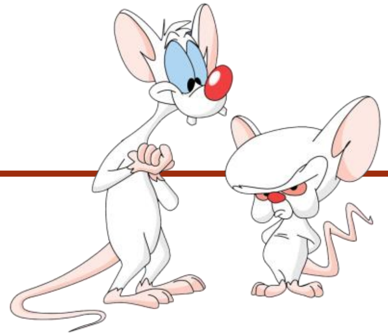
COMP 250
INTRODUCTION TO COMPUTER SCIENCE

35 – Graphs Traversals

Giulia Alberini, Fall 2022

Slides adapted from Michael Langer's



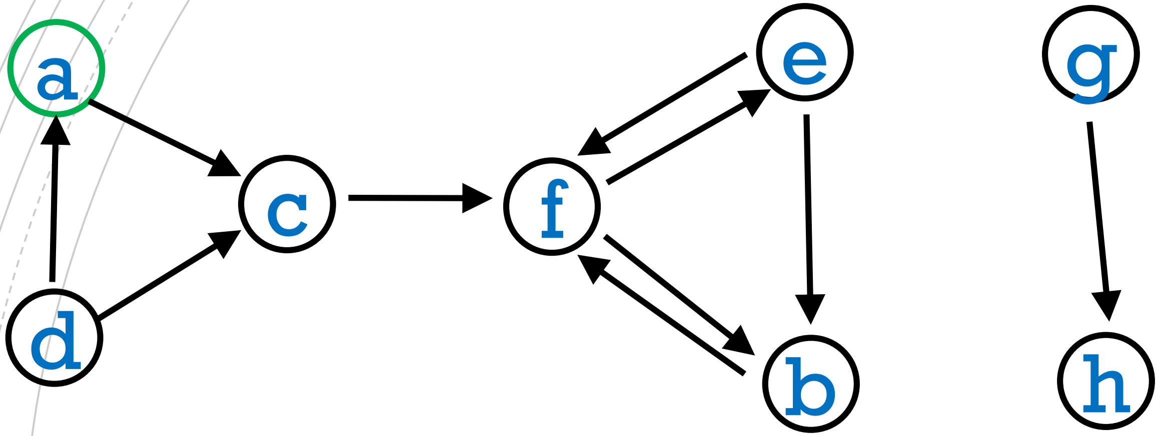
WHAT ARE WE GOING TO DO TODAY?

- Non-recursive graph traversal
  - depth first
  - breadth first
- Shortest path
  - Dijkstra's Algorithm

GRAPH TRAVERSAL (RECURSIVE)

```
depthFirst_Graph (v){
   v.visided = true
   visit v // do something with v
   for each w such that (v,w) is in E
   // i.e. for each w in v.adjList
      if !(w.visited) // avoid cycles!
         dephFirst_Graph(w)
}
```

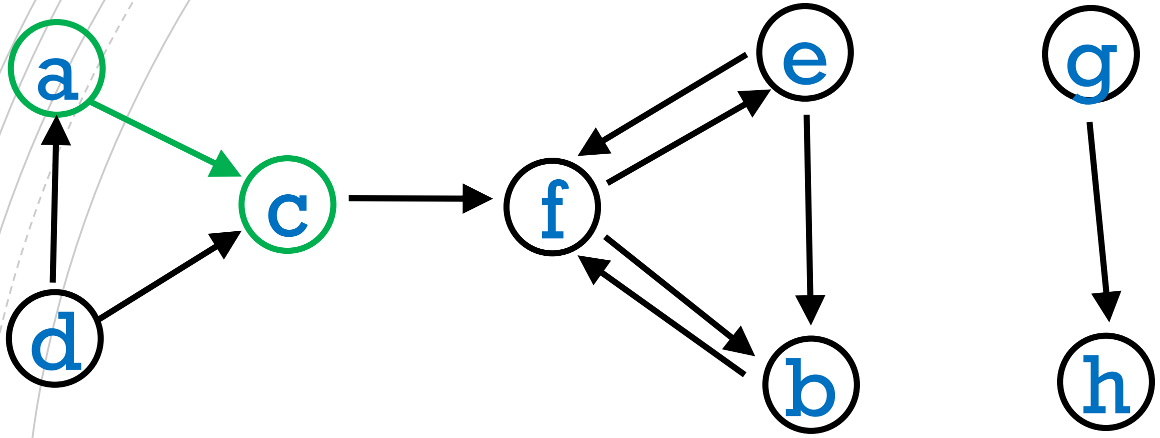
CALL STACK FOR depthFirst(a)

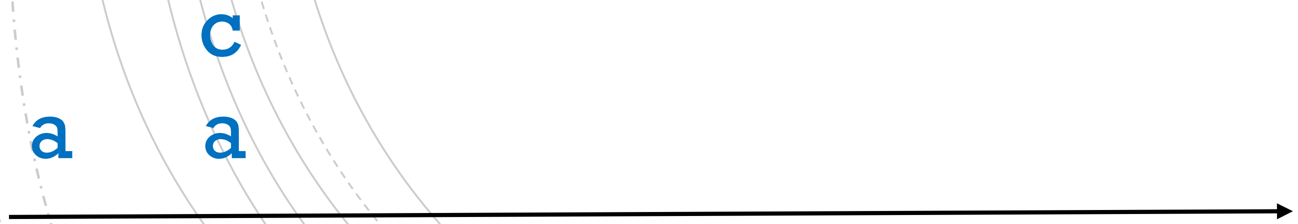



```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

a

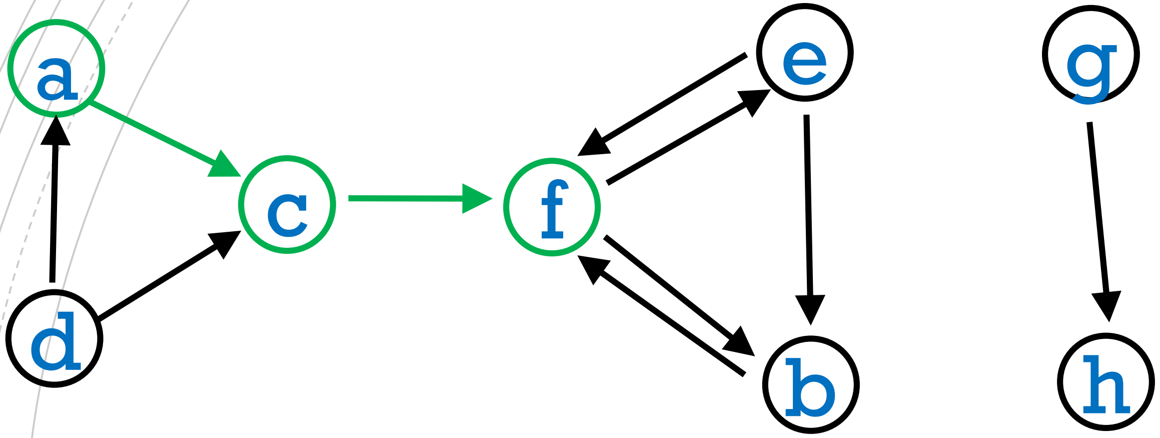
CALL STACK FOR depthFirst(a)

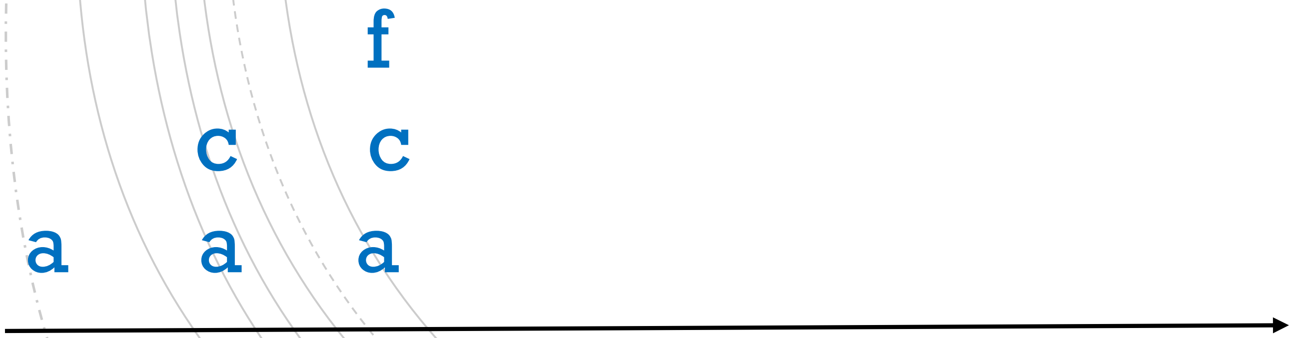





```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

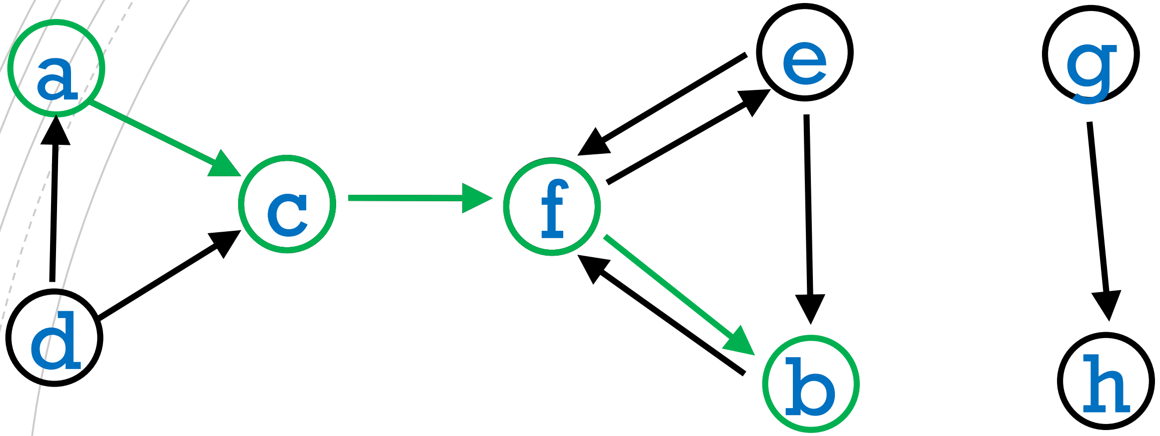
CALL STACK FOR depthFirst(a)






```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

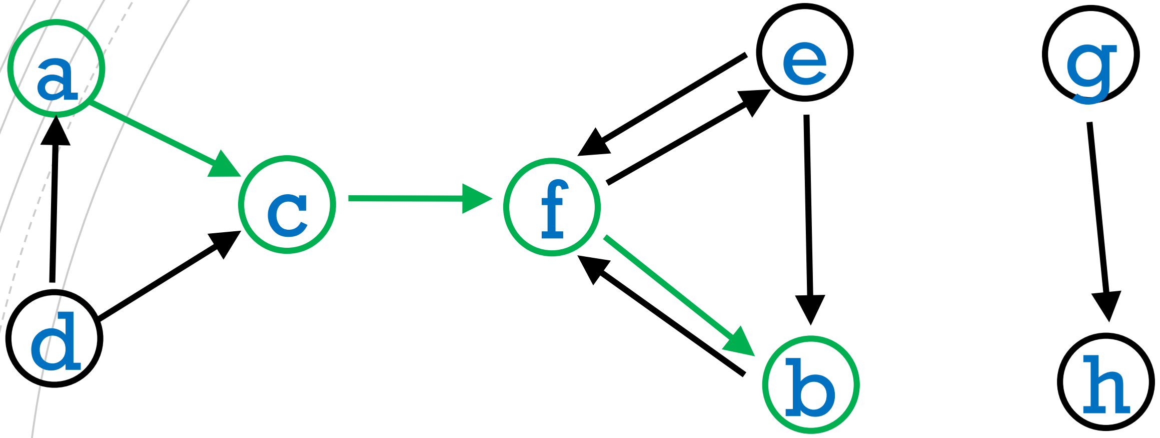
CALL STACK FOR depthFirst(a)



			b
		f	f
	c	c	c
a	a	a	a

