CSCI 2270: Data Structures

Recitation #10 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours**
  - **12pm to 2pm on Mondays**
  - **12:30pm to 2:30pm on Fridays**
  - **Same Zoom ID as that of recitation - https://cuboulder.zoom.us/j/3112555724**

- In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.

Logistics

- In case you have a question during the recitation, just unmute yourself and speak up. Let’s try to make it as interactive as possible.

- You can ask questions via chat as well, but I would prefer if you guys ask your questions verbally.

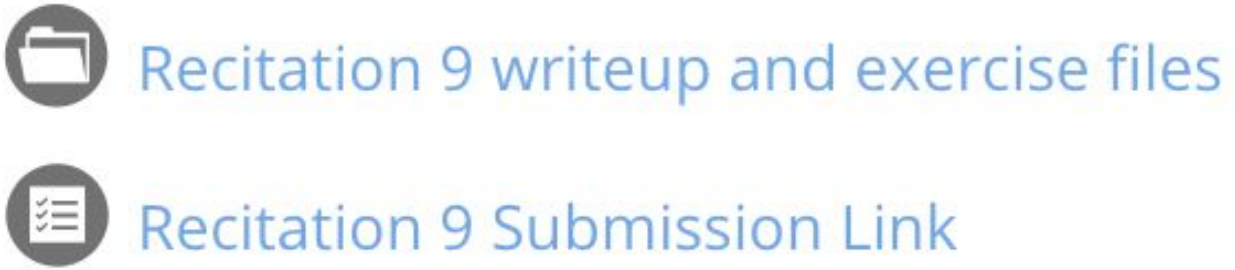
- I would highly encourage you guys to switch on your cameras as well (if possible). It helps in making the session more lively and interactive.

Logistics

- **Attendance for Recitations**
    - You get points for the recitation only if you submit your recitation exercise on moodle.
    - Download starter code from moodle. Make necessary changes to finish the recitation exercise. Compress all the files into one single zip file and upload it on moodle.

Logistics: Attendance for Recitation 9

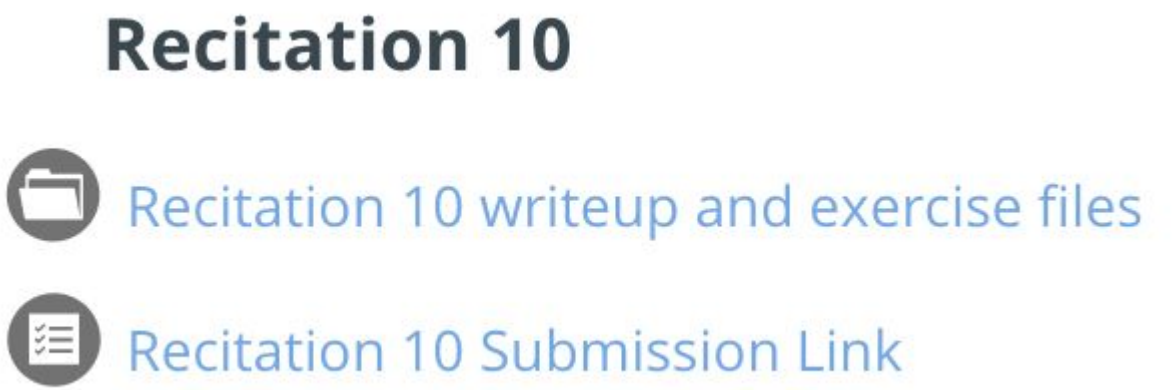
- Upload a single zip file with solved code to Moodle.
- Your points for Recitation 9 depend on this.



- **Due Date - Sunday, March 22 2020, 11:59 PM**

Logistics: Attendance for Recitation 10

- Upload a single zip file with solved code to Moodle.
- Your points for Recitation 10 depend on this.

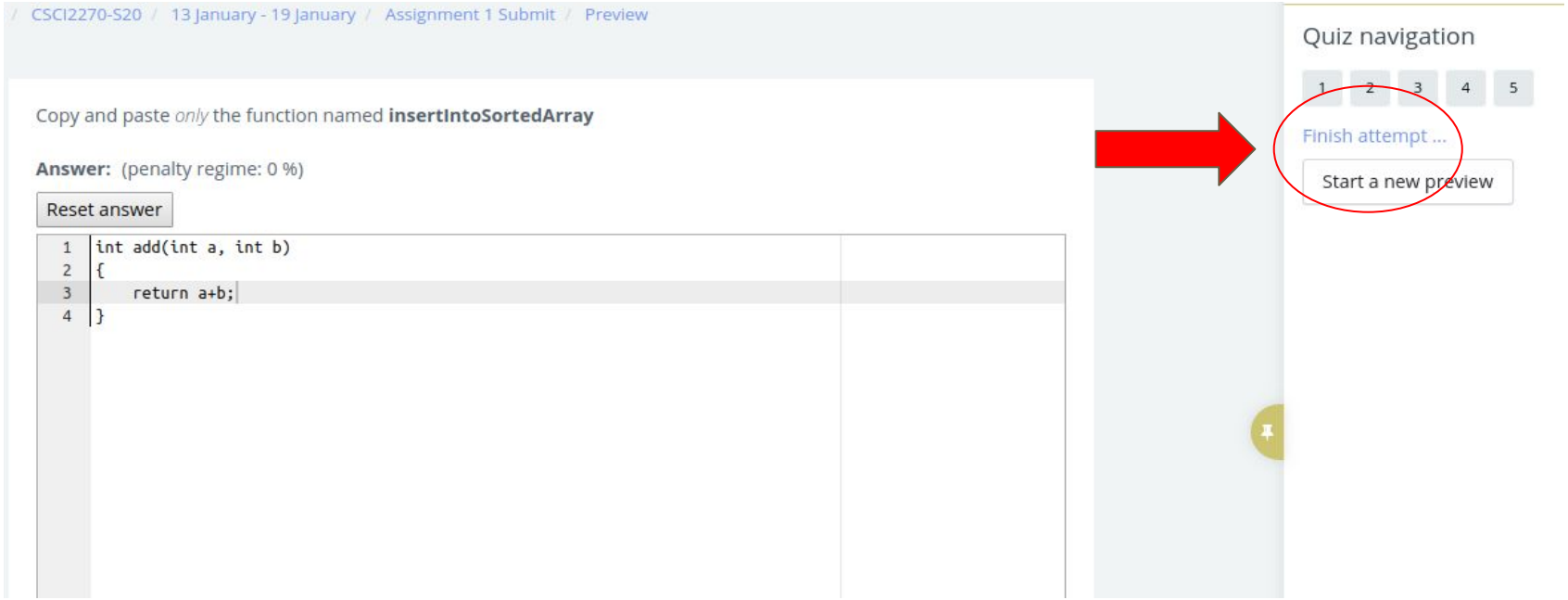


- **Due Date - Sunday, March 22 2020, 11:59 PM**

Logistics

- **Assignment 7 is due on Sunday, March 22 2020, 11:59 PM.**
  GOOD LUCK!

Please click on “Finish Attempt” after you are done!



Any questions?

Agenda

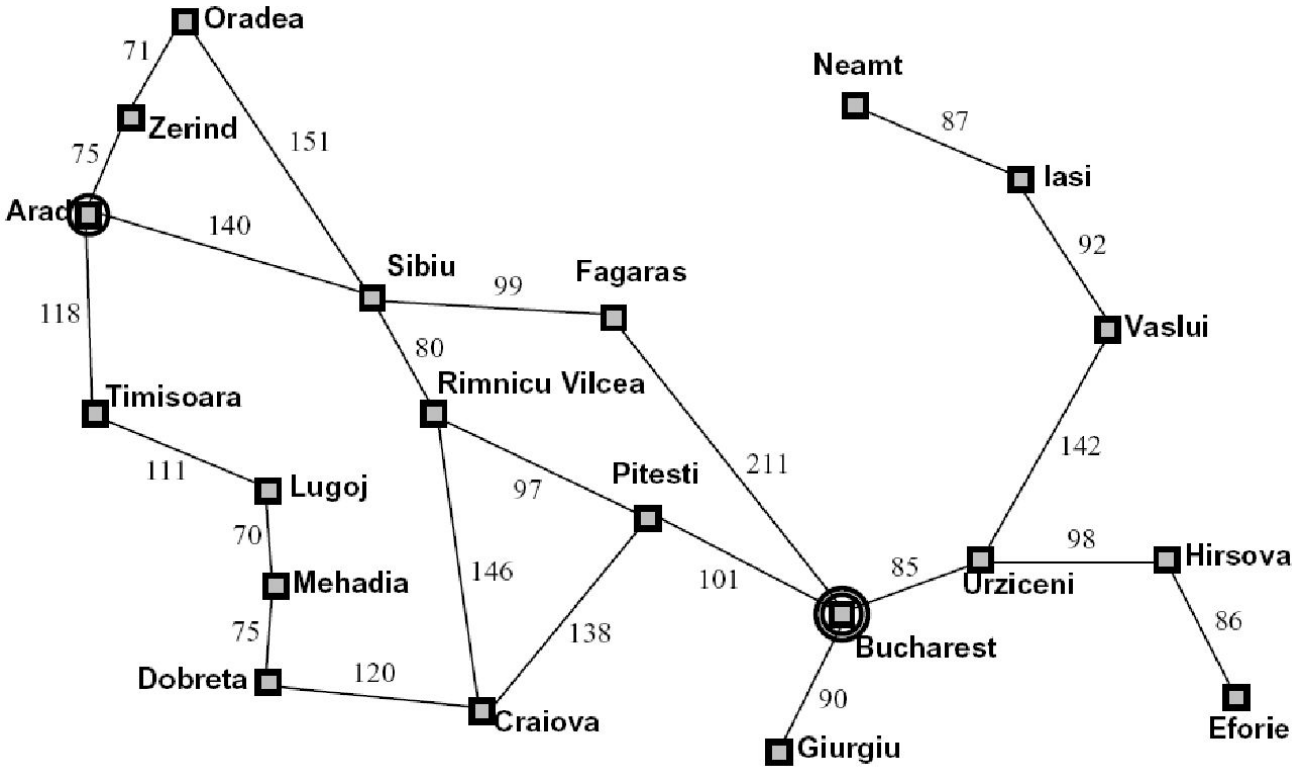
- Reviewing Recitation 9 material
  - Graph representations

- Reviewing Recitation 10 material
  - Breadth First Search
  - Depth First Search

- Exercise
  - **Silver Problem:** Find the length of the shortest path between two vertices
  - **Gold Problem:** Print out the shortest path between two vertices.

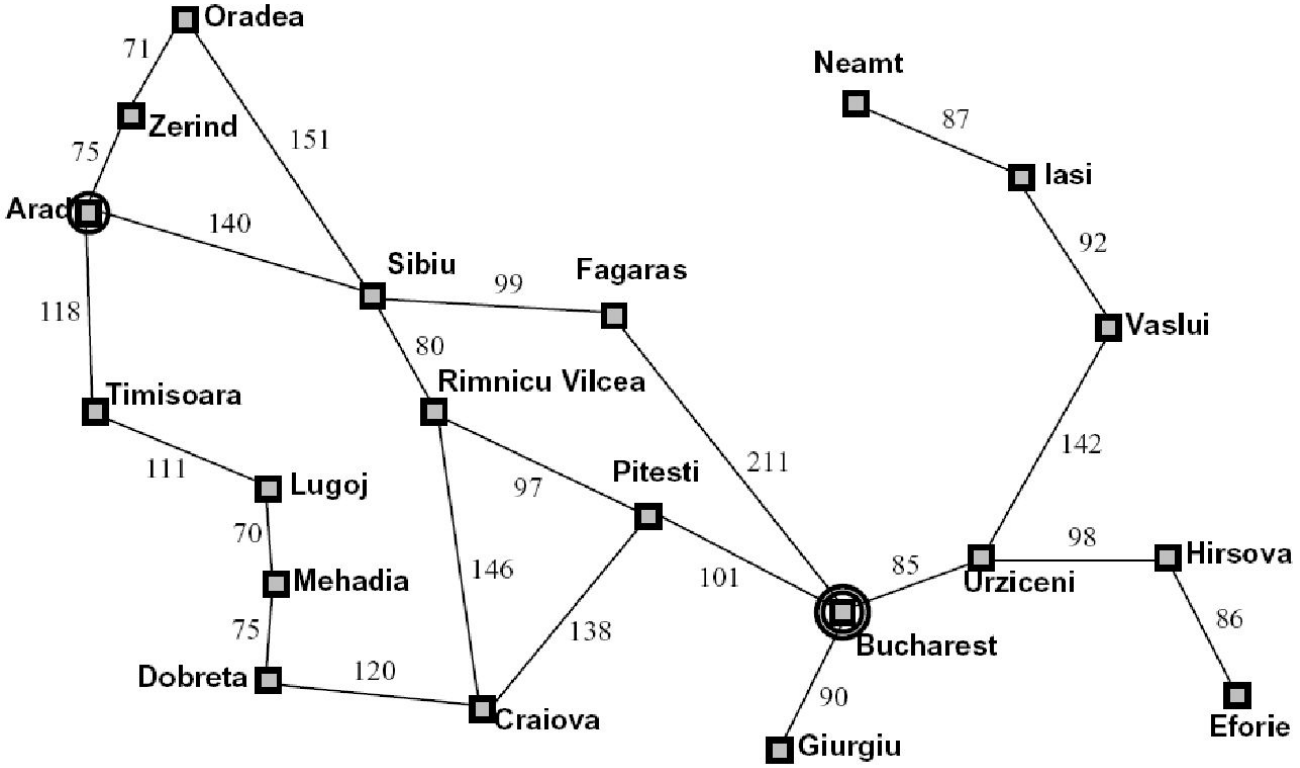
Reviewing Recitation 9 material

Yo! Find me a path from Arad to Bucharest



Yo! Find me a path from Arad to Bucharest

We use a non-linear data structure called “GRAPHS” to solve problems like these.



