
- AFCEHBIGD
- **DECBHFAIG**

# Applied Traversal

A connected component of a graph is a collection of vertices in which any two vertices in a component have a path between them. Given an unweighted and undirected graph represented as an adjacency list, write a function using pseudocode or C++ code which will **return the number of vertices in the largest component of the graph**. You do not need to write main method for reading and creating a graph. Assume that the graph is passed into your function which has the following signature:

```
unsigned int largest_component(unordered_map<int, vector<int>>& adjListGraph){  
    // code here  
}
```

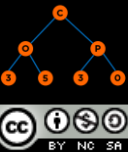
Example input:

```
0: 2, 3  
1: 4  
2: 0, 6  
3: 0  
4: 1, 5  
5: 4  
6: 2  
7:
```

Example output: 4

Explanation: There are three connected components: (0,2,3,6); (1,4,5); and (7). Of these, the largest has 4 vertices.

Also, state the Big O complexity of the solution in the worst case.



# Applied Traversal

```
#include <unordered_map>
#include <vector>
#include <unordered_set>
using namespace std;

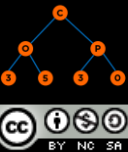
unsigned int dfs(int node, unordered_map<int, vector<int>>& adjListGraph, unordered_set<int>& visited) {
    visited.insert(node);
    unsigned int size = 1; // Start with the current node

    for (int neighbor : adjListGraph[node]) {
        if (visited.find(neighbor) == visited.end()) {
            size += dfs(neighbor, adjListGraph, visited);
        }
    }
    return size;
}

unsigned int largest_component(unordered_map<int, vector<int>>& adjListGraph) {
    unordered_set<int> visited;
    unsigned int maxComponentSize = 0;

    for (const auto entry : adjListGraph) {
        int node = entry.first;
        if (visited.find(node) == visited.end()) {
            unsigned int componentSize = dfs(node, adjListGraph, visited);
            if (componentSize > maxComponentSize) {
                maxComponentSize = componentSize;
            }
        }
    }

    return maxComponentSize;
}
```



# BFS

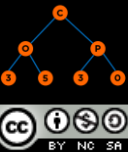
vs

# 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:  $O(V+E)$



# BFS Pseudocode

- Write pseudocode/code for implementing the **Breadth First Search Algorithm** of a graph,  $G$  that takes a source vertex  $S$  as input. (8).
- Also, state the Big  $O$  complexity of the traversal in the worst case (2).

# Graph Algorithm Mix n Match

- Finds the shortest paths in a weighted graph
- Find the minimum cost connected network
- Scheduling algorithm, list steps in a process
- Finds the shortest path in an unweighted graph

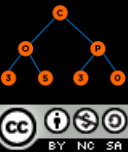
Prim's or Kruskals

BFS

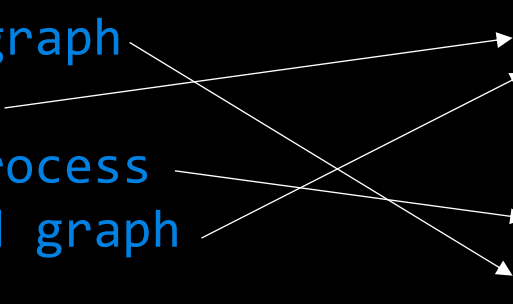
DFS

Topological Sort

Dijkstra's Algorithm



# Graph Algorithm Mix n Match

- Finds the shortest paths in a weighted graph
  - Find the minimum cost connected network
  - Scheduling algorithm, list steps in a process
  - Finds the shortest path in an unweighted graph
- Prim's or Kruskals  
BFS  
DFS  
Topological Sort  
Dijkstra's Algorithm
- 
- The diagram shows four arrows connecting the list items to the algorithm names on the right. The first arrow connects 'Finds the shortest paths in a weighted graph' to 'Dijkstra's Algorithm'. The second arrow connects 'Find the minimum cost connected network' to 'Prim's or Kruskals'. The third arrow connects 'Scheduling algorithm, list steps in a process' to 'Topological Sort'. The fourth arrow connects 'Finds the shortest path in an unweighted graph' to 'BFS'.