```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

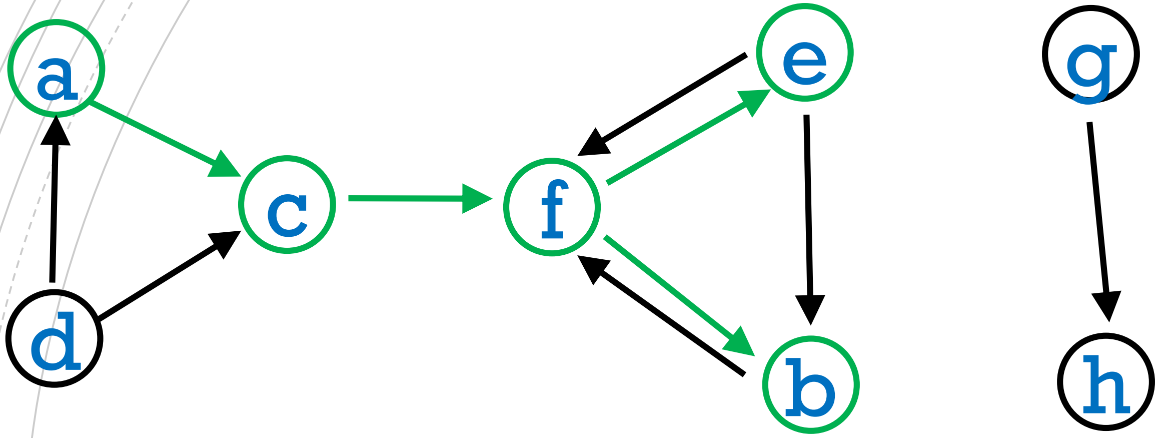
CALL STACK FOR depthFirst(a)

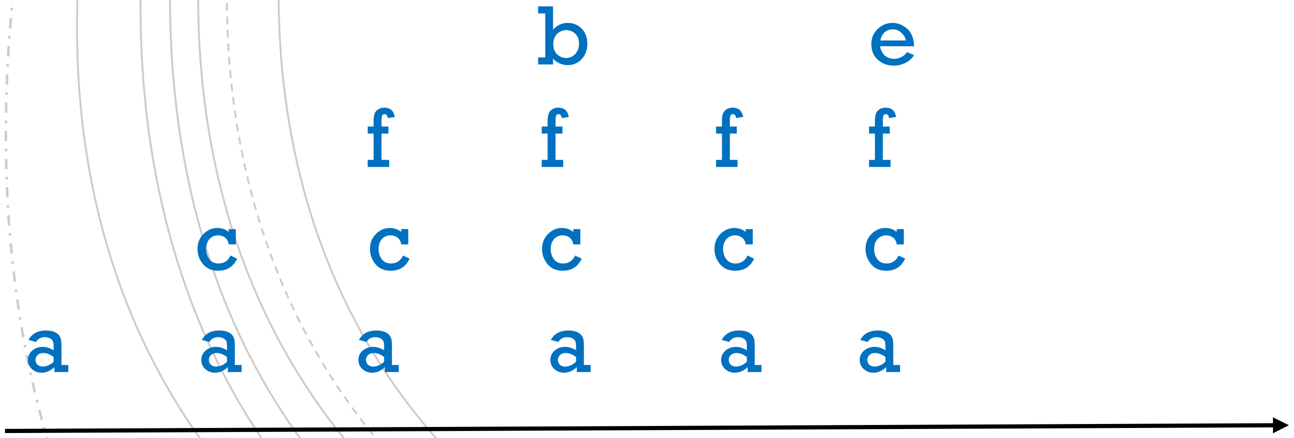


			b	
		f	f	f
	c	c	c	c
a	a	a	a	a

```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
       if !(w.visited)
          dephFirst_Graph(w)
}
```

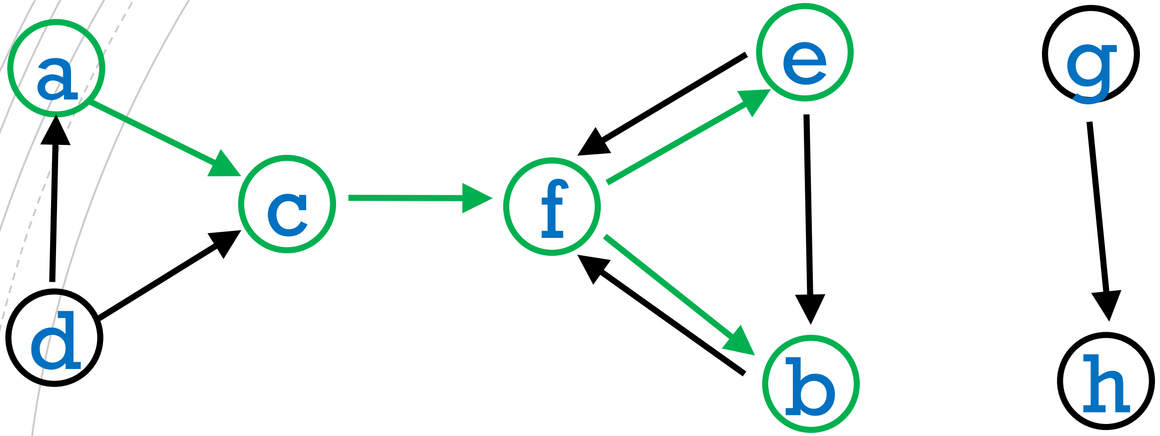
CALL STACK FOR depthFirst(a)

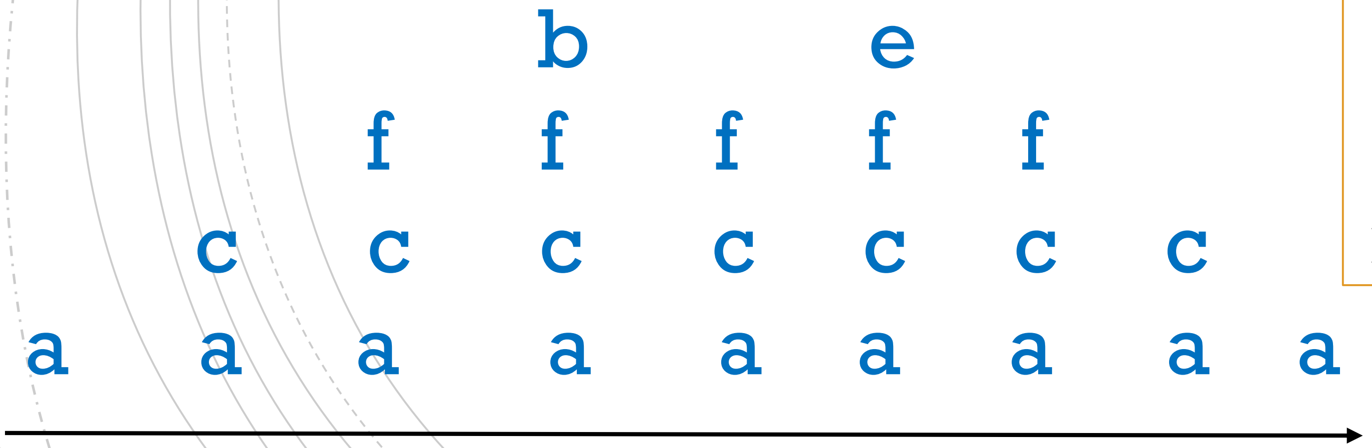





```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

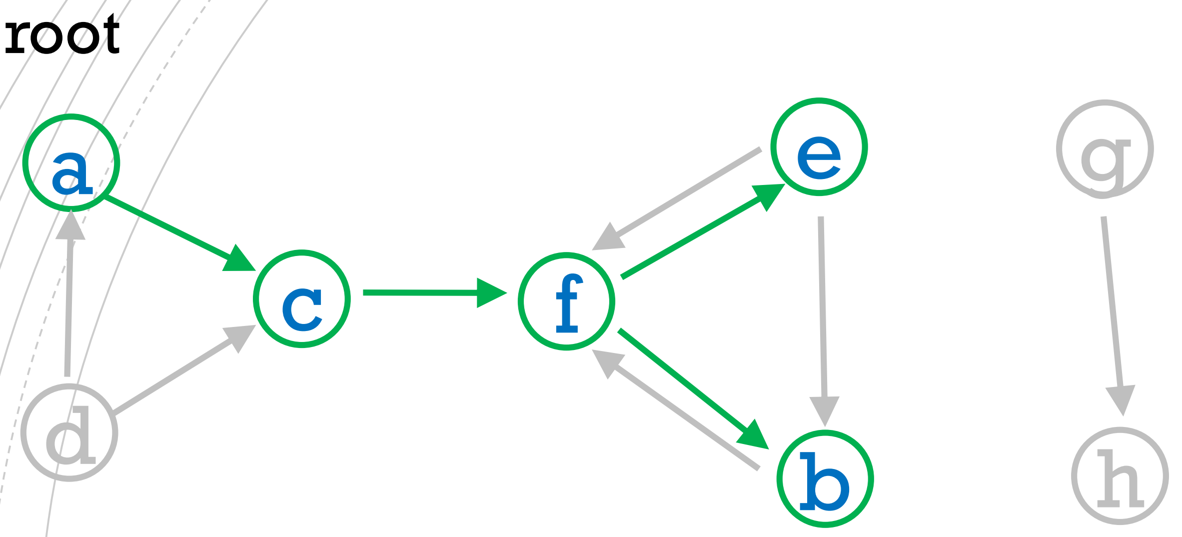
CALL STACK FOR depthFirst(a)

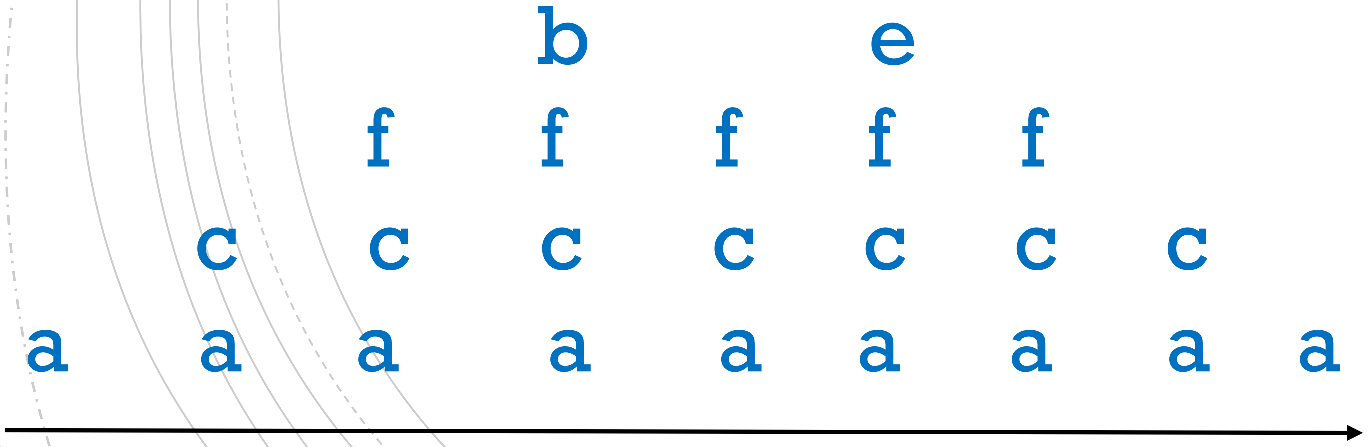





```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

CALL TREE





GRAPH TRAVERSALS

- Unlike tree traversal for rooted tree, a graph traversal started from some arbitrary vertex does not necessarily reach all other vertices.
- *Knowing which vertices can be reached by a path from some starting vertex is itself an important problem. You will learn about such graph `connectivity' problems in COMP 251.*
- The order of nodes visited depends on the order of nodes in the adjacency lists.

