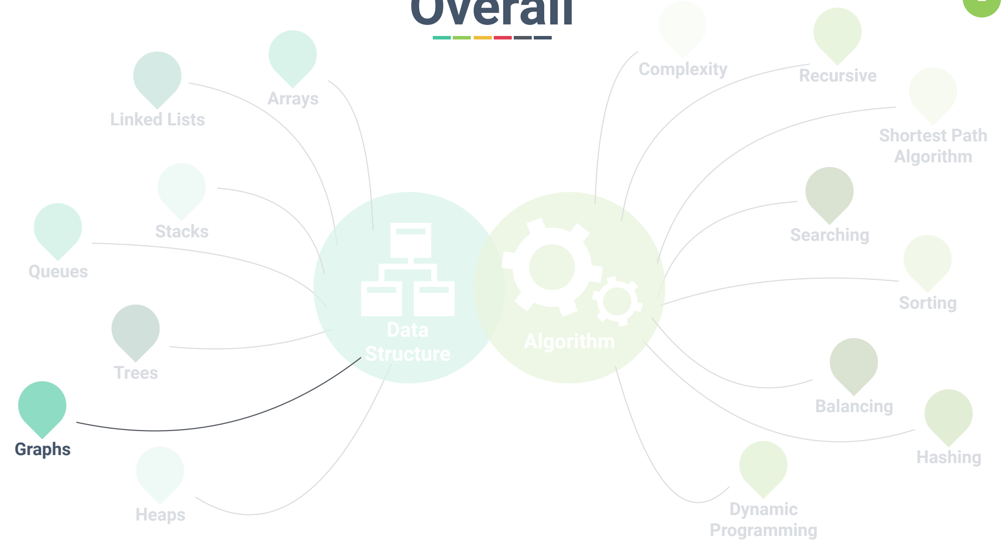
# Chapter 11: Graph Algorithms Part 1

Dr. Sirasit Lochanachit



# Overall






# Outline

Graphs

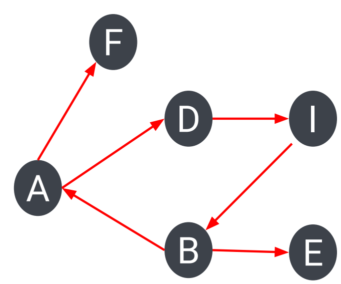
- Definition, elements and types

Graph Algorithms

- Traversal
- Minimum Spanning Tree (next week)
- Shortest Path



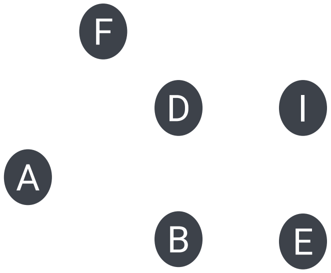
# Graphs




- A **graph** is a set of objects, called vertices or **nodes**, where the actual data is stored and a collection of connections between them, called **edges** or arcs.
- A graph can be used to represent relationships between pairs of objects.
- Applications that require efficient processing between networks such as mapping, transportation and computer networks (Internet).

# The Basic Elements of Graph

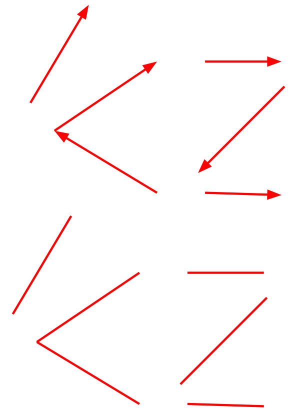





- Formally, a graph *G* is a set *V* of vertices and a collection *E* of pairs of vertices, called edges.
- V = {A, B, D, E, F, I}

# The Basic Elements of Graph

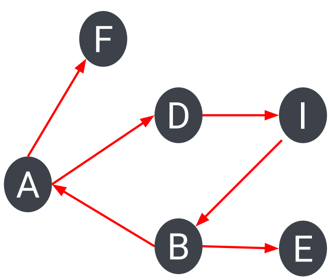




- Formally, a graph *G* is a set *V* of vertices and a collection *E* of pairs of vertices, called edges.

# The Basic Elements of Graph

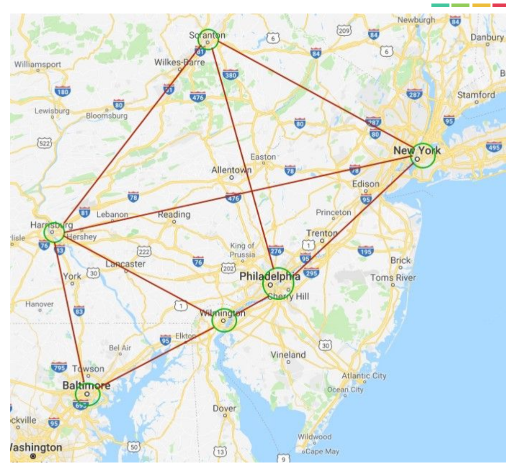


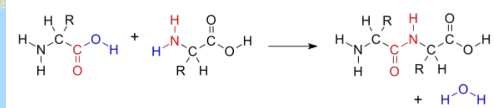



- Formally, a graph *G* is a set *V* of vertices and a collection *E* of pairs of vertices, called edges.
- V = {A, B, D, E, F, I}
- E = {(A, F), (A, D), (B, A), (D, I), (I, B), (B, E)}

# Graphs

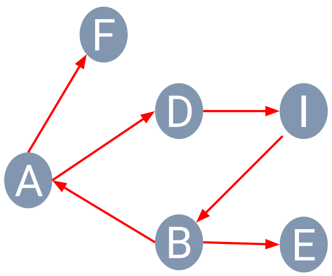




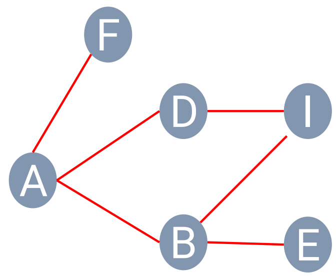



Ref: https://miro.medium.com/max/700/1*2jL0bKbT8ttskAIEVDv1yg.jpeg

# Graph Types




Directed

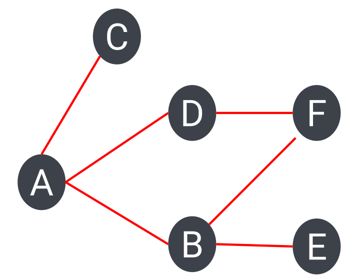



Undirected

- An edge (*u*, *v*) is directed from *u* to *v* if the pair (*u*, *v*) is ordered.
- An edge (*u*, *v*) is undirected if the pair (*u*, *v*) is not ordered.