# Alt Text for the Graph on Next Slide

Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: -

B: C, E

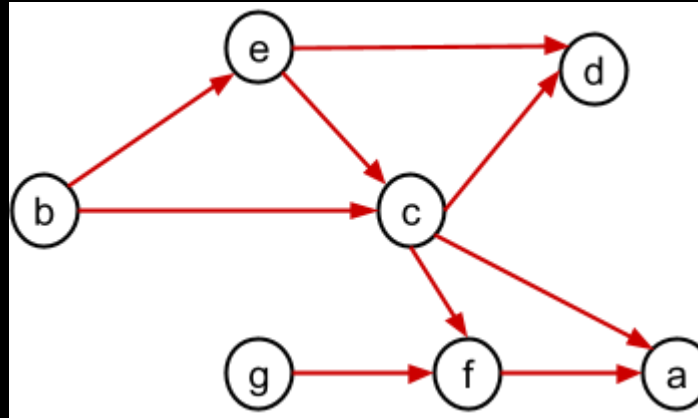
C: A, D, F

D: -

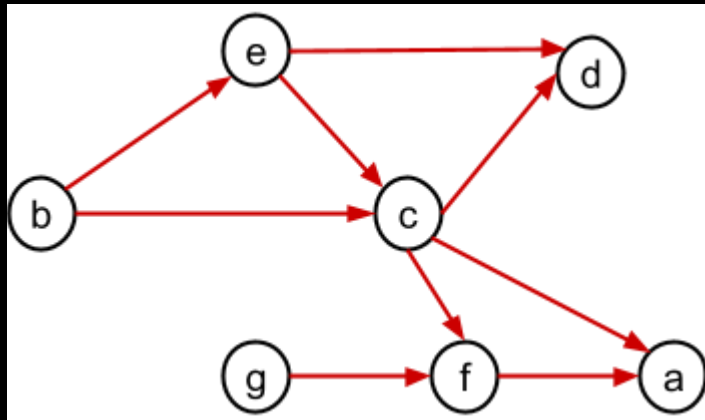
E: C, D

F: A

G: F

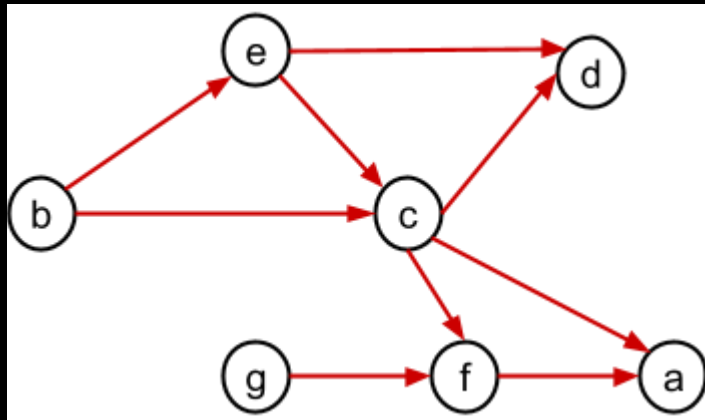


**Which of the choices below represent a valid topological sort ordering of this graph?**



- b, e, c, g, f, a, d
- b, a, c, g, f, e, d
- b, g, f, c, e, a, d
- b, e, c, g, a, f, d
- b, g, e, c, d, f, a
- b, f, c, g, a, e, d

Which of the choices below represent a valid topological sort ordering of this graph?

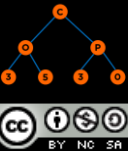


- b, e, c, g, f, a, d
- b, a, c, g, f, e, d
- b, g, f, c, e, a, d
- b, e, c, g, a, f, d
- b, g, e, c, d, f, a
- b, f, c, g, a, e, d

# What does this code do?

```
#include <set>
#include <stack>
using namespace std;

bool doSomething(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end())) {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```



# What does this code do?

```
#include <set>
#include <stack>
using namespace std;

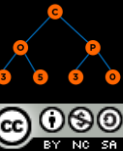
bool doSomething(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end())) {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```

Returns whether a given vertex is reachable from another vertex using DFS

# Scenario

A county government maintains a network of roads. The county government has tabulated the cost of maintaining each road. They need to minimize the cost of road maintenance but ensure that all places in the county are accessible.

Which graph algorithm that we discussed in class could they use to solve this problem? What are the vertices, what are the edges, what are the edge values?



# Scenario

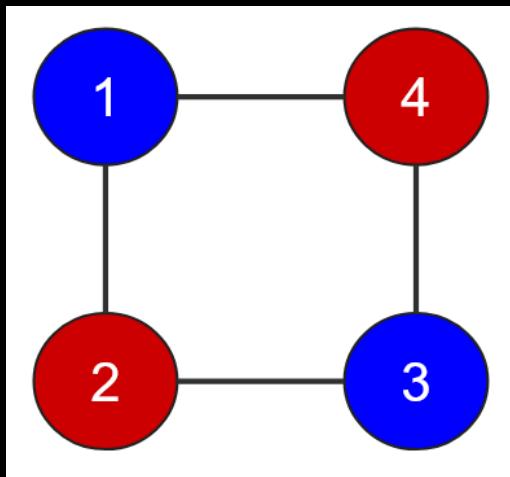
A county government maintains a network of roads. The county government has tabulated the cost of maintaining each road. They need to minimize the cost of road maintenance but ensure that all places in the county are accessible.

Which graph algorithm that we discussed in class could they use to solve this problem? What are the vertices, what are the edges, what are the edge values?

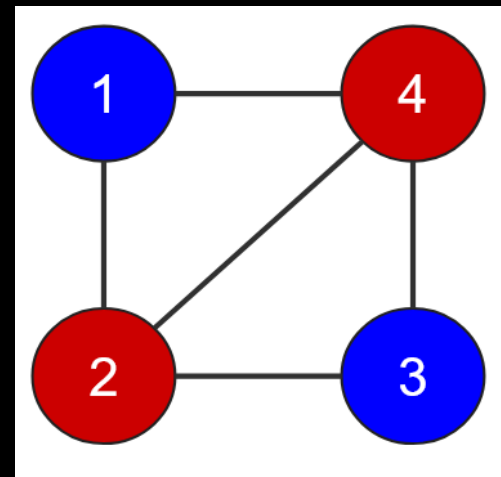
- Prim's or Kruskal's algorithm for minimum spanning tree.
- Roads are edges.
- Ends of roads are vertices.
- Edge weights are cost for maintaining roads.

# Scenario

A **bipartite graph** is known formally as an undirected graph whose nodes can be grouped into two sets such that no edge connects a node to another node in its own set (that is, every edge connects a node from one set to another). If the following graphs are colored using two colors denoting different sets, are these graphs bipartite?



Graph 1

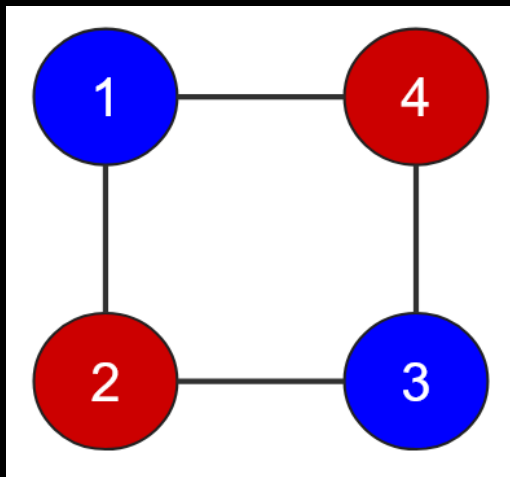


Graph 2



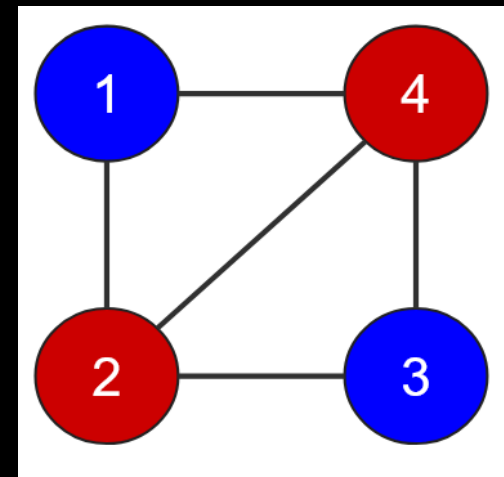
# Scenario

A **bipartite graph** is known formally as an undirected graph whose nodes can be grouped into two sets such that no edge connects a node to another node in its own set (that is, every edge connects a node from one set to another). If the following graphs are colored using two colors denoting different sets, are these graphs bipartite?



