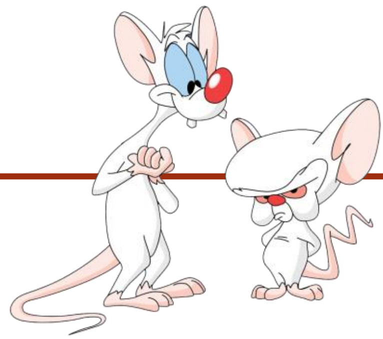
COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Week 14-1 : Graphs 2

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Recursive graph traversal
  - depth first

- Non-recursive graph traversal
  - depth first
  - breadth first

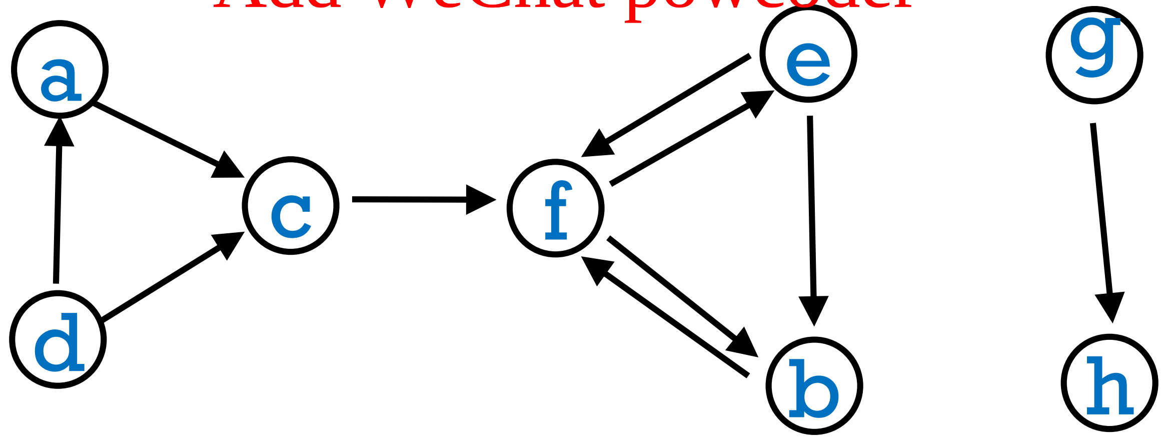
RECALL: TREE TRAVERSAL (RECURSIVE)

```
depthFirst_Tree (root){
   if (root is not empty){
      visit root // preorder
      for each child of root
         depthfirst_Tree( child )
   }
}
```

GRAPH TRAVERSAL (RECURSIVE)

Need to specify a starting vertex.

Visit all nodes that are "reachable" by a path from a starting vertex.



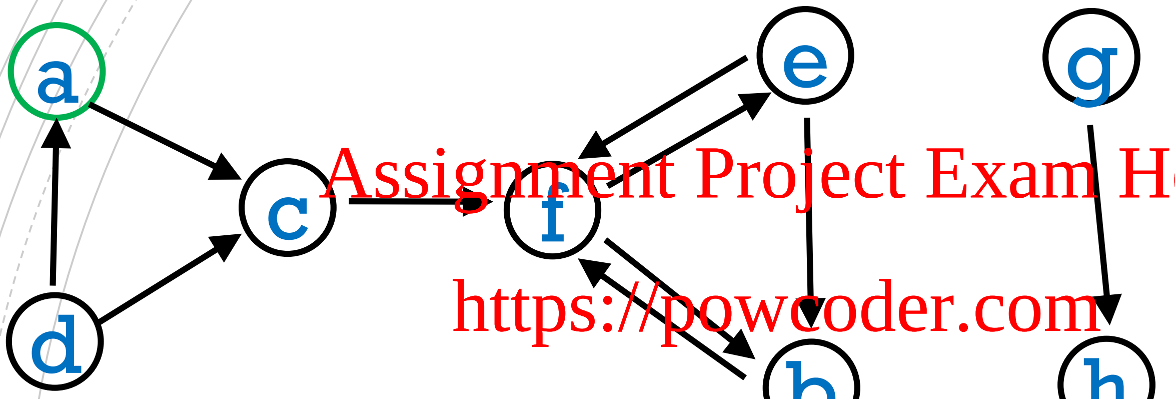
GRAPH TRAVERSAL (RECURSIVE)

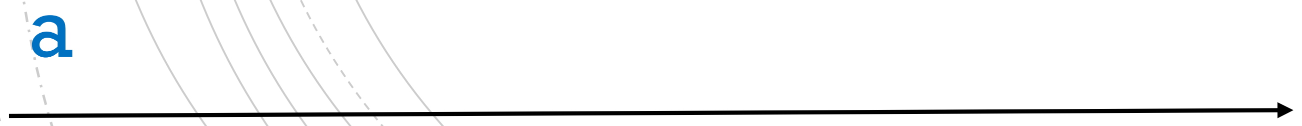
```
depthFirst_Graph (v){
    v.visided = true
    for each w such that (v,w) is in E
    // i.e. v.adjList.contains(w) returns true

        ??
}
```