GRAPH TRAVERSALS

- Q: Can we do non-recursive graph traversals?

GRAPH TRAVERSALS

- Q: Can we do non-recursive graph traversals?
- A: Yes, similar to tree traversal: use a stack or a queue.

RECALL: DEPTH FIRST TREE TRAVERSAL
(WITH A SLIGHT VARIATION)

```
treeTraversalUsingStack(root){
   initialize empty stack s
   s.push(root)
   while s is not empty {
      cur = s.pop()
      visit cur
      for each child of cur {
         s.push(child)
      }
   }
}
```

Visit a node *after popping* it from the stack.

Every node in the tree gets pushed, and popped, and visited.

GENERALIZE TO GRAPH

```
graphTraversalUsingStack(v){
   initialize empty stack s
   v.visited = true
   s.push(v)
   while s is not empty {
      cur = s.pop()
      visit cur // do something
      for each w in cur.adjList
            if(!w.visited) {
               w.visited = true
               s.push(w)
            }
      }
   }
}
```

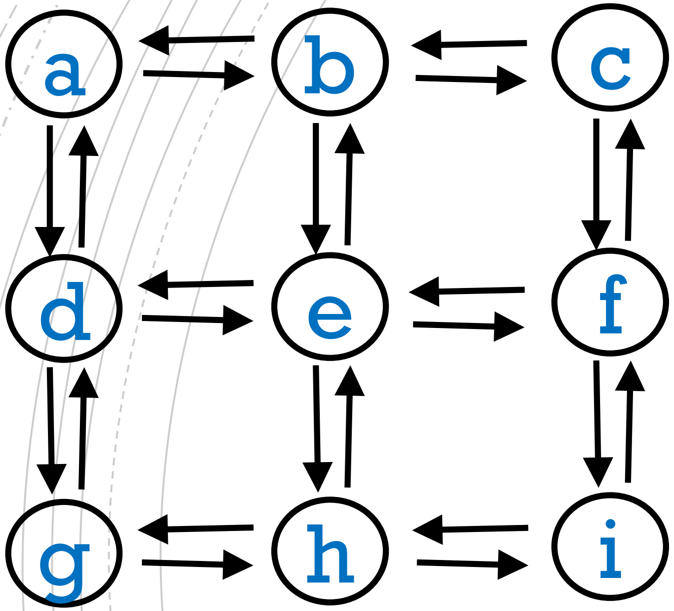
Indicate as "reached" a node *before pushing* it onto the stack. We do that by updating the field `visited`.

Visit the node (perform some operations) after it gets popped from the stack.

Every node in the graph gets *reached*, pushed, popped, and visited.

EXAMPLE: graphTraversalUsingStack(a)

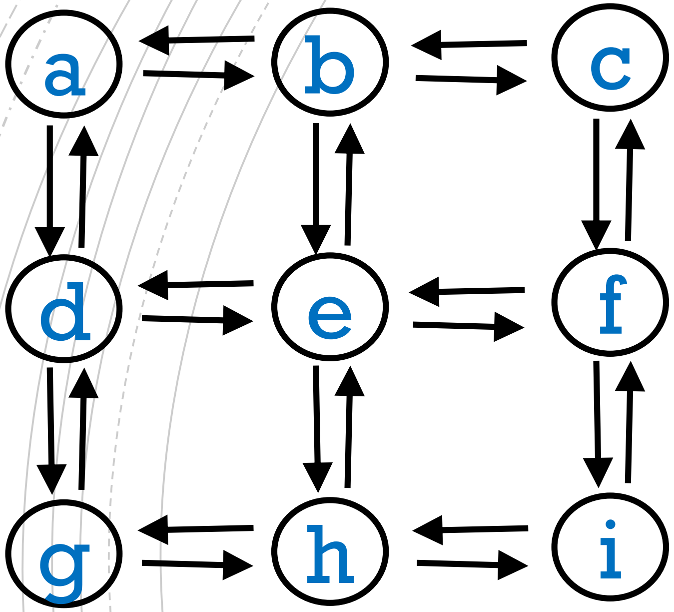
Order of visit: **a**



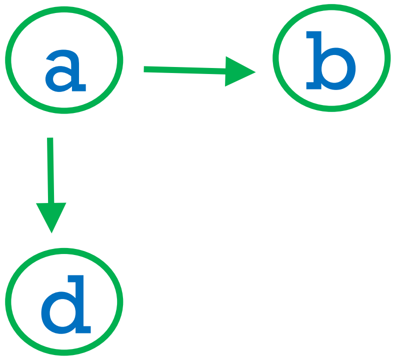
a

a

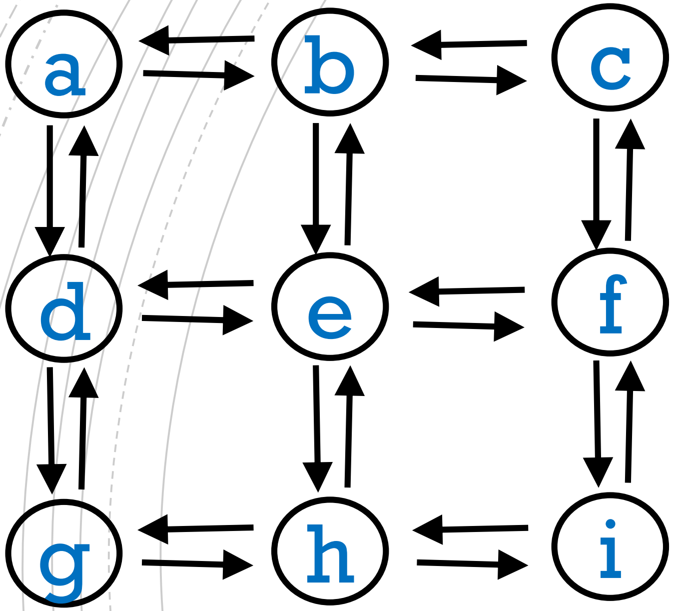
EXAMPLE: graphTraversalUsingStack(a)



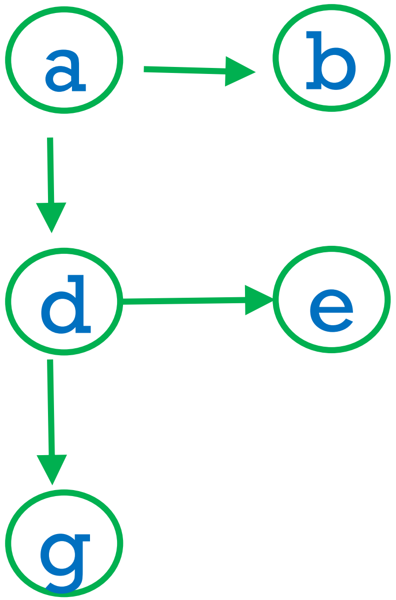
Order of visit: **a**



EXAMPLE: graphTraversalUsingStack(a)