Graph 1

Bipartite



Graph 2

Non-Bipartite (Edge 2-4) in the same set

# Alt Text for the Graph on Next Slide

Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

A: <B, 4>, <E, 2>, <F, 9>

B: <A, 4>, <C, 11>, <F, 1>, <H, 12>

C: <B, 11>, <D, 7>, <F, 10>, <G, 15>, <H, 14>, <I, 17>

D: <C, 5>, <E, 7>, <F, 16>, <G, 3>

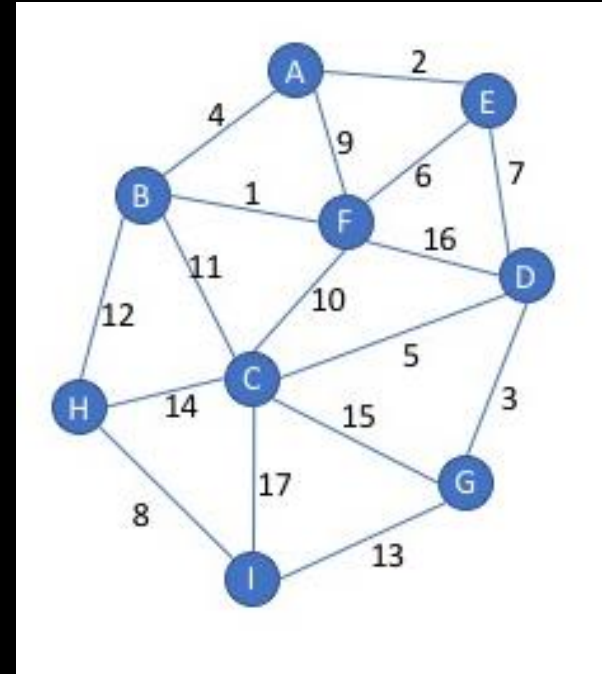
E: <A, 2>, <D, 7>, <F, 6>

F: <A, 9>, <B, 1>, <C, 10>, <D, 16>, <E, 6>

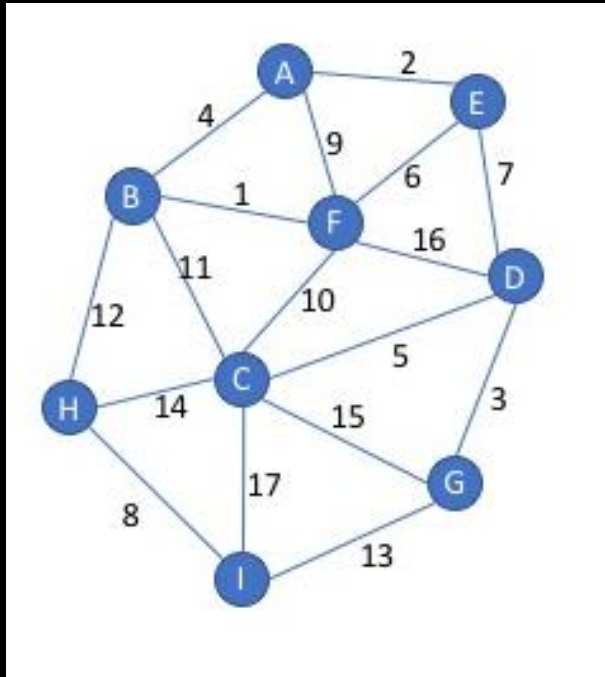
G: <C, 15>, <D, 3>, <I, 13>

H: <B, 12>, <C, 14>, <I, 8>

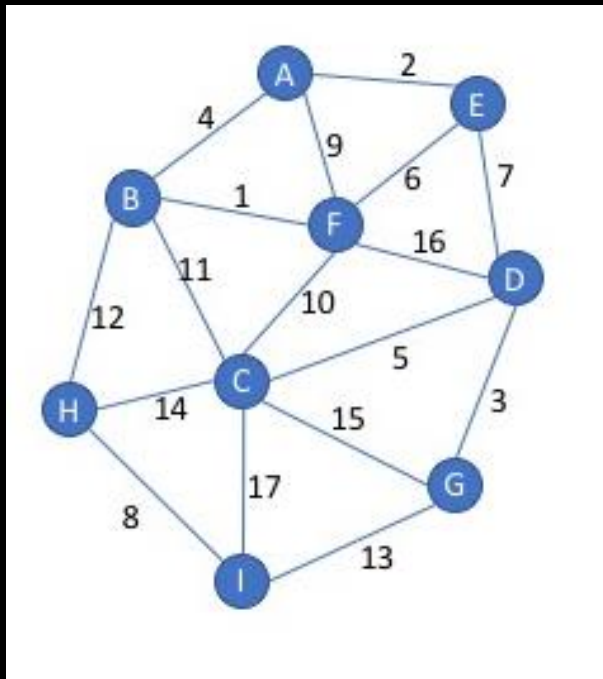
I: <C, 17>, <G, 13>, <H, 8>



# MST using Prim's starting from "I"



# MST using Prim's starting from "I"



I H B F A E D G C

# Alt Text for the Graph on Next Slide

Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

A: <B, 1>, <D, 3>, <F, 2>

B: <A, 1>, <C, 2>, <D, 1>, <E, 7>

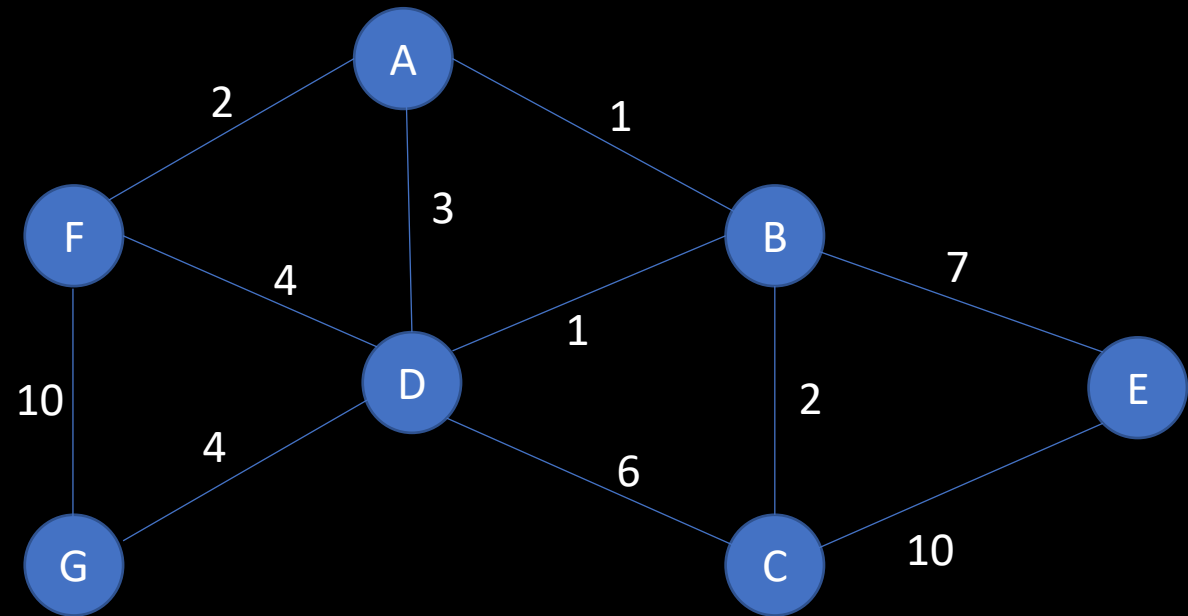
C: <B, 2>, <D, 6>, <E, 10>

D: <A, 3>, <B, 1>, <C, 6>, <F, 4>, <G, 4>

E: <B, 7>, <C, 10>

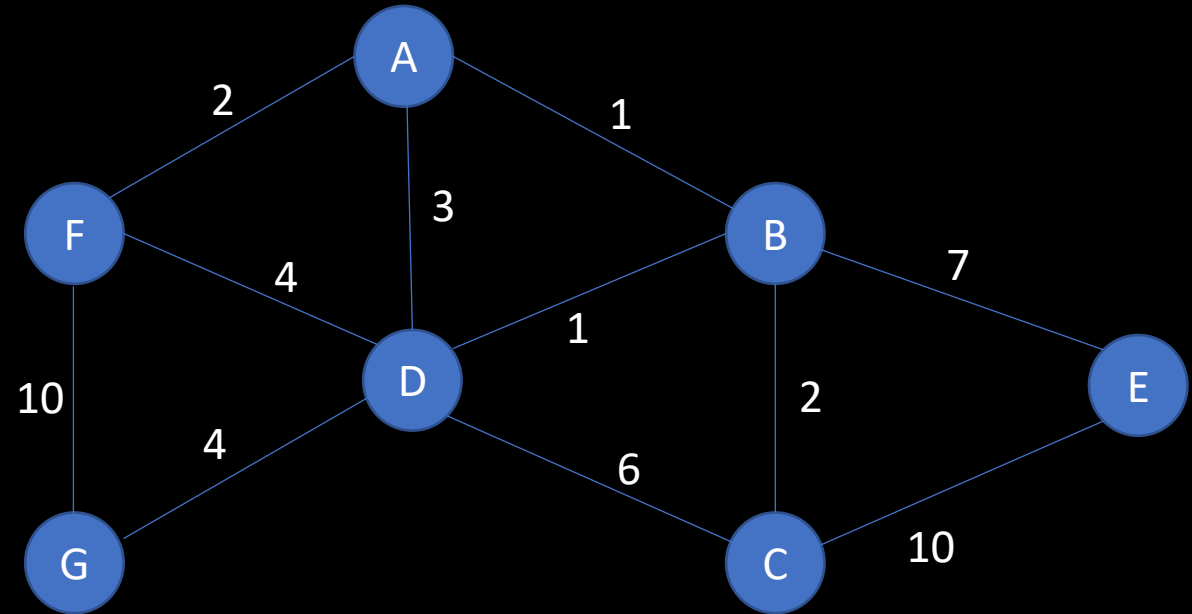
F: <A, 2>, <D, 4>, <G, 10>

G: <D, 4>, <F, 10>



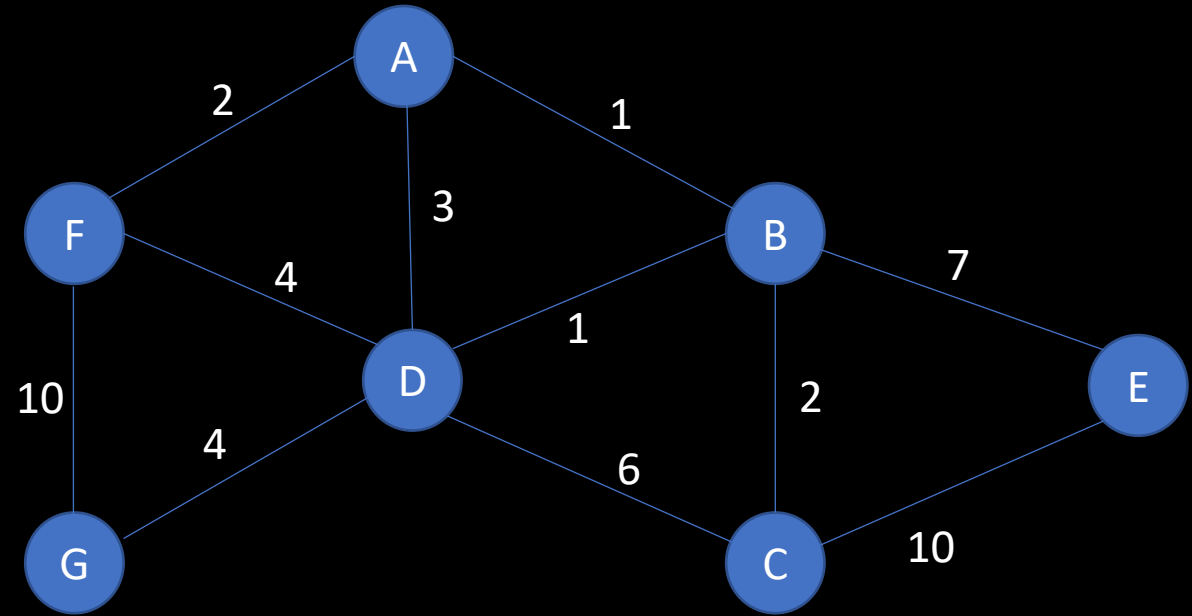
# Dijkstra with A as source

v	D(v)	P(v)
A		
B		
C		
D		
E		
F		
G		



# Dijkstra with A as source

v	D(v)	P(v)
A	0	NA
B	1	A
C	3	B
D	2	B
E	8	B
F	2	A
G	6	D



# Alt Text for the Graph on Next Slide

Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

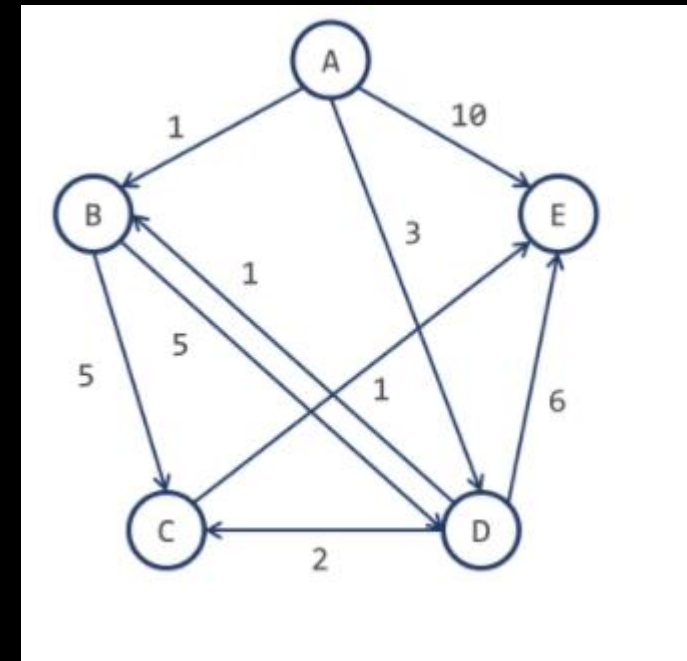
A: <B, 1>, <D, 3>, <E, 10>

B: <C, 5>, <D, 5>

C: <E, 1>

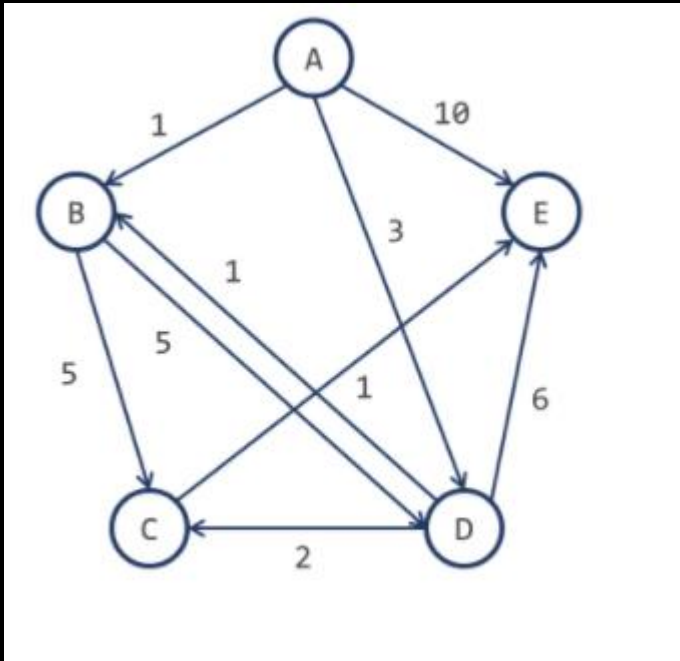