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)
}
```

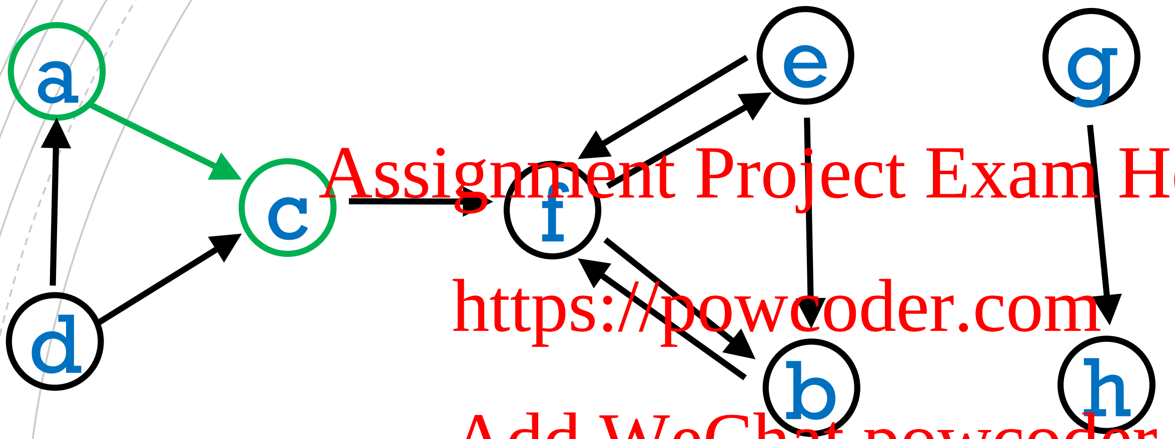
CALL STACK FOR depthFirst(a)

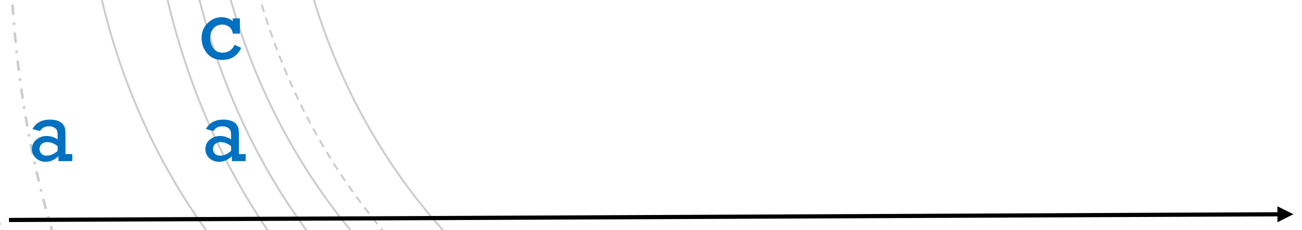




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

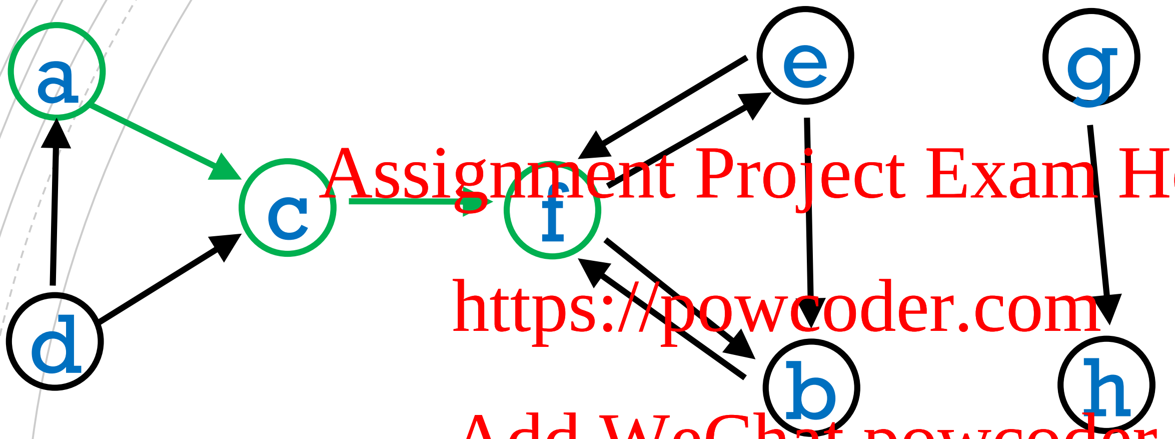
CALL STACK FOR depthFirst(a)

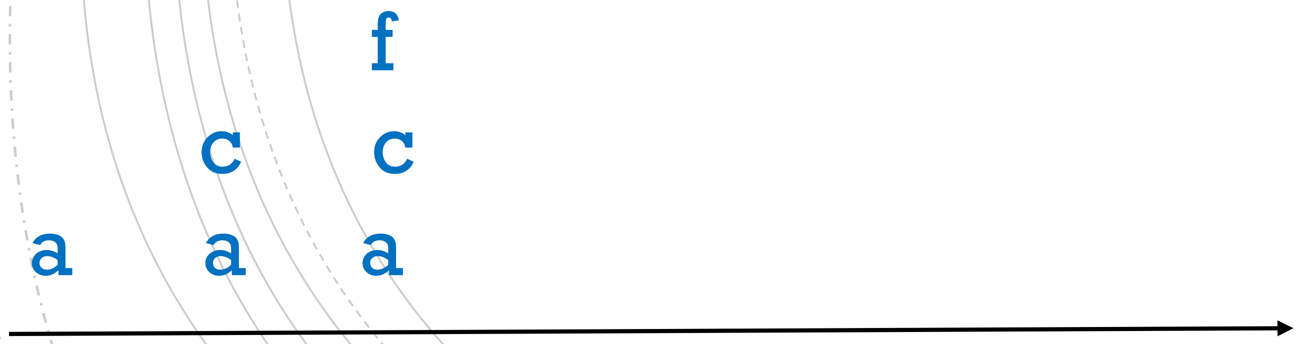




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

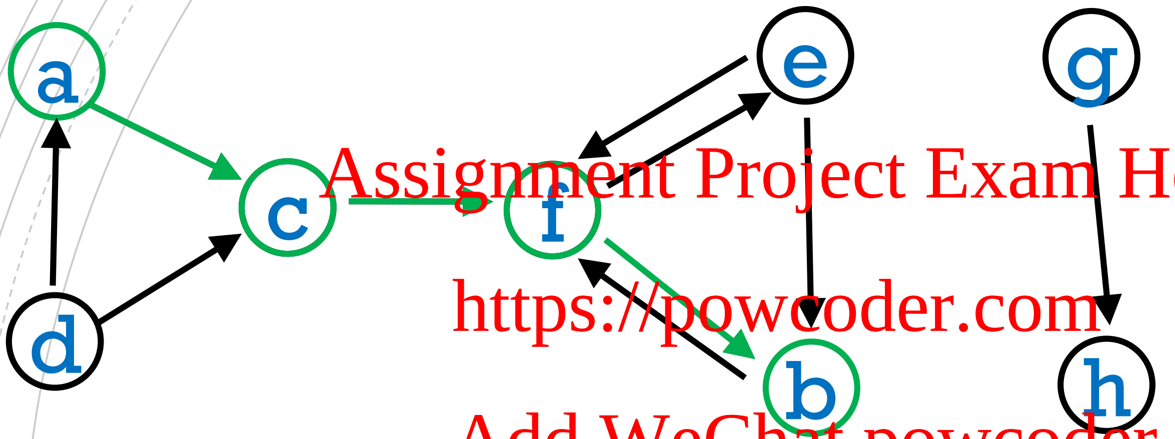
CALL STACK FOR depthFirst(a)

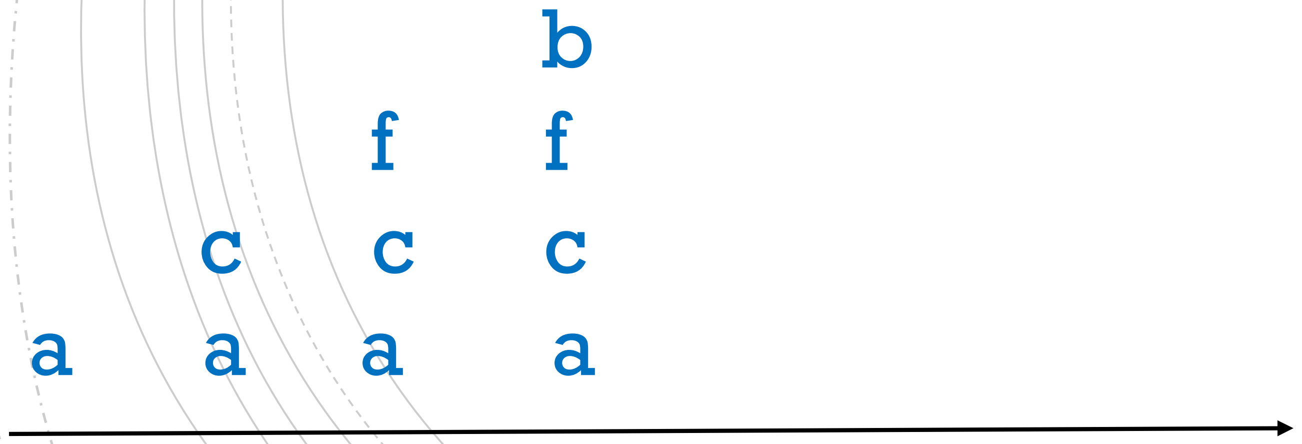




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

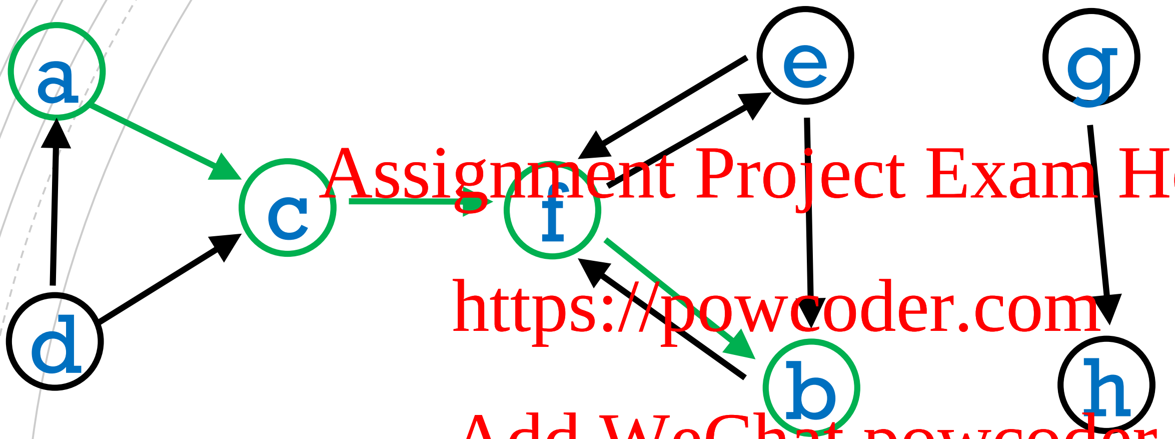
CALL STACK FOR depthFirst(a)

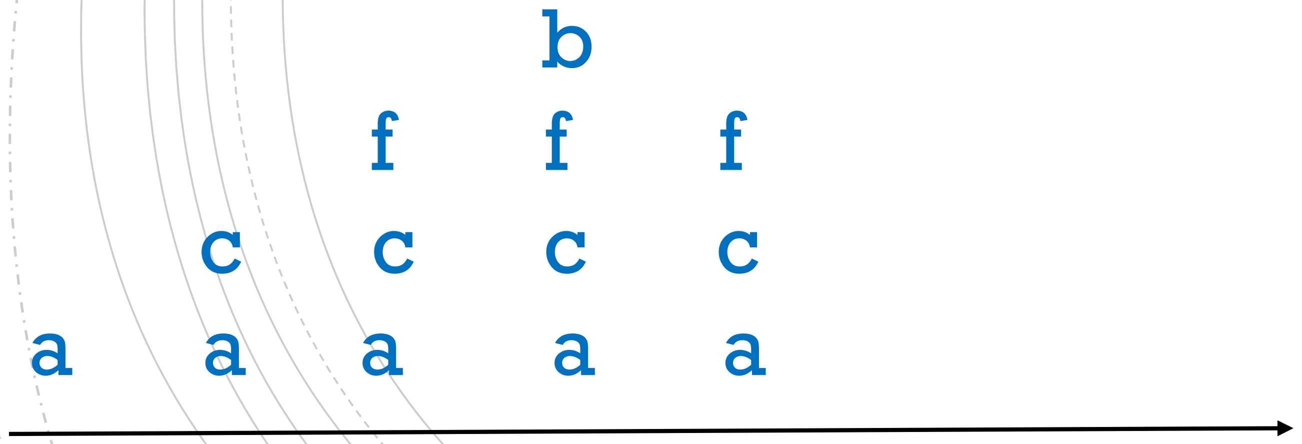




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

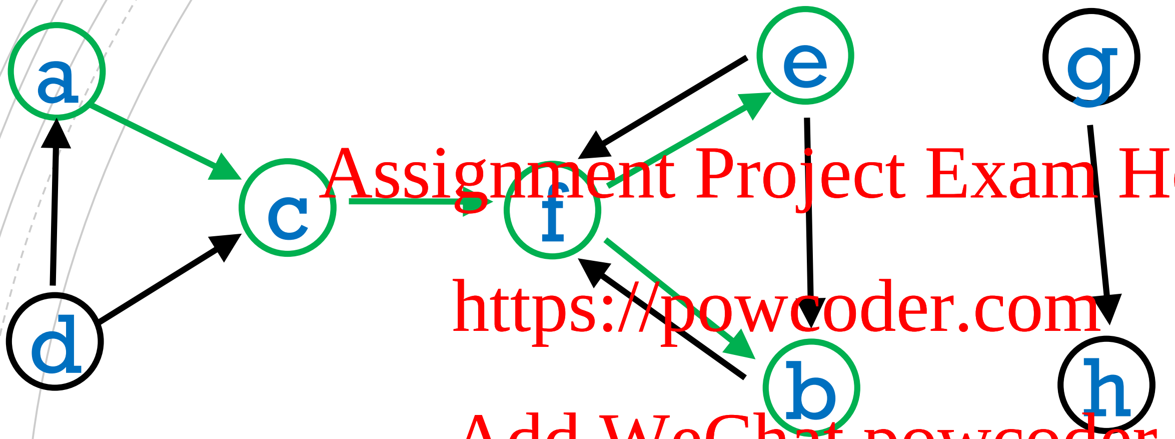
CALL STACK FOR depthFirst(a)

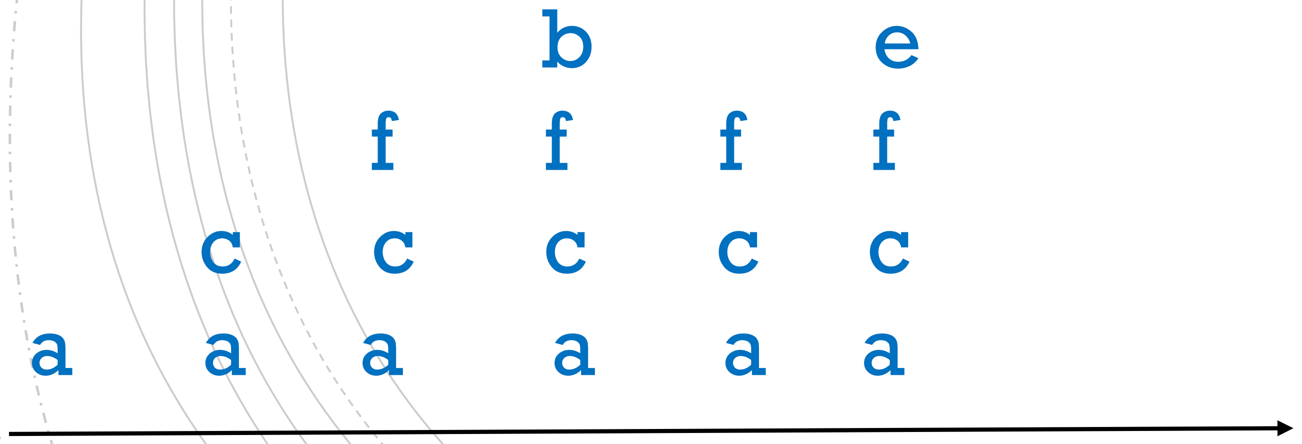




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

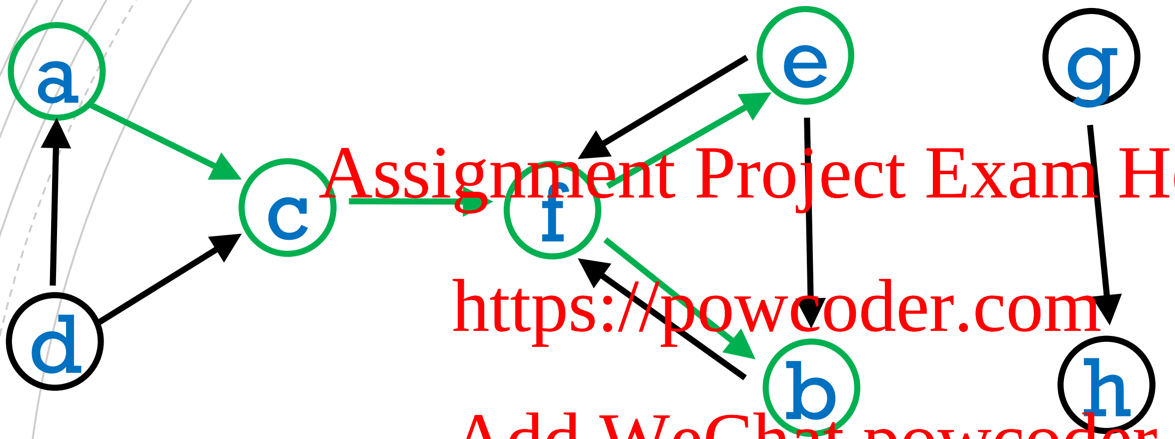
CALL STACK FOR depthFirst(a)

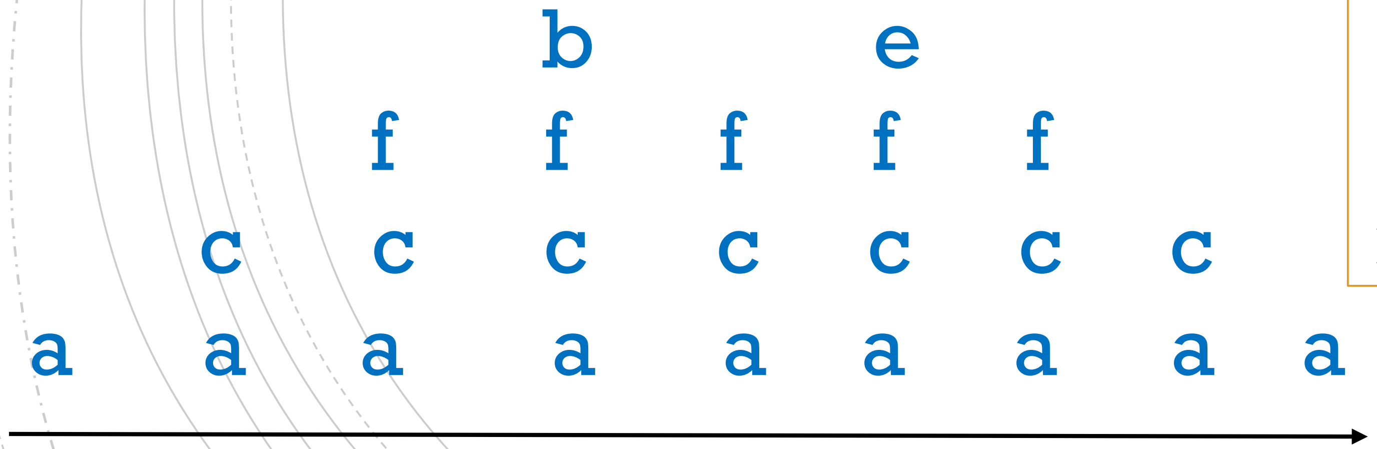




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

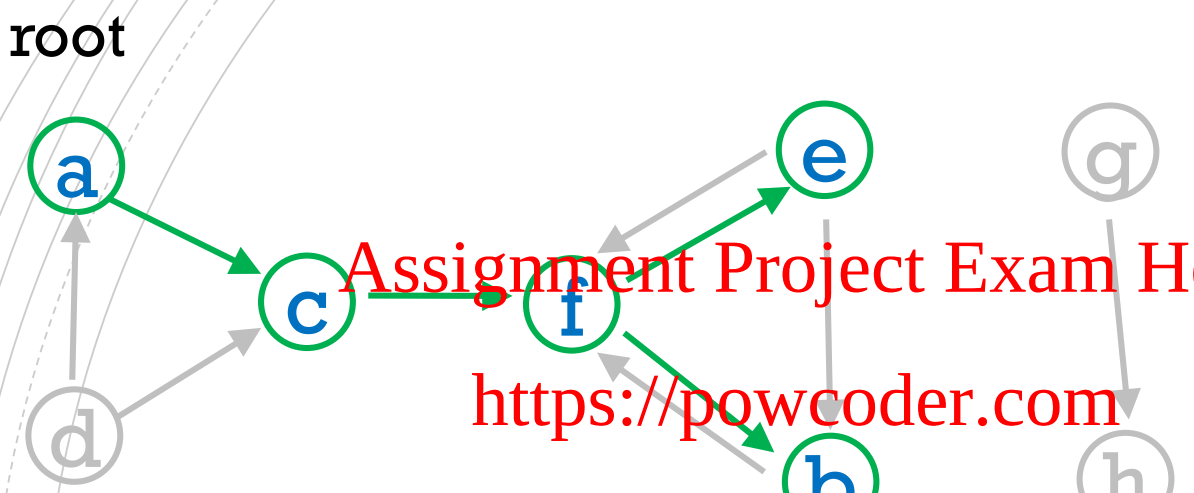
CALL STACK FOR depthFirst(a)

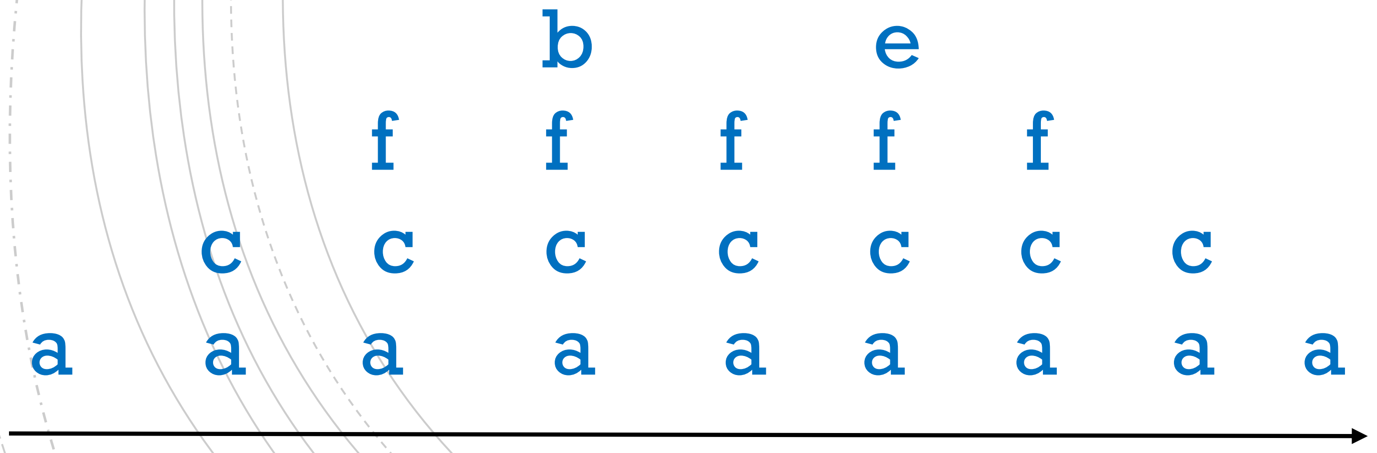




