COMP 250

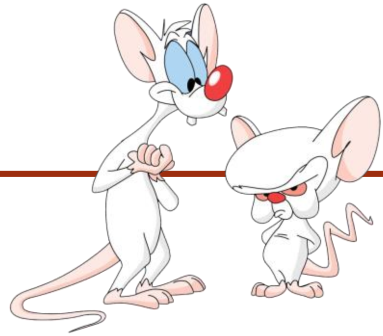
INTRODUC TER SCIENCE

Week 14-1 :

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

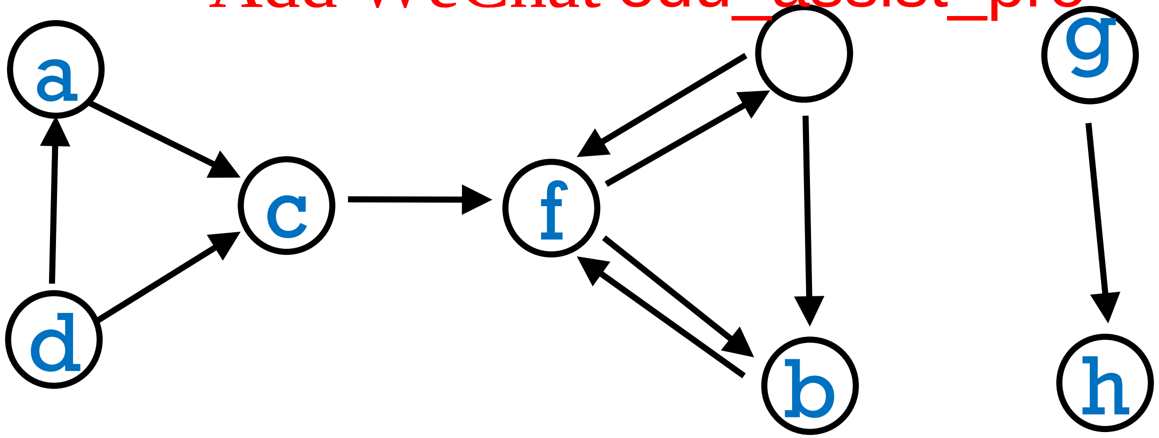
RECALL: TREE TRAVERSAL (RECURSIVE)

```
depthFirst_Tree (root){
    if (root is no
        visit root
        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.



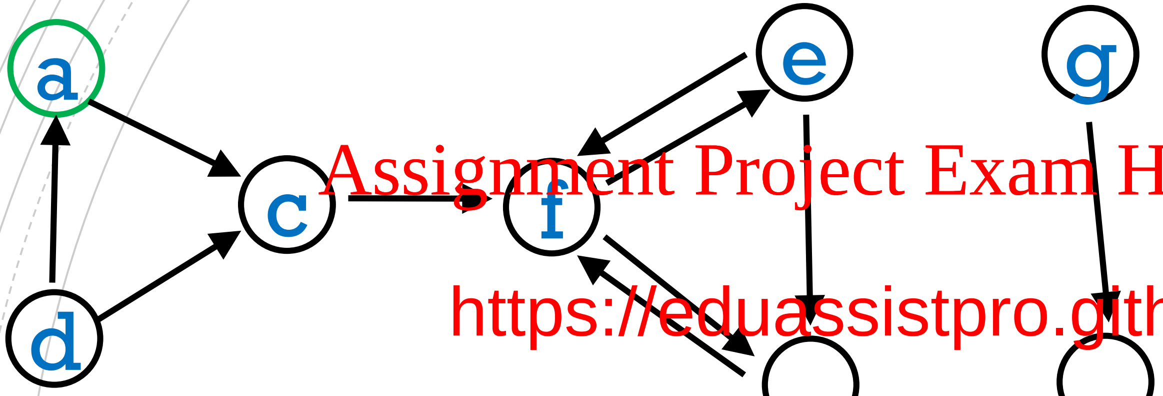
GRAPH TRAVERSAL (RECURSIVE)

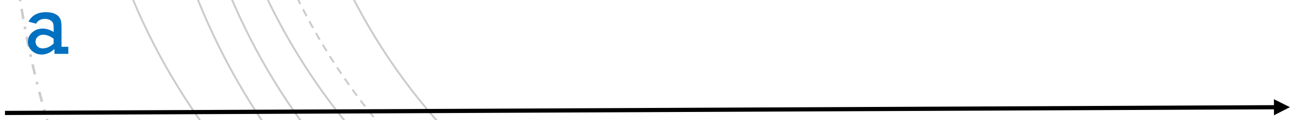
```
depthFirst_Graph (v){
    v.visided = true
    for each w s                              in E
    // i.e. v.ad                              r          ue

        ??
}
```

GRAPH TRAVERSAL (RECURSIVE)