D: <B, 1>, <C, 2>, <E, 6>

E: -





# Dijkstra with A as source

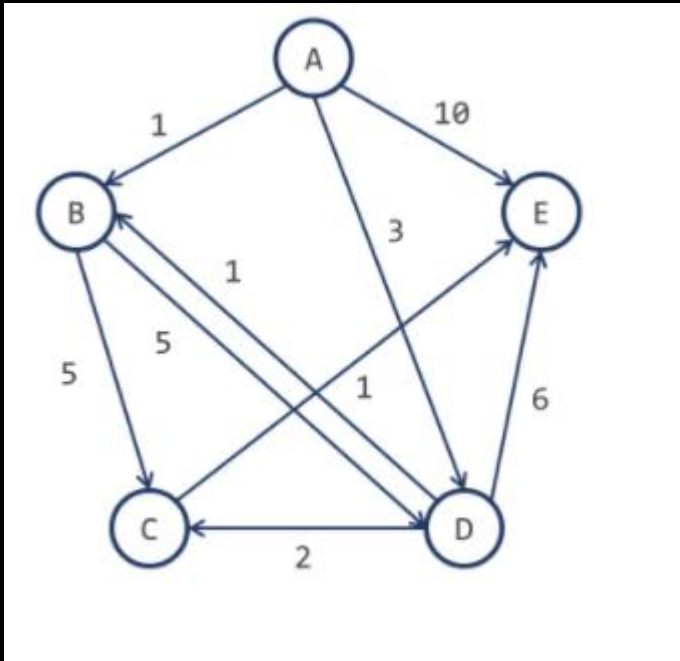


V  
B  
C  
D  
E

$d(V)$

$p(V)$

# Dijkstra with A as source



V  
B  
C  
D  
E

$d(V)$   
1  
5  
3  
6

$p(V)$   
A  
D  
A  
C

# Algorithmic Paradigms

# Algorithmic Paradigms

	Properties	Examples
Brute Force	<ul style="list-style-type: none"><li>▪ Generate and Test an Exhaustive Set of all possible combinations</li><li>▪ Can be computationally very expensive</li><li>▪ Guarantees optimal solution</li></ul>	<ul style="list-style-type: none"><li>▪ Finding divisors of a number, <math>n</math> by checking if all numbers from <math>1..n</math> divides <math>n</math> without remainder</li><li>▪ Finding duplicates using all combinations</li><li>▪ Bubble/Selection Sort</li></ul>
Divide and Conquer	<ul style="list-style-type: none"><li>▪ Break the problem into subcomponents typically using recursion</li><li>▪ Solve the basic component</li><li>▪ Combine the solutions to sub-problems</li></ul>	<ul style="list-style-type: none"><li>▪ Quick Sort</li><li>▪ Merge Sort</li><li>▪ Binary Search</li><li>▪ Peak Finding</li></ul>
Dynamic Programming	<ul style="list-style-type: none"><li>▪ Optimal substructure: solution to a large problem can be obtained by solution to a smaller optimal problems</li><li>▪ Overlapping sub-problems: space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.</li><li>▪ Guarantees optimal solution</li></ul>	<ul style="list-style-type: none"><li>▪ Fibonacci Sequence</li><li>▪ Assembly Scheduling</li><li>▪ Knapsack</li></ul>