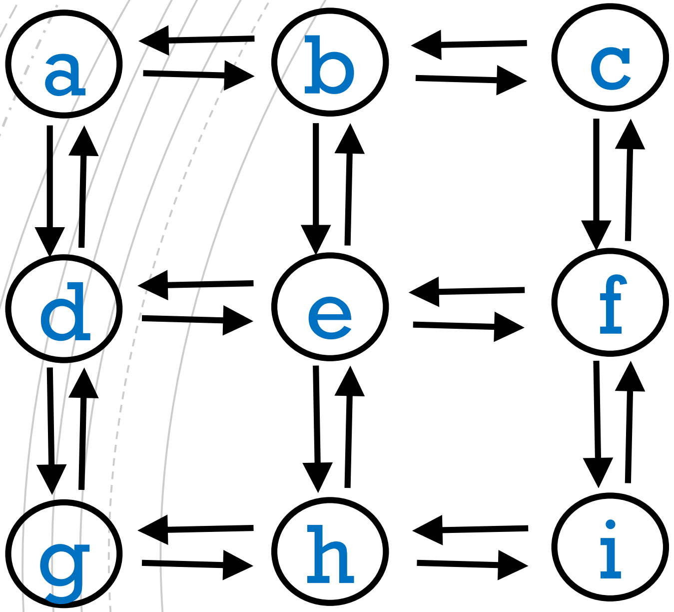


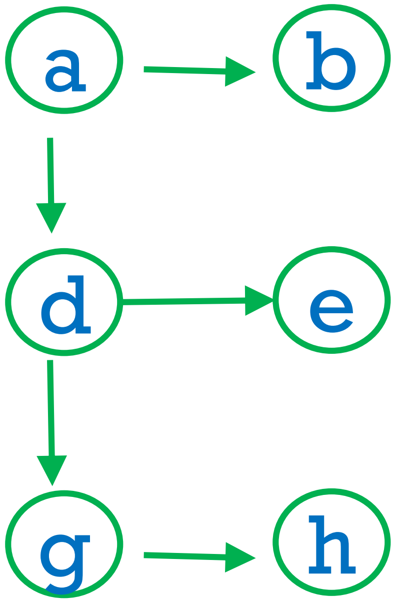
Order of visit: **ad**

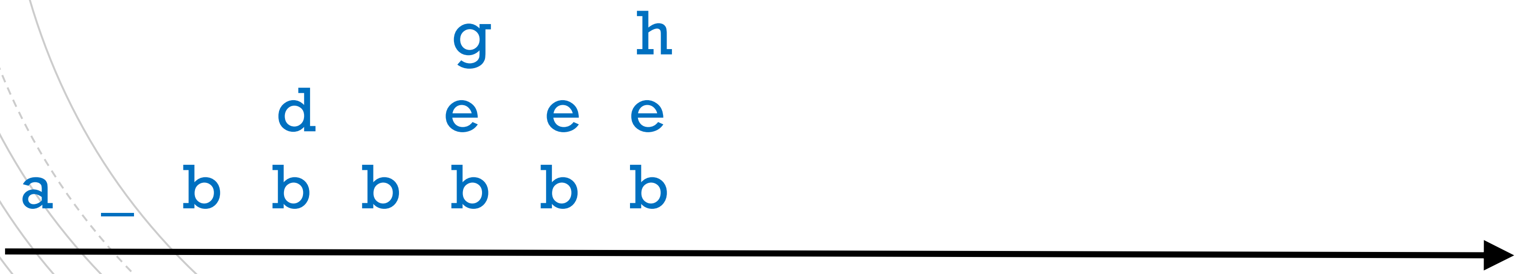


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adg**

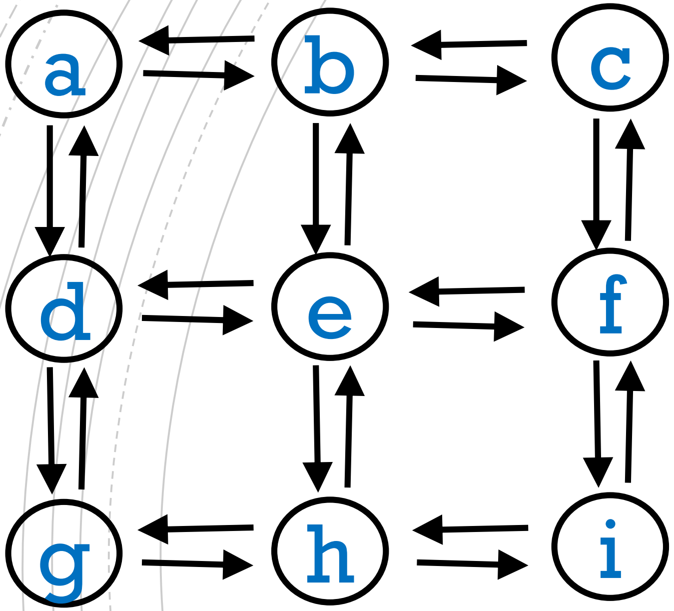


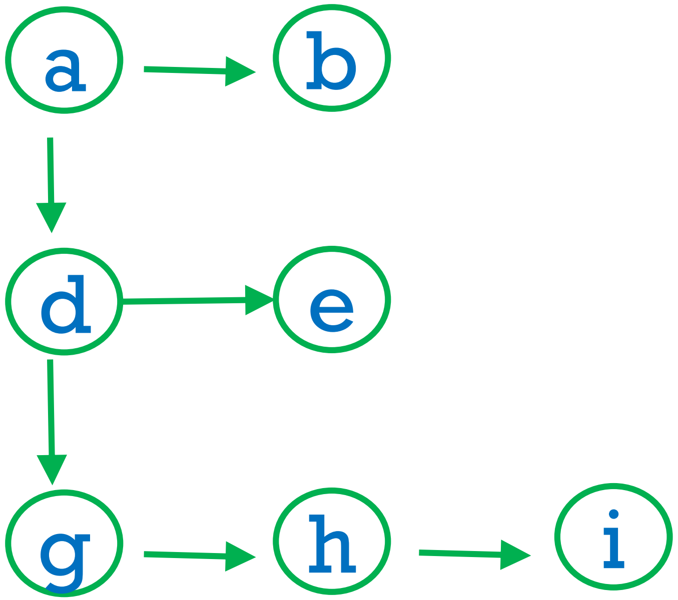


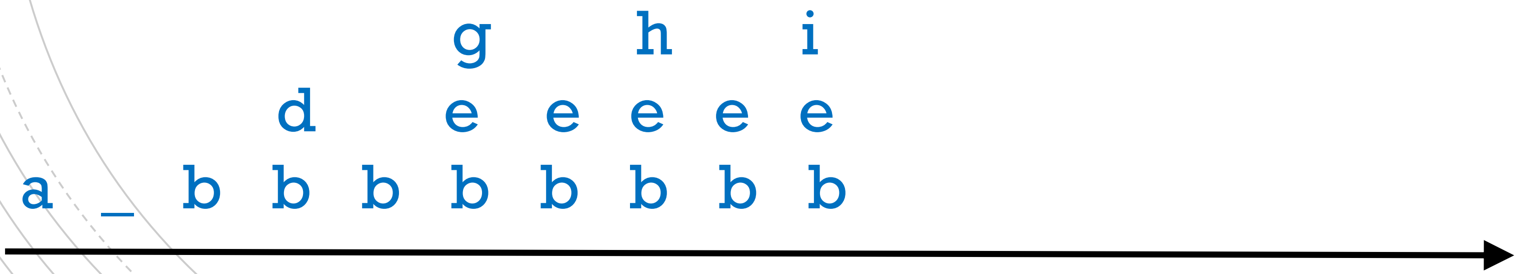


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adgh**

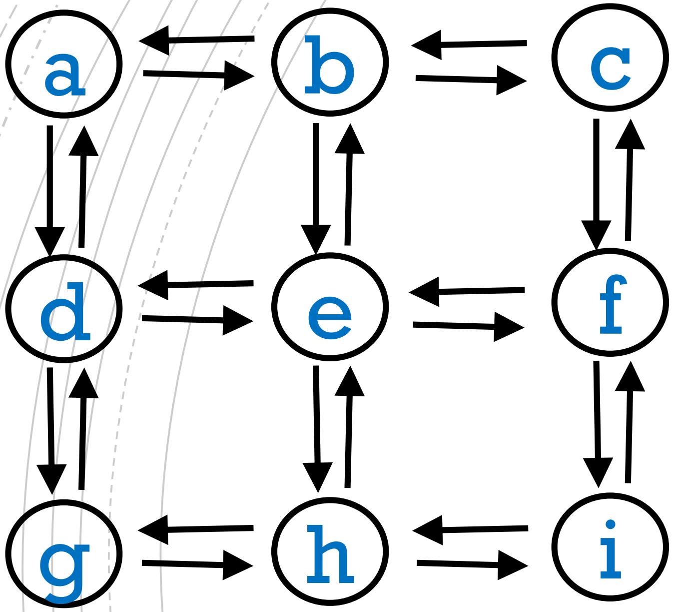


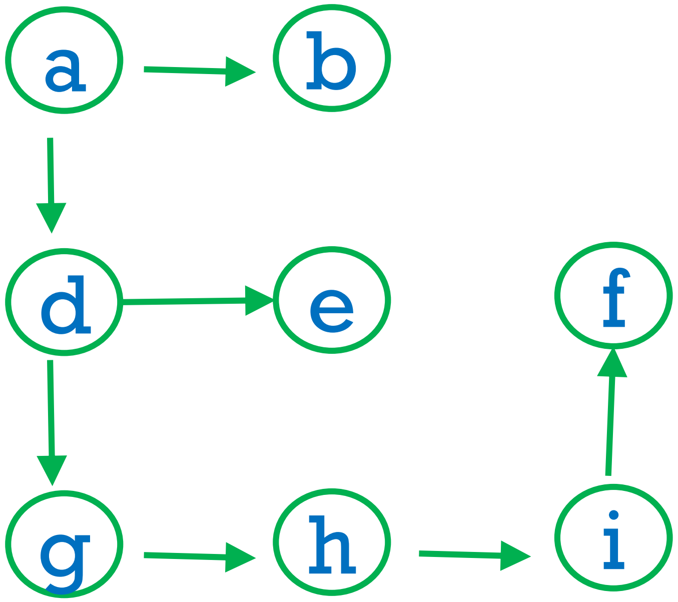


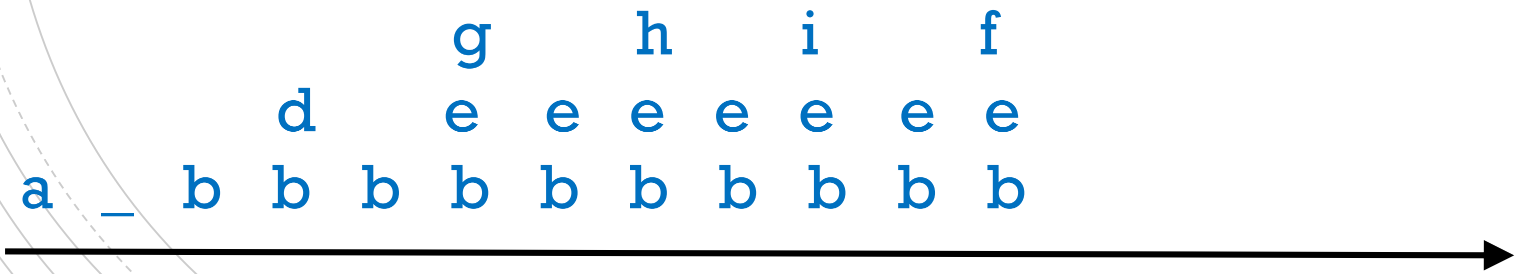


EXAMPLE: graphTraversalUsingStack(a)

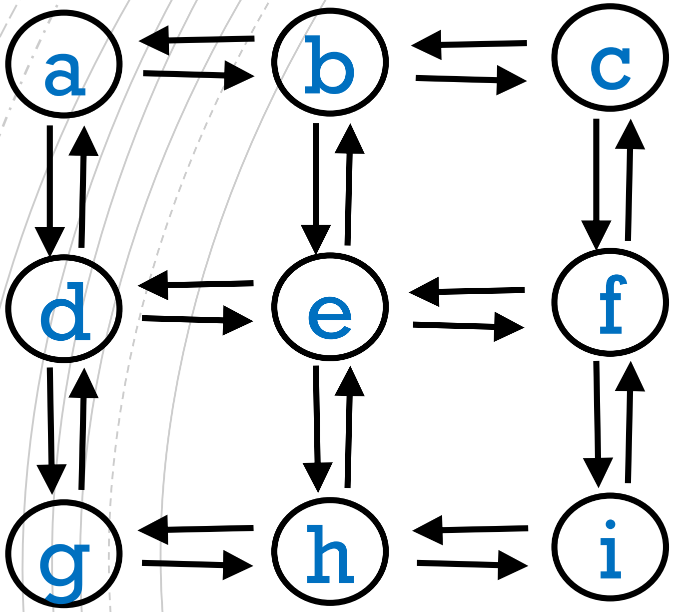
Order of visit: **adghi**



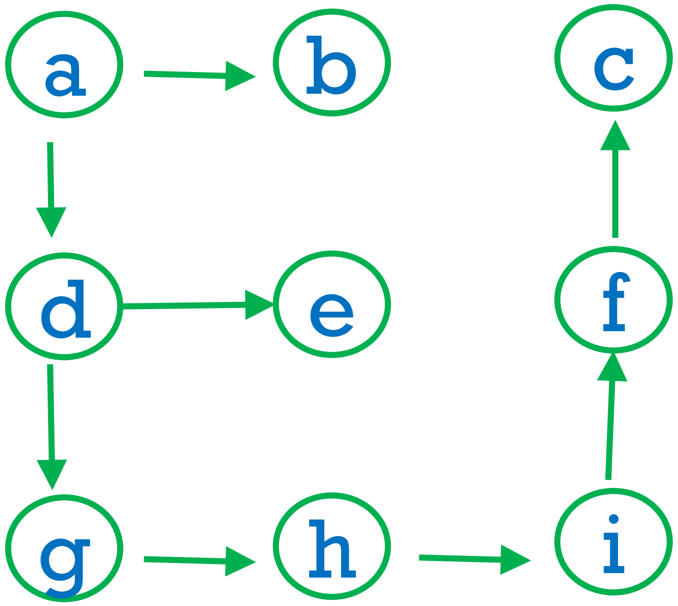




EXAMPLE: graphTraversalUsingStack(a)