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

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.

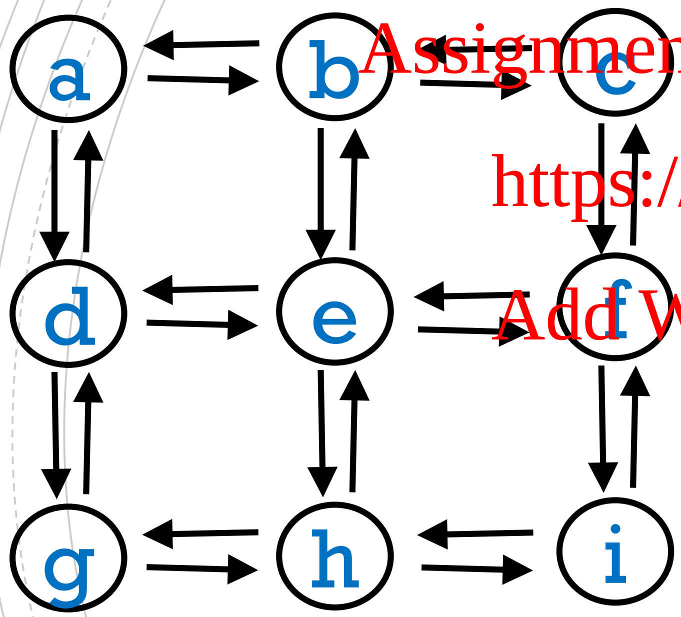
EXAMPLE 2

$$\begin{tikzcd}
a \arrow[r, shift left] \arrow[d, shift right] & b \arrow[l, shift left] \arrow[r, shift left] \arrow[d, shift right] & c \arrow[l, shift left] \arrow[d, shift right] \\
d \arrow[u, shift right] \arrow[r, shift left] \arrow[d, shift right] & e \arrow[u, shift right] \arrow[l, shift left] \arrow[r, shift left] \arrow[d, shift right] & f \arrow[u, shift right] \arrow[l, shift left] \arrow[d, shift right] \\
g \arrow[u, shift right] \arrow[r, shift left] & h \arrow[u, shift right] \arrow[l, shift left] \arrow[r, shift left] & i \arrow[u, shift right] \arrow[l, shift left]
\end{tikzcd}$$

Adjacency List

a - (b,d)
b - (a,c,e)
c - (b,f)
d - (a,e,g)
e - (b,d,f,h)
f - (c,e,i)
g - (d,h)
h - (e,g,i)
i - (f,h)

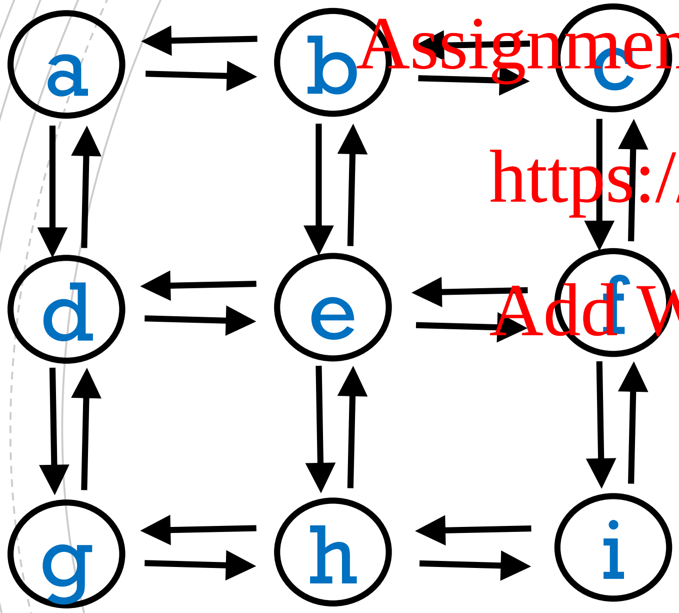
EXAMPLE 2

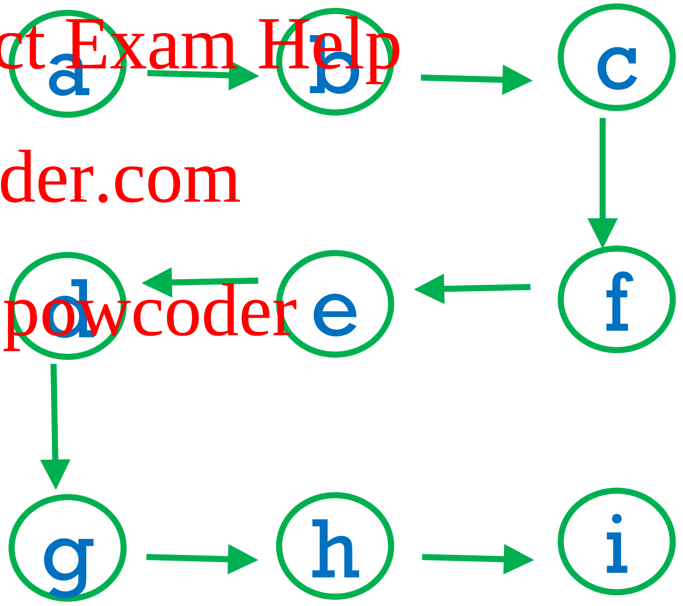


What is the call tree for **`depthFirst`**`(a)` ?

(Do it in your head)

EXAMPLE 2





call tree for **`depthFirst(a)`**

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)
            }
      }
   }
}
```