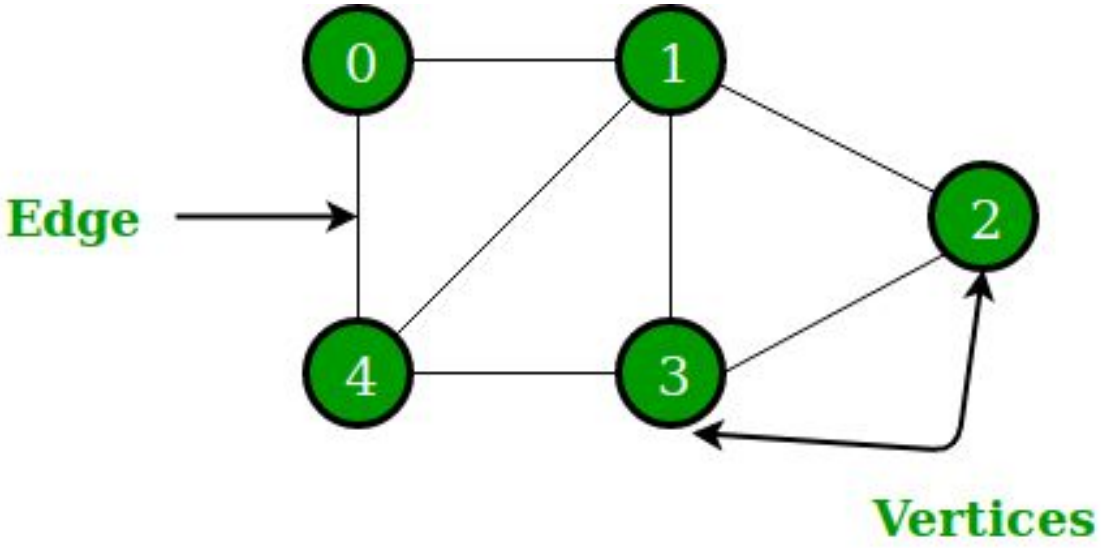
Graph

- A graph is a non linear data structure.
  - Generally denoted as G (V , E)

- A graph consists of a finite set of **vertices** (also called nodes or points).
  - V is used to denote the set of vertices/nodes of the graph.
  - Let x,y be two vertices in our graph.

- These vertices are often connected to each other via lines/links and these links represent that there is some relationship between those vertices. Such links are called **edges**.
  - E is used to refer to the collection of edges in that graph.
  - If the tuple (x,y) belong to the set E, we say that there is an edge between x and y in our graph.

Graph



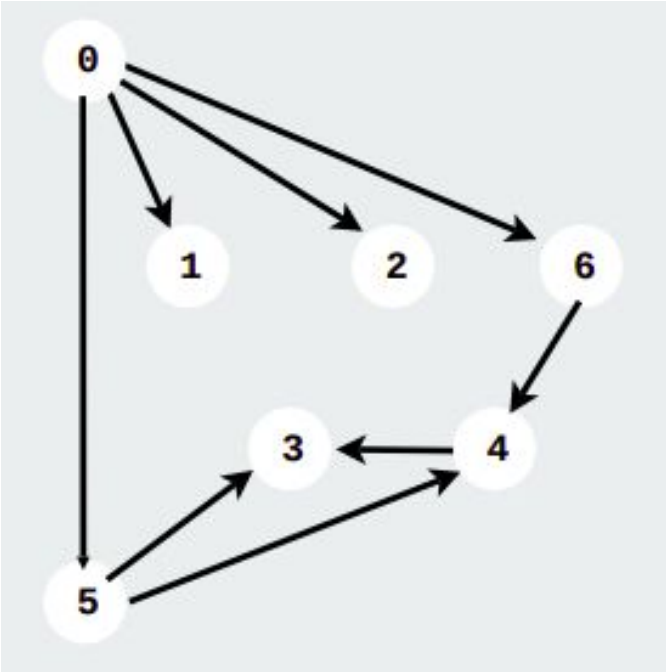
Directed Graph

- A graph is a directed graph if all the edges in that graph are directed.
- If the tuple (u, v) belongs in your set E, then we say that there is an edge from vertex “u” to vertex “v”.

- The opposite need not be true, i.e. you can’t assume that there exists an edge from vertex “v” to vertex “u” just because (u,v) belongs to E.
- In the same graph, edge from vertex “v” to vertex “u” exists only if (v,u) belongs to E.

Directed Graph (example)

- Following is a directed graph with 7 vertices and 8 directed edges.
  - Few Edges are (0,1); (6,4); (5,4)



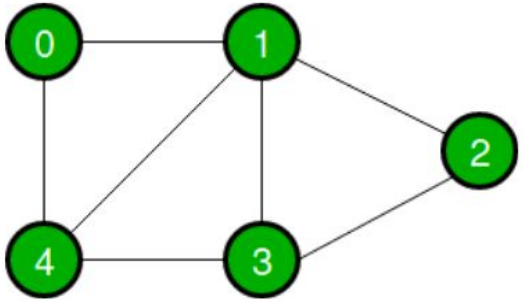
Undirected Graph

- A graph is an undirected graph if all the edges in that graph are undirected.

- If the tuple (u, v) belongs in your set E, then we say that there is an edge from vertex “u” to vertex “v” and there is also an edge from vertex “v” to vertex “u”.

Undirected Graph (example)

- Following is a directed graph with 5 vertices and 7 undirected edges.
  - Few Edges are (0,1); (1,4); (2,3)



Any questions?

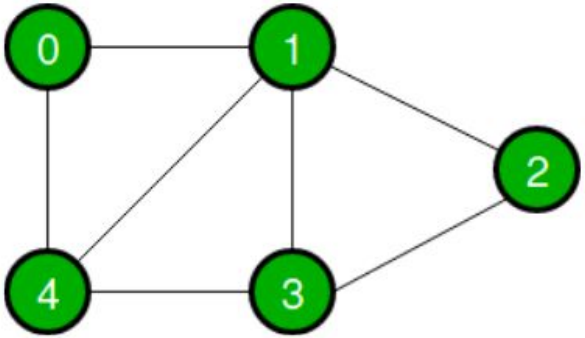
How to represent all the edges in a Graph?

Two most commonly used methods:

- Adjacency Matrix
- Adjacency List
  - **This is what we are using for our recitation exercises.**

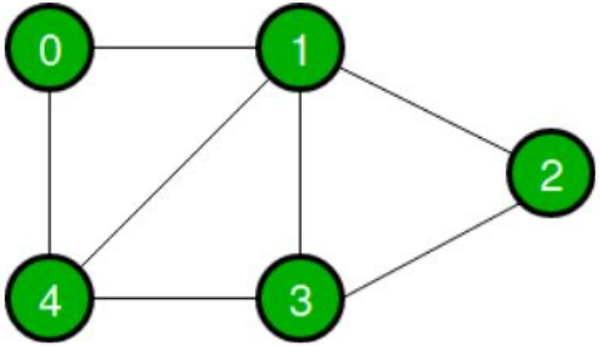
Adjacency Matrix

- When you have an undirected graph with 5 vertices,



Graph

Adjacency Matrix



Graph

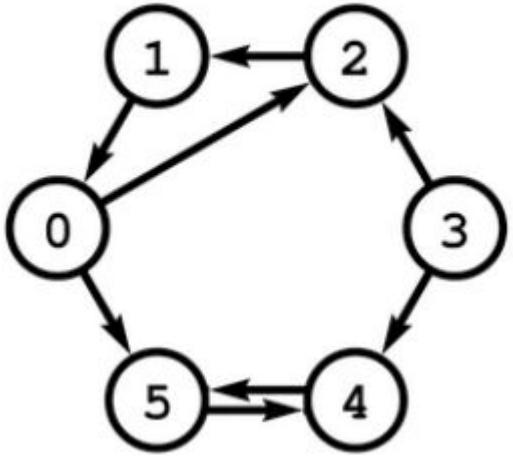
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix (5 x 5)

- **For a graph with n vertices, the adjacency matrix is of size n x n**

Adjacency Matrix

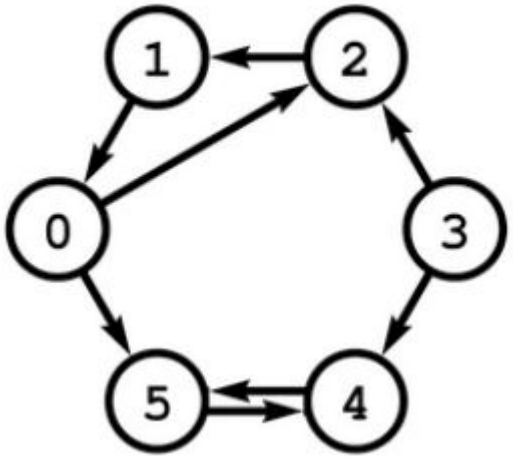
- When you have a directed graph with 6 vertices,



Graph

Adjacency Matrix

- When you have a directed graph with 6 vertices,



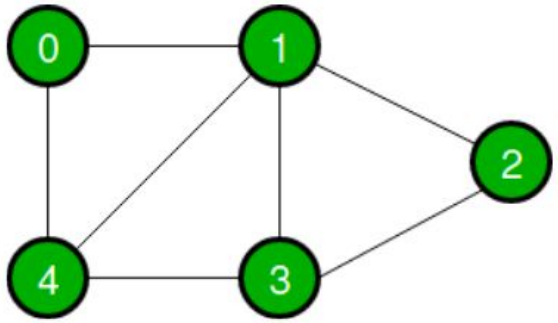
Graph

	0	1	2	3	4	5
0	**0**	0	1	0	0	1
1	1	**0**	0	0	0	0
2	0	1	**0**	0	0	0
3	0	0	1	**0**	1	0
4	0	0	0	0	**0**	1
5	0	0	0	0	1	**0**

Adjacency Matrix (6 x 6)

Adjacency List

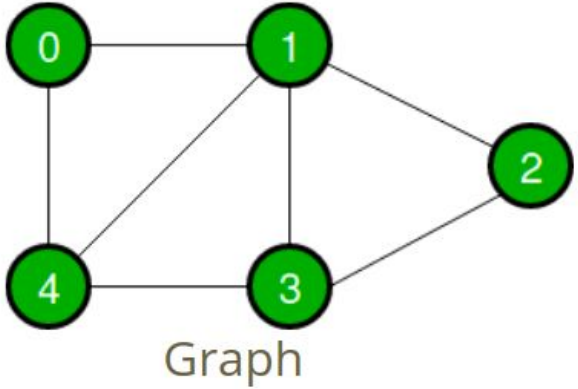
- When you have an undirected graph with 5 vertices,



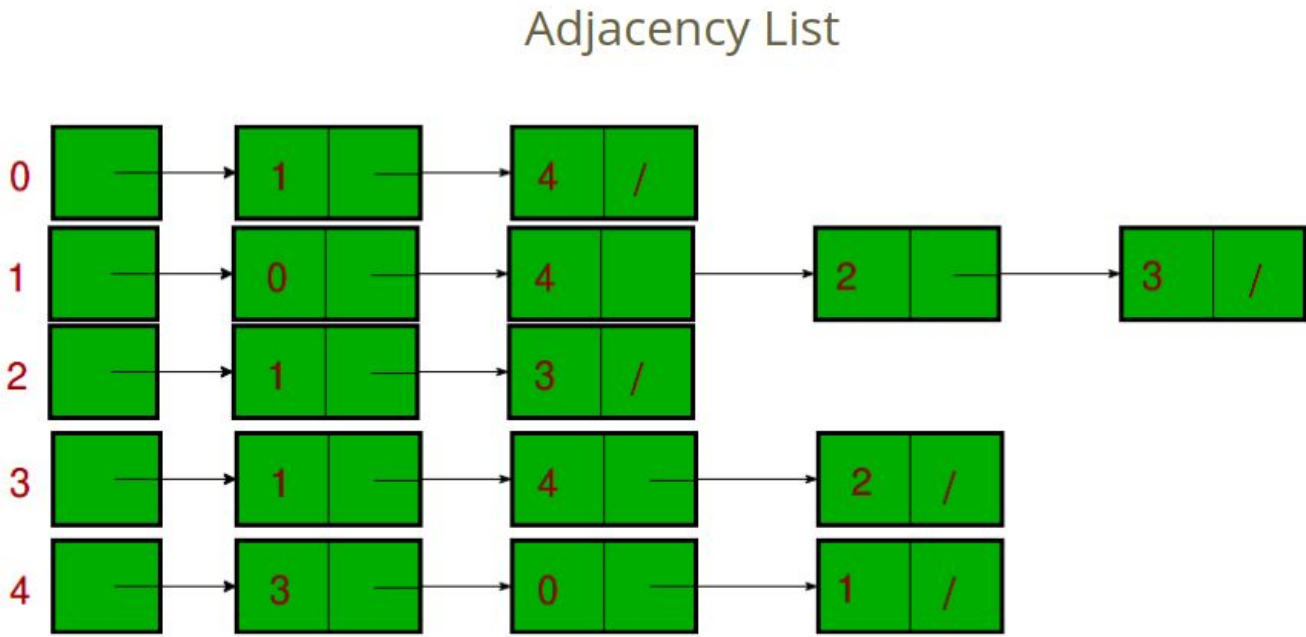
Graph

Adjacency List

- When you have an undirected graph with 5 vertices,

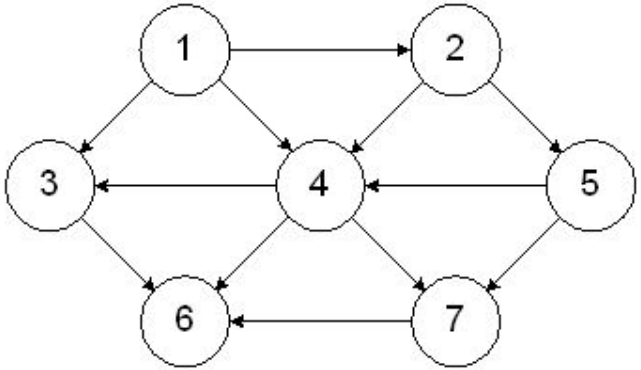


Graph



Adjacency List

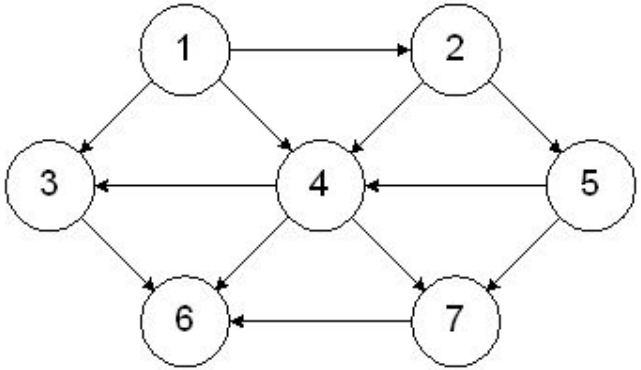
- When you have a directed graph with 7 vertices,