Greedy Algorithms	<ul style="list-style-type: none"><li>▪ Local optimal solutions at each stage</li><li>▪ Does not guarantee optimal solution</li></ul>	<ul style="list-style-type: none"><li>▪ Prim's Algorithm</li><li>▪ Dijkstra's Algorithm</li><li>▪ Kruskal's Algorithm</li></ul>

# Bin Packing

If we have packets that each require 7 units, 8 units, 2 units and 3 units of space, how many minimum bins are required to store all the four packets if each bin can take at most 10 units of space using the following Greedy strategies

- **First Fit:** scan the bins and place the new item in the first bin that is large enough.
- **Best Fit:** scan the bins and place the new item in the bin that finds the spot that creates the smallest empty space

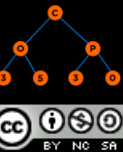
# Algorithm for Huffman Encoding

Given this file, generate a Huffman Tree and identify the codes of each character.

```
care racecar era
```

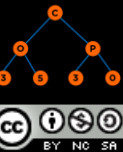
Use the following constraints when building the tree:

- the node with a lower frequency is attached to the left in case two nodes are merged after extraction from the priority queue;
- if two nodes have the same priority when merging, then the merged nodes are resolved as follows:
  - if both nodes have letters (a.k.a. leaf nodes) then the letter with lower ascii value will be the left node when combining two nodes into a tree.