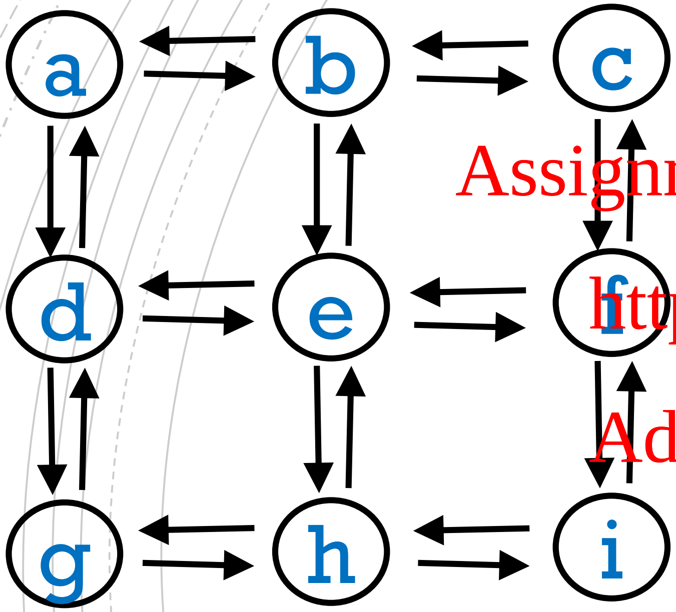
Indicate as “reached” a node *before pushing* it onto the stack. We do that by updating the field `visited`.