# Graph Representation



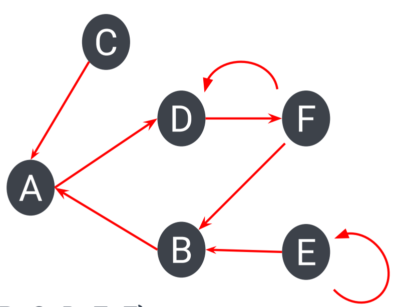

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 0 | 0 | 1 | 1 |
| C | 1 | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 1 | 0 | 0 | 0 | 0 |
| F | 0 | 1 | 0 | 1 | 0 | 0 |

Adjacency Matrix

- V = {A, B, C, D, E, F}
- E = {(A, C), (A, D), (A, B), (D, F), (B, E), (B, F)}



# Graph Representation



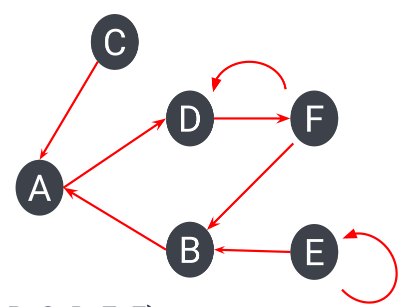

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 1 | 0 | 0 |
| B | 1 | 0 | 0 | 0 | 0 | 0 |
| C | 1 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 1 | 0 | 0 | 1 | 0 |
| F | 0 | 1 | 0 | 1 | 0 | 0 |

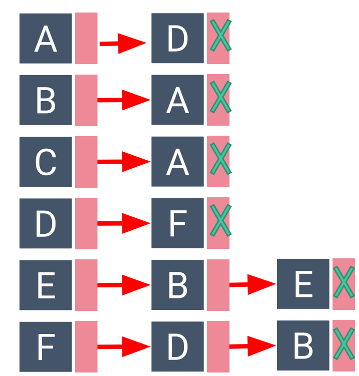
Adjacency Matrix

- V = {A, B, C, D, E, F}
- E = {(C, A), (A, D), (B, A), (D, F), (F, D), (E, B), (F, B), (E, E)}



# Graph Representation




Adjacency List

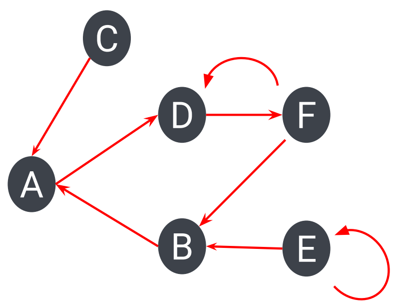
- V = {A, B, C, D, E, F}
- E = {(C, A), (A, D), (B, A), (D, F), (F, D), (E, B), (F, B), (E, E)}

# Graph Representation



Node Representation

| Node | Name | Phone |
|---|---|---|
| A | Able | |
| B | Baker | |
| C | Charlie | |
| D | Denver | |
| E | Ethan | |
| F | Fred | |



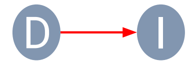

Edge Representation

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 1 | 0 | 0 |
| B | 1 | 0 | 0 | 0 | 0 | 0 |
| C | 1 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 1 | 0 | 0 | 1 | 0 |
| F | 0 | 1 | 0 | 1 | 0 | 0 |

Adjacency Matrix

# Graph Notations





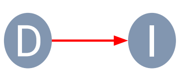

V = {D, I}

E = {(D, I)}

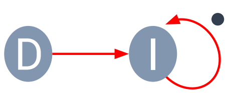
- **Endpoints**: Two nodes (*u*, *v*) that are joined by an edge.
    - These two nodes are **adjacent**.
- **Origin**: First endpoint (*u*) on a directed edge.
- **Destination**: Second endpoint (*v*) on a directed edge.

# Graph Notations





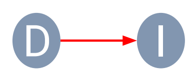

- A **path** is a sequence of nodes and edges that starts at a node and ends at a node such that each node is adjacent to the next one.
- Formally, a path is a sequence of nodes $V_1, V_2, V_3, \ldots, V_n$ where $(V_1, V_2), (V_2, V_3), \ldots, (V_{n-1}, V_n) \in E$.




- A **loop** is a special case of path where two endpoints are the same.
    - An edge that starts and ends with the same node.

# Graph Notations

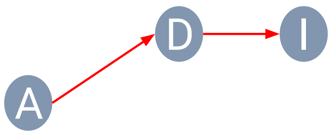





- A **cycle** is a path that starts and ends at the same node, having at least one edge.
- A **simple path** is when each node in the path is distinct.
- A **simple cycle** is when each node in the cycle is distinct, except for the first and last one.

# Graph Properties

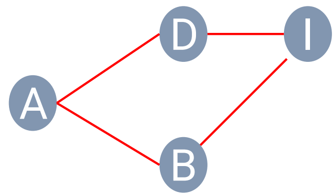





A, D, I

Simple Path

Acyclic (No Cycle)



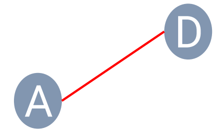

A, D, I, B, A

Non Simple Path

Undirected Simple Cycle

# Graph Properties

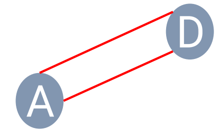





A, B, A

Non Simple Path

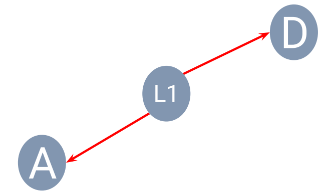
Acyclic (No Cycle)

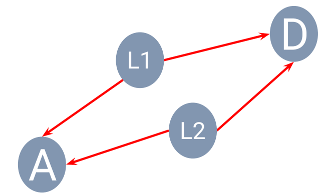



A, D, A

Non Simple Path

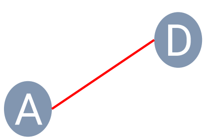
Acyclic (No Cycle)




# Graph Properties

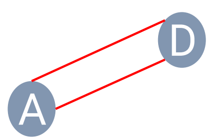





A, B, A

Non Simple Path

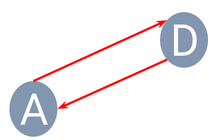
Acyclic (No Cycle)




A, D, A

Non Simple Path

Acyclic (No Cycle)



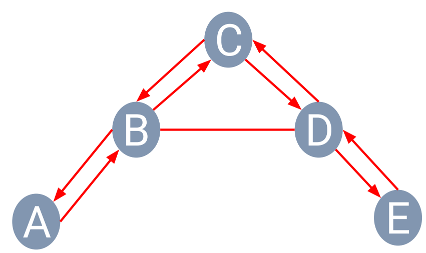

A, D, A

Non Simple Path

Directed cycle

# Graph Properties

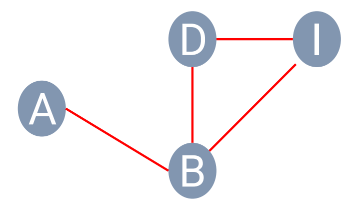





A, B, C, D, E, D, B, A

Non Simple Path

Cycle



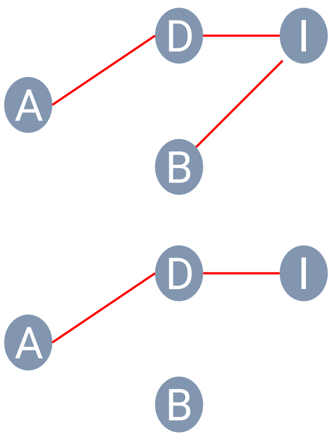

A, B, I, D, B, A

Non Simple Path

Undirected Cycle

# Graph Notations






- A graph is **connected** if, for any two nodes, there is a path between them.
- The **in-degree** of a node v is the number of the incoming edges of v.
- The **out-degree** of a node v is the number of the outgoing edges of v.
- For instance, node D has in-degree = 2, out-degree = 2.

# Graph Algorithms



- Traversals
- Minimum Spanning Tree
- Shortest Path

# Graph Traversals



- A **traversal** is a systematic procedure for exploring a graph by examining all of its nodes and edges.
- Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of **reachability**, that is, in determining how to travel from one node to another while following paths of a graph.
- Two efficient graph traversal algorithms: **depth-first search** and **breadth-first search**.