```
depthFirst_Graph (v){
    v.visided = true
    visit v // d
    for each w s                                  in
    // i.e. for each w in v
        if !(w.visited) // avoid cycles!
            dephFirst_Graph(w)
}
```

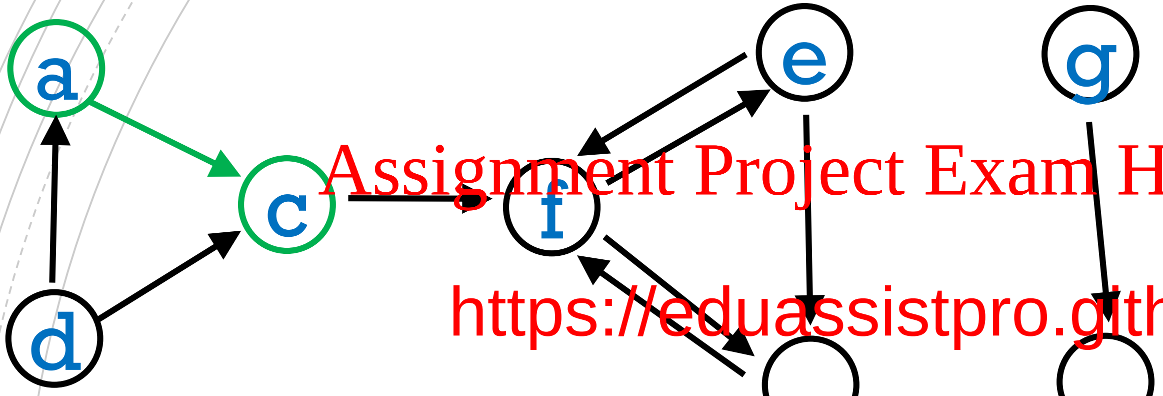
CALL STACK FOR depthFirst(a)

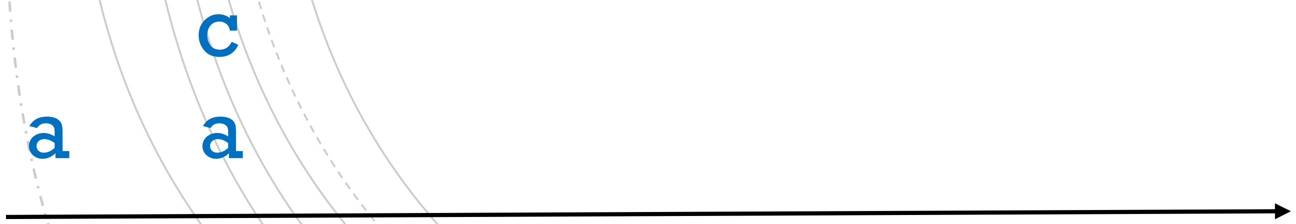




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

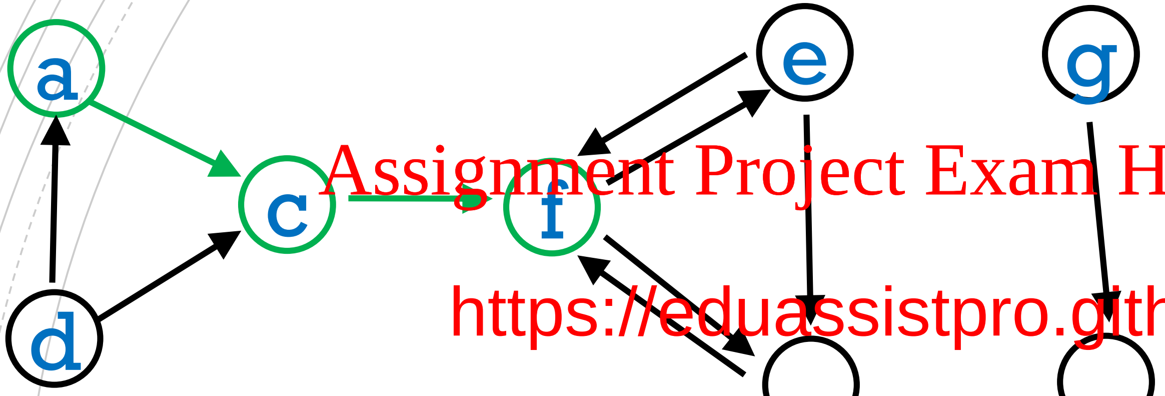
CALL STACK FOR depthFirst(a)

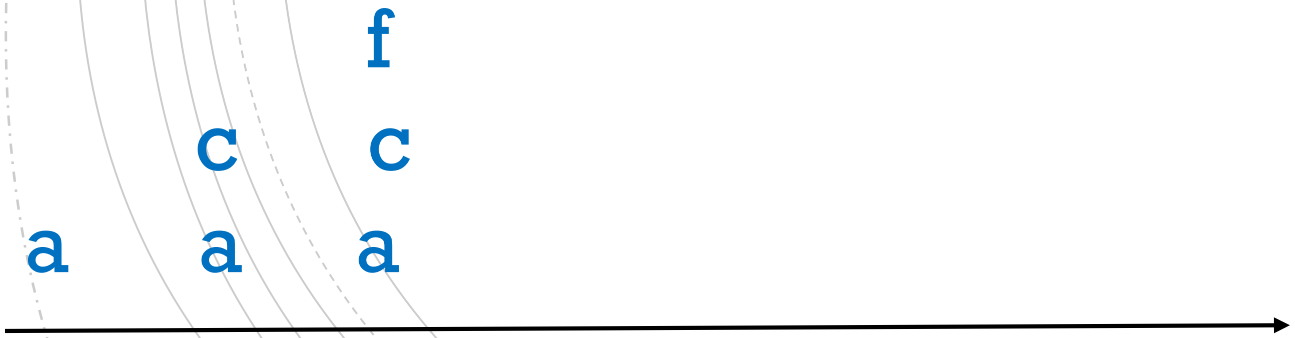




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

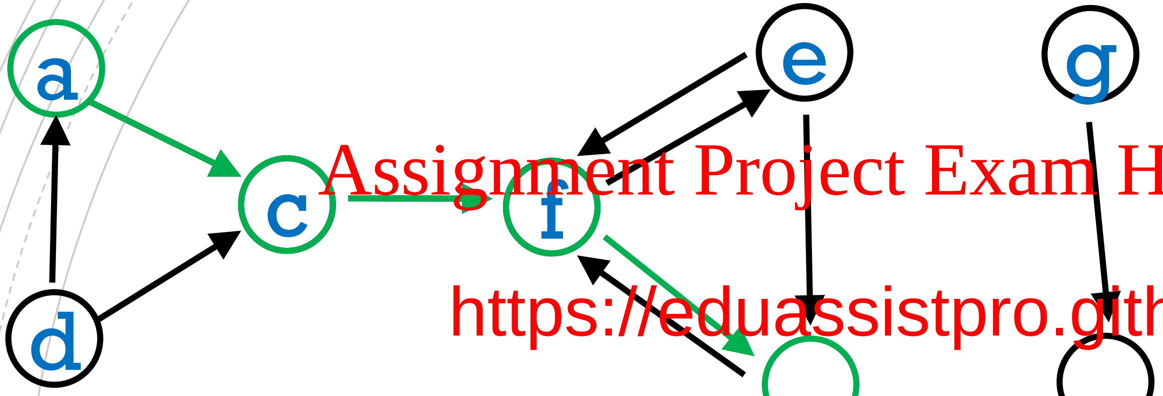
CALL STACK FOR depthFirst(a)

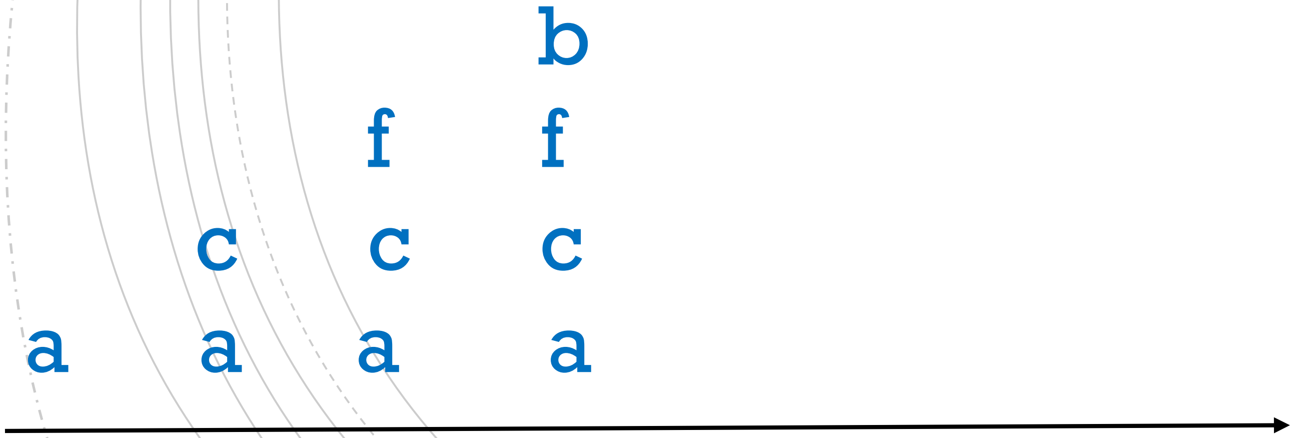