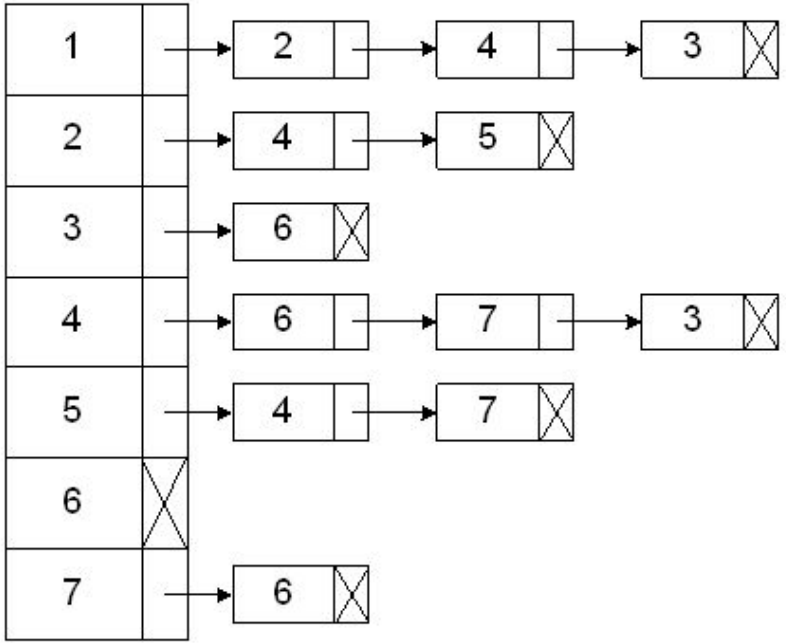
Graph

Adjacency List

- When you have a directed graph with 7 vertices,



Graph



Adjacency List

Any questions?

Why use Adjacency List instead of Adjacency Matrix?

Why use Adjacency List instead of Adjacency Matrix?

For a graph with n vertices and m edges, in terms of space complexity

- Adjacency matrix takes $O(n^2)$
- Adjacency list takes O(n+m)

Thus, for graphs with few edges, also known as sparse graphs, adjacency list is the desired choice.

Now let's look at how to represent graphs in code.

- Vertex

```
struct vertex{
    int key;
    bool visited = false;
    std::vector<adjVertex> adj;
};
```

```
struct adjVertex{
    vertex *v;
};
```

- **key** stores the value of that vertex
- **visited** allows us to infer if a node has been visited or not while traversing the graph.
- **adj** is a vector of type adjVertex that is our adjacency list. It stores all the vertices that are directly connected to the current vertex/node via an edge/link.
- **adjVertex** is just a struct that has a pointer of type vertex in it.

Now let's look at how to represent graphs in code.

- **addVertex** function adds a new vertex with value v
- **addEdge** function adds an edge between vertex v1 and v2
- **vertices** is a vector that stores pointers to all the vertices in the graph

```
class Graph
{
    public:
        void addEdge(int v1, int v2);
        void addVertex(int v);
        void printGraph();

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

};
```

Few basic vector commands

- Access element at index **i** in a vector graph_vertices
  - graph_vertices[i] or graph_vertices.at(i)

- Size of a vector graph_vertices
  - graph_vertices.size()

- **Access the value of j^{th} vertex in the adjacency list of a vertex at index i in vector graph_vertices**
  - **graph_vertices[i]->adj[j].v->key**

Recitation 9 exercise

- Complete the printGraph() function that prints all the graph vertices with their adjacency list.

  graph_vertices[i]->adj[j].v->key

- ^ This lets you access the value of the jth adjacent vertex of the vertex at index i in graph_vertices vector.

- Now, use appropriate for loops to finish the function and get the expected output.

Any questions?

Reviewing Recitation 10 material

Graph Traversal

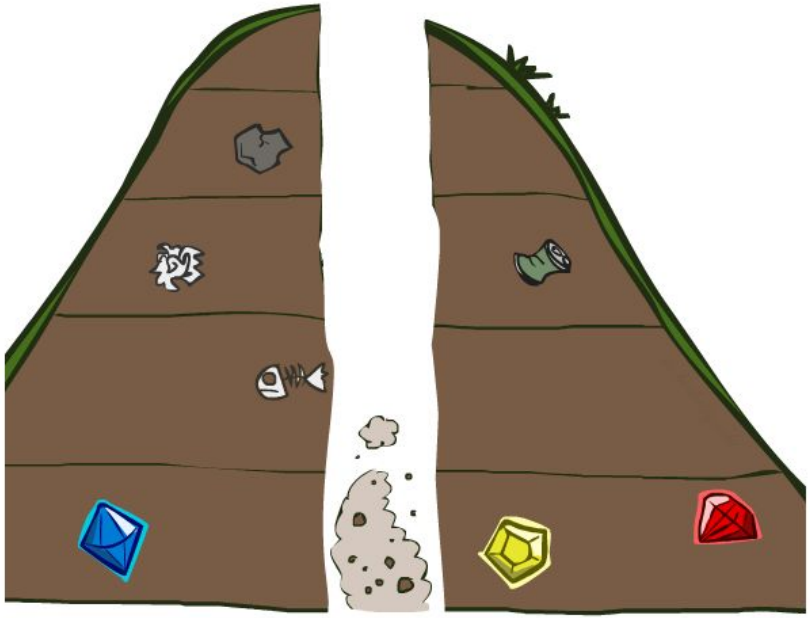
- Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph.

- Such traversals are classified by the order in which the vertices are visited.

- Traversal techniques we will be discussing today -
  - **Depth First Search (DFS)**
  - **Breadth First Search (BFS)**

Depth First Search (DFS)

- Strategy: expand the deepest node first. We use **stack** data structure for doing DFS



Depth First Search (DFS)

- A depth-first search (DFS) is an algorithm for traversing a finite graph.
- DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth.
- Iterative implementation of DFS - creating and using a stack for it
- Recursive implementation of DFS - using recursion to traverse the graph, but those recursive calls are implicitly stored on the system stack anyways.

DFS Psuedocode to search for a node in the graph

- **Iterative implementation**

```
function DFS(G, v)
        Create a stack S
        S.push(v)
        while S is not empty do
                w = S.pop()
                if w is what we are looking for then
                        return w
                if w.visited ==false
                        w.visited = true
                        for all vertices X in G.adjacencyList(w) do
                                S.push(X)
        return null
```

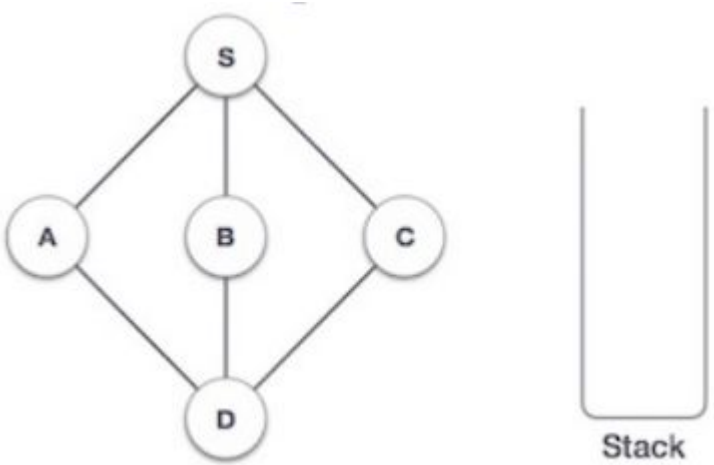
DFS Psuedocode to search for a node in the graph

- **Recursive implementation**

```
function DFS(G, v)
        v.visited = true
        if v is what we are looking for then
                return v
        for all vertices X in G.adjacencyList(v) do
                if X.visited == false, then
                        To_be_returned = DFS(G,X)
                        if(To_be_returned != NULL)
                                return To_be_returned
        return null
```

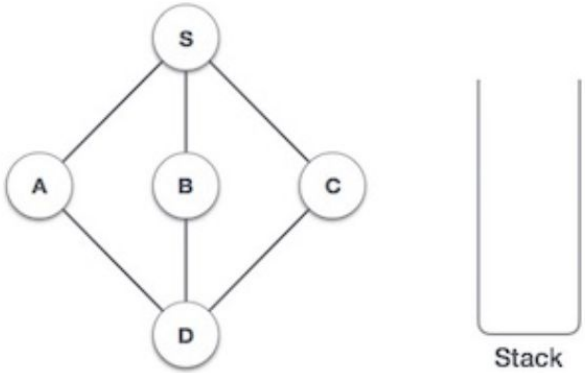
DFS Example

- Let’s start traversing the graph from the vertex **S**
- A vertex’s color is white if it hasn’t been visited yet. Its color changes to Grey as soon as it is visited.

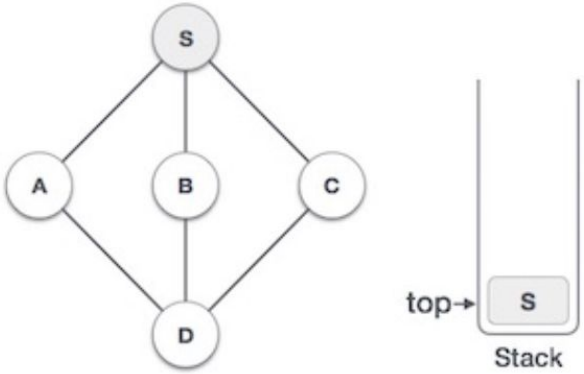


DFS Example

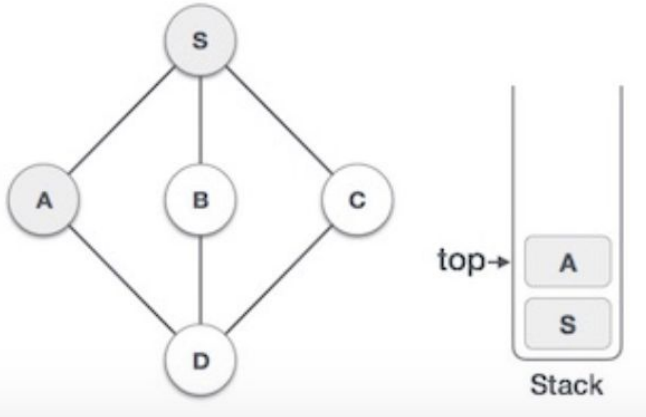
1



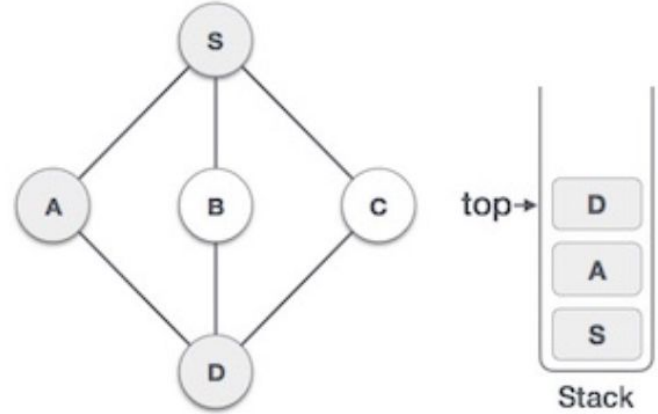
2



3

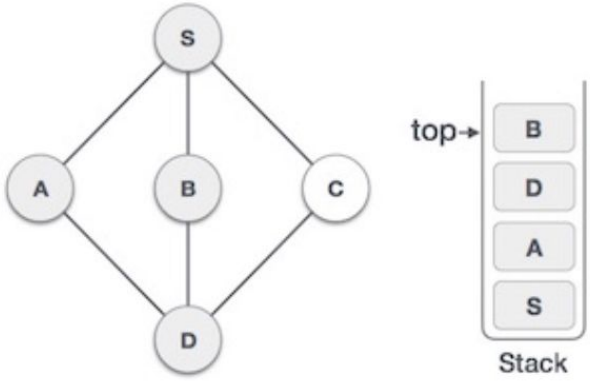


4

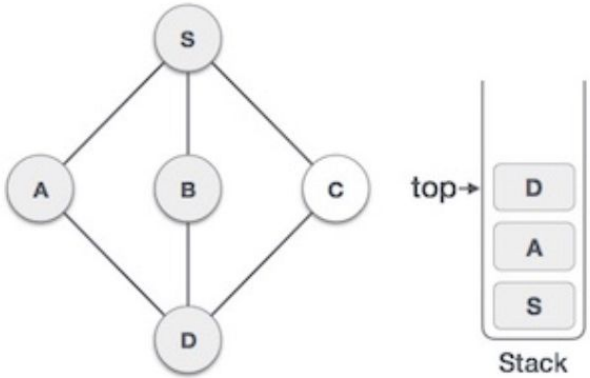


DFS Example

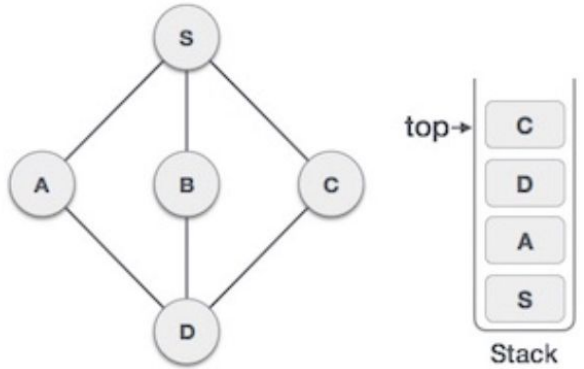
5



6



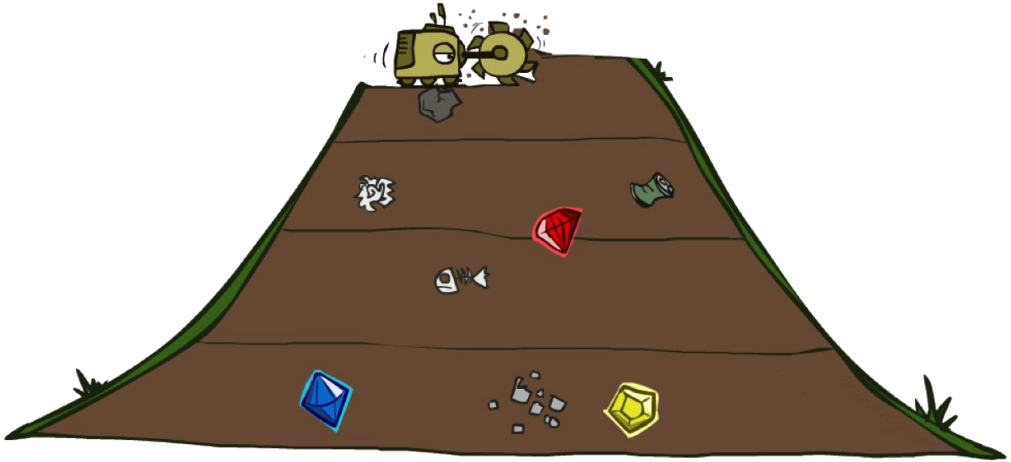
7



Any Questions?