  - if one or both nodes have cumulative frequencies, then the node with more number of nodes in the tree will be attached to the right, in other words, the smaller tree will be towards the left.
- traversing left from a node appends '0' to the Huffman code and traversing right appends '1'.



# Algorithm for Huffman Encoding

1. Create a table with symbols and their frequencies
2. Construct a set of trees with root nodes that contain each of the individual symbols and their weight (frequency).
3. Place the set of trees into a min priority queue.
4. while the priority queue has more than one item
  - Remove the two trees with the smallest weights.
  - Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.
  - Insert the newly created tree back into the priority queue.
5. Traverse the resulting tree to obtain binary codes for characters



# Algorithm for Huffman Encoding

1. Create a table with symbols and their frequencies

care racecar era

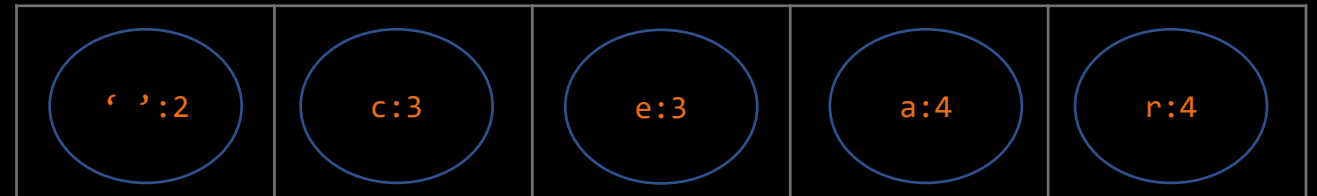
Character	Frequency
a	4
r	4
c	3
e	3
' ,	2



# Algorithm for Huffman Encoding

- Construct a set of trees with root nodes that contain each of the individual symbols and their weight (frequency).
- Place the set of trees into a min priority queue.

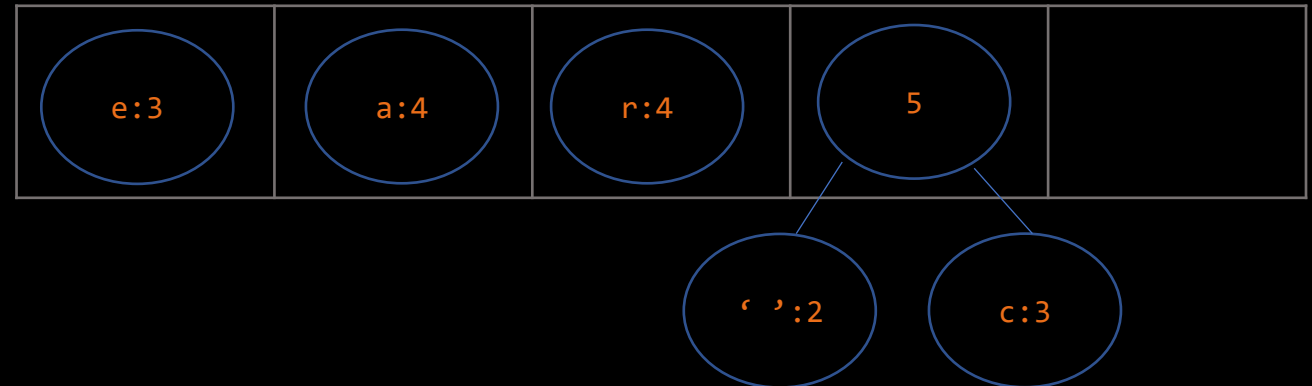
Character	Frequency
a	4
r	4
c	3
e	3
' '	2



# Algorithm for Huffman Encoding

- Construct a set of trees with root nodes that contain each of the individual symbols and their weight (frequency).
- Place the set of trees into a min priority queue.