```
depthFirst_Graph (v){
    v.visided = true
    for each w s.t. (v,w) is in E
        if !(w.visited)
            dephFirst_Graph(w)
}
```

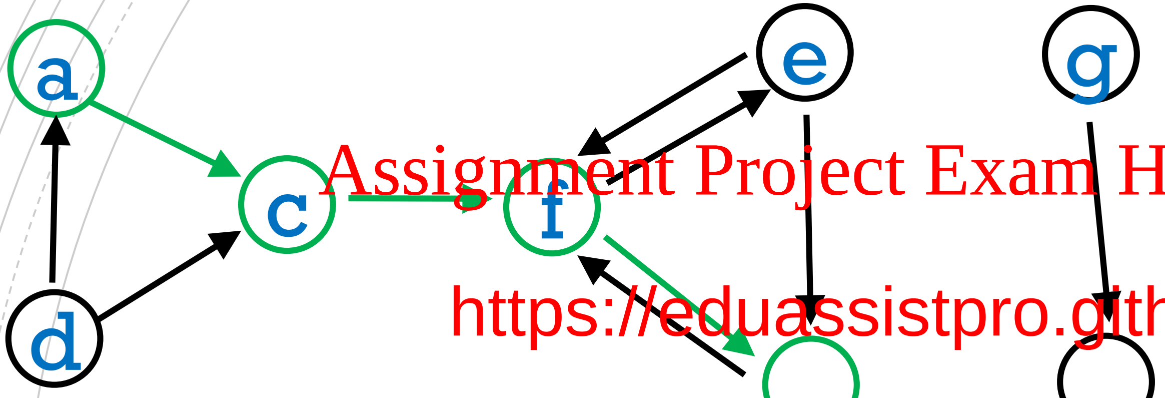
CALL STACK FOR depthFirst(a)





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

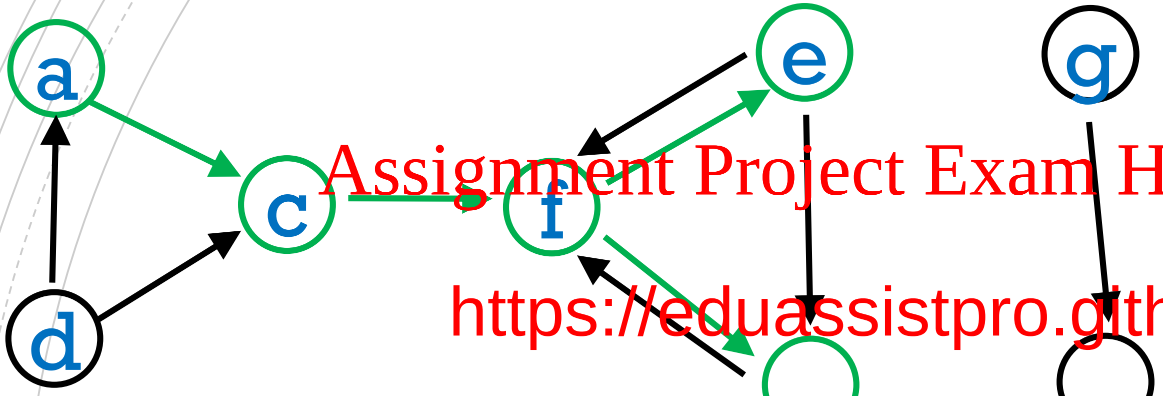
CALL STACK FOR depthFirst(a)

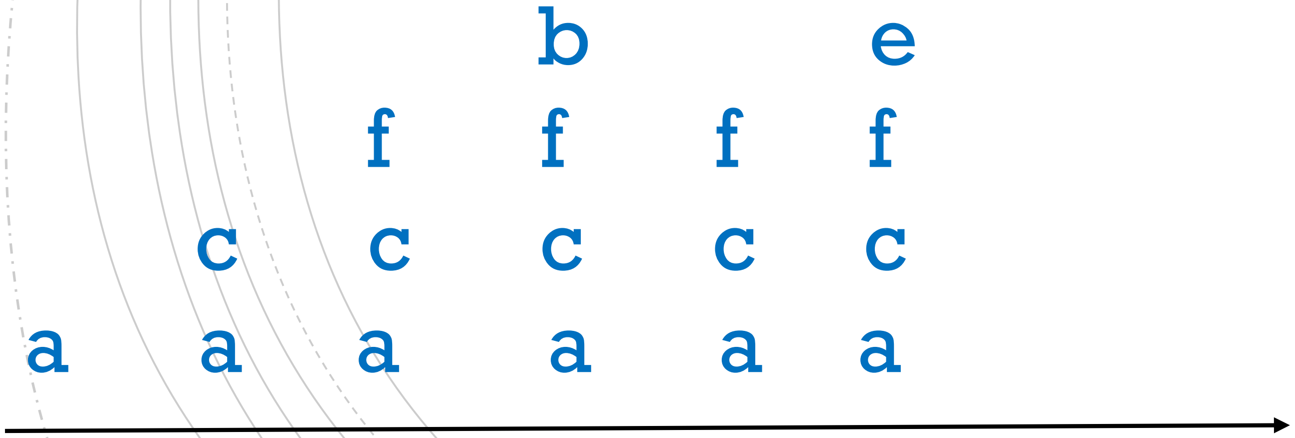


			b	
		f	f	f
	c	c	c	c
a	a	a	a	a

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

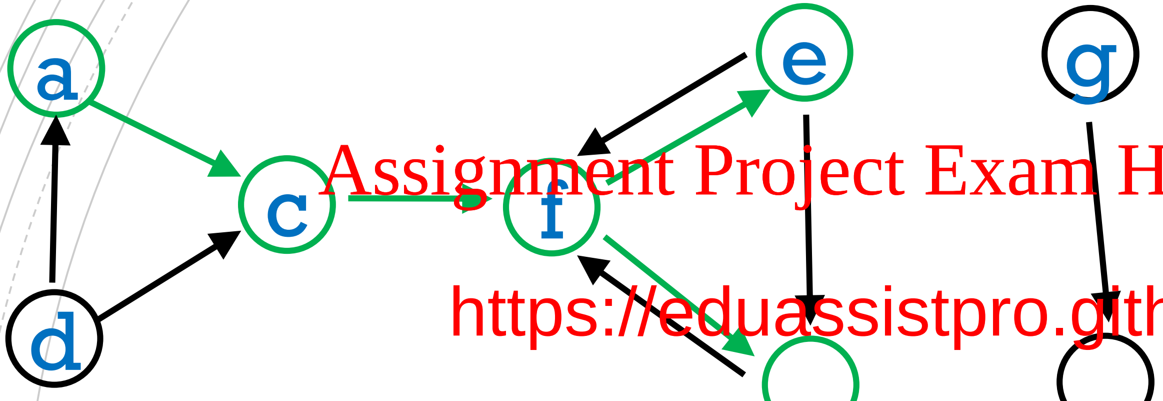
CALL STACK FOR depthFirst(a)