Breadth First Search (BFS)

- Strategy: expand the shallowest node first. We use **queue** data structure for doing BFS



Breadth First Search (BFS)

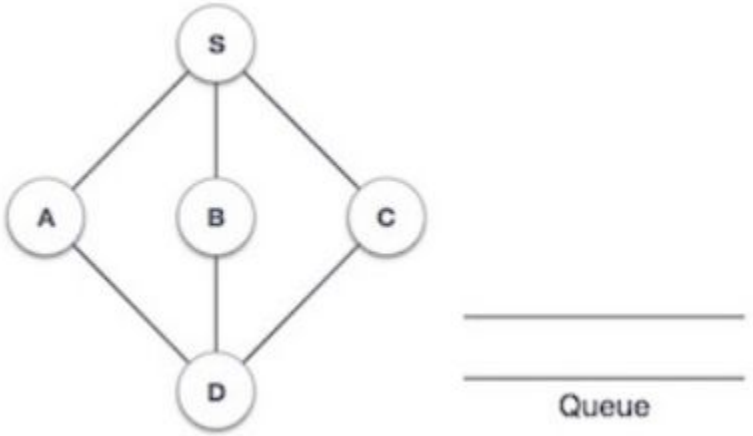
- A breadth-first search (DFS) is an algorithm for traversing a finite graph.
- BFS visits the sibling vertices before visiting the child vertices, and a queue is used in the search process.

BFS Psuedocode to search for a node in the graph

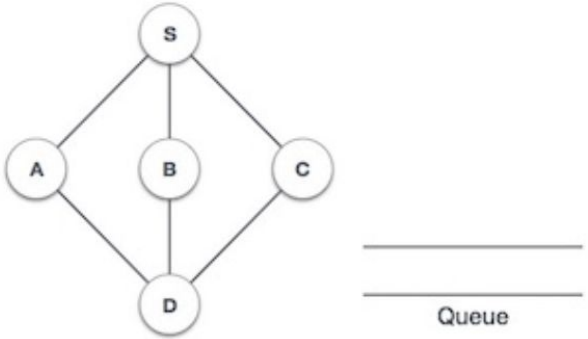
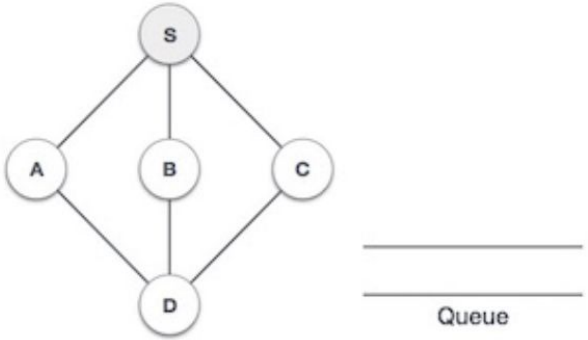
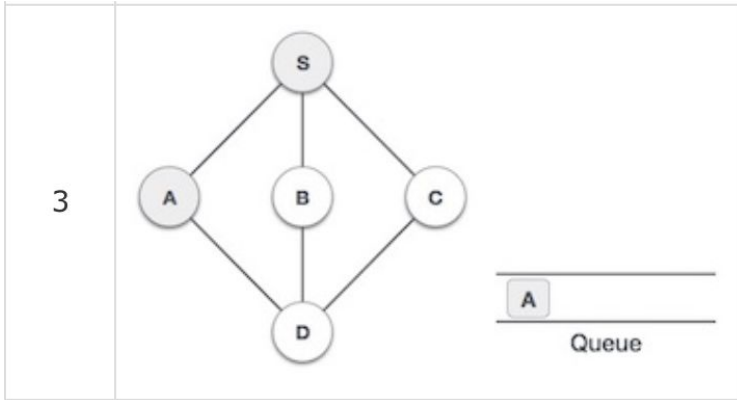
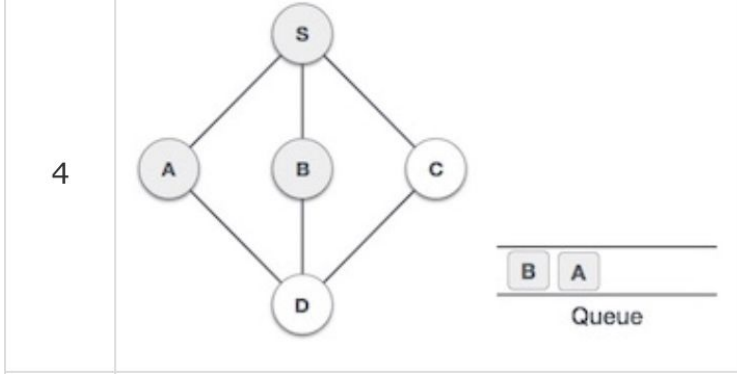
```
function BFS(G, v)
    create a queue Q
    enqueue v onto Q
    v.visited = true
    while Q is not empty do
        w ← Q.dequeue()
        if w is what we are looking for then
            return w
        for all vertices X in G.adjacencyList(w) do
            if X.visited == false, then
                X.visited = true
                enqueue X onto Q
    return null
```

BFS Example

- Let’s start traversing the graph from the vertex **S**
- A vertex’s color is white if it hasn’t been visited yet. Its color changes to Grey as soon as it is visited.

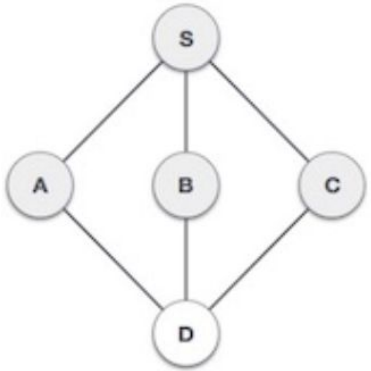


BFS Example

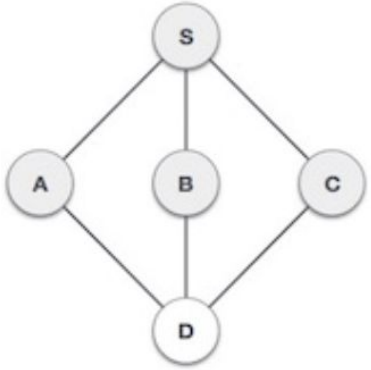
Step	Traversal
1	
2	
3	
4	

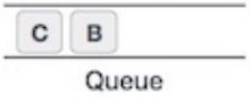
BFS Example

5

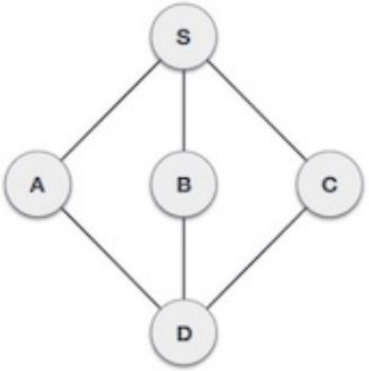


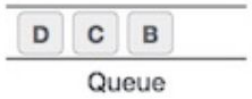
6





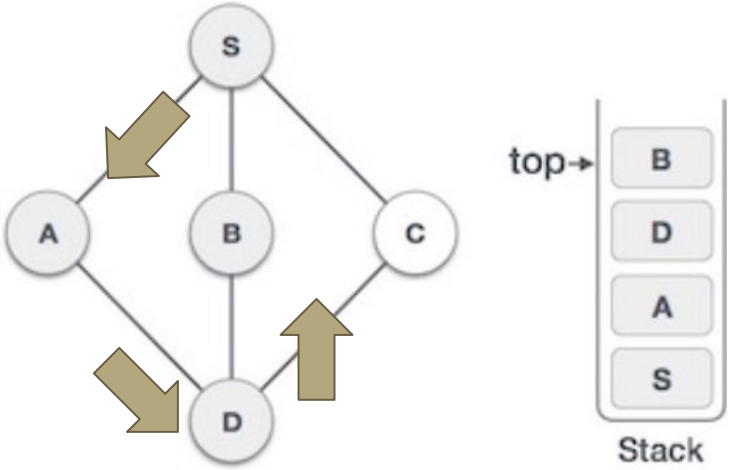
7



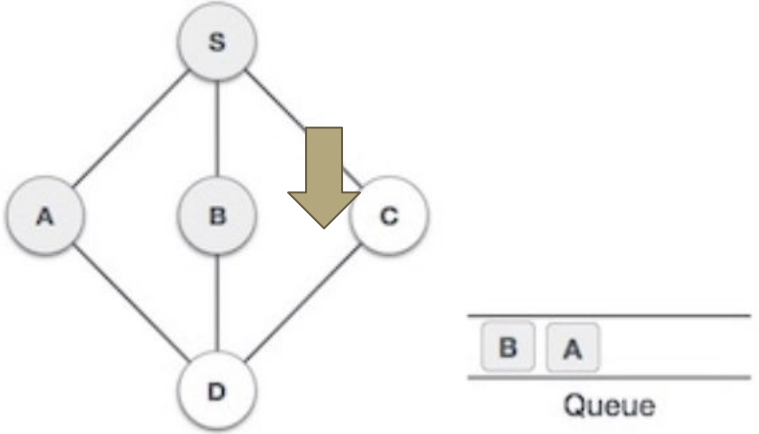


DFS vs BFS - Path Traversed to reach B from S

- DFS: S->A->D->B

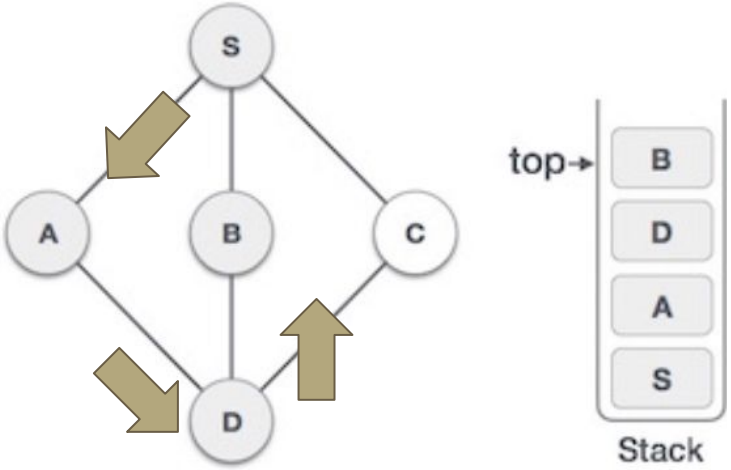


- BFS: S->B

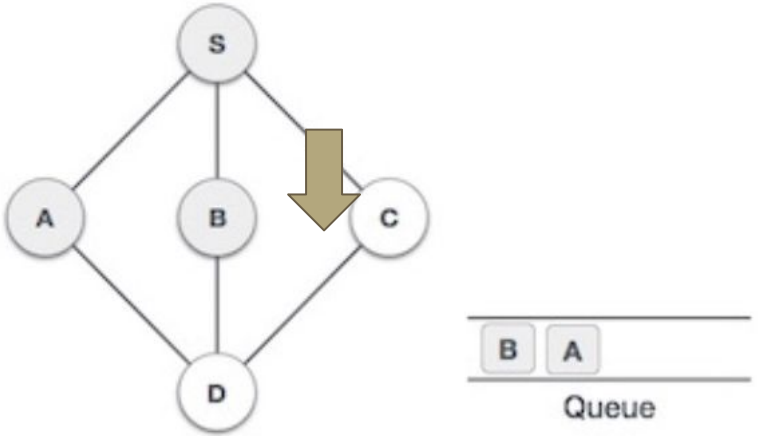


DFS vs BFS - Path Traversed to reach B from S

- DFS: S->A->D->B



- BFS: S->B

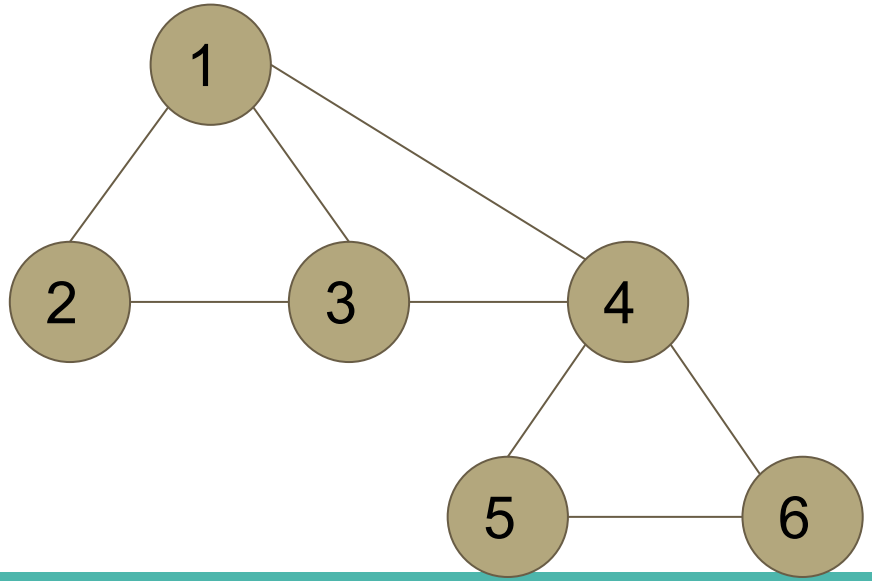


BFS is used to find the shortest path between nodes in a graph.