# Depth-First Search (DFS)



- Imagine wandering in a labyrinth/maze with a string and a can of paint.
- Starts at node *s* in graph *G*, fix one end of the string to *s* and paint *s* as “visited”.
  - The node *s* is now a “current” node, which is a current node *u*.
  - Painting is analogous to putting a visited node in a stack.
- Then, traverse *G* by considering an edge (*u*, *v*) that is connected to *u*.
  - If the edge (*u*, *v*) leads to a node *v* that is already existed/painted, ignore.
  - Otherwise, if (*u*, *v*) leads to an unvisited node *v*, then unroll the string and go to *v*. Then painted *v* as “visited” and make it the current node *u*.
- Repeat the step above until reaching a “dead end”, that is, a current node *v* such that all the edges connected to *v* lead to nodes already visited.

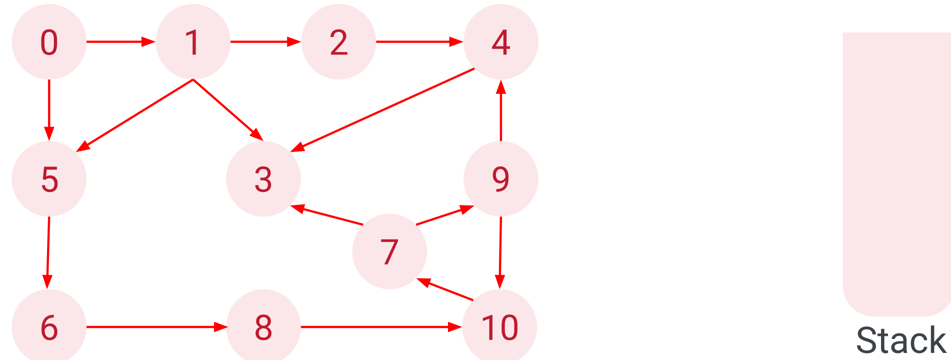
# Depth-First Search (DFS)



- To get out of the “dead end”, roll the string back up, backtracking along the edge to a previously visited node *u*.
- Then make *u* the current node and repeat the traversal step for any edges connected to *u* that have not yet considered.
- The traversal is continued until the process terminates when the backtracking leads back to the start node *s*, and there are no more unexplored edges connected to *s*.
- DFS can have many solutions, however, for consistency, when there are multiple edges available, the node with fewer values should be selected first.

# Depth-First Search (DFS)






“Parent ➪ Children ➪ Grandchild”

Output : ?

# Depth-First Search (DFS)




Output : 0

# Depth-First Search (DFS)




Output : 0

# Depth-First Search (DFS)


Output : 0 1

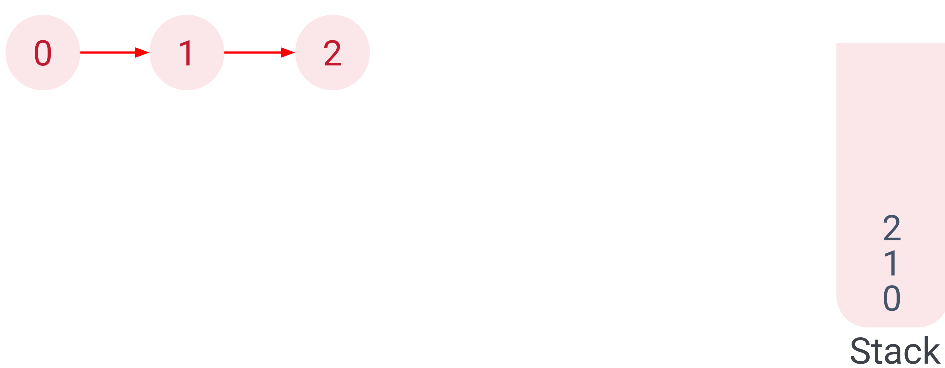


# Depth-First Search (DFS)


Output : 0 1



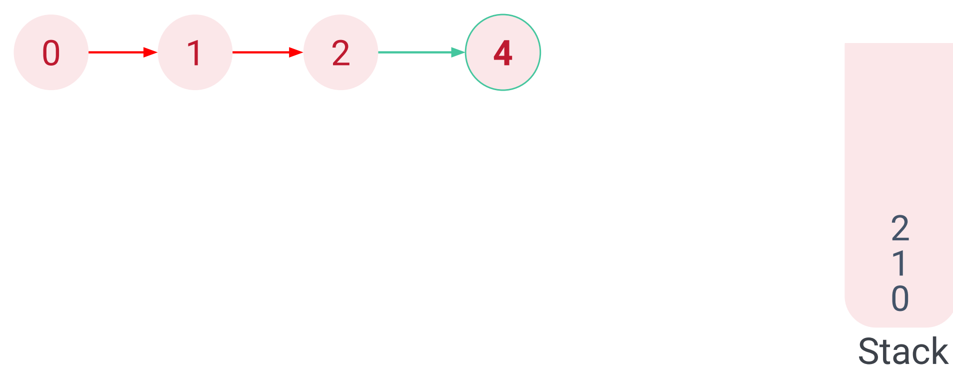
# Depth-First Search (DFS)




Output : 0 1 2

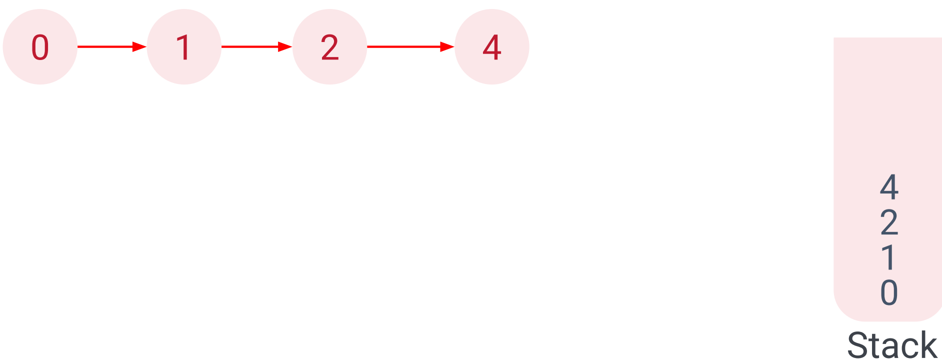


# Depth-First Search (DFS)




Output : 0 1 2

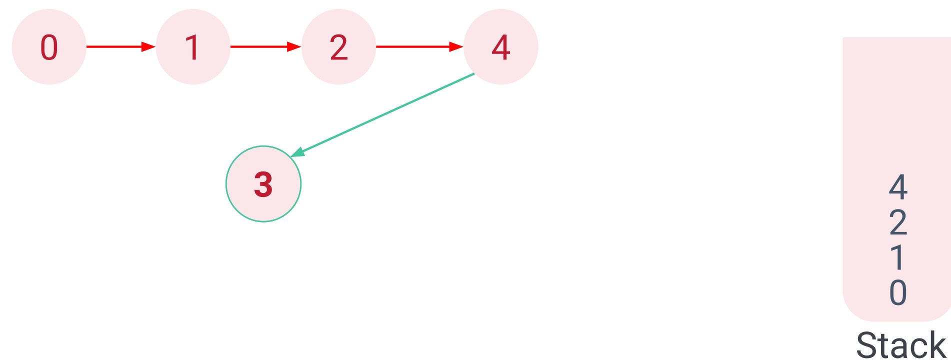
# Depth-First Search (DFS)