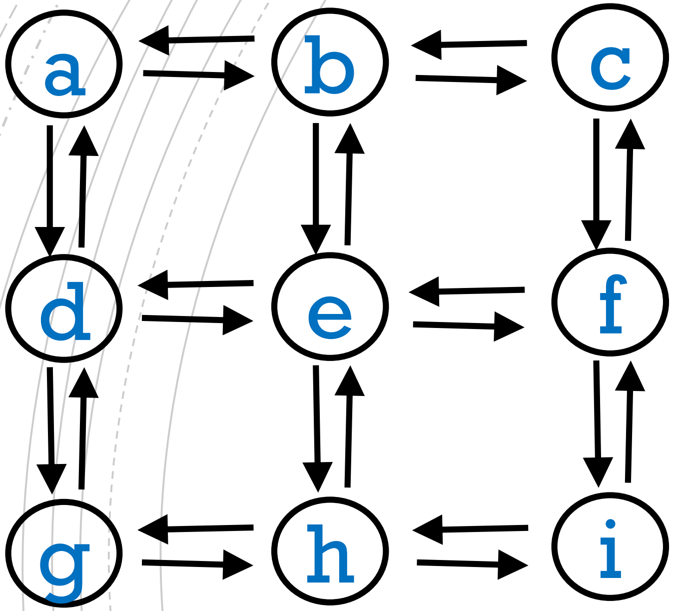
Order of visit: **adghif**

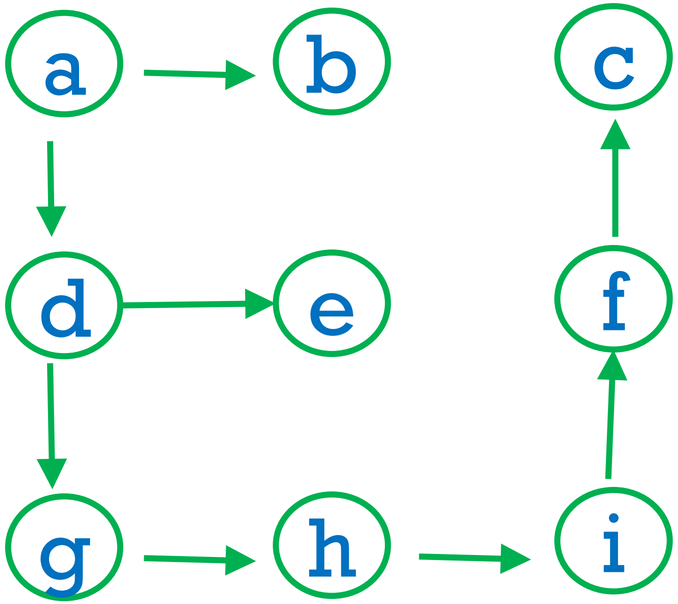


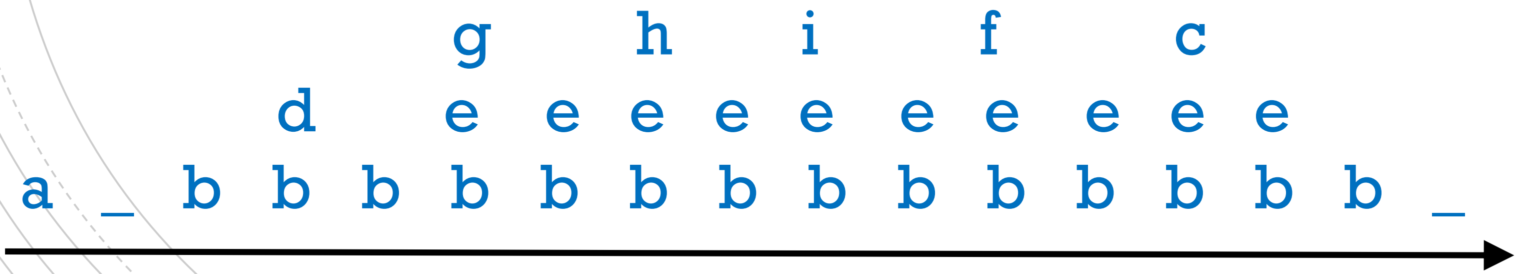
					g		h		i		f		c
			d		e	e	e	e	e	e	e	e	e
a	_	b	b	b	b	b	b	b	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghifceb**

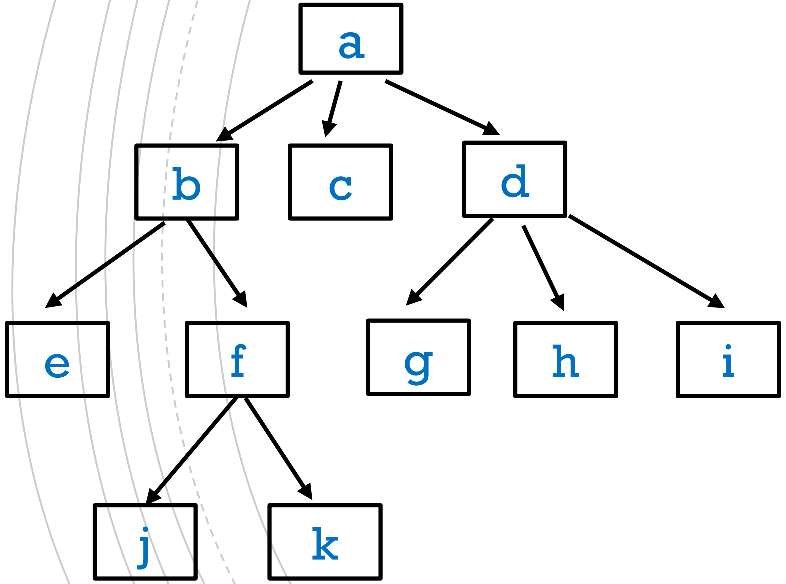






RECALL: BREADTH FIRST TREE TRAVERSAL

for each level i
    visit all nodes at level i




```
treeTraversalUsingQueue(root){
   initialize empty queue q
   q.enqueue(root)
   while q is not empty {
      cur = q.dequeue()
      visit cur
      for each child of cur
         q.enqueue(child)
   }
}
```

BREADTH FIRST GRAPH TRAVERSAL

Given an input vertex, visit all vertices that can be reached by paths of length 1, 2, 3, 4, ….

BREADTH FIRST GRAPH TRAVERSAL

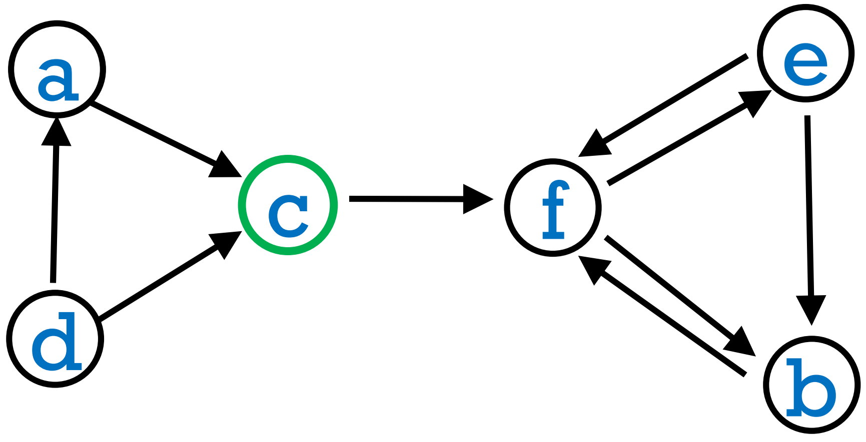
```
graphTraversalUsingQueue(v){
   initialize empty queue q
   v.visited = true
   q.enqueue(v)
   while q is not empty {
      cur = q.dequeue()
      for each w in cur.adjList {
         if(!w.visited) {
            w.visited = true
            q.enqueue(w)
         }
      }
   }
}
```

EXAMPLE

graphTraversalUsingQueue(c)

<u>queue</u>

c



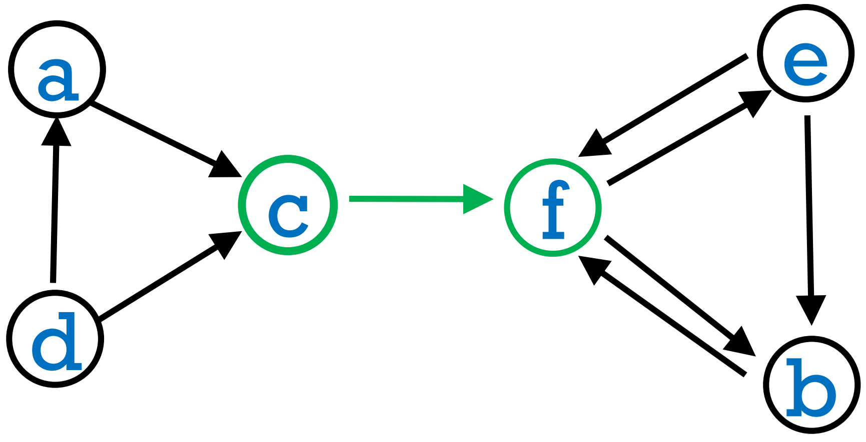
EXAMPLE

graphTraversalUsingQueue(c)

queue

c

f



EXAMPLE

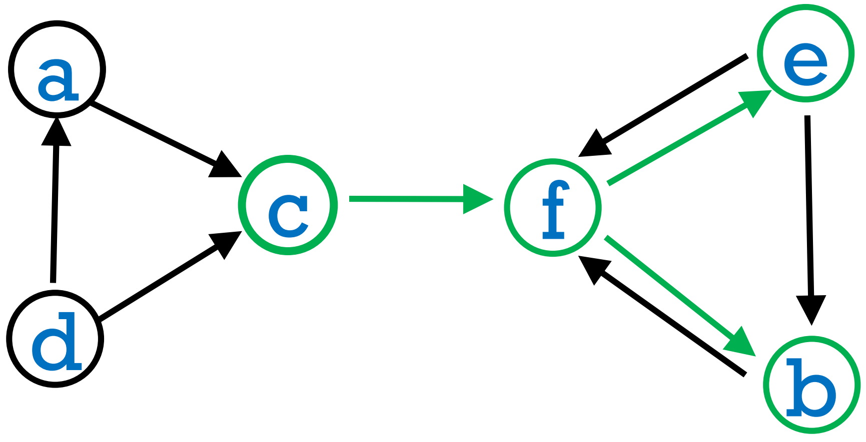
graphTraversalUsingQueue(c)

queue

c

f

be



Both ‘b’, ‘e’ are visited and enqueued before ‘b’ is dequeued.