Any questions?

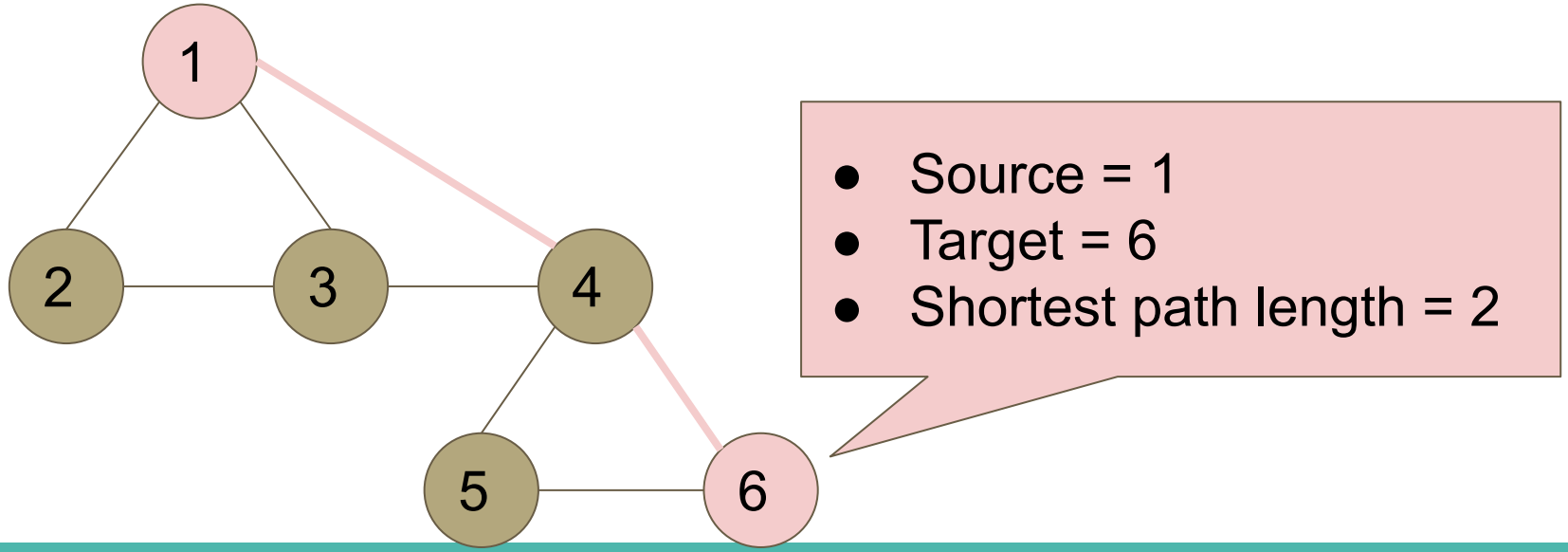
Recitation 10 Exercise (Silver Problem)

- Calculate the length of the shortest path between a source and destination vertex



Recitation 10 Exercise (Silver Problem)

- Calculate the length of the shortest path between a source and destination vertex

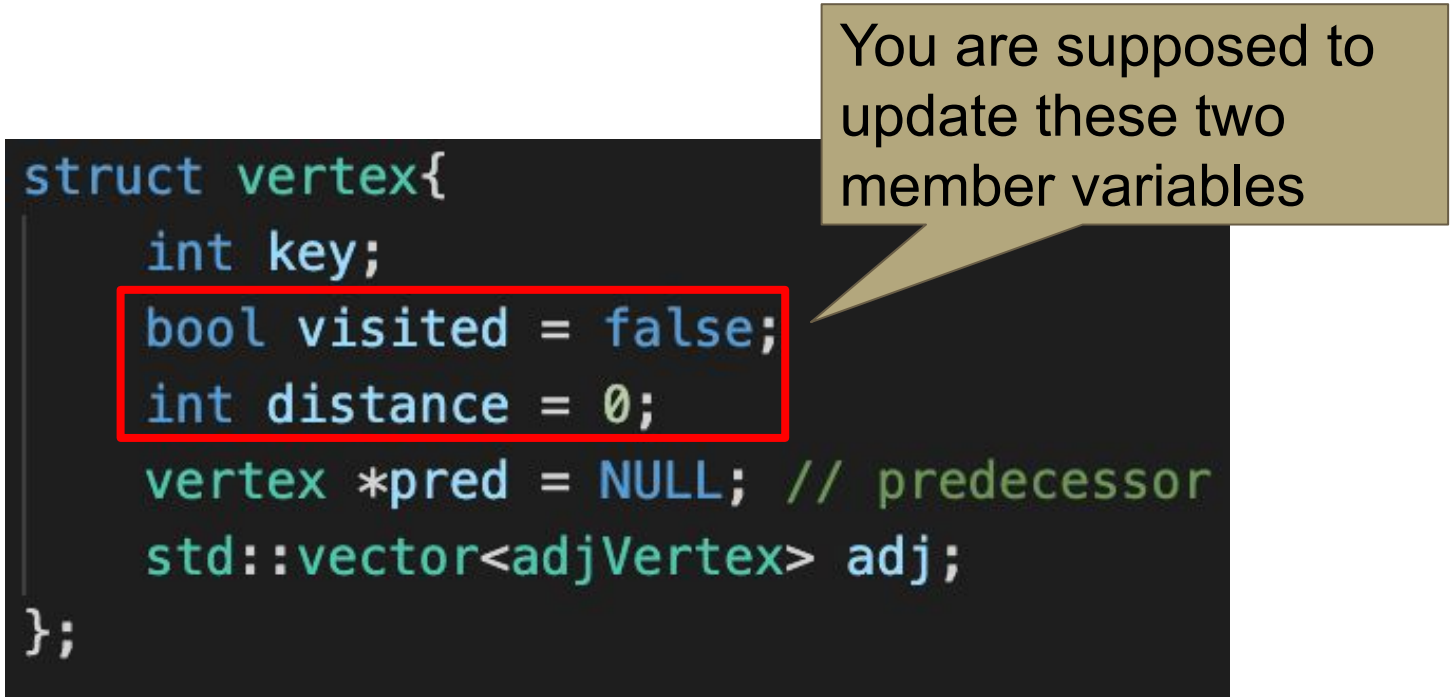


Recitation 10 Exercise (Silver Problem)