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

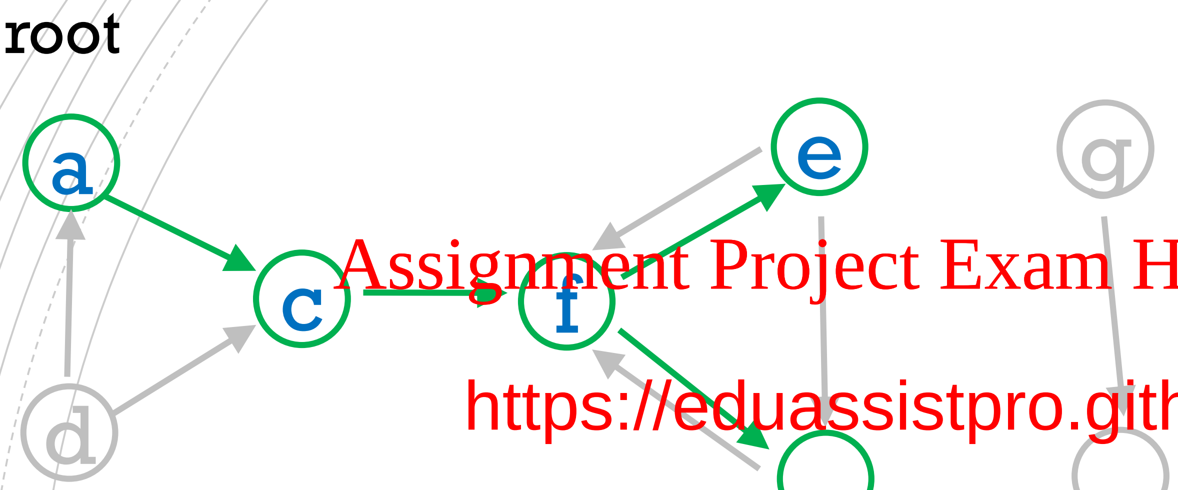
CALL STACK FOR depthFirst(a)

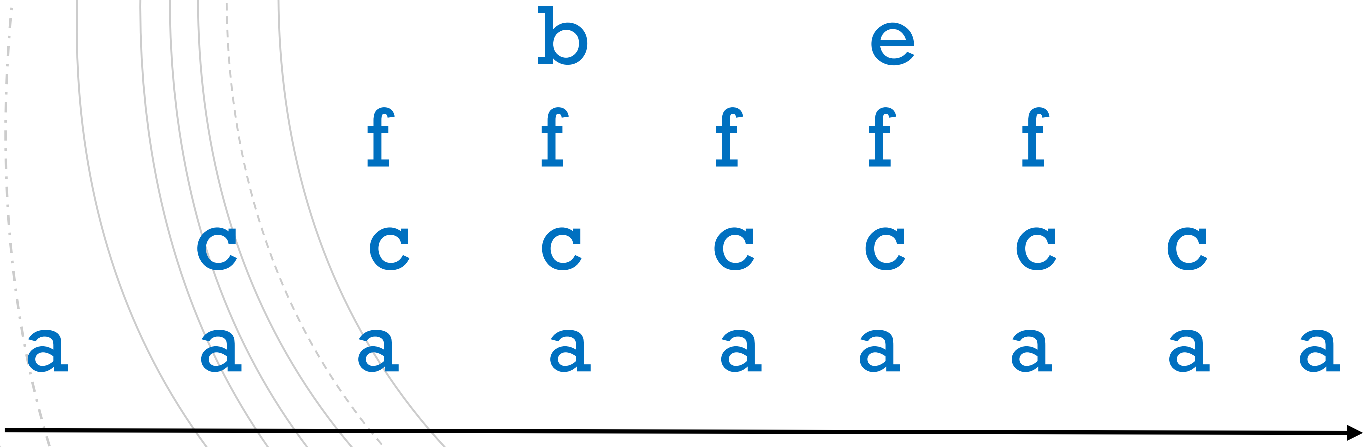


			b		e			
		f	f	f	f	f		
	c	c	c	c	c	c	c	
a	a	a	a	a	a	a	a	a

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

CALL TREE





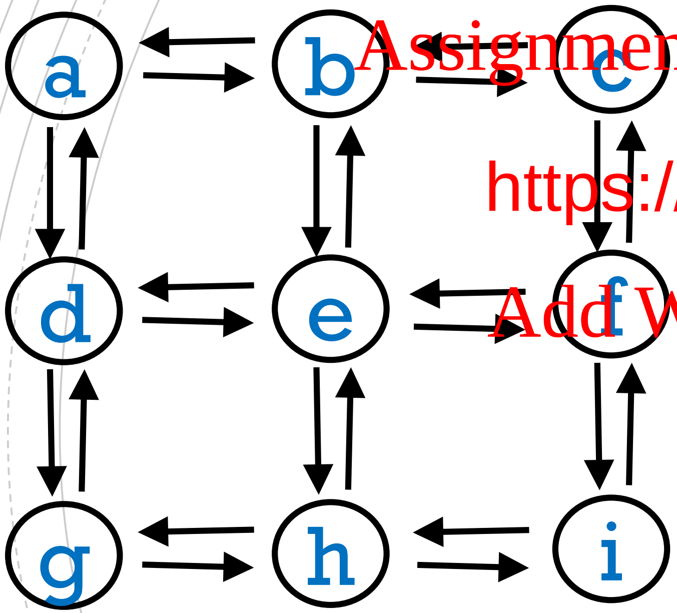
GRAPH TRAVERSALS

- Unlike tree traversal for rooted tree, a graph traversal started from some arbitrary vertex does not necessarily reach all other vertices.

- *Knowing which vertices th from some starting vertex is itself an important problem. Y n 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