EXAMPLE

graphTraversalUsingQueue(c)

queue

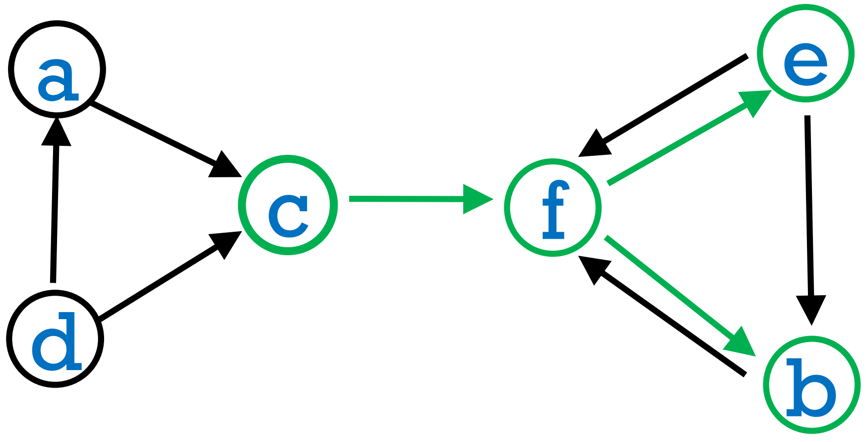
c

f

be

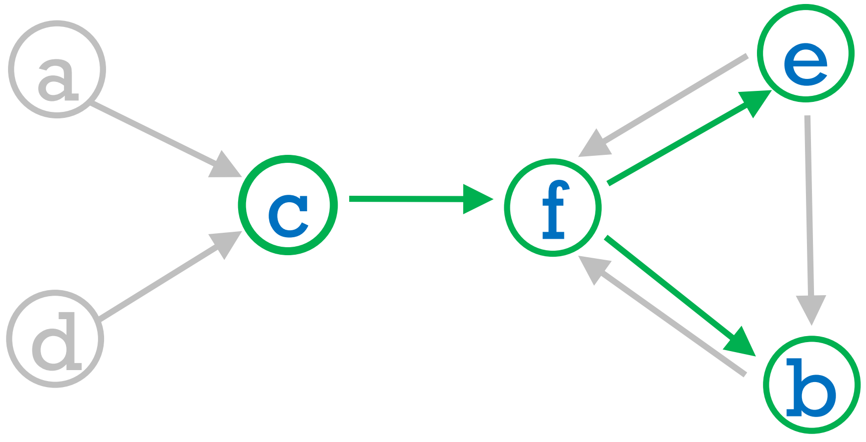
e

_



EXAMPLE

graphTraversalUsingQueue(c)



It defines a tree whose root is the starting vertex.
It finds the shortest path (number of edges) to all vertices reachable from the starting vertex.

RECALL: HOW TO IMPLEMENT A GRAPH CLASS IN JAVA?

```
class Graph<T> {
   ArrayList<Vertex<T>> vetexList;

   class Vertex<T> {
      ArrayList<Edge> adjList;
      T element;
      boolean visited;
   }

   class Edge {
      Vertex endVertex;
      double weight;
            :
   }
}
```

PRIOR TO TRAVERSAL!

```
for each w in V
   w.visited = false
```

How should we implement this?

PRIOR TO TRAVERSAL!

```
for each w in V
   w.visited = false
```

```
class Graph<T> {
   ArrayList<Vertex<T>> vetexList;
   :
   public void resetVisited() {



   }
}
```

PRIOR TO TRAVERSAL!

```
for each w in V
   w.visited = false
```

```
class Graph<T> {
   ArrayList<Vertex<T>> vetexList;
   :
   public void resetVisited() {
      for(Vertex<T> v : vertexList)
         v.visited = false;
   }
}
```

SHORTEST PATH

MODELING AS GRAPHS

Input:

- Directed graph $G = (V, E)$
- Weight function $w\colon E \rightarrow \mathbb{R}$

Weight of path $p = \langle v_0, v_1, \ldots, v_n \rangle$

$$w(p) = \sum_{k=1}^{n} w(v_{k-1}, v_k) \quad = \text{sum of edges weights on path } p$$

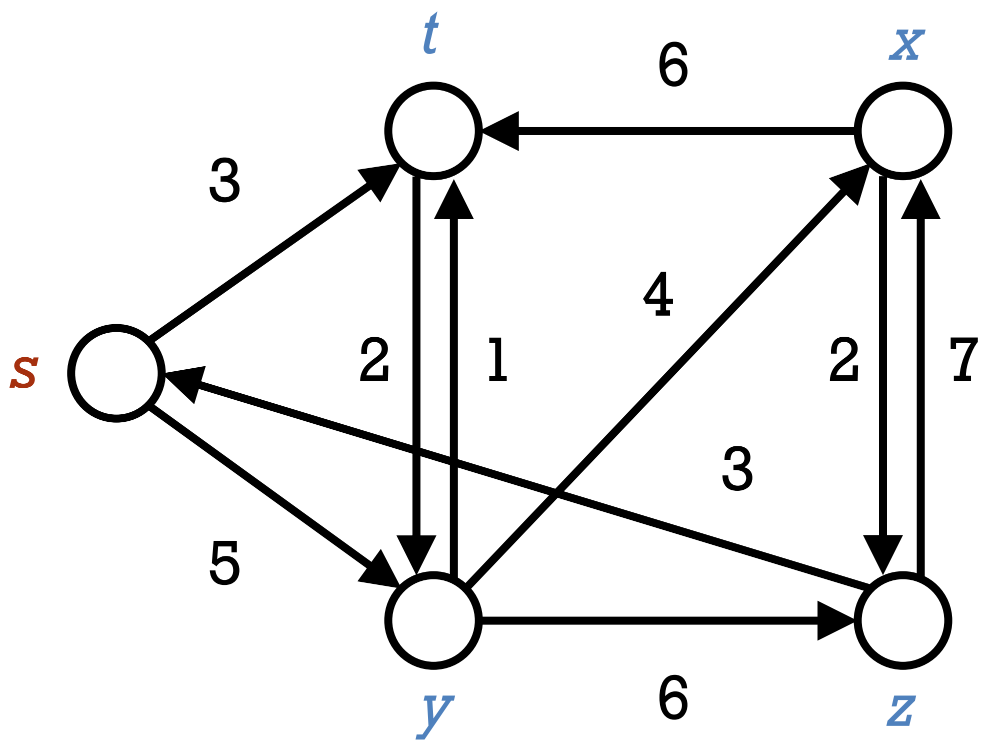
Shortest-path weight u to v:

$$\delta(u,v) = \begin{cases} \min\left\{ w(p) : u \overset{p}{\mapsto} v \right\} & \text{If there exists a path } u \rightsquigarrow v. \\ \infty & \text{Otherwise.} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$

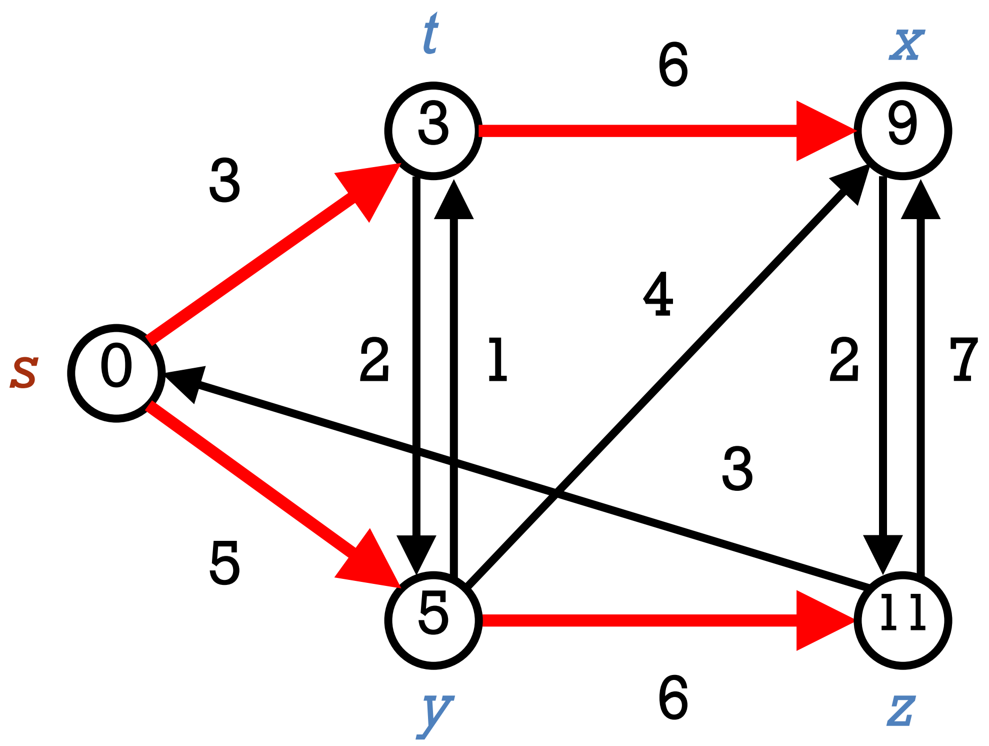
Generalization of breadth-first search to weighted graphs

EXAMPLE



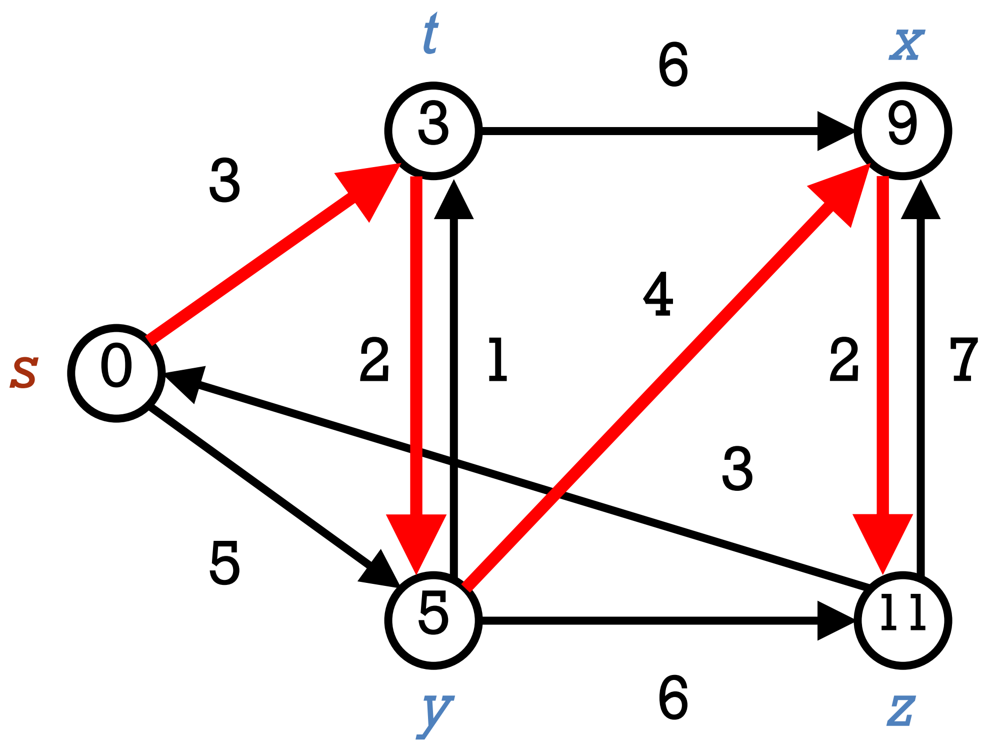
Shortest path from s?

EXAMPLE



Shortest paths are organized as a tree.
Vertices store the length of the shortest path from s.