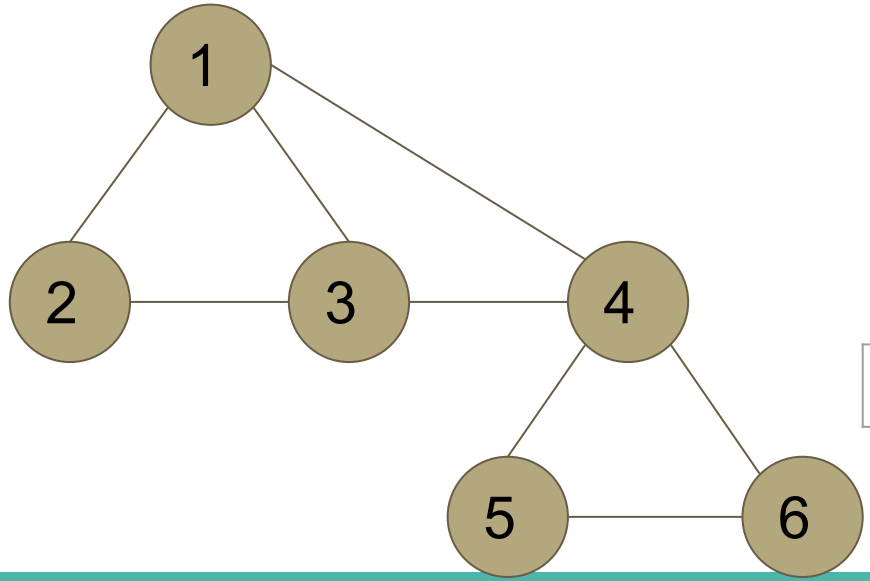
- Structure of a vertex is available in Graph.hpp
- I encourage you to read it to get a better understanding

```
struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

Recitation 10 Exercise (Silver Problem)



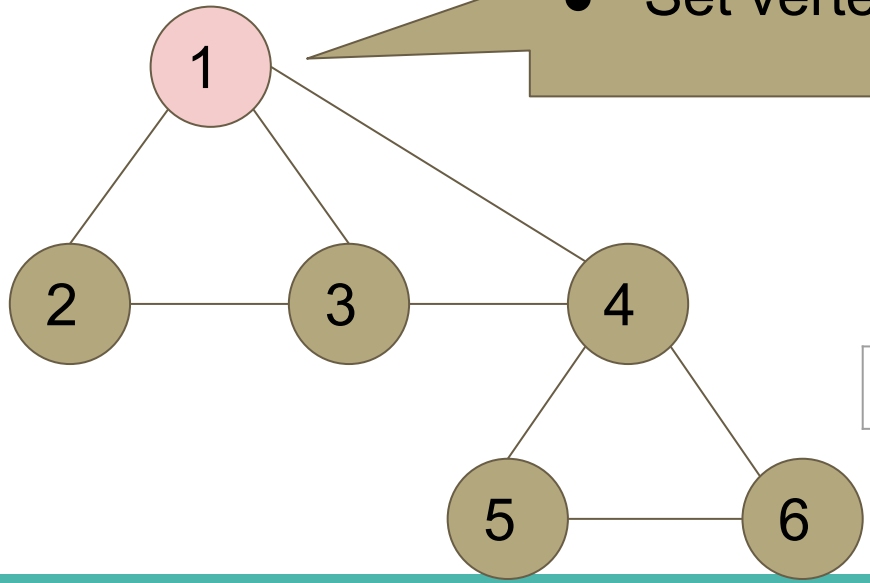
Recitation 10 Exercise (Silver Problem)



Queue

Recitation 10 Exercise (Silver Problem)

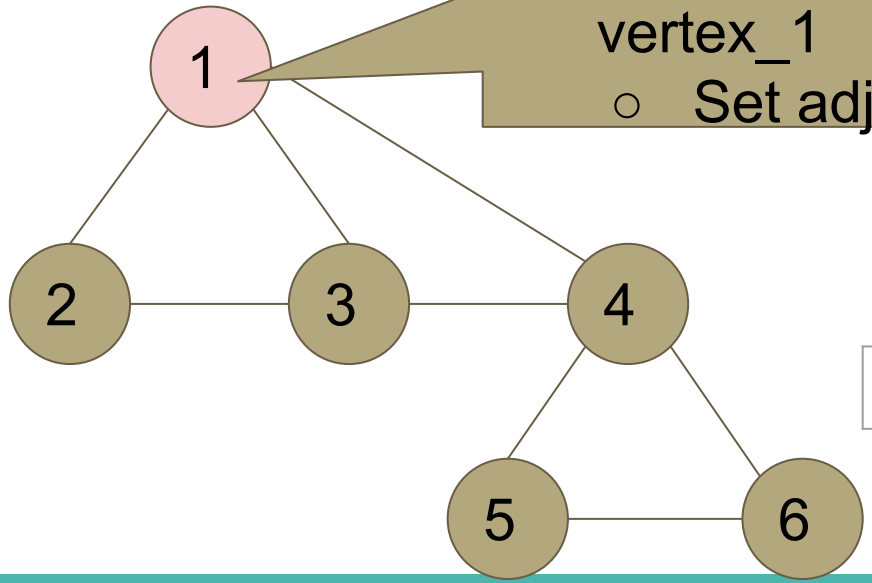
- Enqueue vertex_1
- Set vertex_1->visited = true



1					

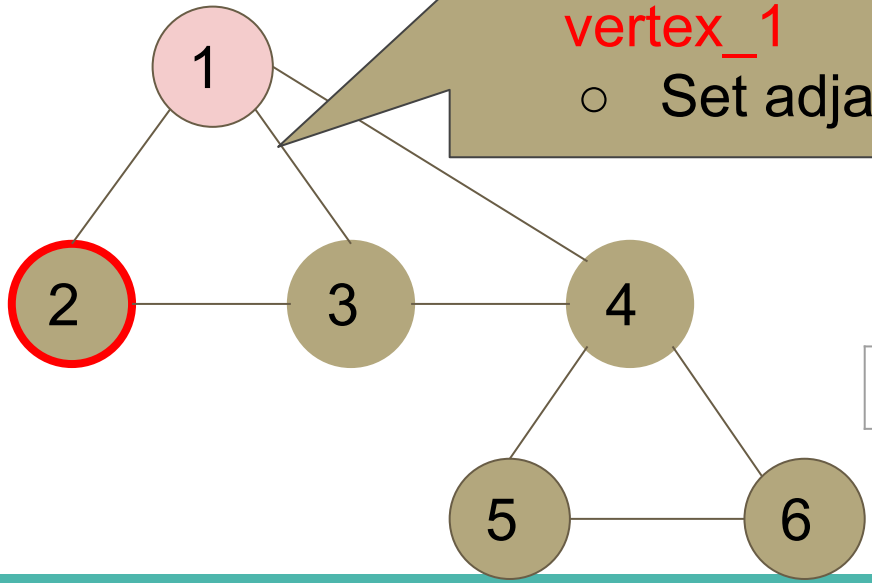
Recitation 10 Exercise (Silver Problem)

- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



Recitation 10 Exercise (Silver Problem)

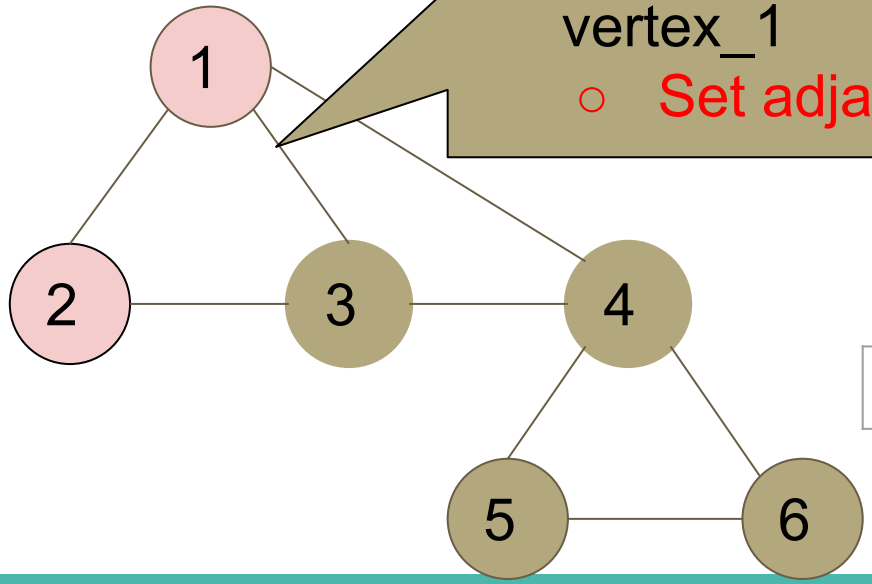
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



2					

Recitation 10 Exercise (Silver Problem)

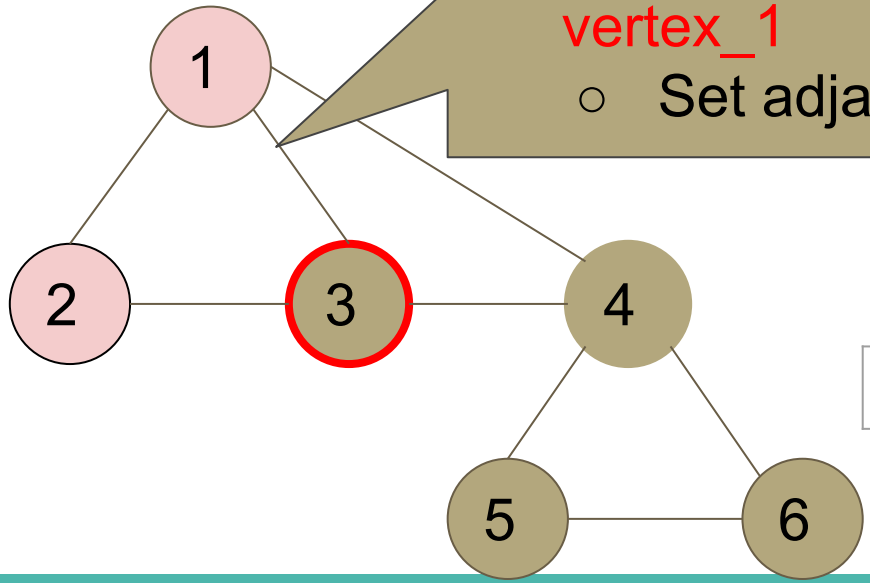
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



2					

Recitation 10 Exercise (Silver Problem)

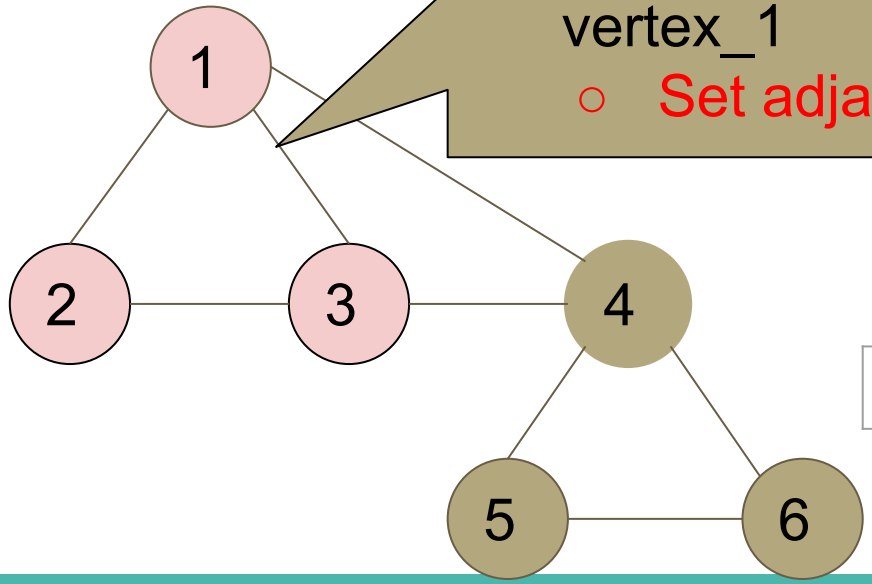
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



3	2				

Recitation 10 Exercise (Silver Problem)

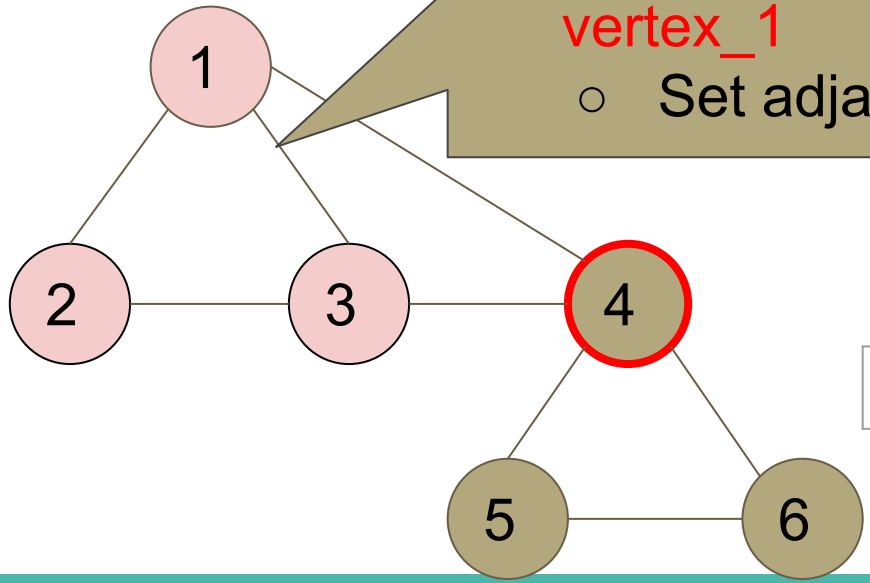
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



3	2				

Recitation 10 Exercise (Silver Problem)

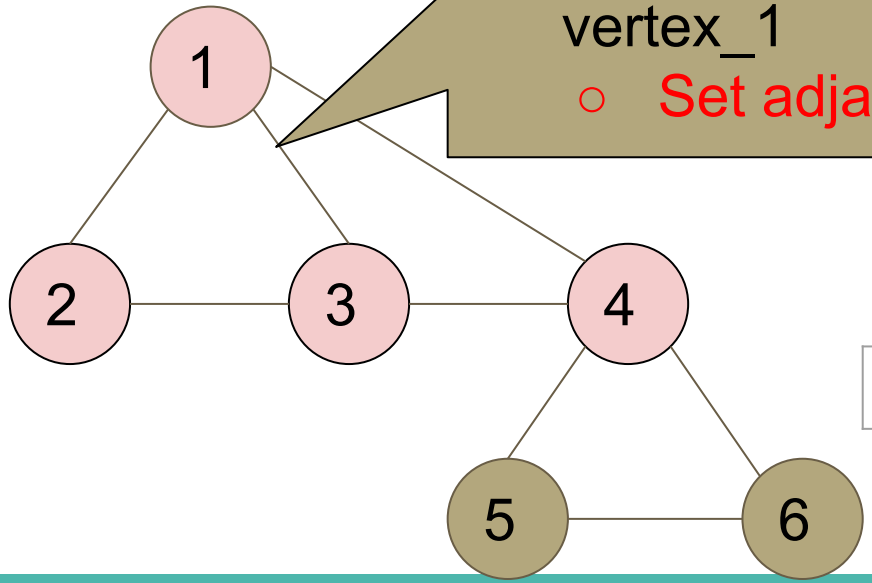
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



4	3	2			

Recitation 10 Exercise (Silver Problem)

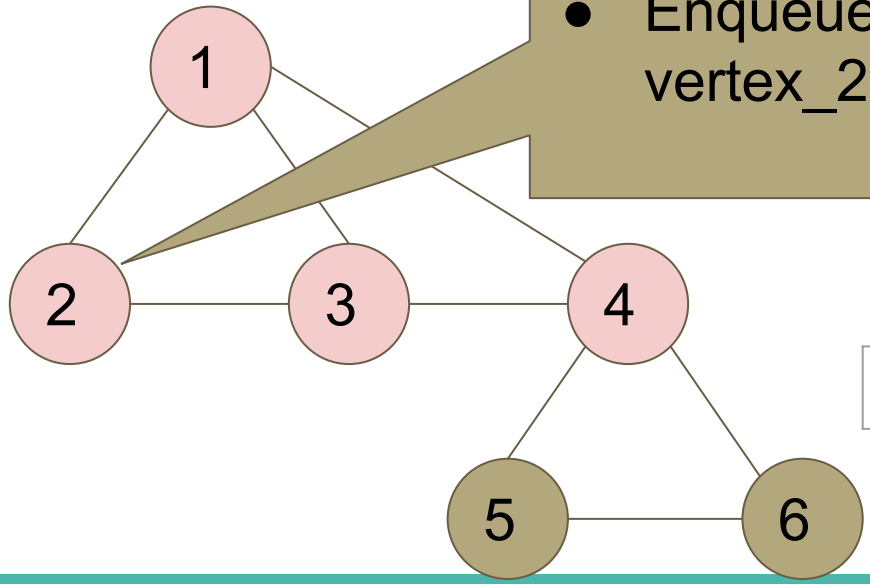
- Dequeue the front vertex_1
- Update vertex_1->distance
- Enqueue unvisited adjacent vertices of vertex_1
  - Set adjacent vertices->visited = true



4	3	2			

Recitation 10 Exercise (Silver Problem)

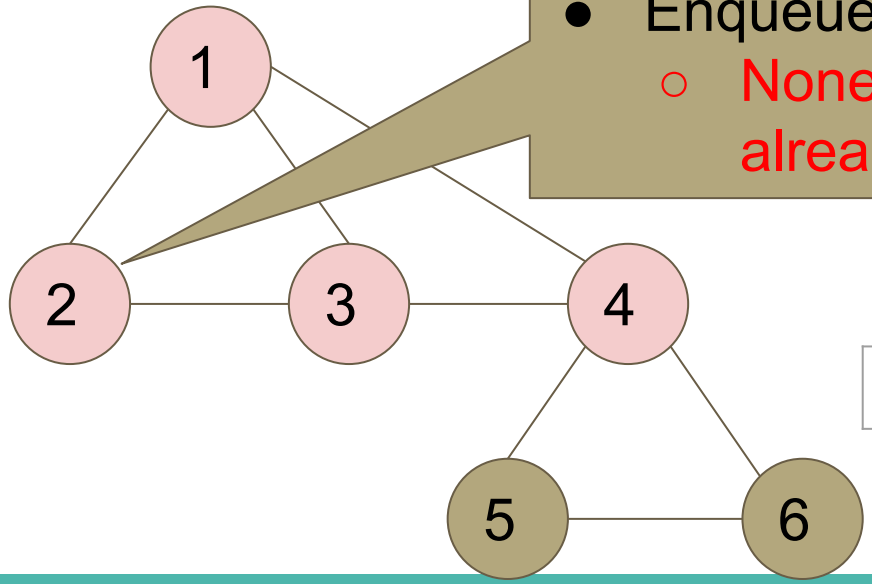
- Dequeue the front vertex_2
- Update vertex_2->distance
- Enqueue unvisited adjacent vertices of vertex_2



4	3				