Adjacency List

a - (b,d)

b - (a,c,e)

c - (b,f)

d - (a,e,g)

e - (b,d,f,h)

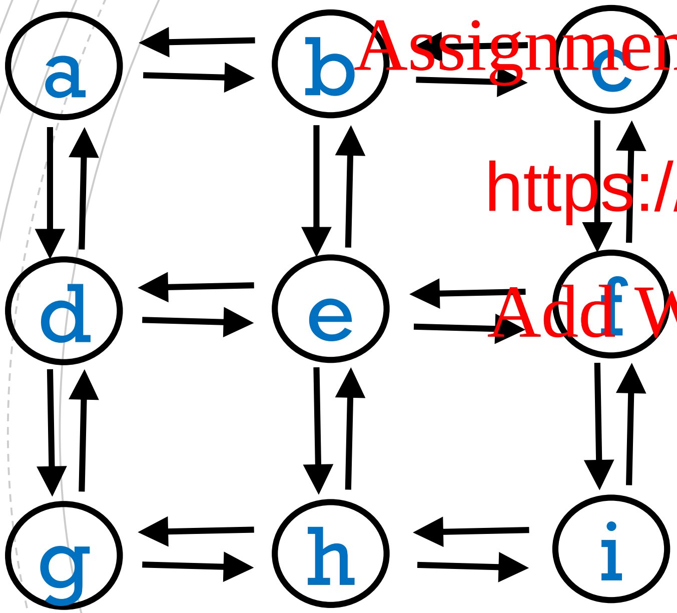
f - (c,e,i)

g - (d,h)

h - (e,g,i)

i - (f,h)

EXAMPLE 2

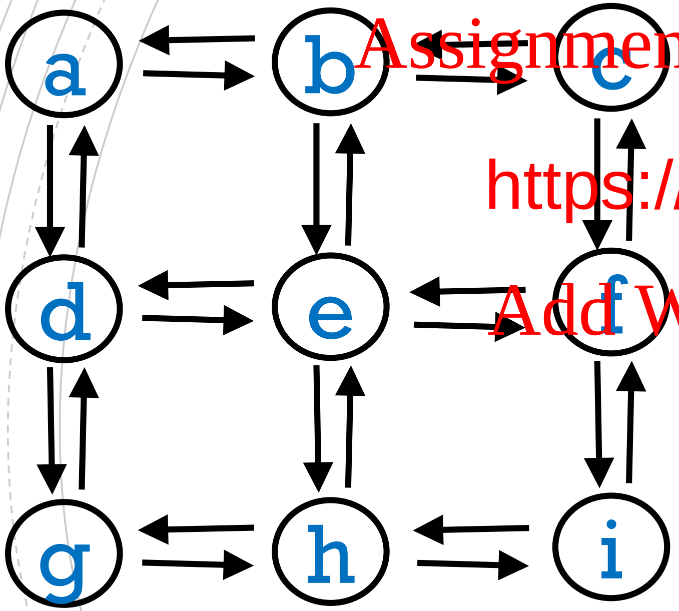


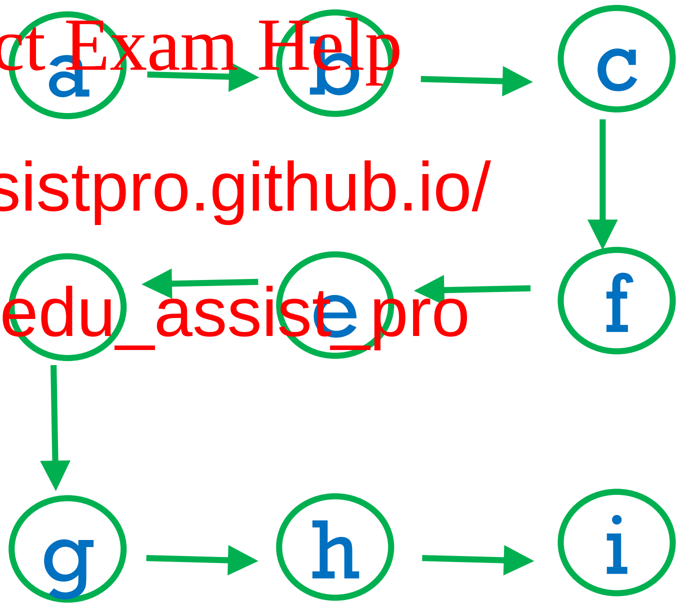
at is the call tree

`pthFirst(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 ck 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 e
      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)
            }
      }
   }
}
```