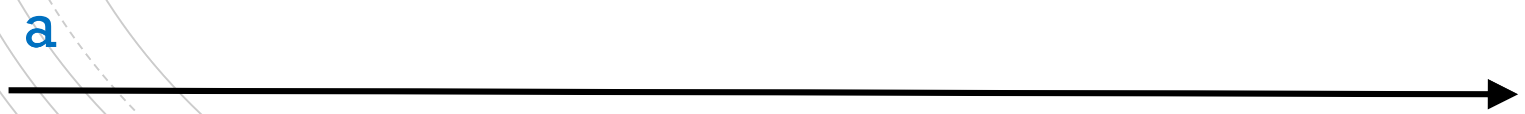
Visit the node (perform some operations) after it gets popped from the stack.

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

EXAMPLE: graphTraversalUsingStack(a)

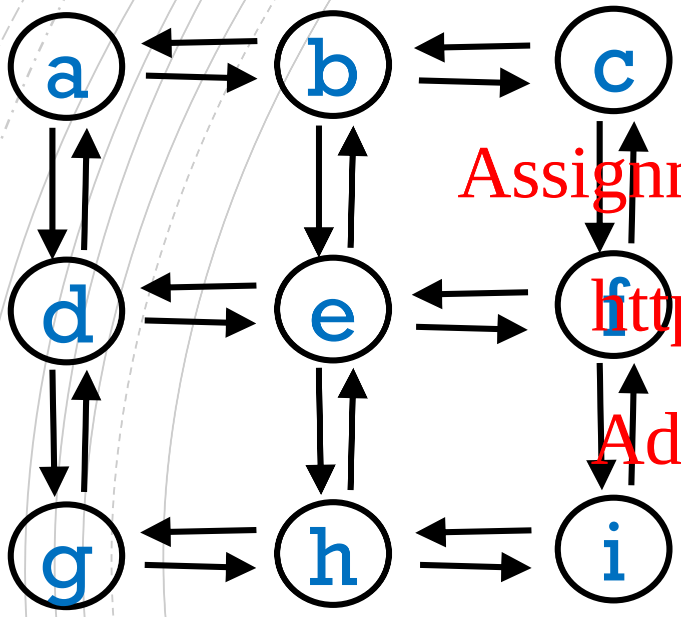
Order of visit: a

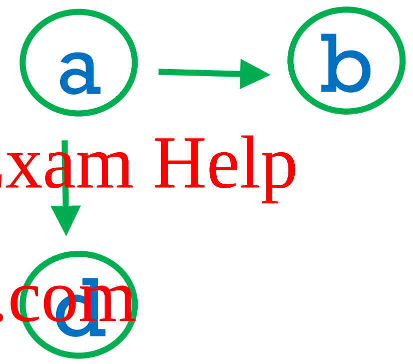


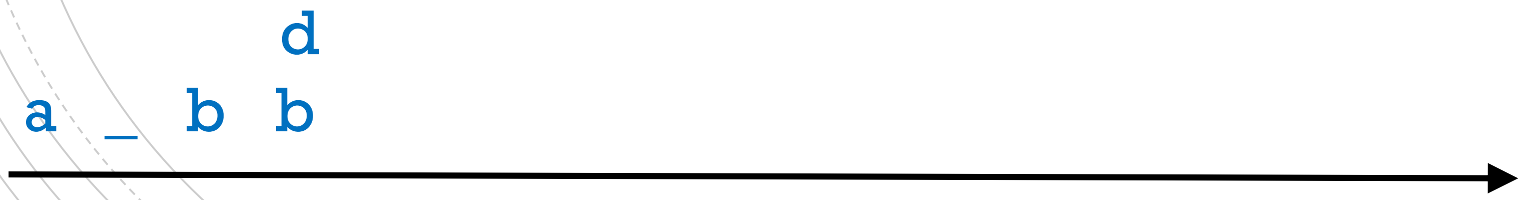


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: a

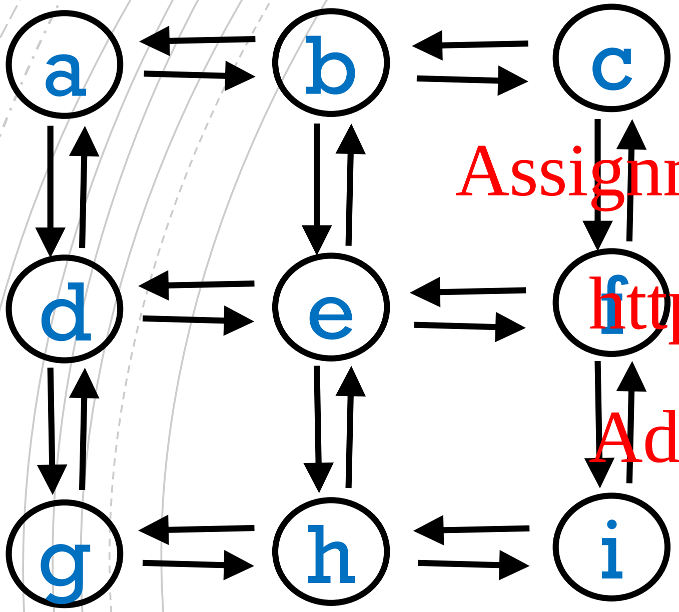


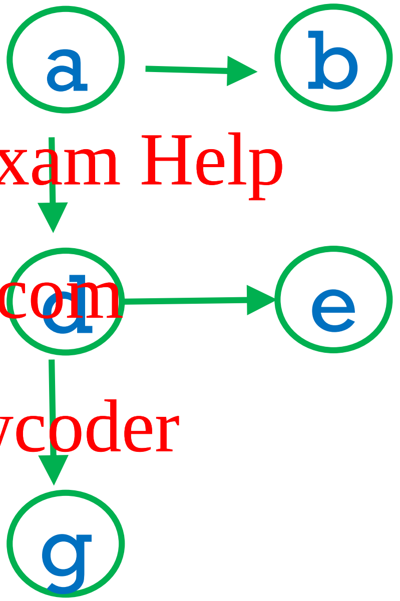


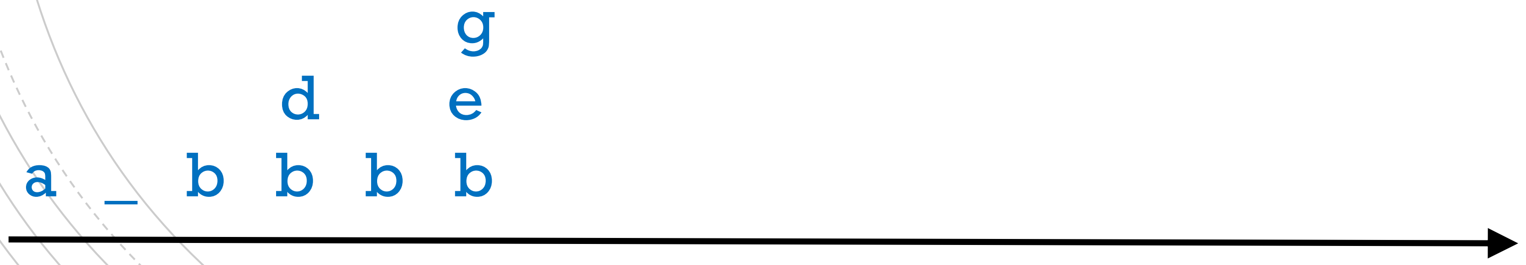


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **ad**

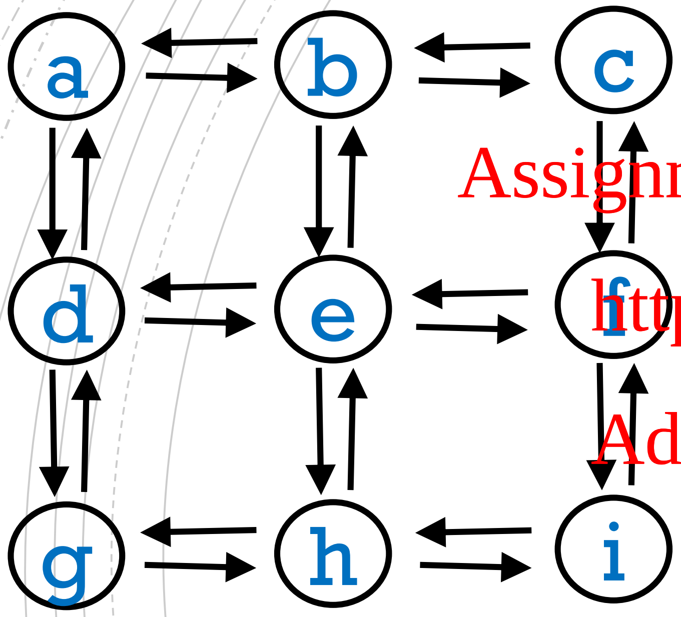


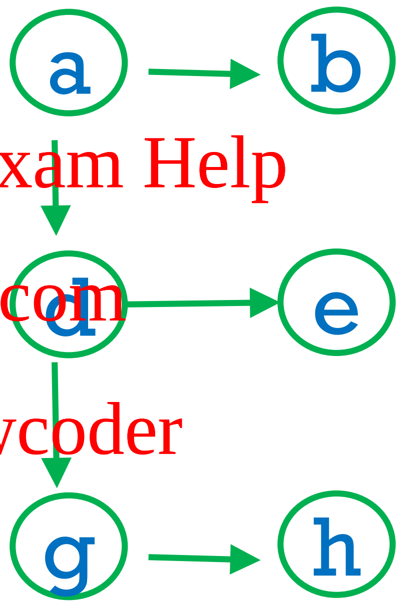


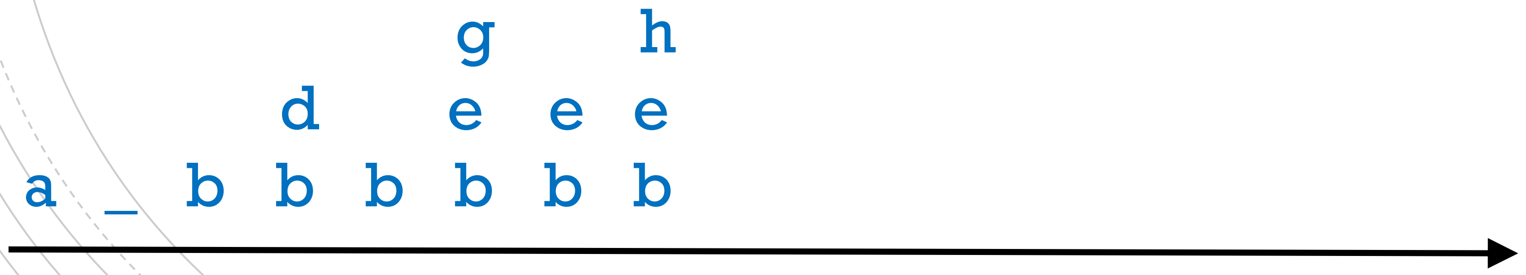


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adg**

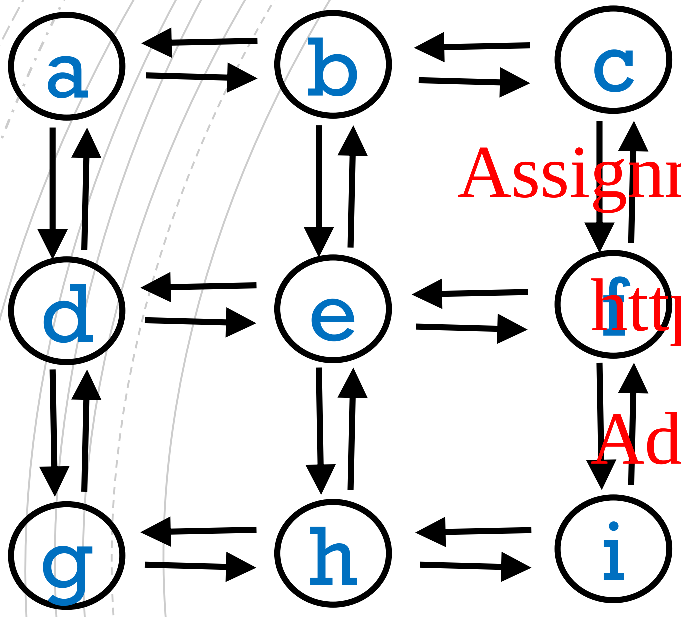


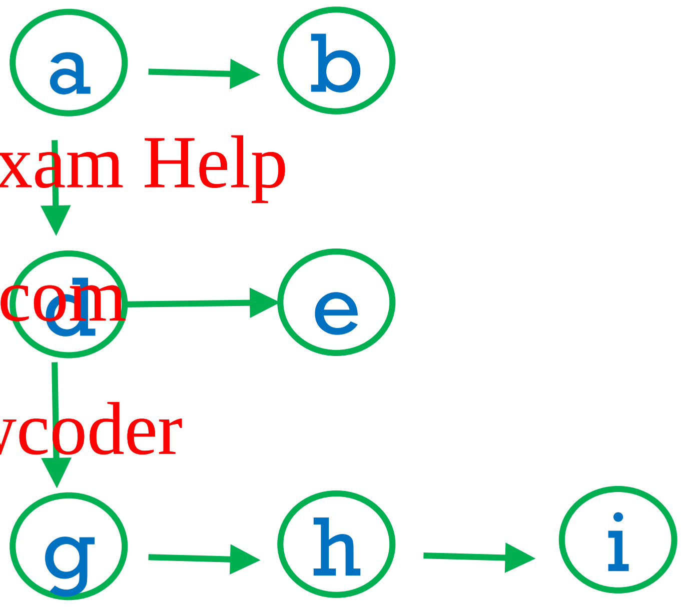


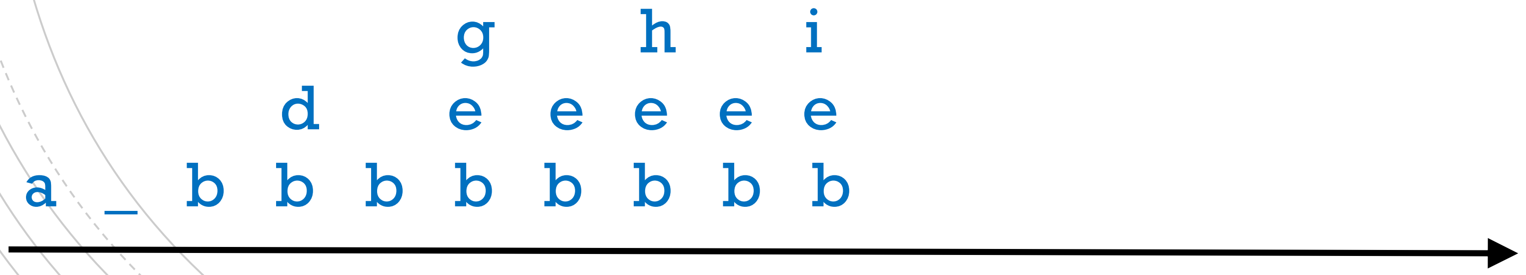