EXAMPLE



Shortest paths are not necessarily unique!

VARIANTS

- **Single-source:** Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- **Single-destination:** Find shortest paths to a given destination vertex.
- **Single-pair:** Find shortest path from u to v.

  Note: No way to known that is better in worst case than solving the single-source problem!
- **All-pairs:** Find shortest path from u to v for all $u, v \in V$
  .

PRINCIPLE OF A SINGLE-SOURCE SHORTEST-PATH ALGORITHM

For each vertex $v \in V$:

$\delta(s, v)$ is the absolute shortest path
$d[v]$ is our current estimate of the shortest path

- $d[v] = \delta(s, v)$.
  - Initially, $d[v] = \infty$.
  - Reduces as algorithms progress, but always maintain $d[v] \geq \delta(s, v)$.
  - Call $d[v]$ a **shortest-path estimate.**

- $\pi[v]$ = predecessor of v on a shortest path from s.
  - If no predecessor, $\pi[v] = null$.
  - π induces a tree - **shortest-path tree**

GENERIC ALGORITHM STRUCTURE

1. Initialization
2. Scan vertices and relax edges

The algorithms differ in the order and how many times they relax each edge.

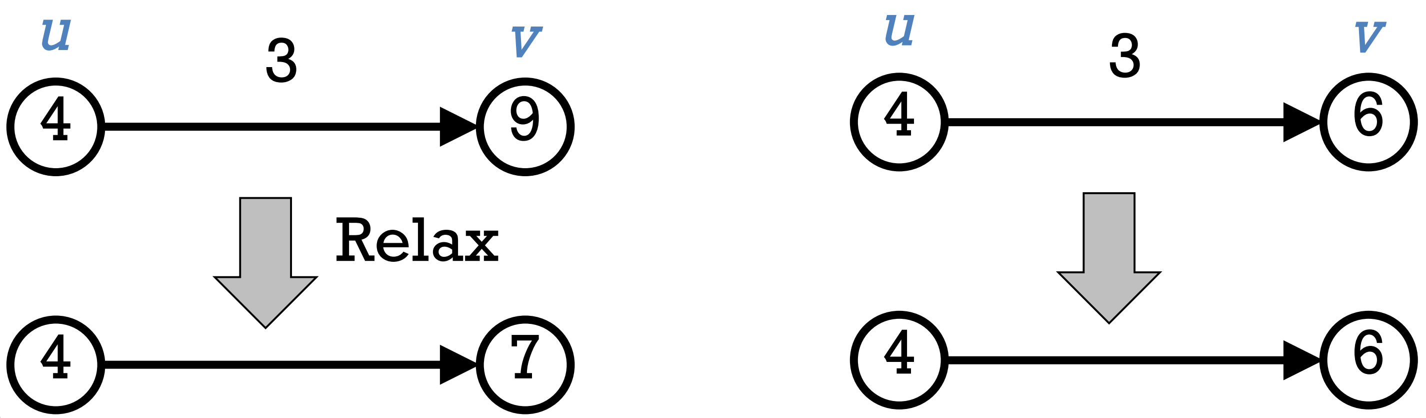
INITIALIZATION

```
INIT-SINGLE-SOURCE(V,s)
   for each v ∈ V do
      d[v]← ∞
      π[v]← null
   d[s]←0
```

This is used to reduce $d[v]$ during the execution of the algorithm.

RELAXING AN EDGE

```
RELAX(u,v,w)
   if d[v]>d[u]+w(u,v) then
      d[v] ← d[u]+w(u,v)
      π[v] ← u
```



DIJKSTRA'S ALGORITHM

- No negative-weight edges.

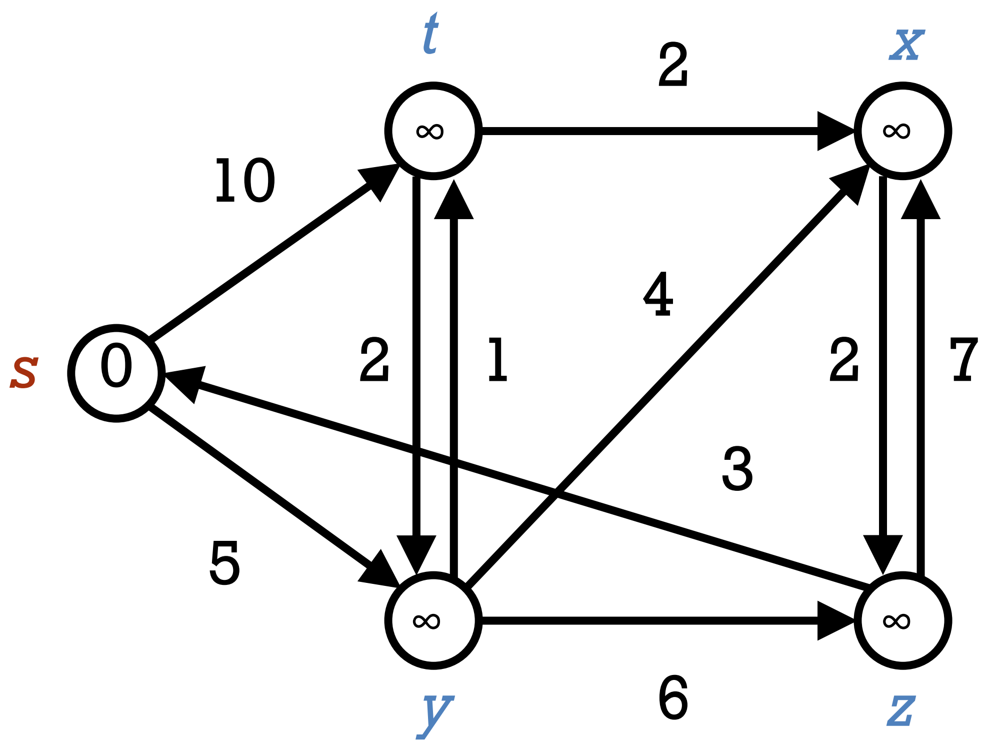
- Weighted version of BFS:
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights ($d[v]$).

- Have two sets of vertices:
  - S = vertices whose final shortest-path weights are determined,
  - Q = priority queue = $V \setminus S$.

DIJKSTRA'S ALGORITHM

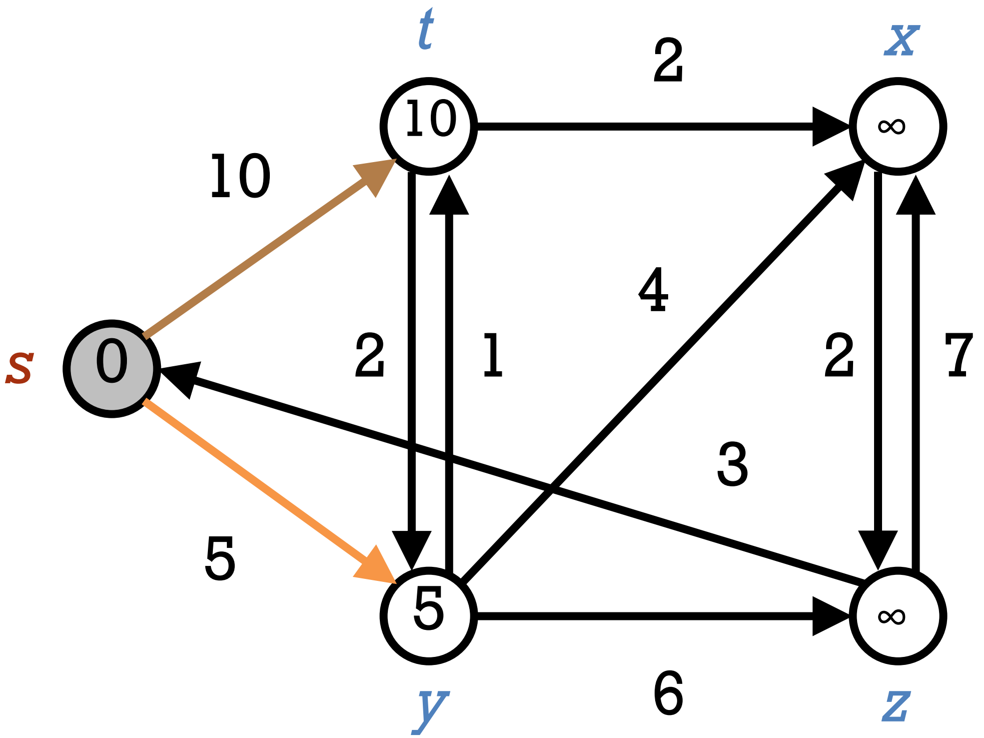
```
DIJKSTRA(V, E,w,s)
INIT-SINGLE-SOURCE(V,s)
S ← ∅
Q ← V
while Q ≠ ∅ do
   u ← REMOVE-MIN(Q)
   S ← S ∪ {u}
   for each vertex v ∈ Adj[u] do
      RELAX(u,v,w)
```