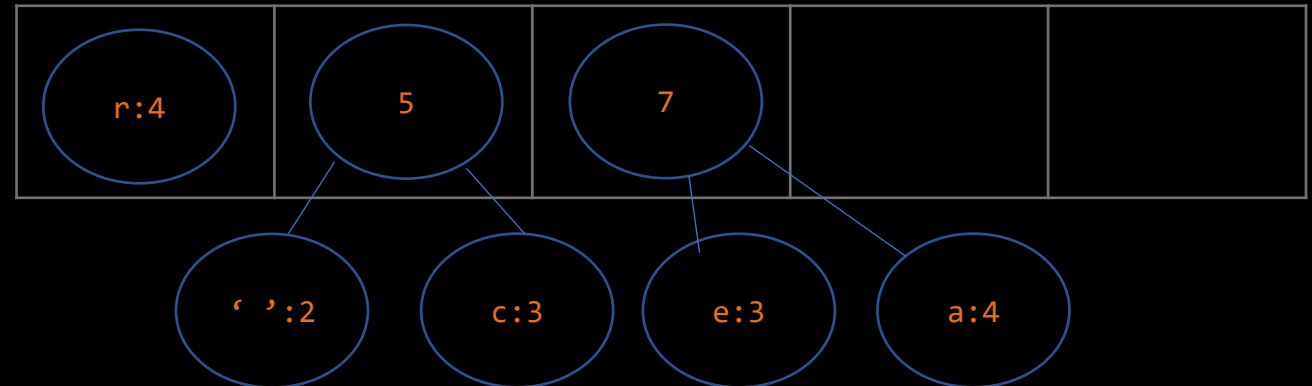
Character	Frequency
a	4
r	4
c	3
e	3
' '	2



# Algorithm for Huffman Encoding

2. Construct a set of trees with root nodes that contain each of the individual symbols and their weight (frequency).
3. Place the set of trees into a min priority queue.

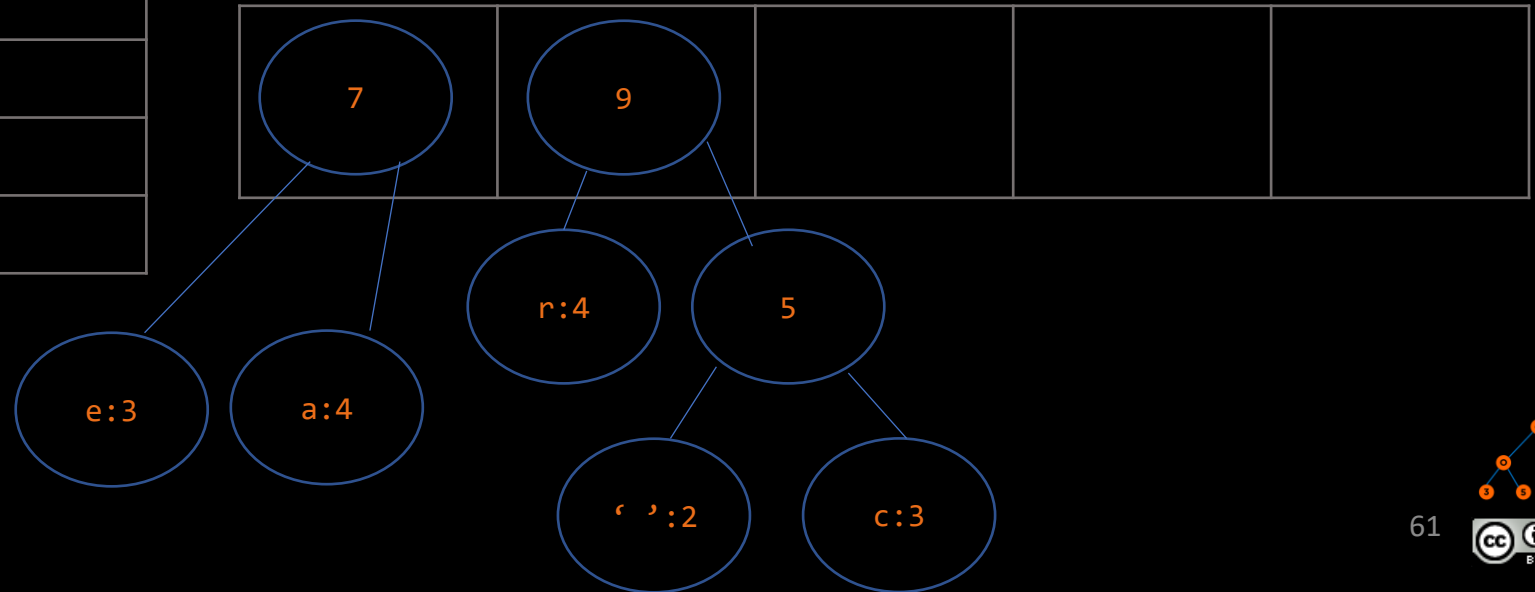
Character	Frequency
a	4
r	4
c	3
e	3
' '	2



# Algorithm for Huffman Encoding

- Construct a set of trees with root nodes that contain each of the individual symbols and their weight (frequency).
- Place the set of trees into a min priority queue.