Output : 0 1 2 4



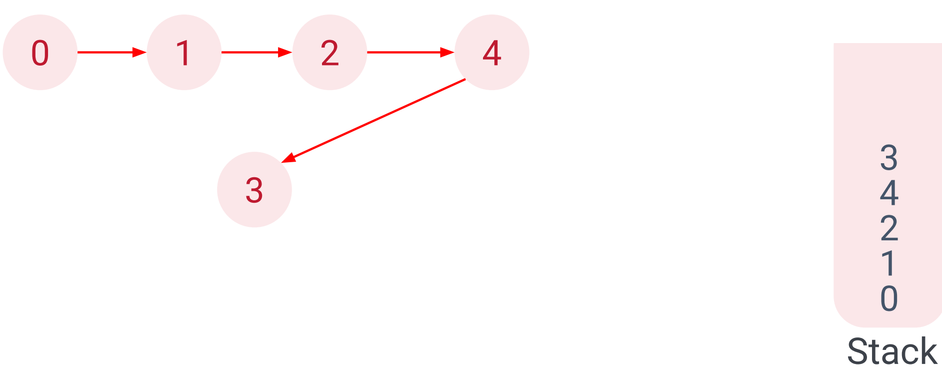
# Depth-First Search (DFS)




Output : 0 1 2 4



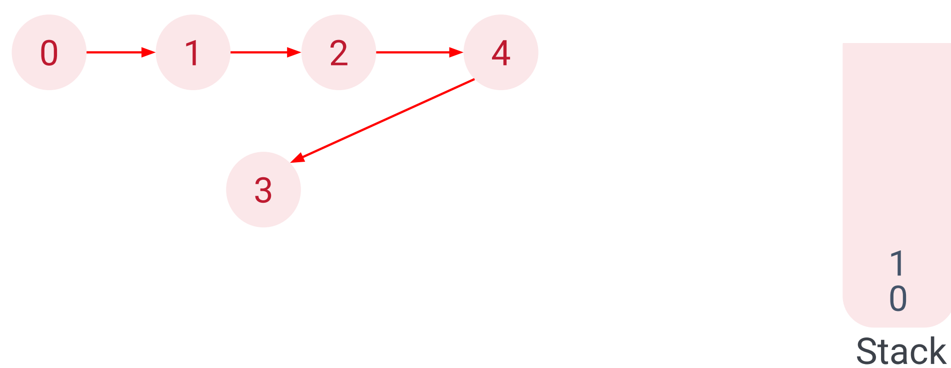
# Depth-First Search (DFS)




Output : 0 1 2 4 3

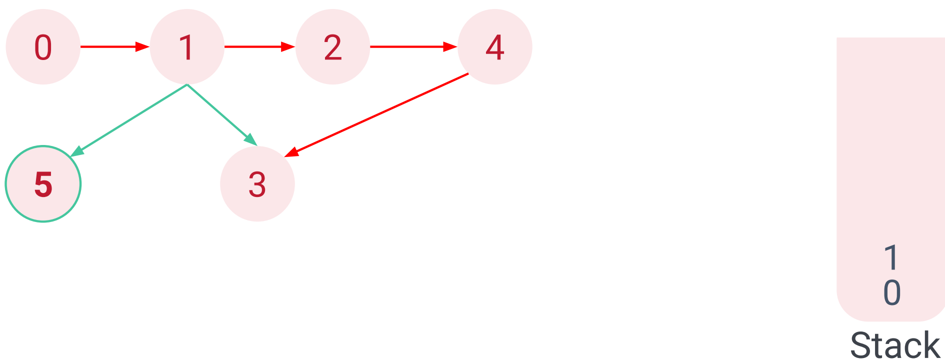


# Depth-First Search (DFS)




Output : 0 1 2 4 3

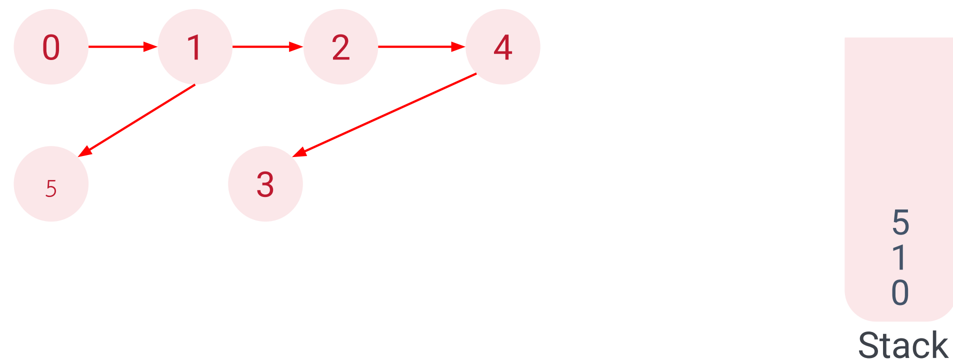
# Depth-First Search (DFS)




Output : 0 1 2 4 3



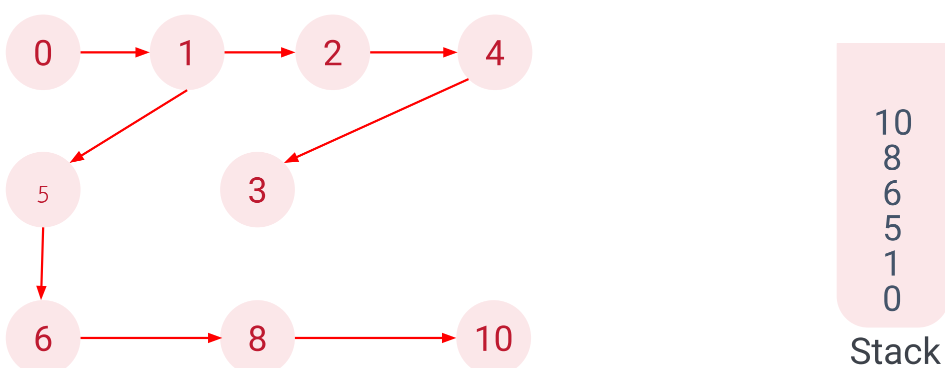
# Depth-First Search (DFS)




Output : 0 1 2 4 3 5



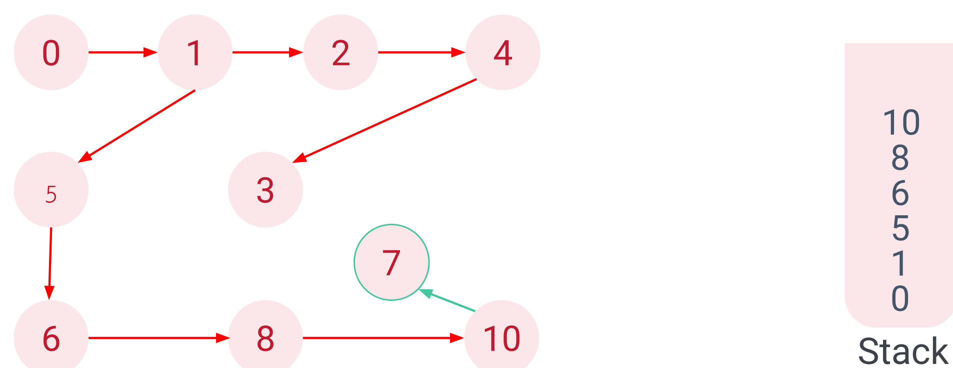
# Depth-First Search (DFS)