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adgh**

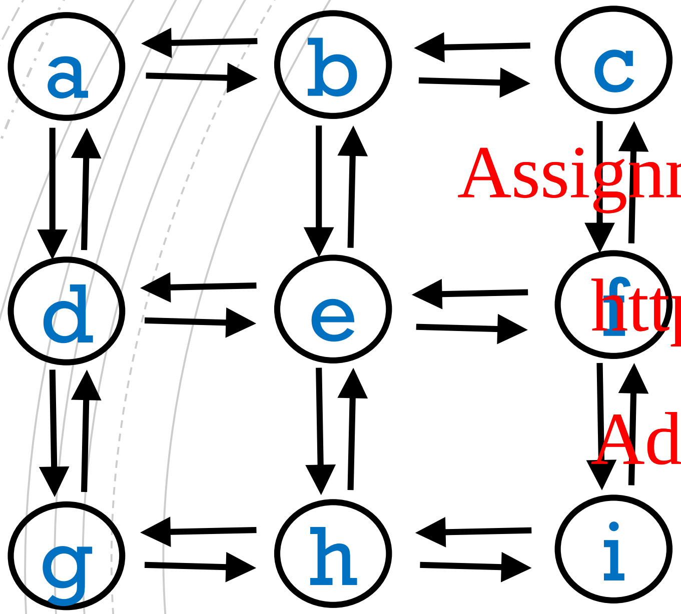


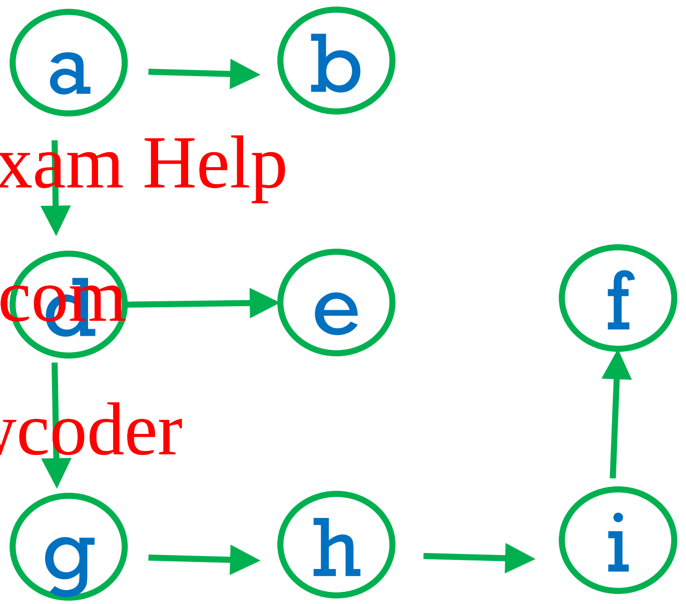


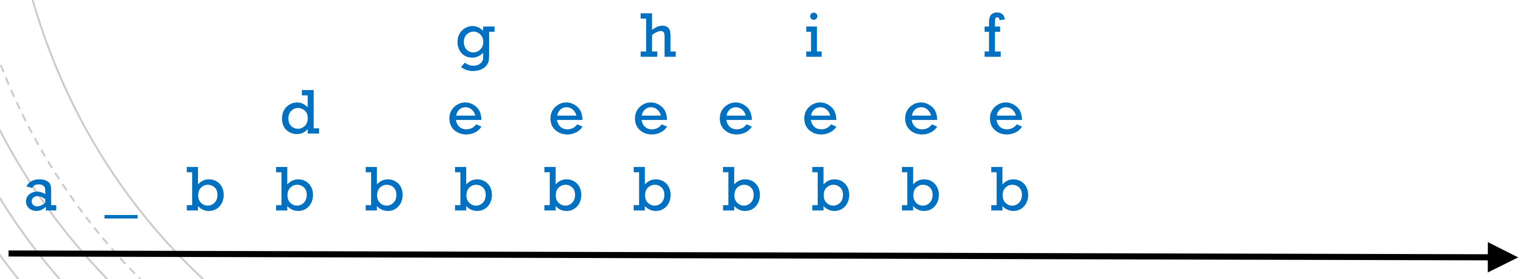


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghi**

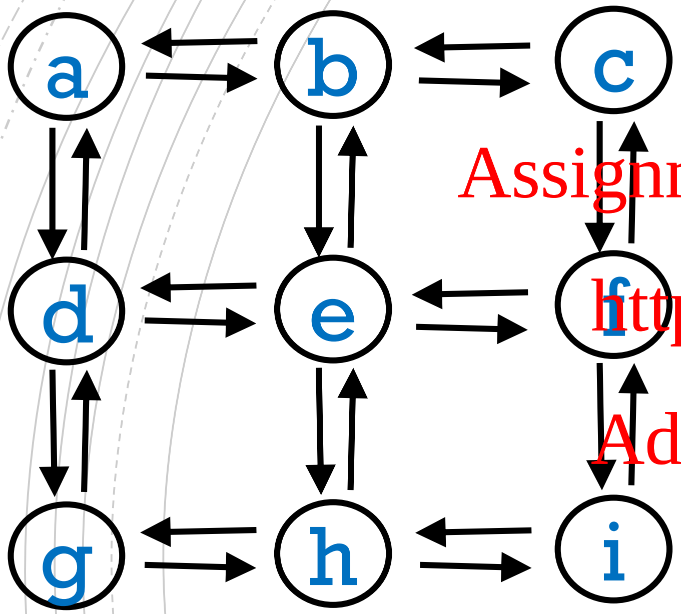


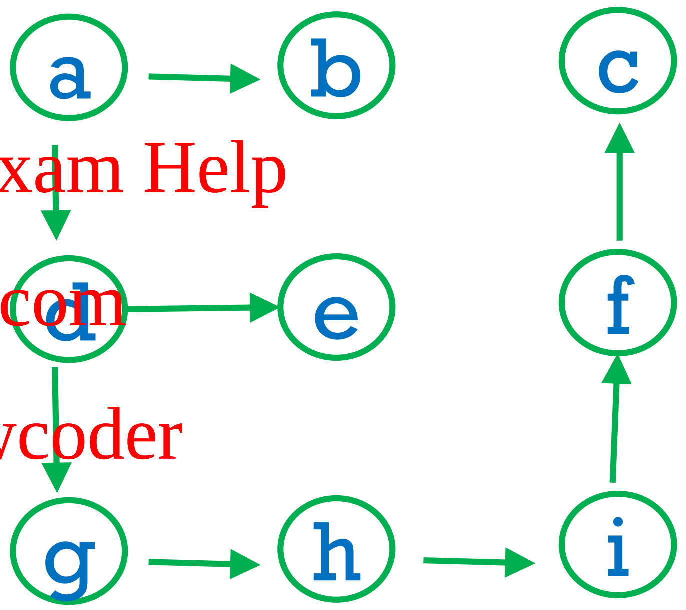


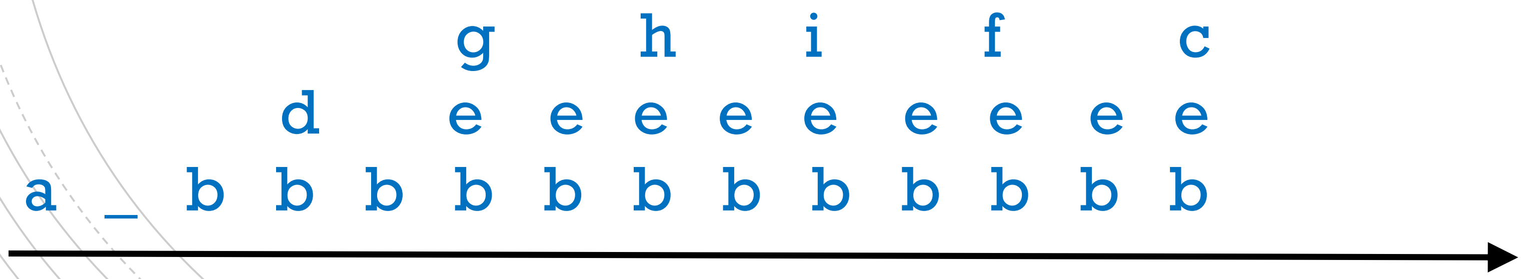


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghif**

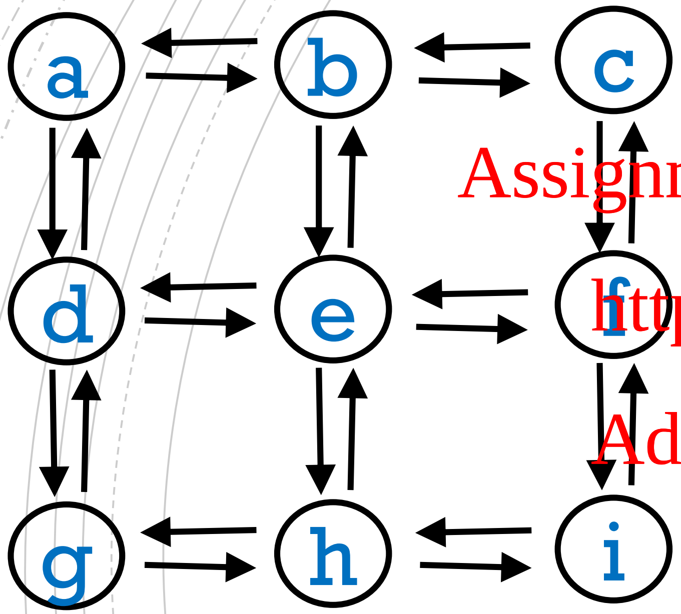


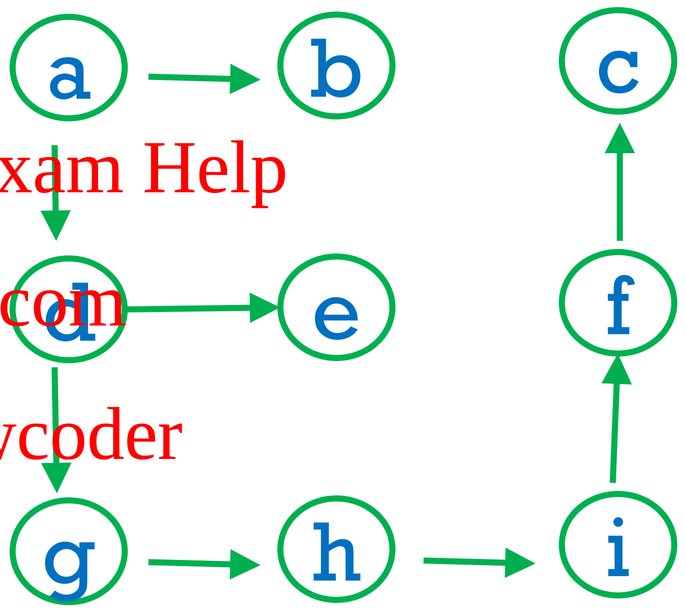




EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghifceb**

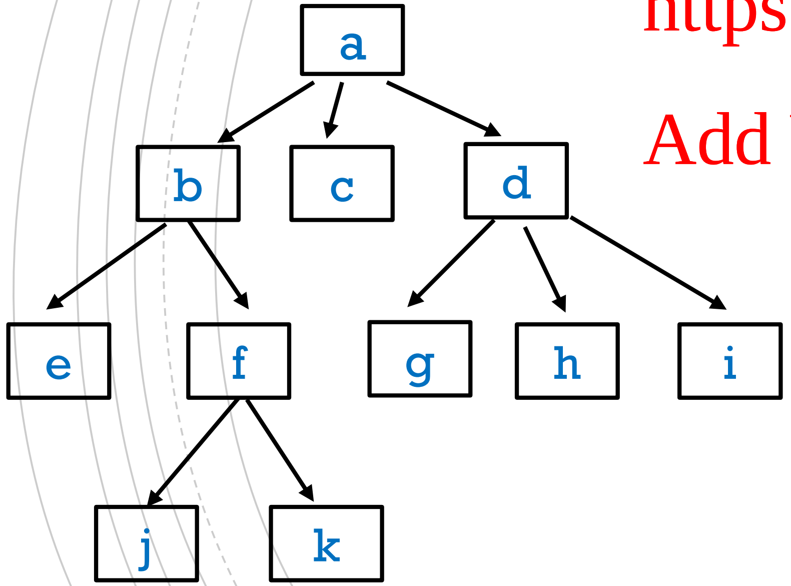




```
          g   h   i   f   c
      d   e e e e e e e e e e
a _ b b b b b b b b b b b b b b _
```

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

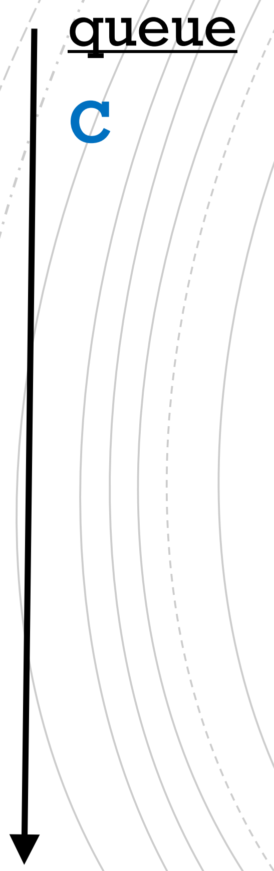
Given an input vertex, visit all vertices that can be reached by paths of length 1, 2, 3, 4,….