EXAMPLE



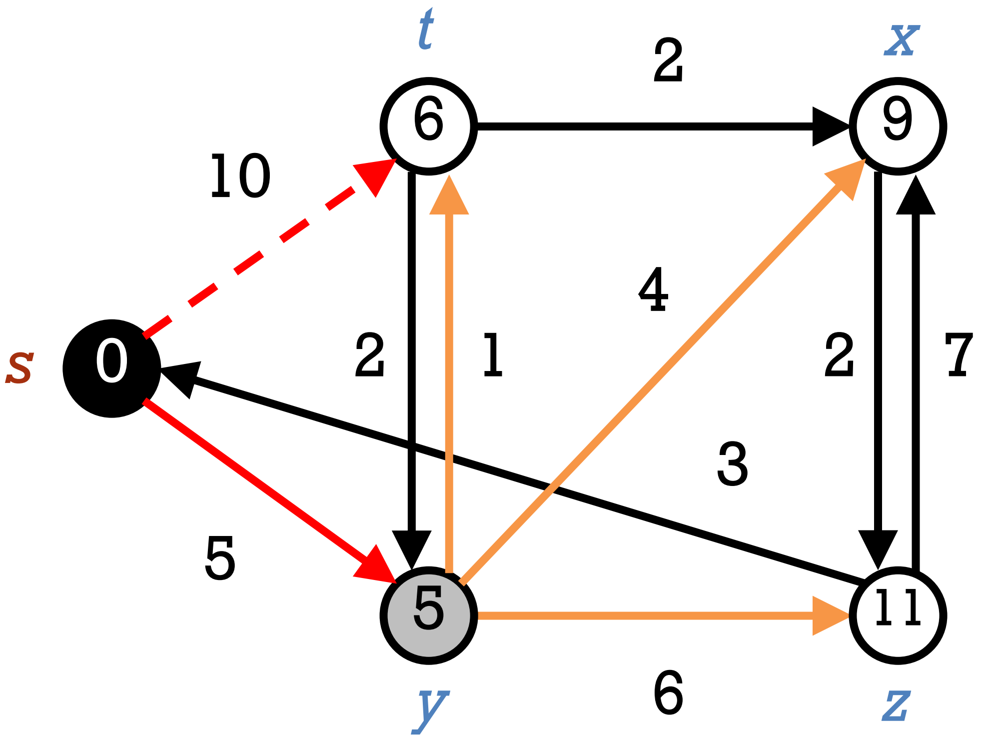
Q	s	t	y	x	z

EXAMPLE



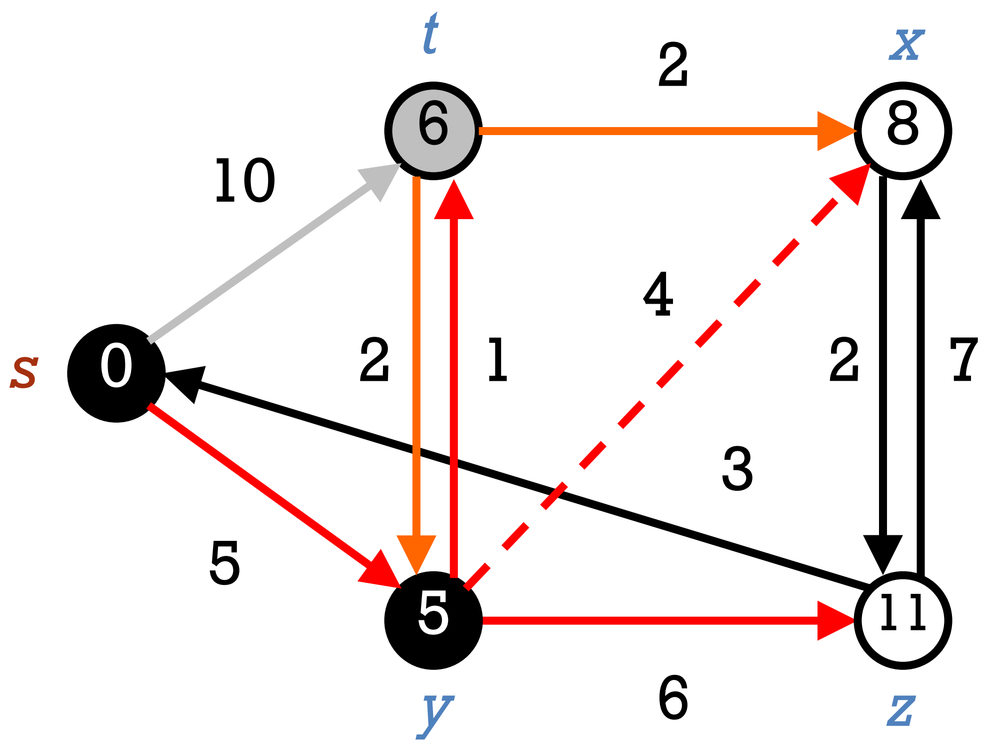
Q	y	t	x	z	

EXAMPLE



Q	t	x	z		

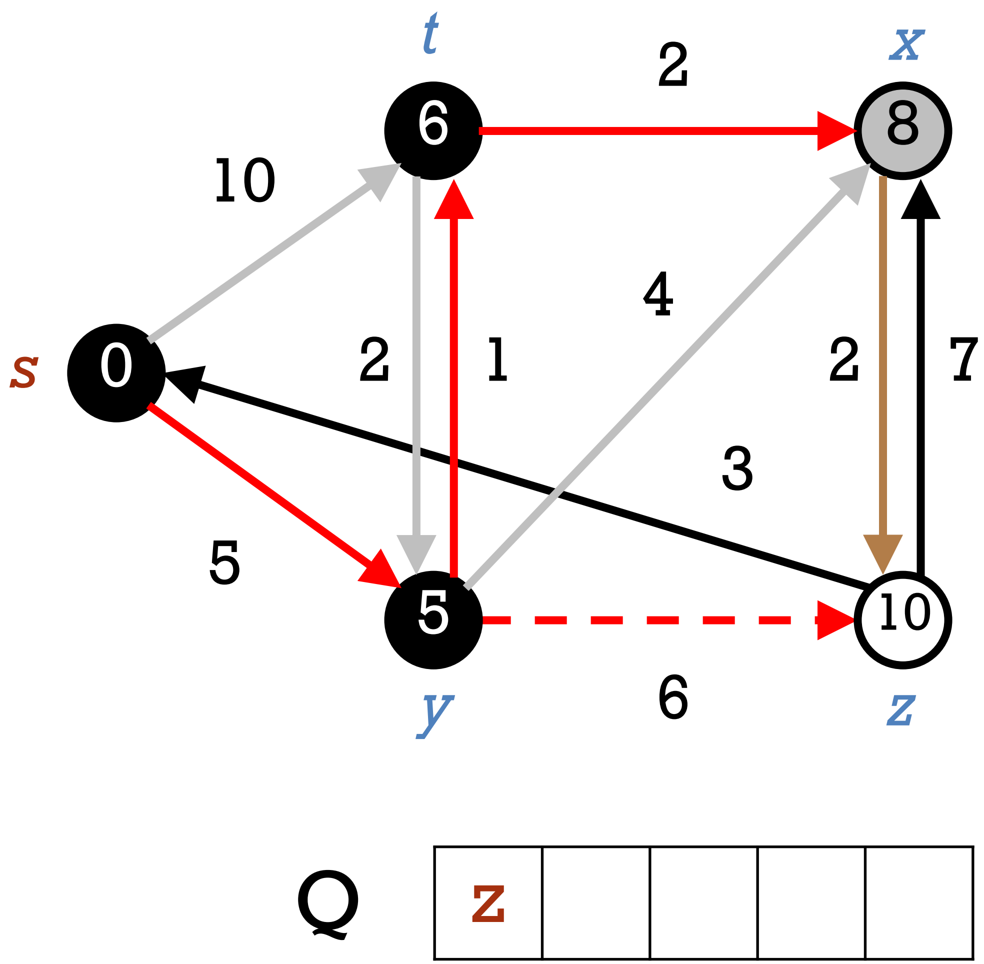
EXAMPLE



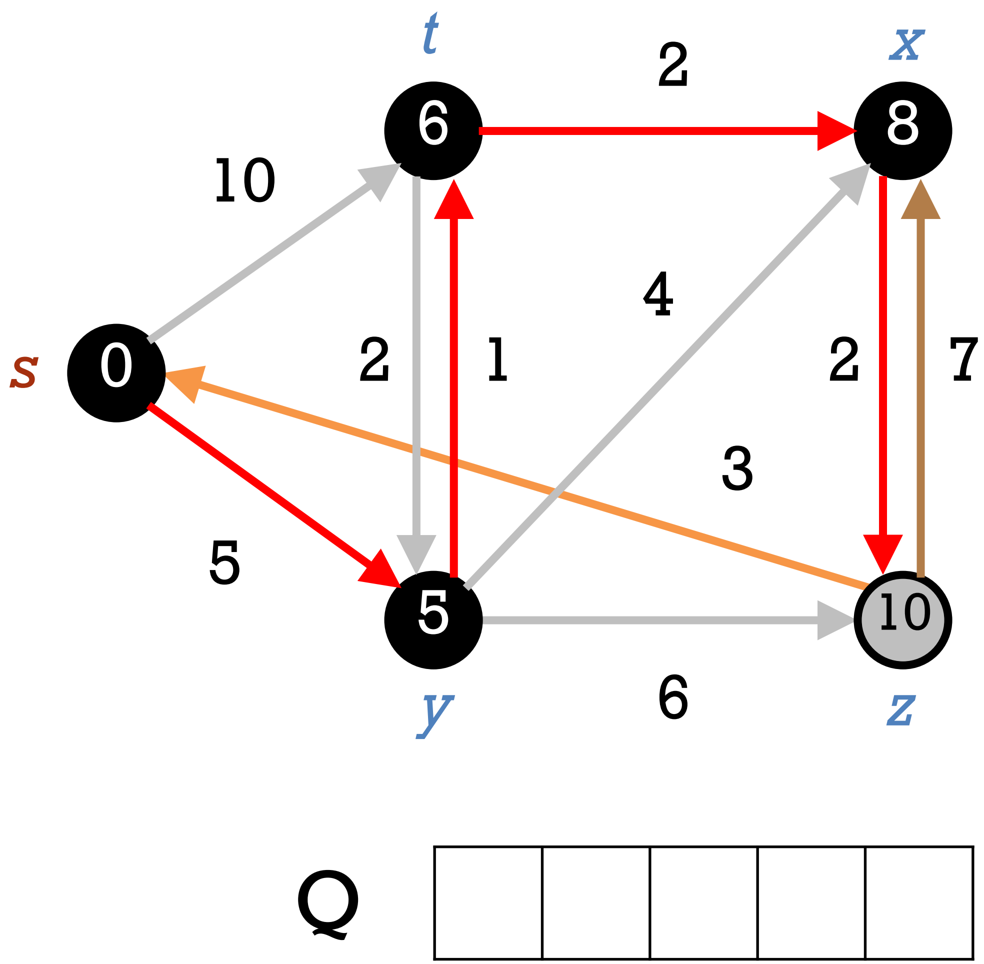
Q

x	z			

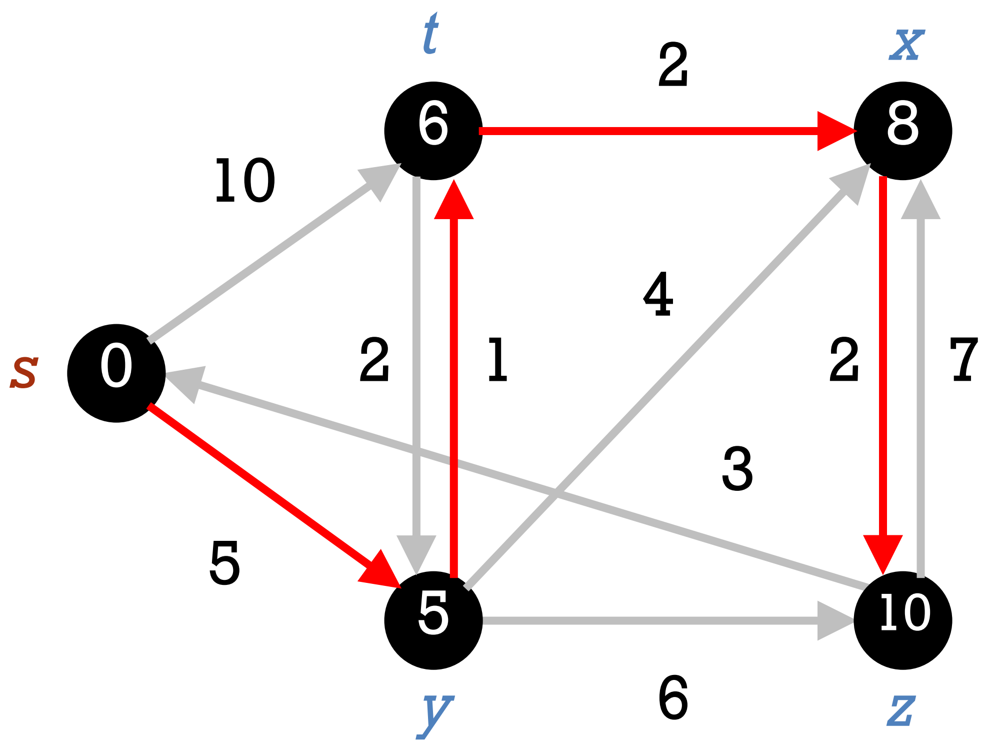
EXAMPLE



EXAMPLE



EXAMPLE




Thank You