Output : 0 1 2 4 3 5 6 8 10

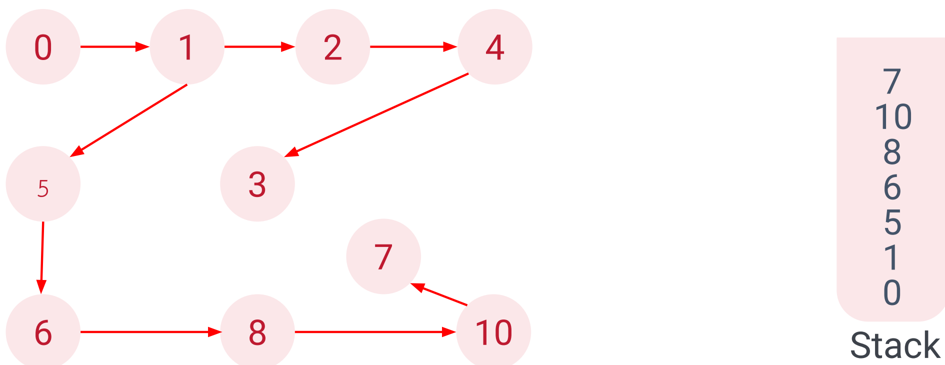


# Depth-First Search (DFS)




Output : 0 1 2 4 3 5 6 8 10

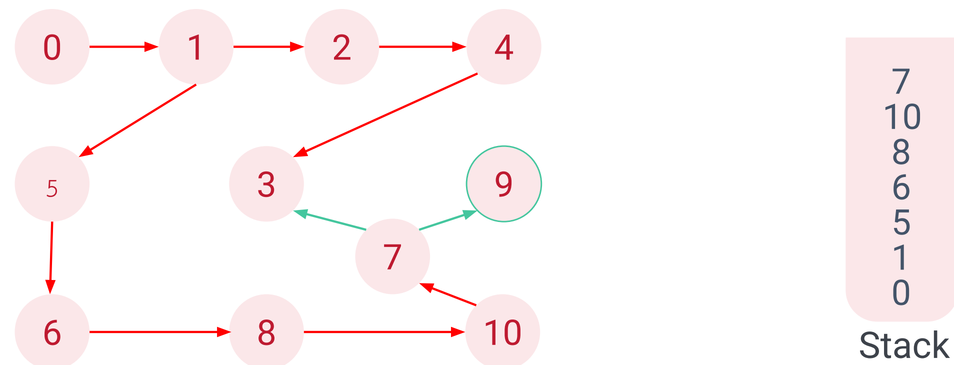
# Depth-First Search (DFS)




Output : 0 1 2 4 3 5 6 8 10 7



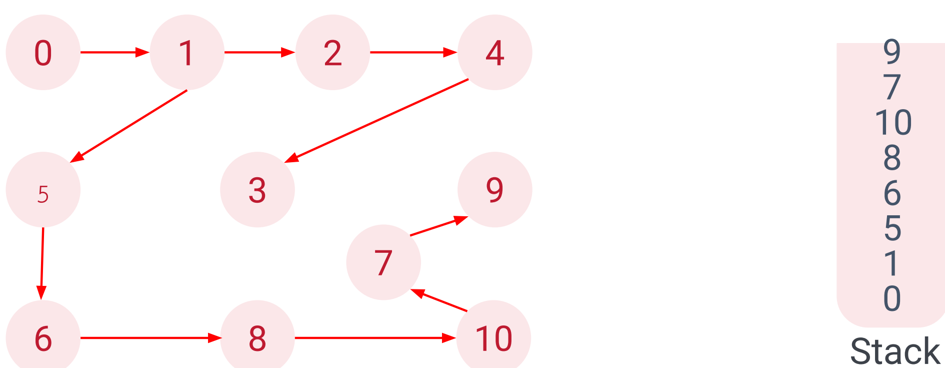
# Depth-First Search (DFS)




Output : 0 1 2 4 3 5 6 8 10 7



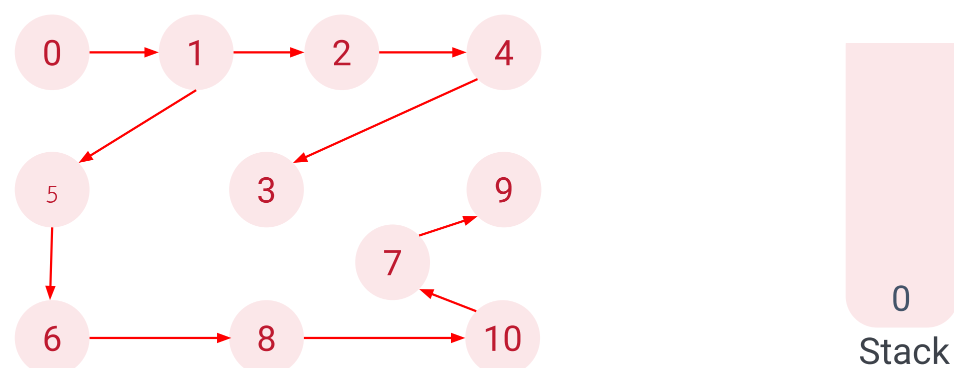
# Depth-First Search (DFS)




Output : 0 1 2 4 3 5 6 8 10 7 9

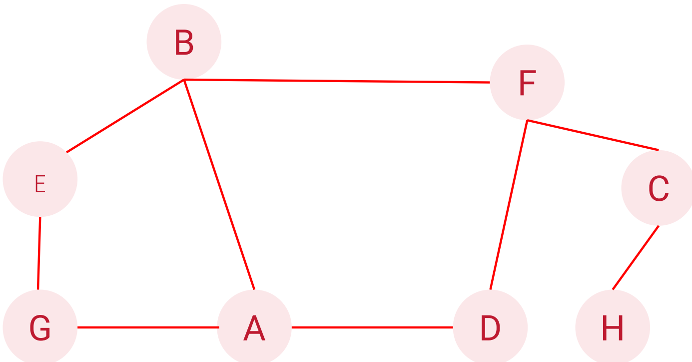


# Depth-First Search (DFS)




**Output : 0 1 2 4 3 5 6 8 10 7 9**

# Depth-First Search (DFS) Exercise




**Output : ?**



# Breadth-First Search (BFS)

- DFS is similar to sending a single person to navigate the graph.
- BFS is more akin to sending out many people in all directions to traverse a graph in coordinated fashion.
- A BFS proceeds in rounds and subdivides the nodes into **levels**.
- Starts at node *s*, which is level 0.
  - 1st Round: paint all nodes adjacent to node *s* as “visited” and placed into level 1.
  - 2nd Round: All nodes adjacent to level 1’s nodes are placed into level 2 and marked as “visited”.
  - This process continues until no new nodes are found in a level.

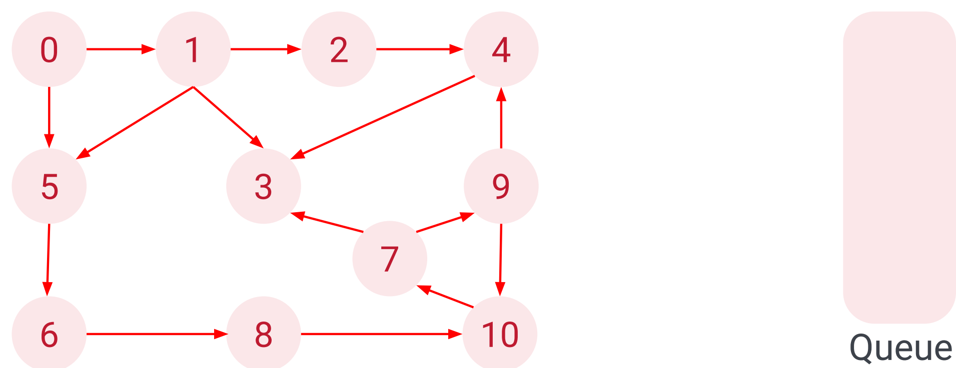


# Breadth-First Search (BFS)

- First, starts with the initial node anywhere on the graph, which is the starting point.
  - Usually starts with a node with lowest value.
- From there, add (enqueue) every node that is directly connected to the current node into a queue.
- Then dequeue the current node and check the next node in queue.