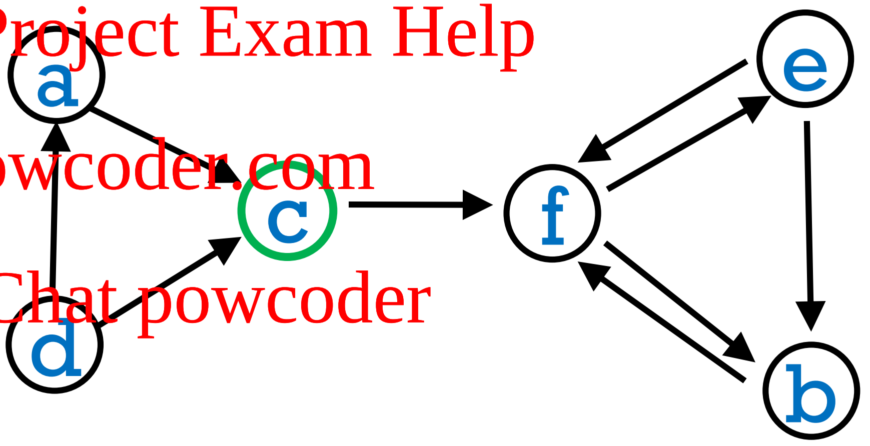
BREADTH FIRST GRAPH TRAVERSAL

```
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

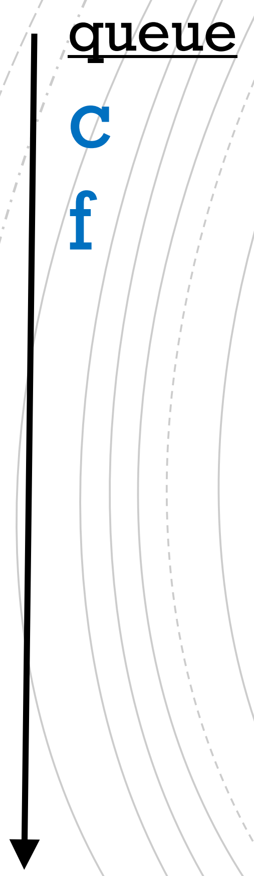
graphTraversalUsingQueue(c)

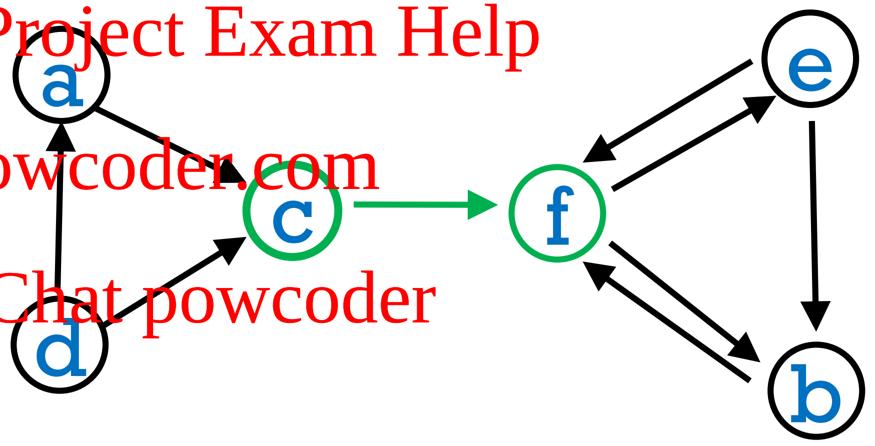




EXAMPLE

graphTraversalUsingQueue(c)





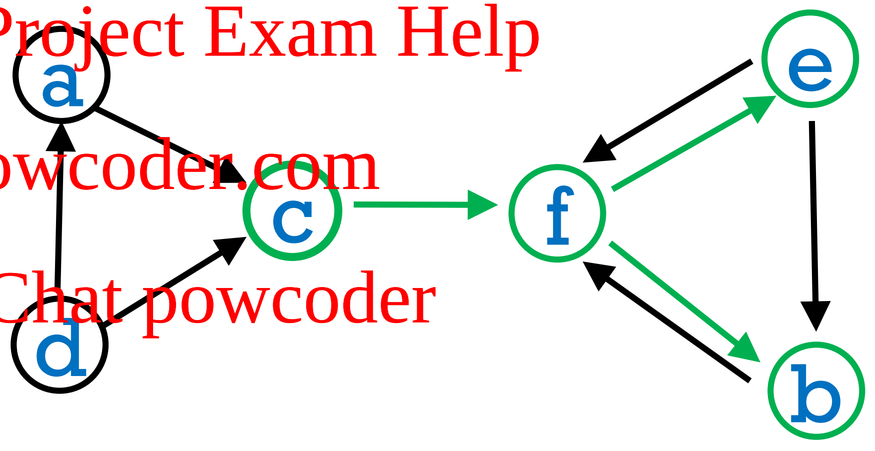
EXAMPLE

graphTraversalUsingQueue(c)

<u>queue</u>

c
f
be

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

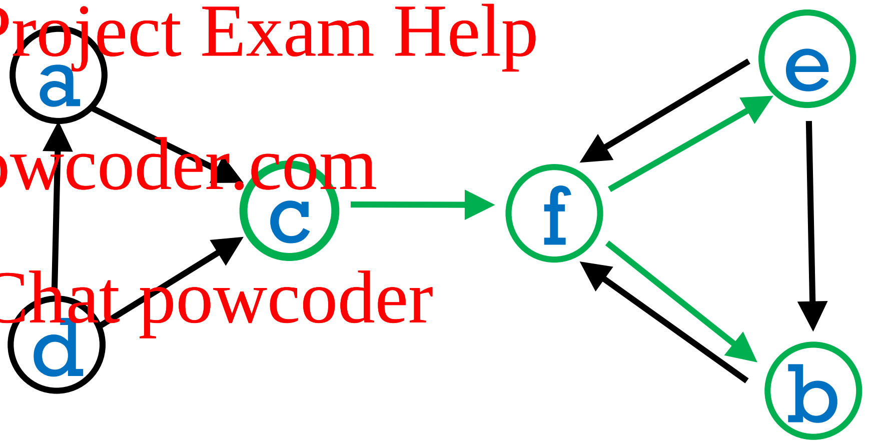


EXAMPLE

graphTraversalUsingQueue(c)

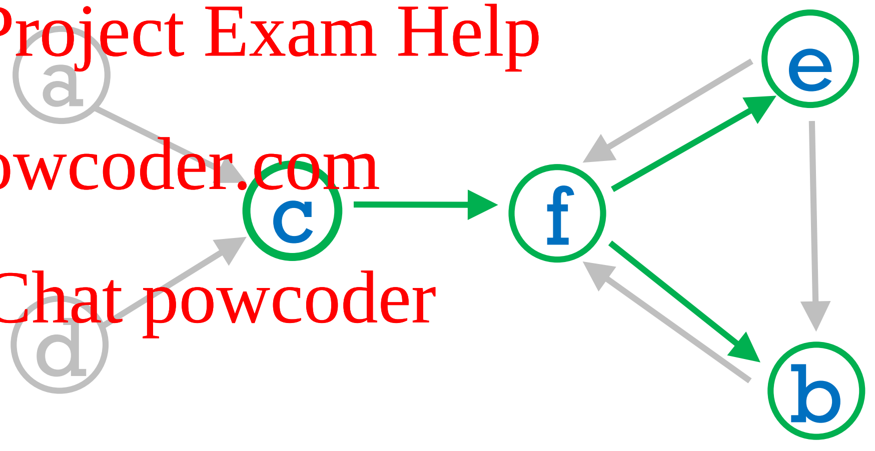
queue

c
f
be
e
_



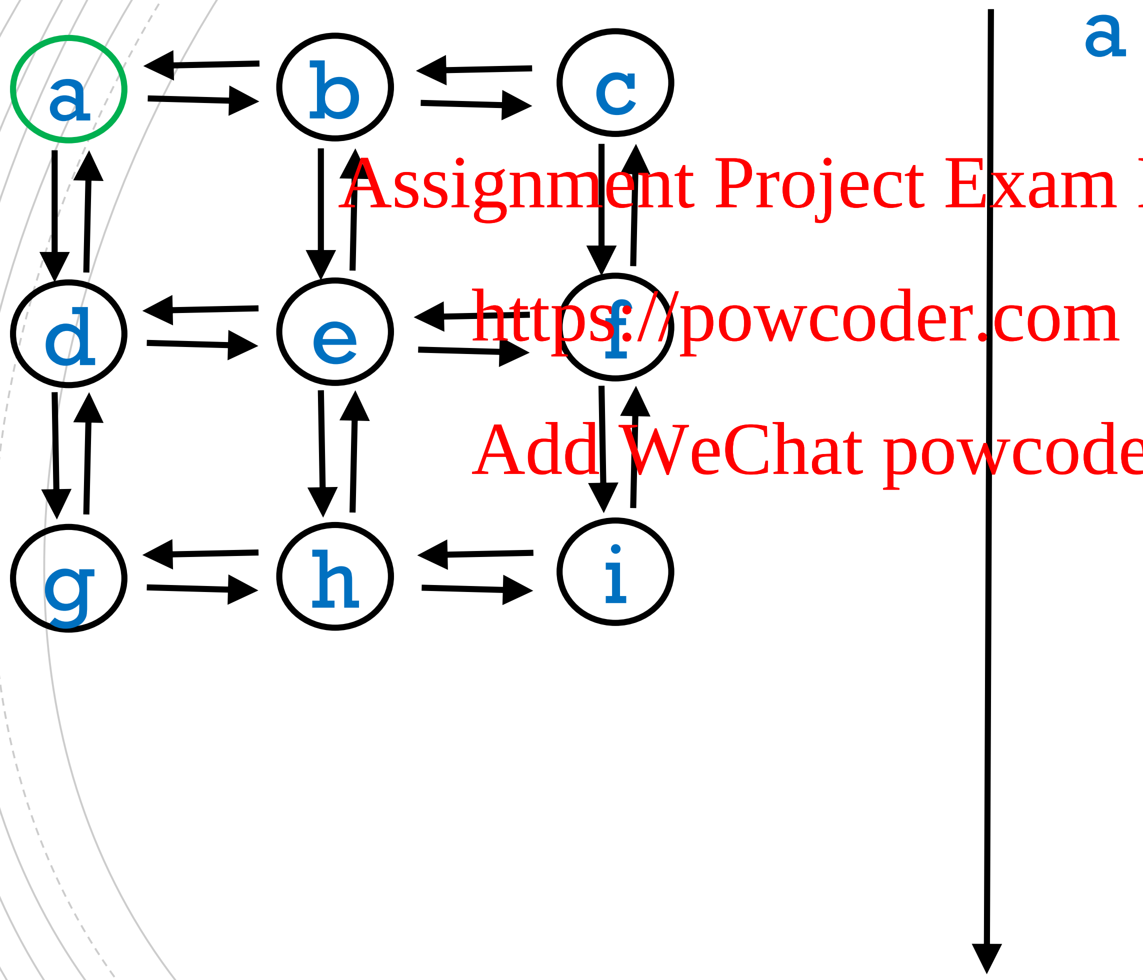
EXAMPLE

graphTraversalUsingQueue(c)

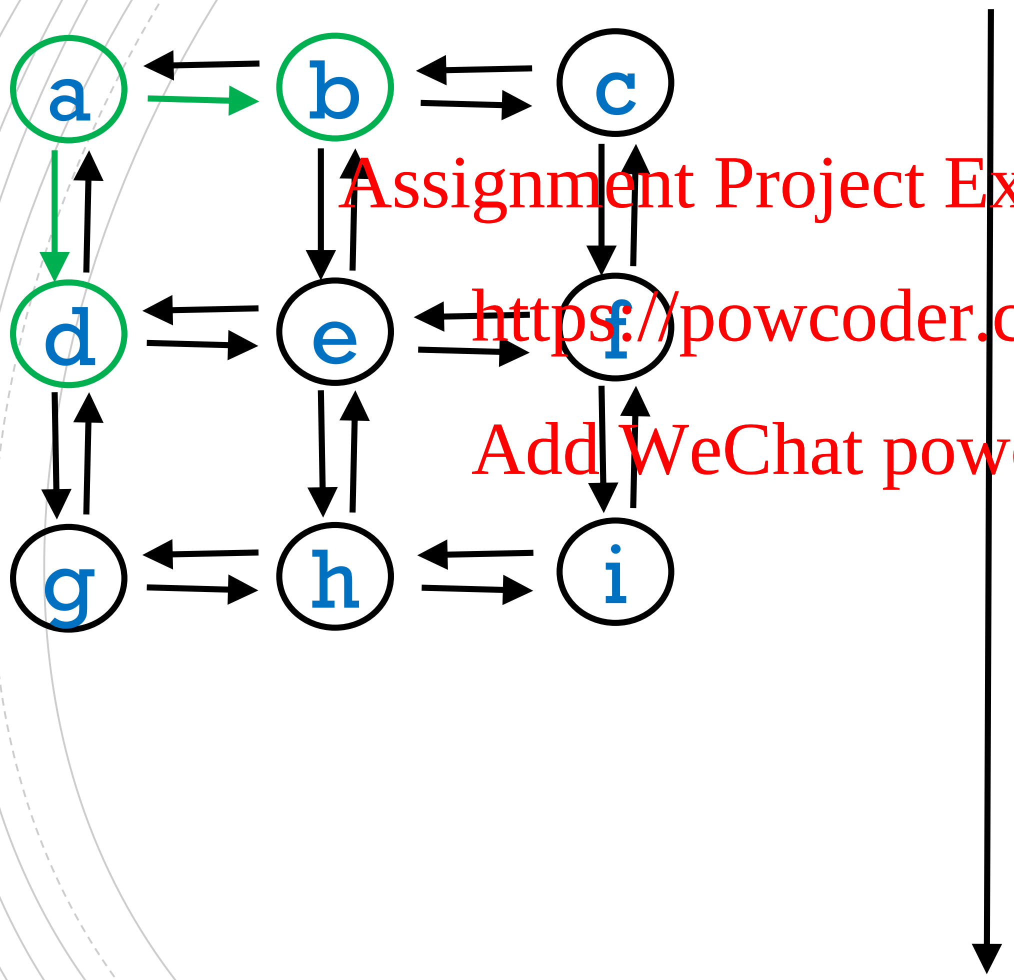


It defines a tree whose root is the starting vertex.
It finds the shortest path (number of edges) to all vertices reachable from the starting vertex.

EXAMPLE: graphTraversalUsingQueue(a)