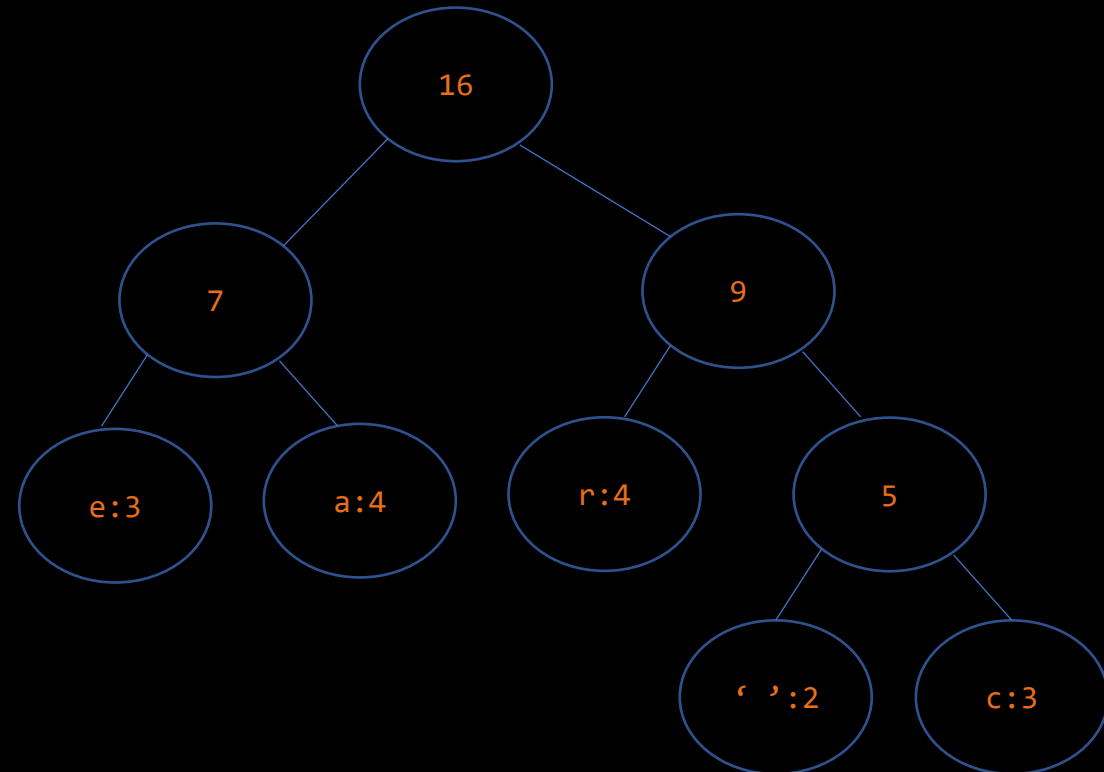
Character	Frequency
a	4
r	4
c	3
e	3
' '	2



# Algorithm for Huffman Encoding

- while the priority queue has more than one item  
Remove the two trees with the smallest weights.  
Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.  
Insert the newly created tree back into the priority queue.

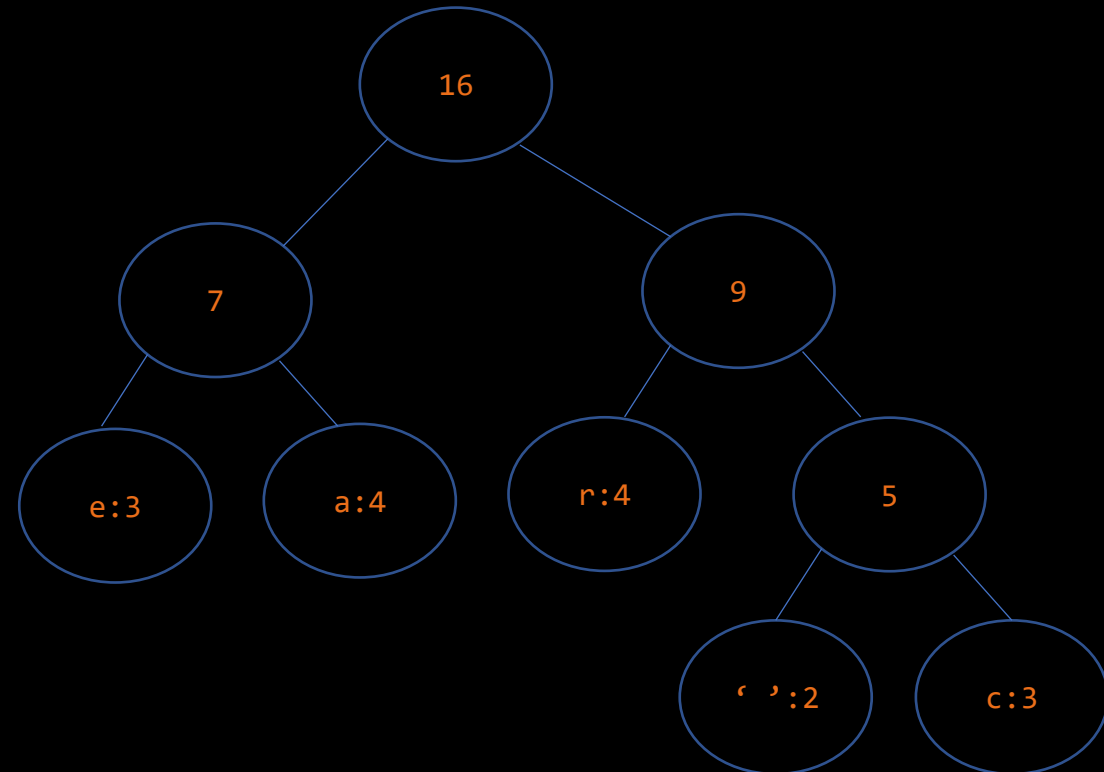
Character	Frequency
a	4
r	4
c	3
e	3
' '	2



# Algorithm for Huffman Encoding

- while the priority queue has more than one item  
Remove the two trees with the smallest weights.  
Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.  
Insert the newly created tree back into the priority queue.

Character	Frequency	Huffman Code
a	4	01
r	4	10
c	3	111
e	3	00
' '	2	110



# Questions