# Breadth-First Search (BFS)




“Parent ⇨ Children ⇨ Grandchild”

Output : ?

# Breadth-First Search (BFS)


Output : 0



# Breadth-First Search (BFS)


Output : 0



# Breadth-First Search (BFS)

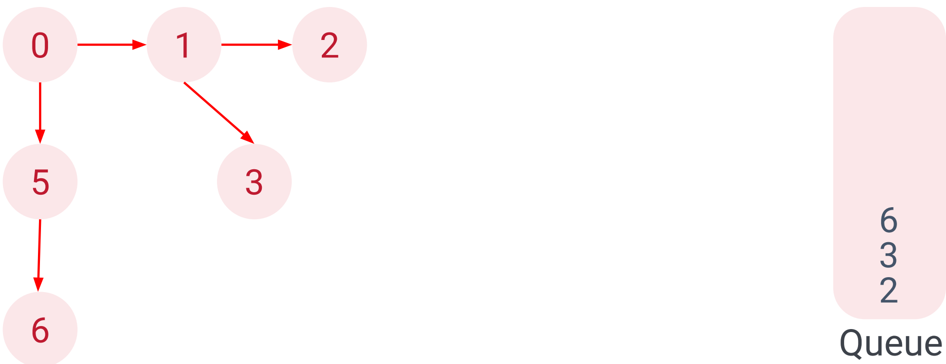

Output : 0 1 5



# Breadth-First Search (BFS)


Output : 0 1 5

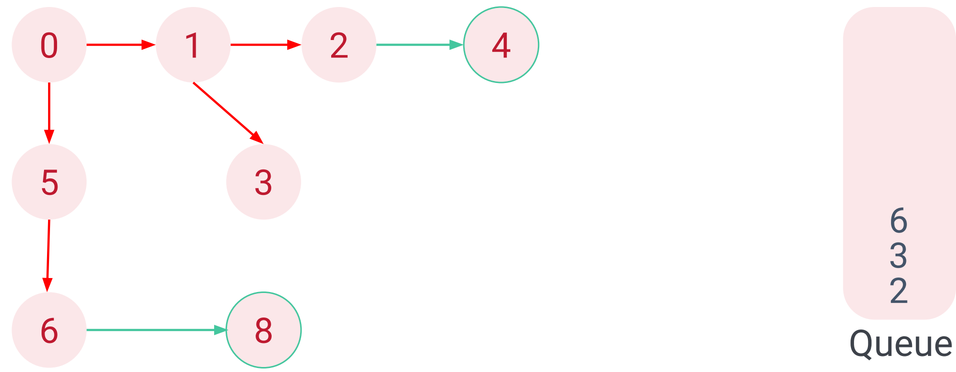
# Breadth-First Search (BFS)




Output : 0 1 5 2 3 6



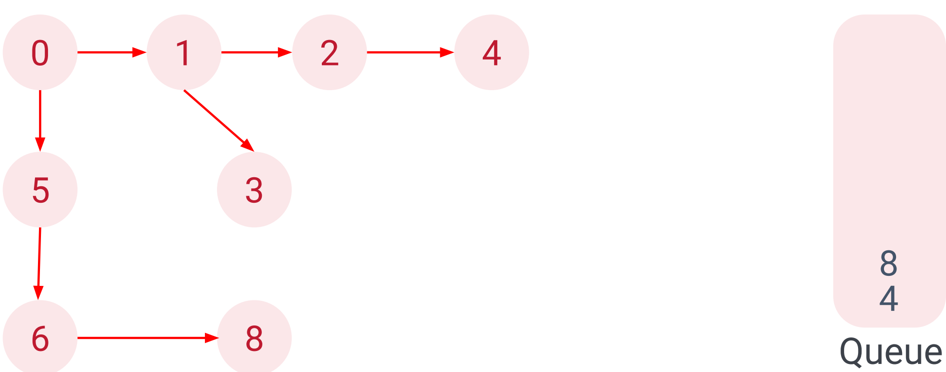
# Breadth-First Search (BFS)




Output : 0 1 5 2 3 6



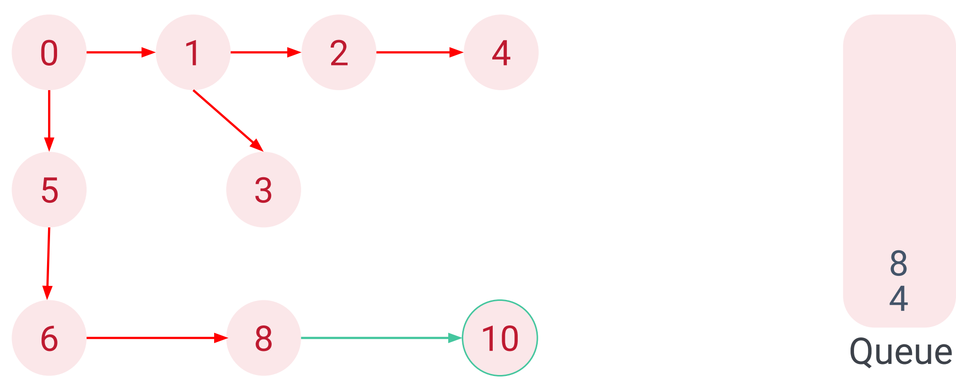
# Breadth-First Search (BFS)




Output : 0 1 5 2 3 6 4 8

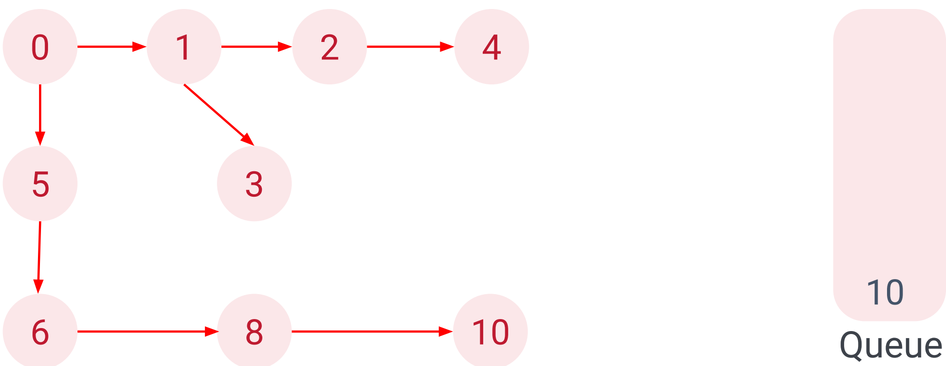


# Breadth-First Search (BFS)




Output : 0 1 5 2 3 6 4 8

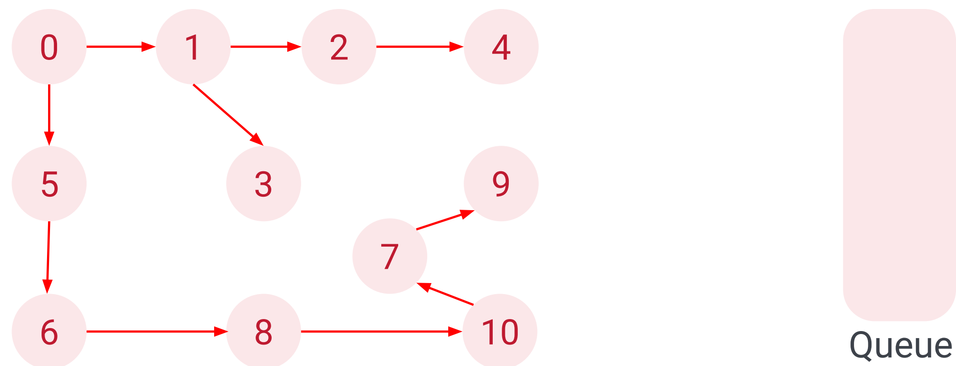
# Breadth-First Search (BFS)