Recitation 10 Exercise (Silver Problem)

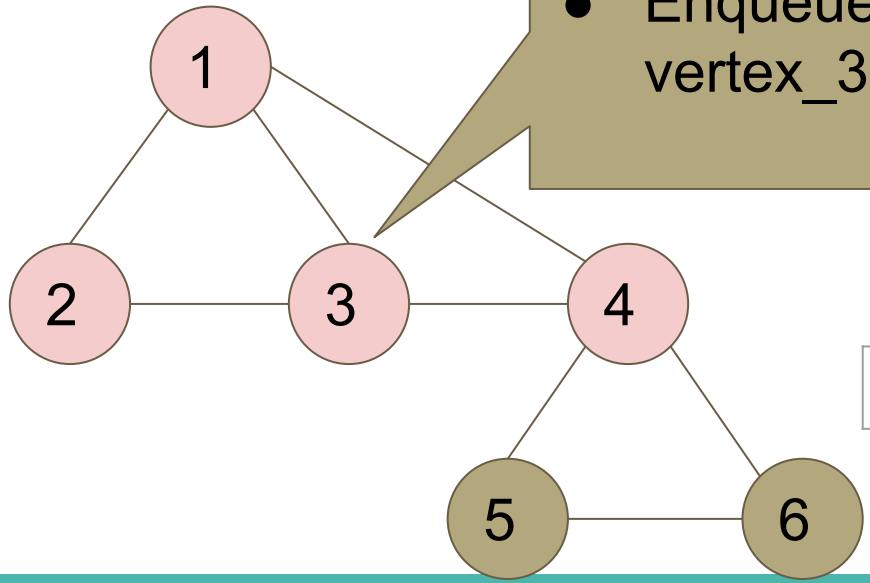
- Dequeue the front vertex_2
- Update vertex_2->distance
- Enqueue unvisited adjacent vertices
  - None! vertex_3 and vertex_1 are already visited



4	3				

Recitation 10 Exercise (Silver Problem)

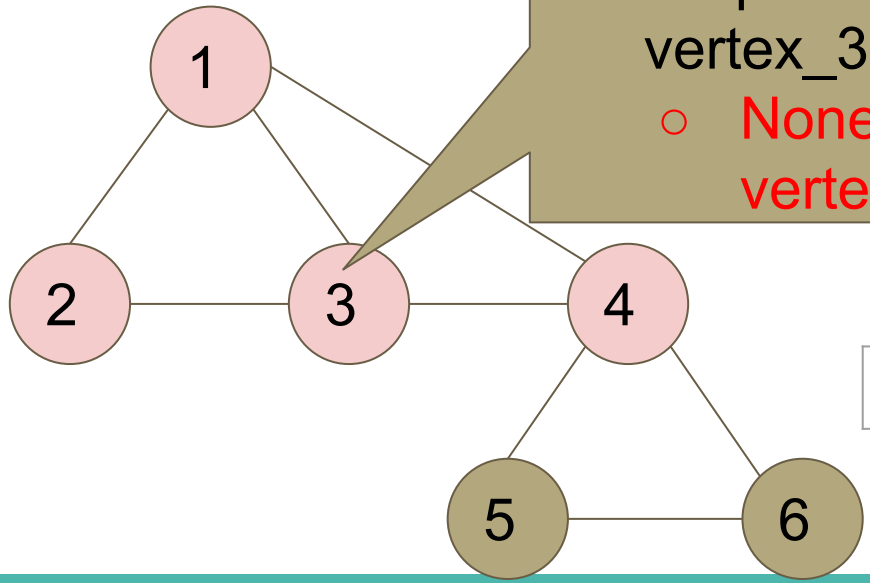
- Dequeue the front vertex_3
- Update vertex_3->distance
- Enqueue unvisited adjacent vertices of vertex_3



4					

Recitation 10 Exercise (Silver Problem)

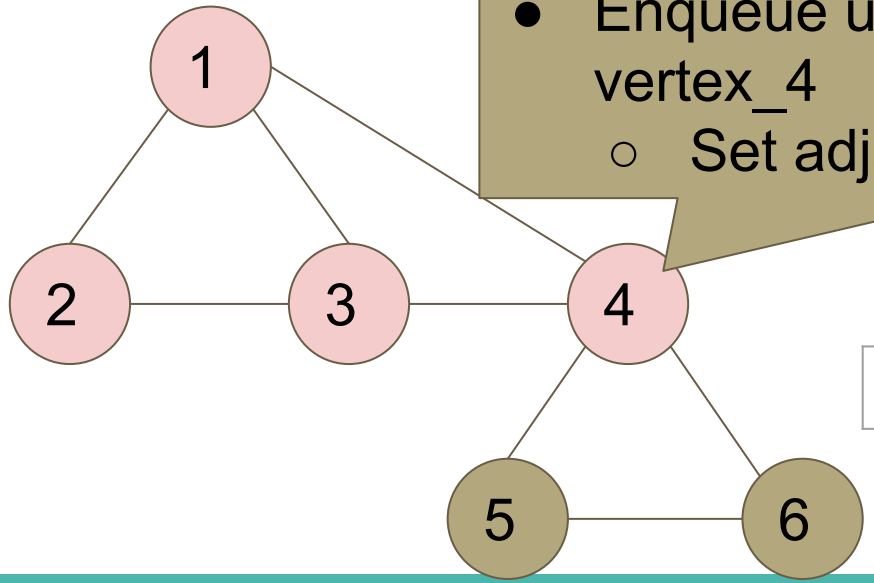
- Dequeue the front vertex_3
- Update vertex_3->distance
- Enqueue unvisited adjacent vertices of vertex_3
  - None! Vertex_1, vertex_2 and vertex_4 are already visited



4					

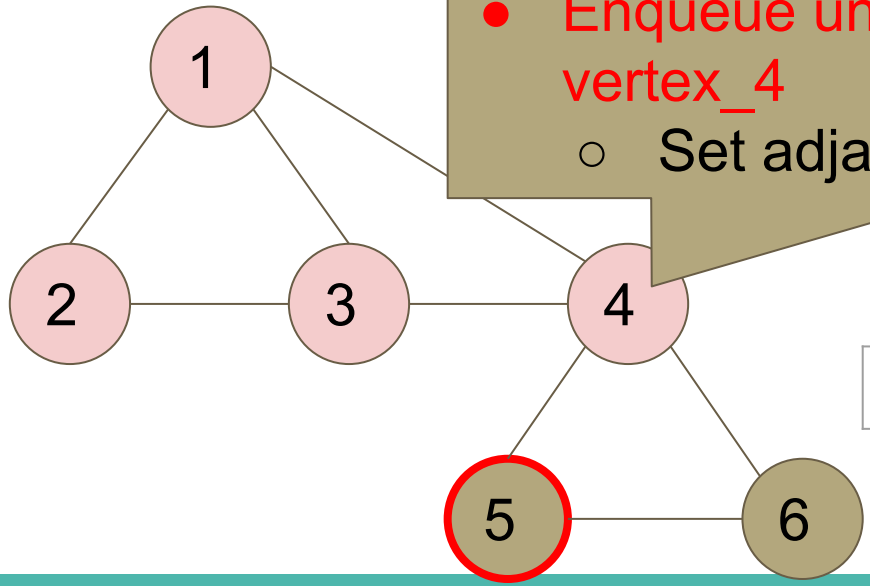
Recitation 10 Exercise (Silver Problem)

- Dequeue the front vertex_4
- Update vertex_4->distance
- Enqueue unvisited adjacent vertices of vertex_4
  - Set adjacent vertices->visited = true



Recitation 10 Exercise (Silver Problem)

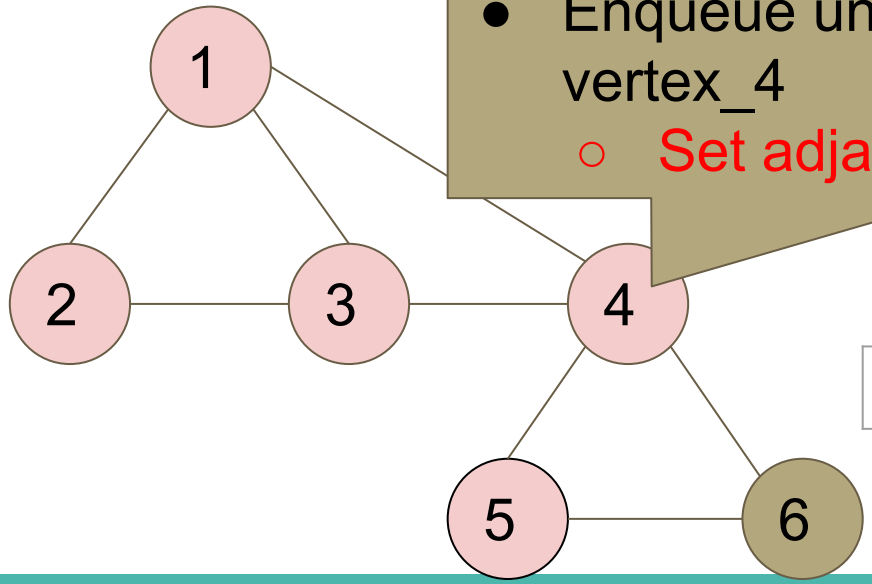
- Dequeue the front vertex_4
- Update vertex_4->distance
- Enqueue unvisited adjacent vertices of vertex_4
  - Set adjacent vertices->visited = true



5					

Recitation 10 Exercise (Silver Problem)

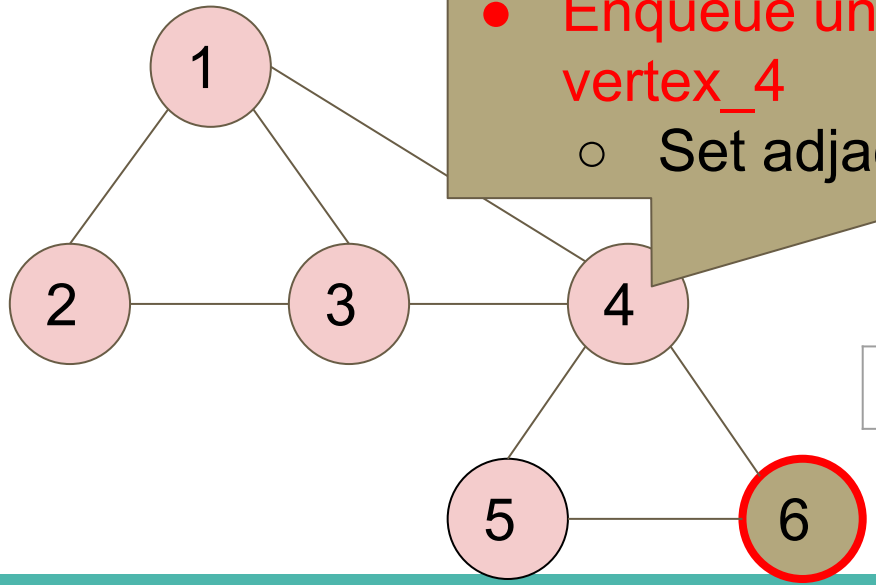
- Dequeue the front vertex_4
- Update vertex_4->distance
- Enqueue unvisited adjacent vertices of vertex_4
  - Set adjacent vertices->visited = true



5					

Recitation 10 Exercise (Silver Problem)

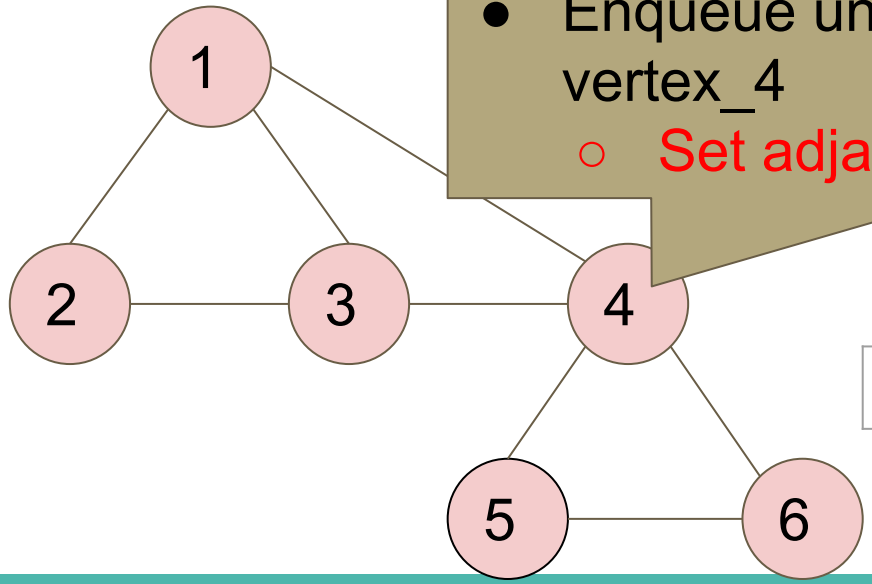
- Dequeue the front vertex_4
- Update vertex_4->distance
- Enqueue unvisited adjacent vertices of vertex_4
  - Set adjacent vertices->visited = true



6	5				

Recitation 10 Exercise (Silver Problem)

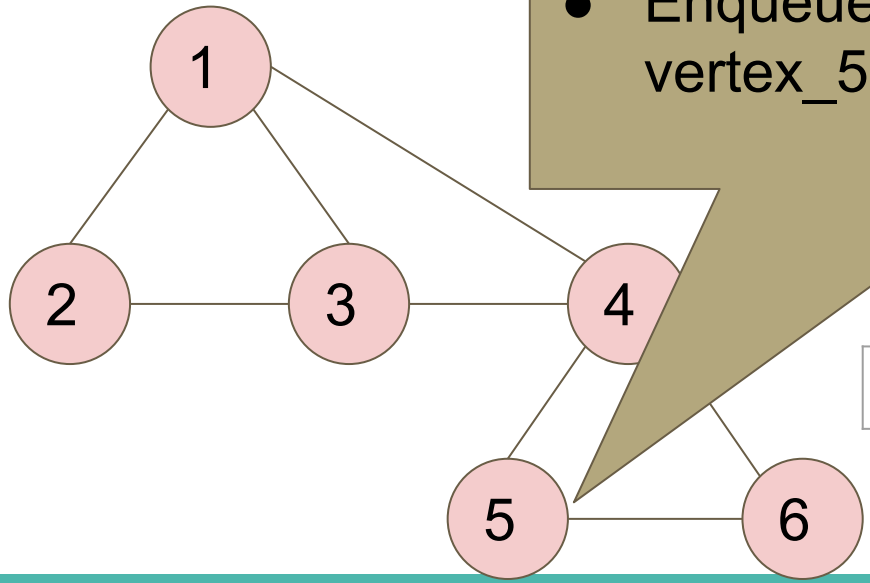
- Dequeue the front vertex_4
- Update vertex_4->distance