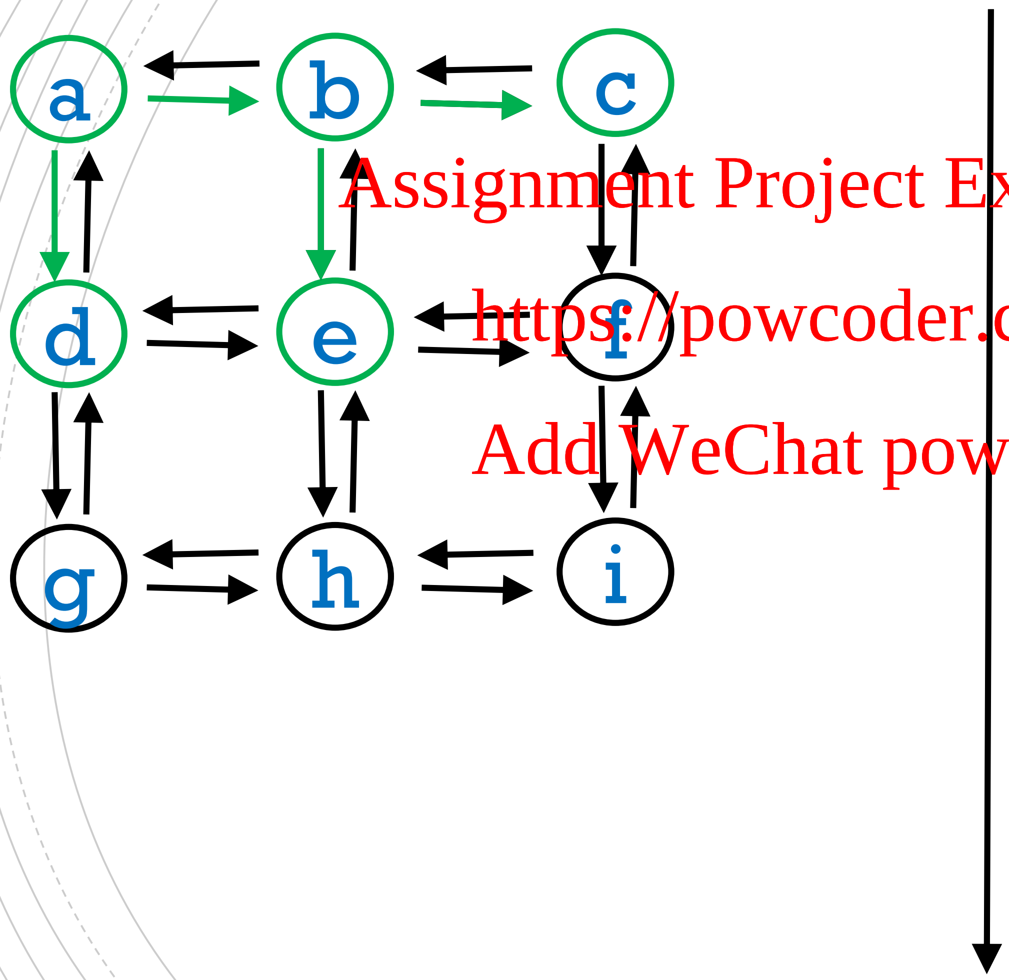
EXAMPLE: graphTraversalUsingQueue(a)



a

bd

EXAMPLE: graphTraversalUsingQueue(a)

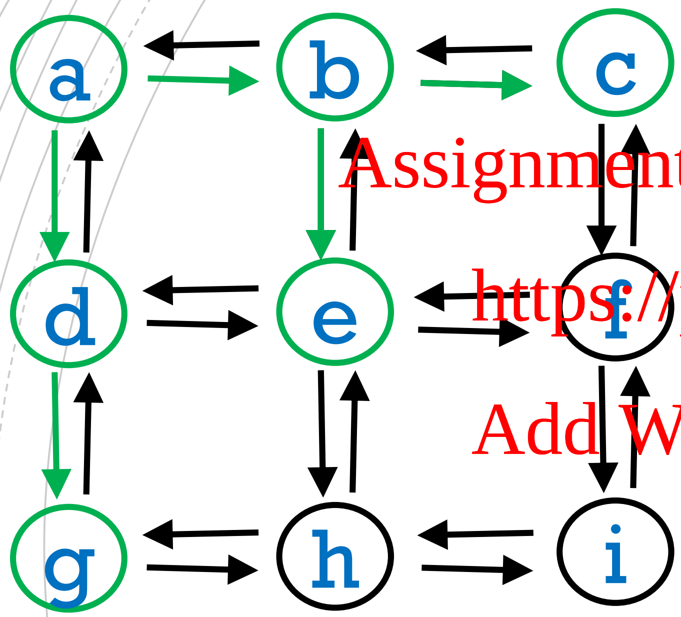


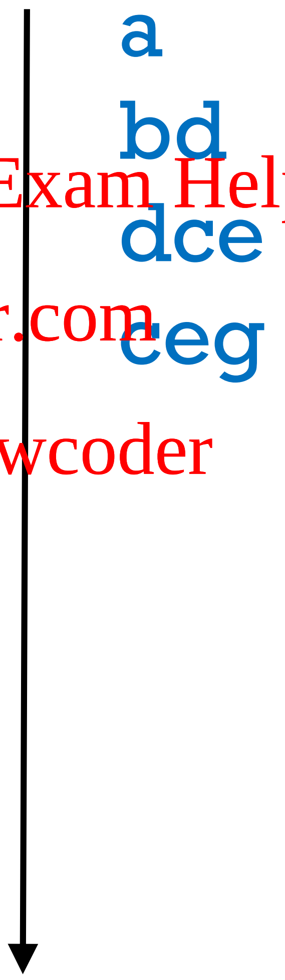
a

bd

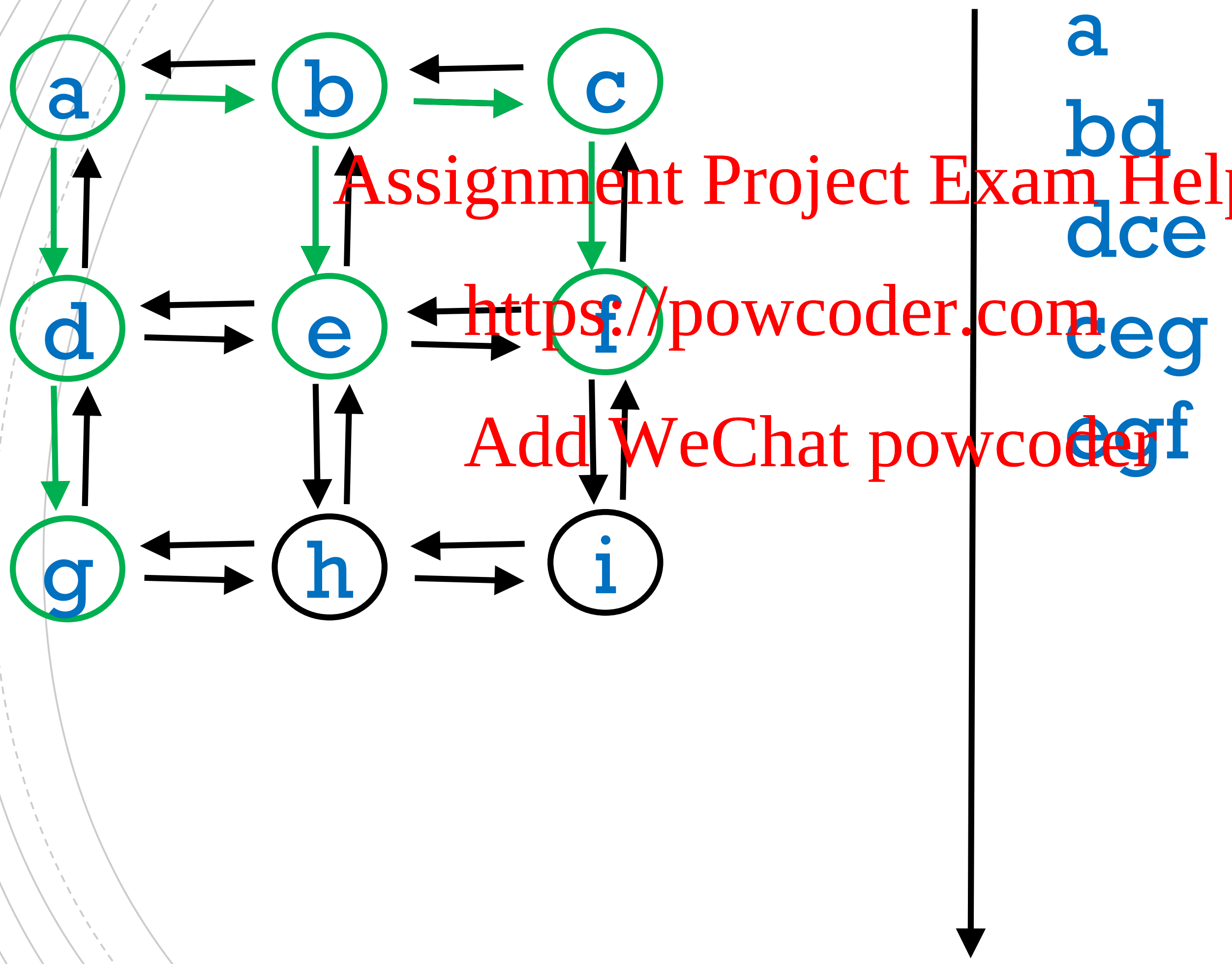
dce

EXAMPLE: graphTraversalUsingQueue(a)

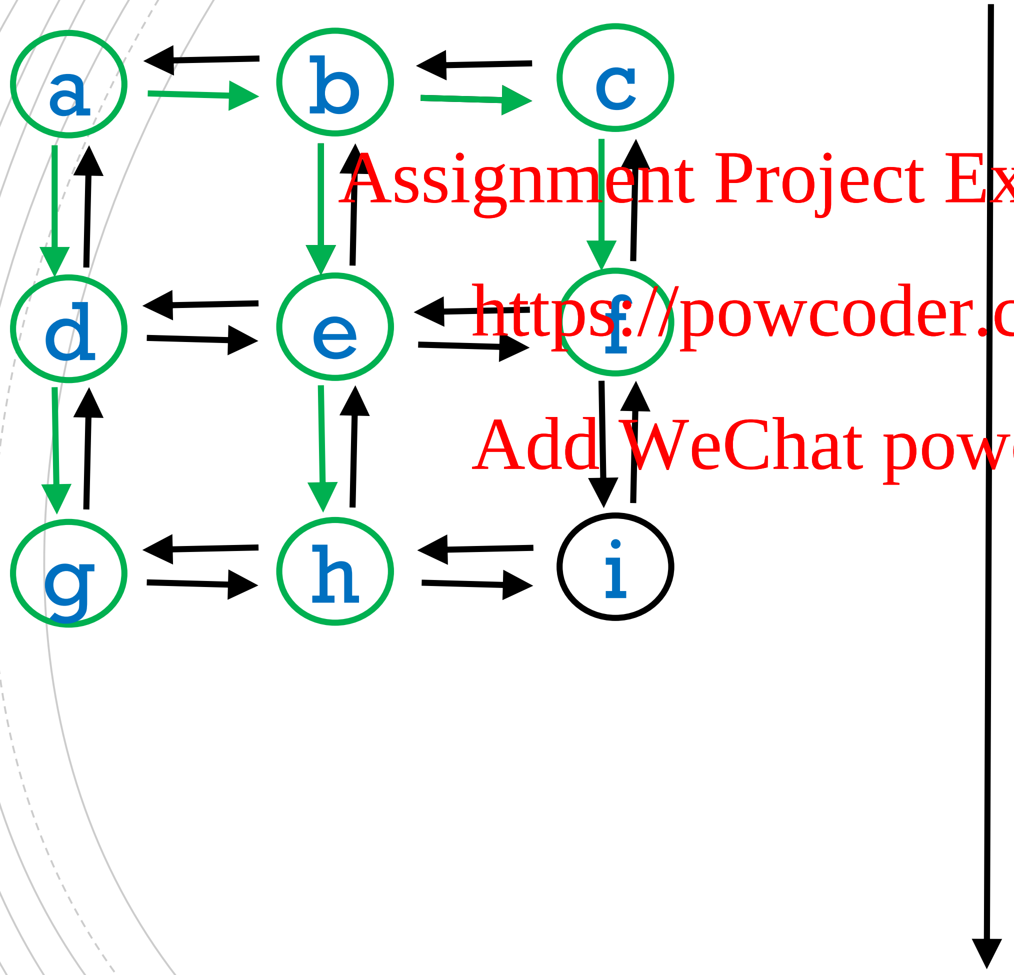




EXAMPLE: graphTraversalUsingQueue(a)

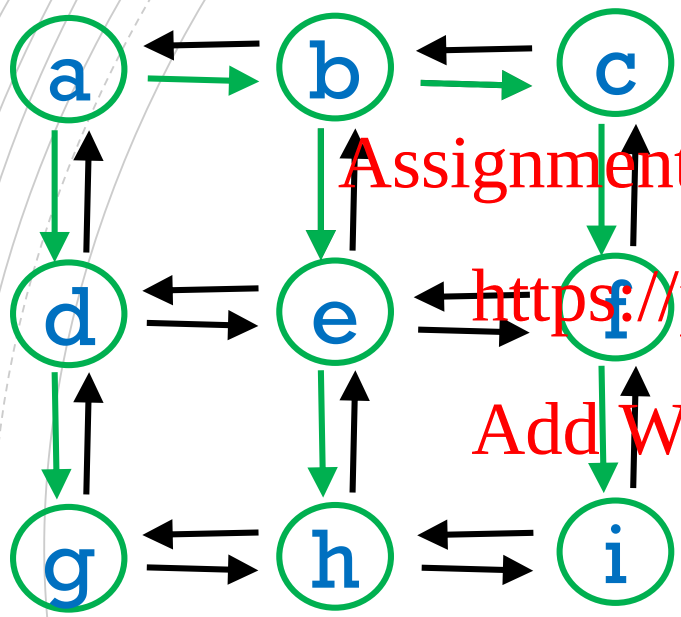


EXAMPLE: graphTraversalUsingQueue(a)



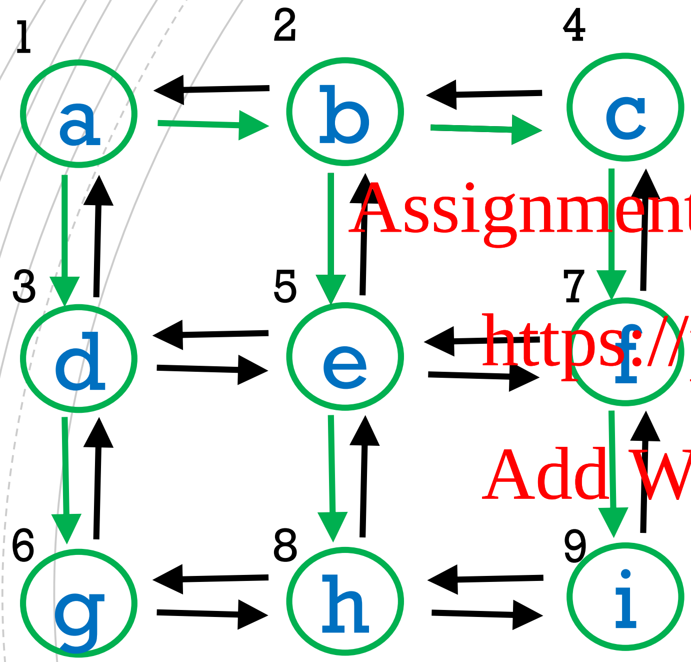
a
bd
dce
ceg
egf
gfh

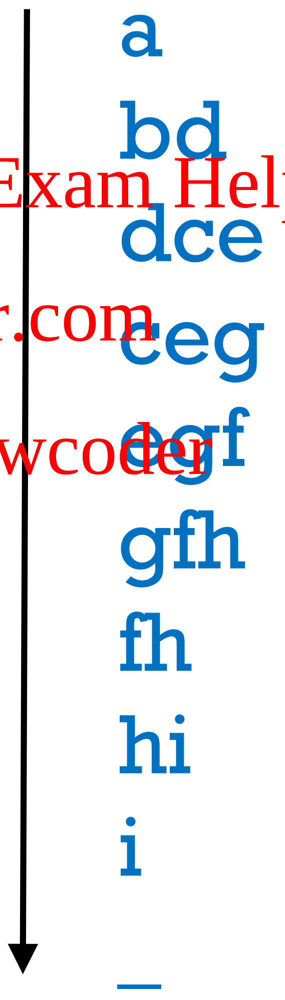
EXAMPLE: graphTraversalUsingQueue(a)



a
bd
dce
ceg
egf
gfh
fh
hi

EXAMPLE: graphTraversalUsingQueue(a)





Note order of nodes visited: We get paths of length 1, the paths of length 2, etc. i.e. breadth first.

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;
   }
}
```

Thank You