Output : 0 1 5 2 3 6 4 8 10



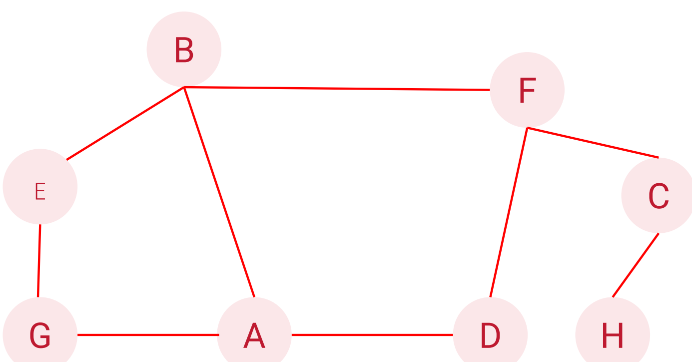
# Breadth-First Search (BFS)




**Output : 0 1 5 2 3 6 4 8 10 7 9**



# Breadth-First Search (BFS) Exercise




**Output : ?**



# Graph Traversals

- On undirected and directed graph with *n* nodes and *m* edges.
  - A DFS traversal can be performed in $O(n + m)$ time.
  - A BFS traversal can be conducted in $O(n + m)$ time

# Directed Acyclic Graphs (DAG)

- Directed graph without directed cycles is a directed acyclic graph (DAG).
- Applications of such graphs, for instance, are:
  - Prerequisites between courses of a degree program.
  - Scheduling constraints between the tasks of a project.
    - Task *a* must be completed before task *b* is started.

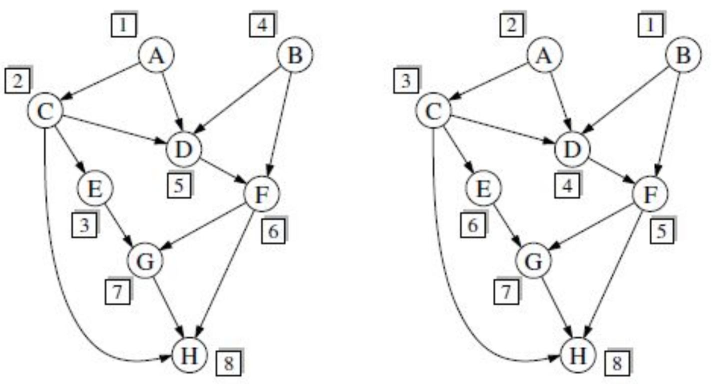


# Topological Ordering

- A topological ordering of graph $G$ is an ordering of the nodes $(v_1, \ldots, v_n)$ such that for every edge $(v_i, v_j)$ of $G$, the condition $i < j$ must be preserved.
- In other words, if there is a part from $v_i$ to $v_j$, $v_j$ must be behind $v_i$ in the ordering.



# Topological Ordering

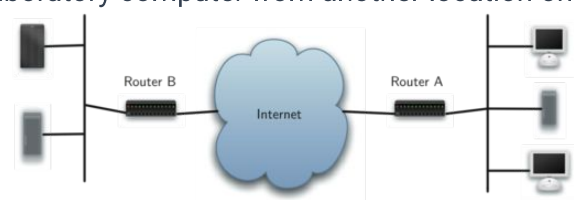



(a) (b)



# Shortest Paths

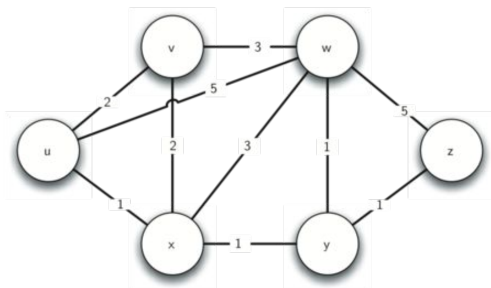
- When an Internet user perform online activities, there is a lot of work going on behind the scenes when transferring information between computers.
  - Examples of online activities: Surf the web, send an email, or log in to a laboratory computer from another location on campus.




Ref: https://runestone.academy/runestone/books/published/pythonds/_images/Internet.png

# Shortest Paths

- The network of routers can be represented as a graph with weighted edges.





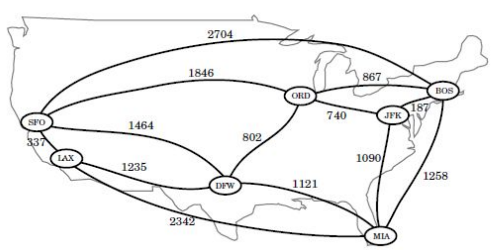

# Shortest Paths

- The BFS strategy can be used to find a **shortest path** from some starting node to every other node in a connected graph.
- This approach is suitable in cases where each edge is equal to others.
- However, for other situations, this approach is not efficient.
- For example, a graph representing the road network between cities, and we would like to find the fastest way to travel from A to B.
- It is natural, therefore, to consider graphs whose edges are <u>not weighted equally</u>.



# Weighted Graphs

- A **weight graph** is a graph that has a numeric label $w(e)$ associated with each edge $e$, called the **weight** of edge $e$.
- For $e = (u, v)$, $w(u, v) = w(e)$.
- Such weights might represent:
  - Costs
  - Lengths
  - Capacities
  - etc.






# Defining Shortest Paths in a Weighted Graph

- Let $G$ be a weight graph.
- The **length** (or **weight**) of a **path** is the sum of the weights of the edges of $P$.
  - P = $((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$
  - Length of P, denoted w(P) is defined as $$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$
- The distance from a node $u$ to a node $v$ in $G$, denoted $d(u, v)$ is the length of a minimum-length path (also called **shortest path**) from $u$ to $v$.

# Shortest Paths Algorithms



- Shortest path with all equal weights (=1) can be solved with BFS traversal algorithm.
- Distances cannot be arbitrarily low negative numbers.
  - For instance, the weight of edges represent the cost to travel between cities. If someone pay you to go between the cities, the cost would be negative.
  - Edge weights in *G* should be nonnegative (that is, *w*(*e*) >= 0) for each edge.

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra’s Algorithm



- An iterative algorithm that provides the shortest path from one starting node to all other nodes in the graph.
- Apply **greedy method** to solve the problem by repeatedly selecting the best choice from among those available in each iteration.
  - Useful for optimising cost function over a collection of objects.
- “Weight” breadth-first search starting at the source node *s*.
- Used in link-state routing protocols in computer network.