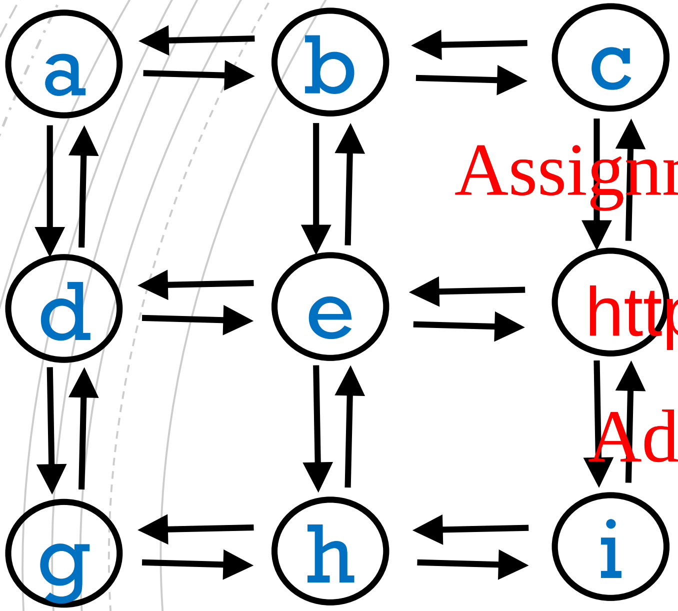
Indicate as "reached" a node *before pushing* it onto the stack. We do that by updating the field visited.