- Enqueue unvisited adjacent vertices of vertex_4
  - Set adjacent vertices->visited = true



6	5				

Recitation 10 Exercise (Silver Problem)

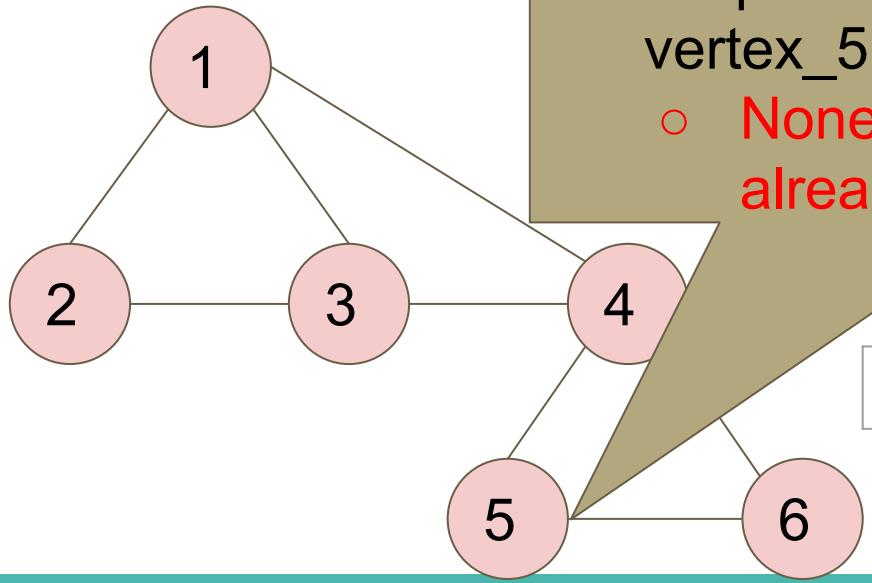
- Dequeue the front vertex_5
- Update vertex_5->distance
- Enqueue unvisited adjacent vertices of vertex_5



6					

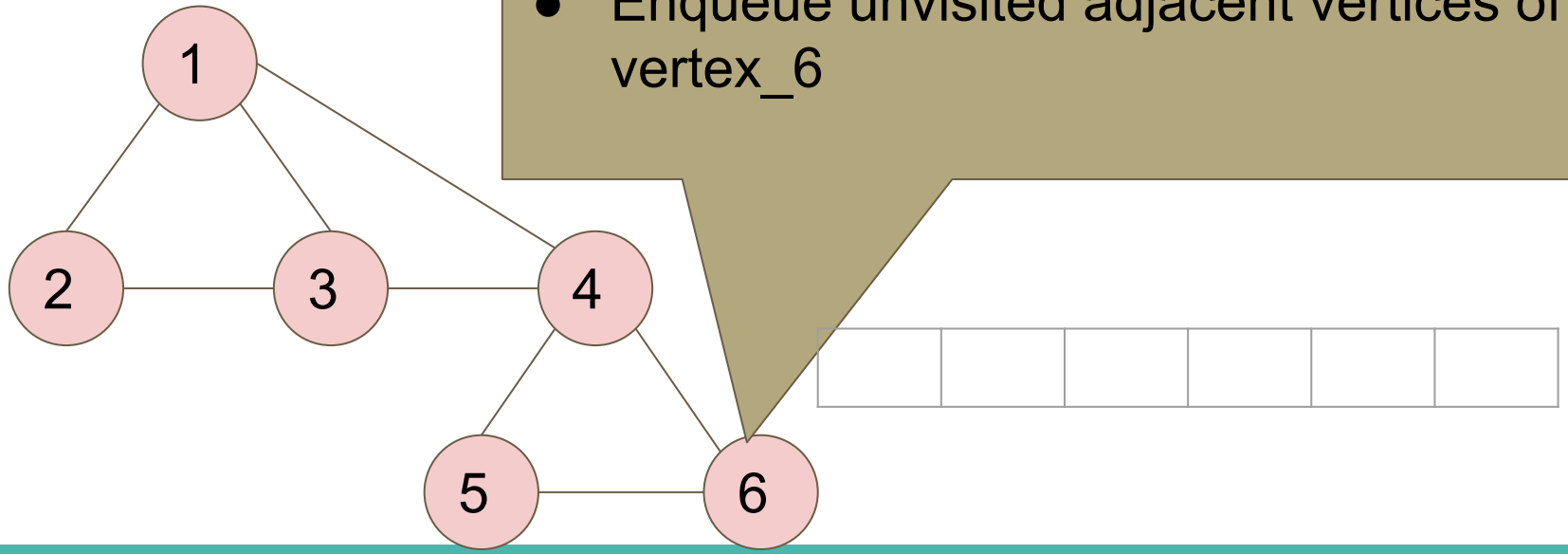
Recitation 10 Exercise (Silver Problem)

- Dequeue the front vertex_5
- Update vertex_5->distance
- Enqueue unvisited adjacent vertices of vertex_5
  - None! vertex_4 and vertex_6 are already visited



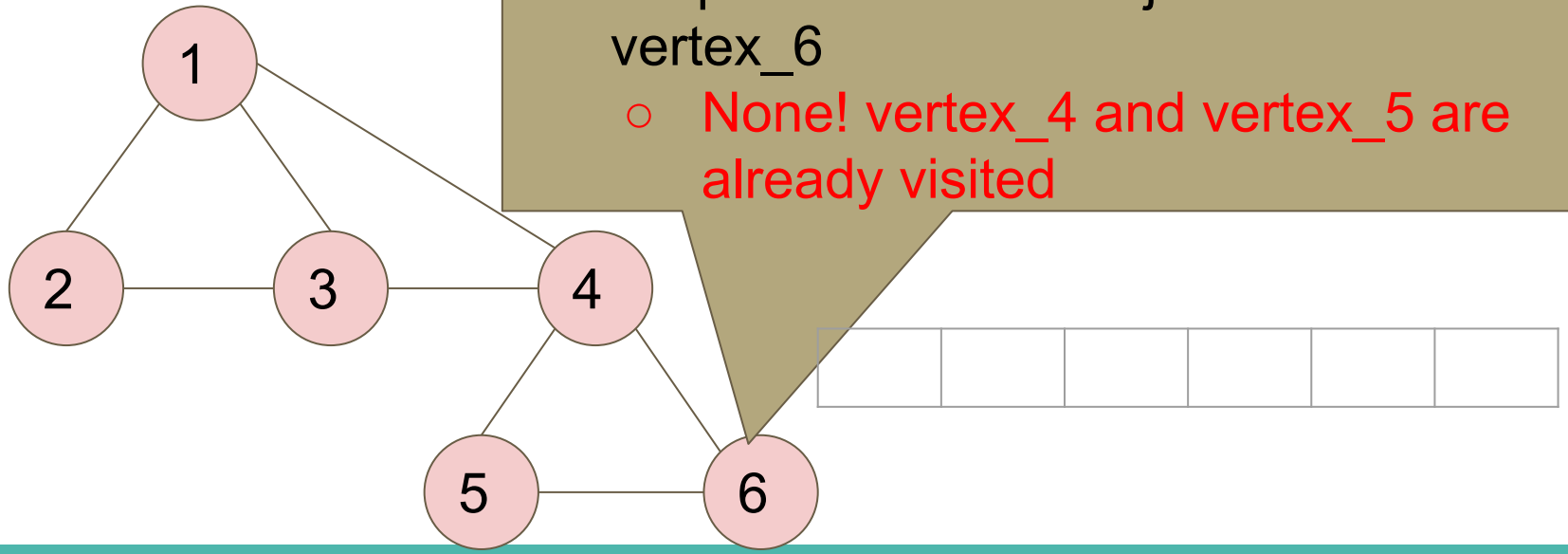
Recitation 10 Exercise (Silver Problem)

- Dequeue the front vertex_6
- Update vertex_6->distance
- Enqueue unvisited adjacent vertices of vertex_6

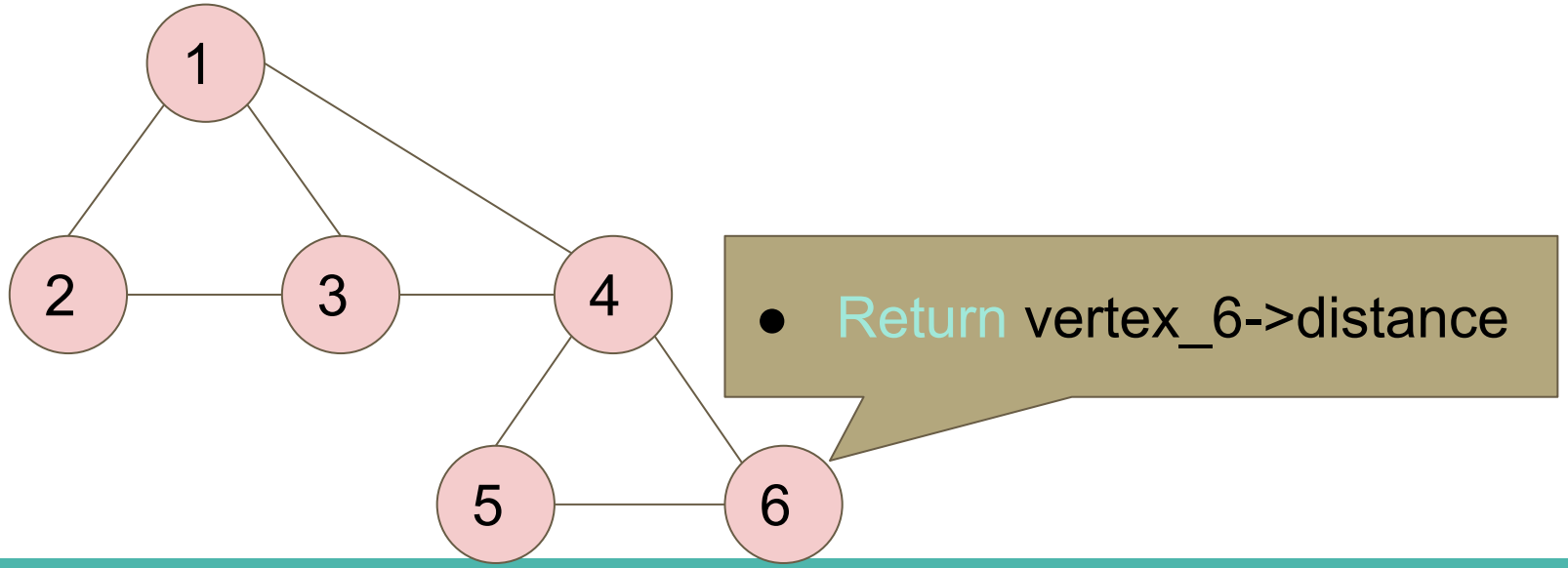


Recitation 10 Exercise (Silver Problem)

- Dequeue the front vertex_6
- Update vertex_6->distance
- Enqueue unvisited adjacent vertices of vertex_6
  - None! vertex_4 and vertex_5 are already visited



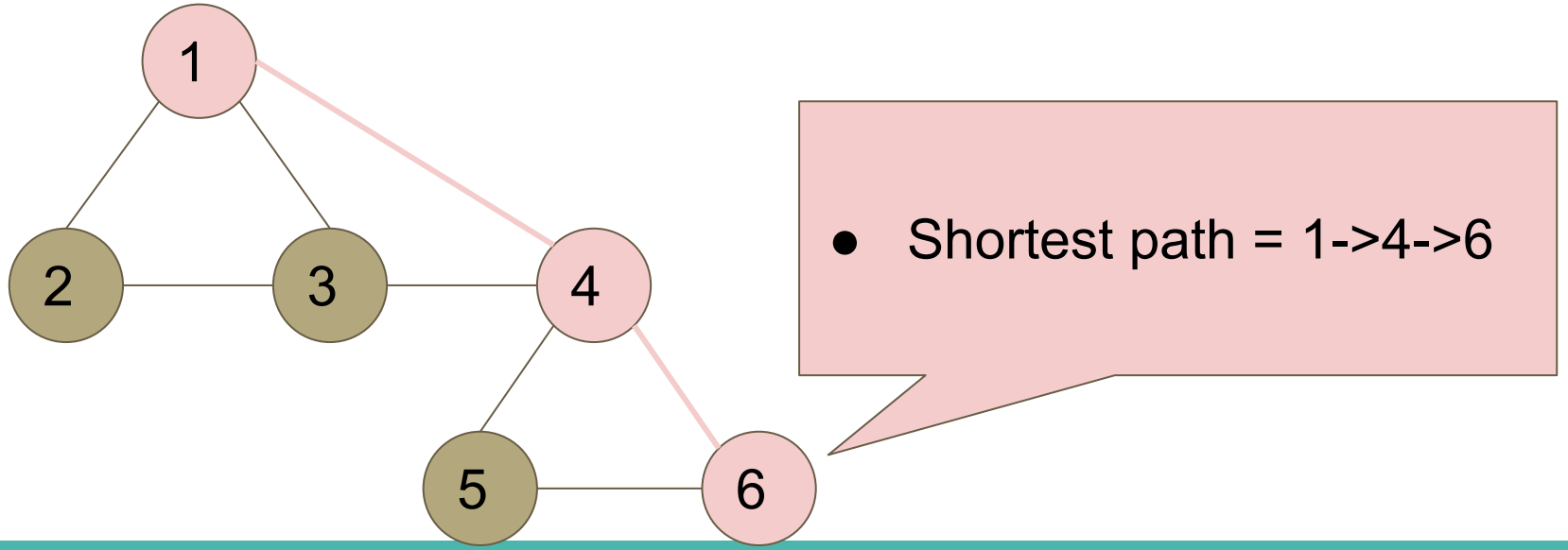
Recitation 10 Exercise (Silver Problem)



Any questions?

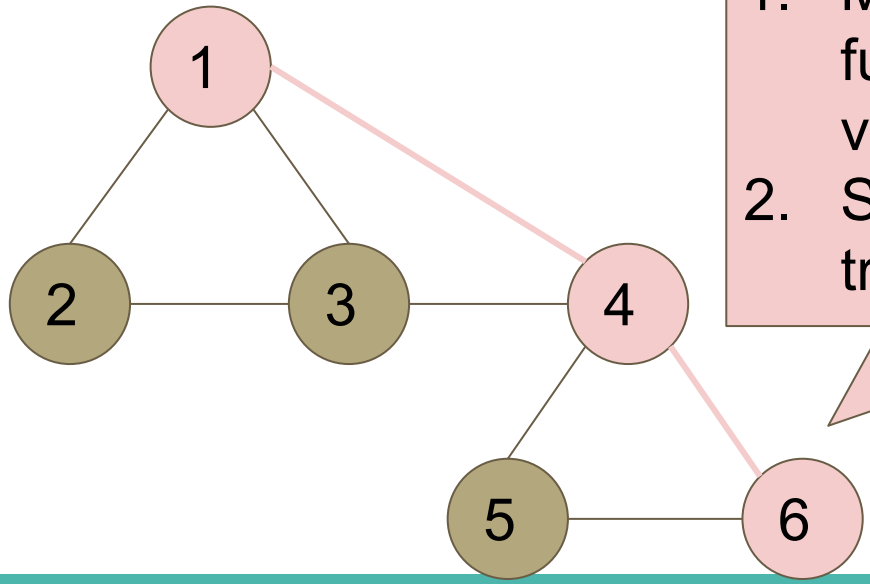
Recitation 10 Exercise (Gold Problem)

- Print the shortest path between a source and destination vertex



Recitation 10 Exercise (Gold Problem)

- Print the shortest path between a source and destination vertex



How to Solve:

1. Modify printShortestPath function to keep track of vertex->pred
2. Start from destination vertex, traverse through vertex->pred

Recitation 10 Exercise (Gold Problem)

```
struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

Need to update the predecessor node while traversing for Gold Problem

Expected Output of both Silver and Gold problems

```
Shortest path length = 2
The shortet path is:
1->4->6
```

START CODING

Also, stay safe and take care of your health!