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra’s Algorithm Variables



- dist[*v*] keeps the length of the best/smallest weight path so far from the source node *s* to each node *v* in the graph.
  - Initially, dist[*s*] = 0 and dist[*v*] = Inf for each *v* != s
  - In practice, dist[v] can be set to a very large number than any real distance in the problem.
- *Q* is a set of all the unvisited nodes, called the <u>unvisited</u> set.
- Array *prev*[v] is used to keep track of the shortest path.

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra’s Algorithm



1. Set the source node as current node.
2. For the current node, consider all of its unvisited neighbors and calculate their distances through the current node.
3. Compare the distances and select the unvisited neighbor node (*v*) with the smallest distance through the current node.
4. Mark the current node as visited and remove it from the unvisited set Q.
   a. A visited node will never be checked again.
5. If the destination node has been marked visited, then stop.
6. If the smallest distance among the nodes in the unvisited set is infinity, then stop.
   a. Occurs when there is no connection between the source node and remaining unvisited nodes
7. Otherwise, set the unvisited node with the smallest distance as the new “current node” and repeat step 3.

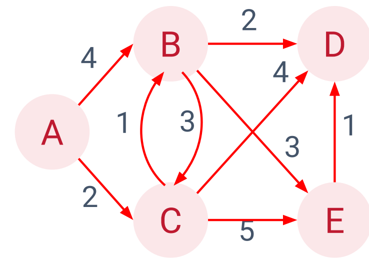
# Dijkstra's Algorithm

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v
that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



# Dijkstra's Algorithm

## Round #0

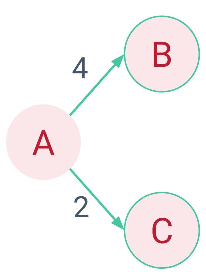



| Node | Cumulative weight |
|---|---|
| A | - |
| B | - |
| C | - |
| D | - |
| E | - |

Q = {'A', 'B', 'C', 'D', 'E'}
D['A'] = 0, D['B'] = D['C'] = D['D'] = D['E'] = Inf
prev['A'] = prev['B'] = prev['C'] = prev['D'] = prev['E'] = None



# Dijkstra's Algorithm




## Round #1

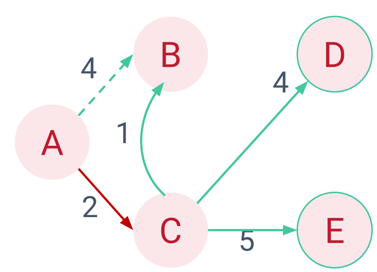
| Node | Cumulative weight |
|---|---|
| A | - |
| B | 4 |
| C | 2 |
| D | - |
| E | - |

Q = {"B', 'C', 'D', 'E'}
D['C'] = 2
prev['C'] = 'A'



# Dijkstra's Algorithm

## Round #2



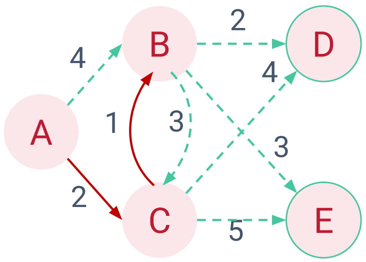

| Node | Cumulative weight |
|---|---|
| A | - |
| B | 4 or 2 + 1 = 3 |
| C | 2 |
| D | 2 + 4 = 6 |
| E | 2 + 5 = 7 |

Q = { 'B', 'D', 'E'}
D['C'] = 2 D['B'] = 3
prev['C'] = 'A' prev['B'] = 'C'

# Dijkstra's Algorithm



## Round #3




| Node | Cumulative weight |
|---|---|
| A | - |
| B | 3 |
| C | 2 |
| D | (ACBD) 3 + 2 = 5 or<br>(ACD) 2 + 4 = 6 |
| E | (ACE) 7 or (ACBE) 3 + 3 = 6 |

Q = { 'D', 'E'}

D['C'] = 2 D['B'] = 3 D['D'] = 5

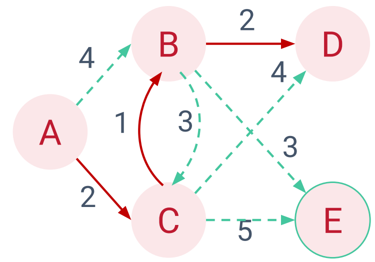
prev['C'] = 'A' prev['B'] = 'C' prev['D'] = 'B'

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra's Algorithm



## Round #4




| Node | Cumulative weight |
|---|---|
| A | - |
| B | 3 |
| C | 2 |
| D | 5 |
| E | (ACE) 7 or (ACBE) 3 + 3 = 6 |

Q = { 'E'}

D['C'] = 2 D['B'] = 3 D['D'] = 5 D['E'] = 6

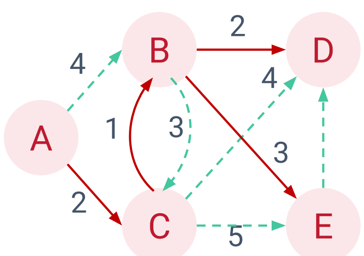
prev['C'] = 'A' prev['B'] = 'C' prev['D'] = 'B' prev['E'] = 'B'

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra's Algorithm



## Round #4




| Node | Cumulative weight |
|---|---|
| A | - |
| B | 3 |
| C | 2 |
| D | 5 |
| E | 6 |

Q = {}

D['C'] = 2 D['B'] = 3 D['D'] = 5 D['E'] = 6

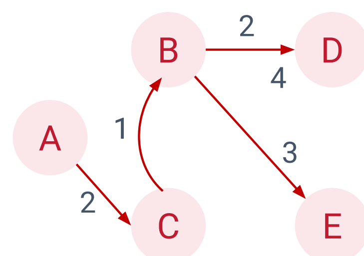
prev['C'] = 'A' prev['B'] = 'C' prev['D'] = 'B' prev['E'] = 'B'

Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

# Dijkstra's Algorithm



## Round #4




| Node | Cumulative weight |
|---|---|
| A | - |
| B | 3 |
| C | 2 |
| D | 5 |
| E | 6 |

## Shortest Path

| Route | Cumulative weight |
|---|---|
| A | 0 |
| A > C > B | 3 |
| A > C | 2 |
| A > C > B > D | 5 |
| A > C > B > E | 6 |

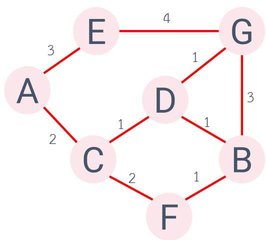
Q = {}

D['C'] = 2 D['B'] = 3 D['D'] = 5 D['E'] = 6

prev['C'] = 'A' prev['B'] = 'C' prev['D'] = 'B' prev['E'] = 'B'

# Dijkstra's Algorithm Exercise






Find the shortest path for each node from node A

Find the distance from A to F

# Dijkstra's Algorithm



- Dijkstra's algorithm works only when the weights are all positive.
  - If there is a negative weight on one of the edges in the graph, the algorithm would never exit.
- Another problem is a complete representation of the graph must be presented for the algorithm to run.
  - Every router has a complete map of all the routers: Not practical.
- Other algorithms allow each router to discover the graph as they go.
  - For instance, distance vector routing algorithm (Computer Networks).
  - Each node computes best path without full view of graph, exchanging link information as they go.

# Individual Assignment



- Assignment#8: Graphs
- Due 09.00 am, Tuesday 02/11/2020.
- Submission
  - Email: sirasit@it.kmitl.ac.th
  - Paper: in classroom next week
- Can be either written by hand or typing.
- Make sure to submit on time!!
  - Late submission has penalty on the score.
- If unable to submit on time for reasonable reasons, let me know asap.