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

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

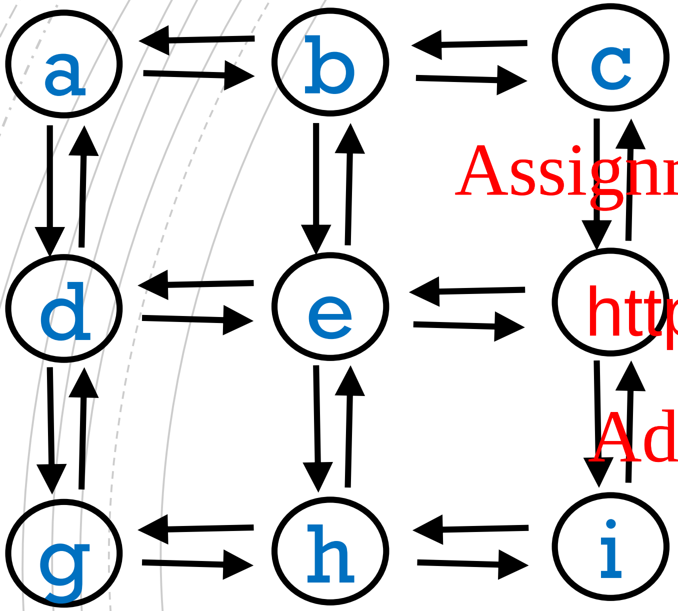
EXAMPLE: graphTraversalUsingStack(a)

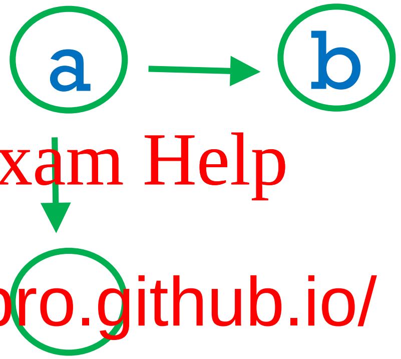
Order of visit: a

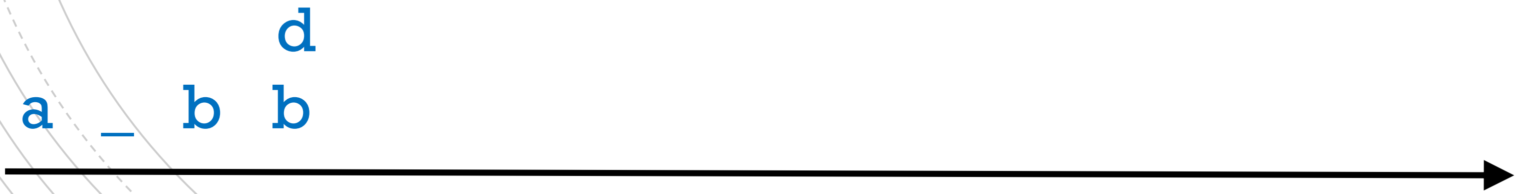


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **a**

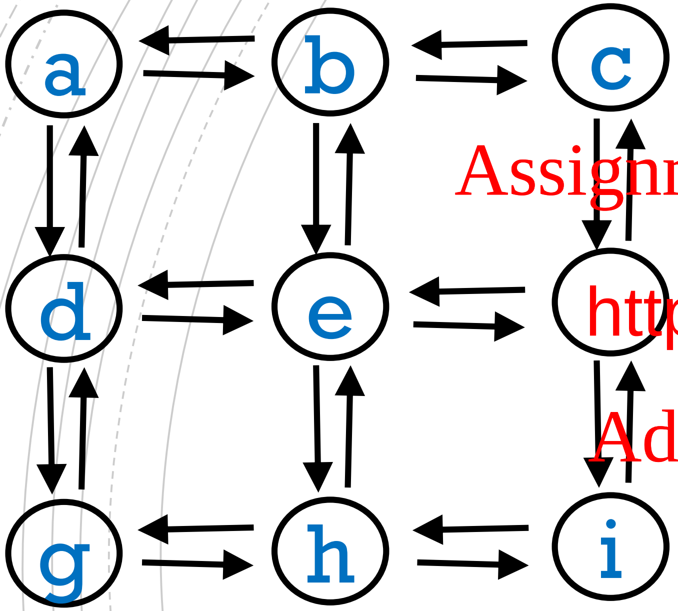


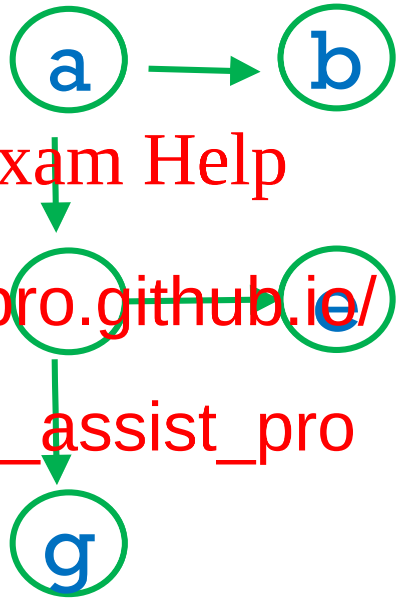


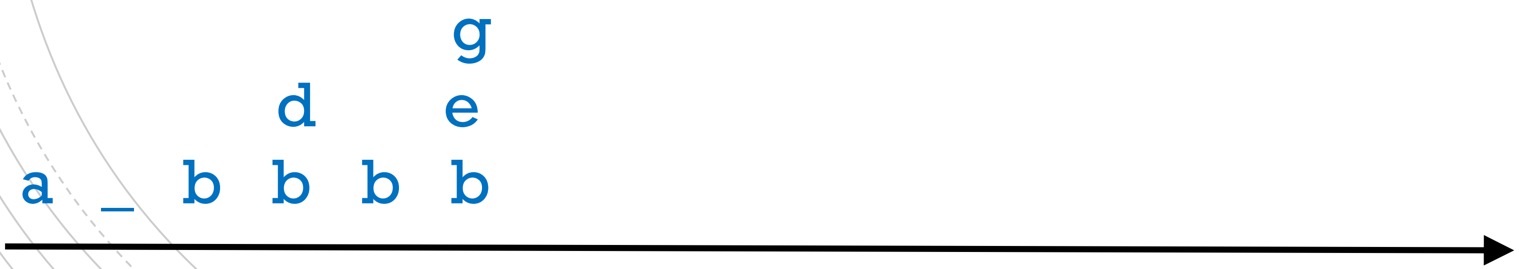


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **ad**

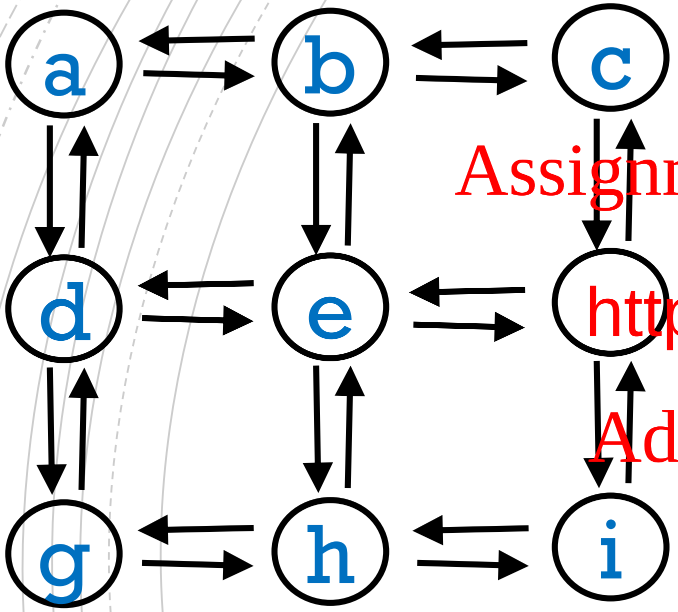


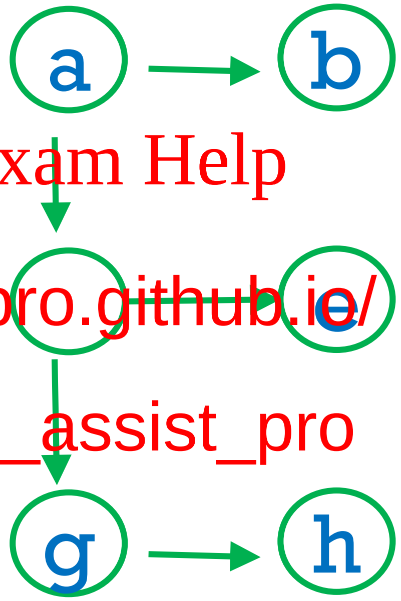


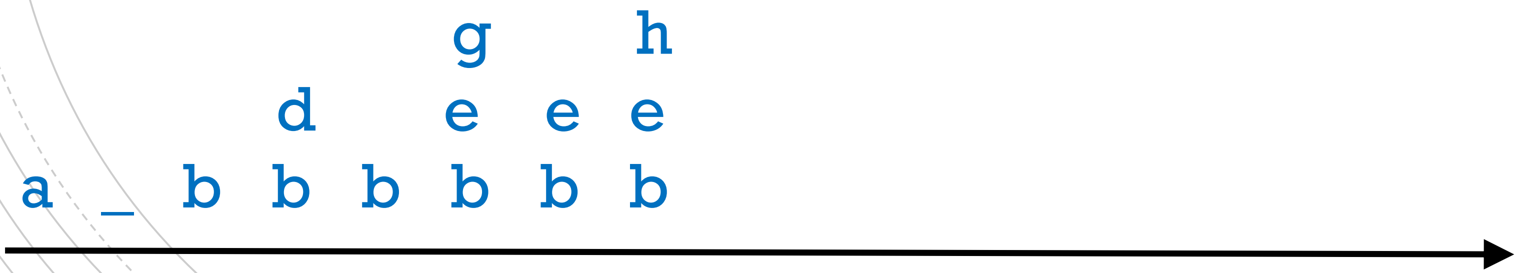


EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adg**

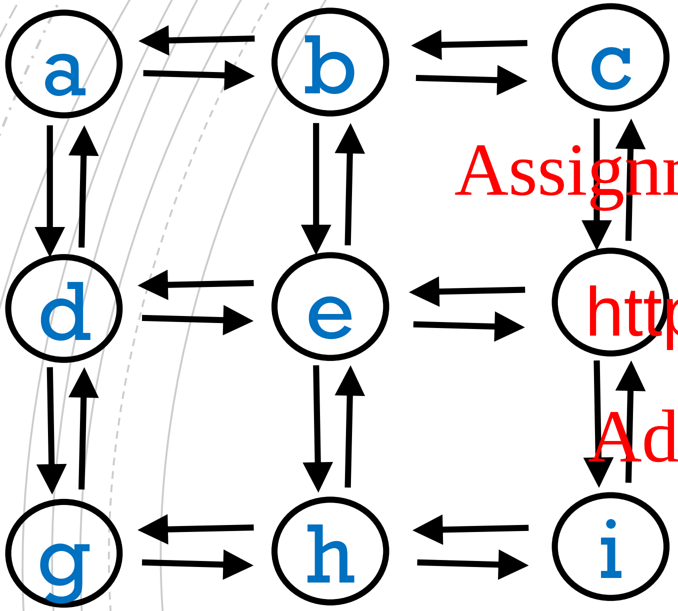


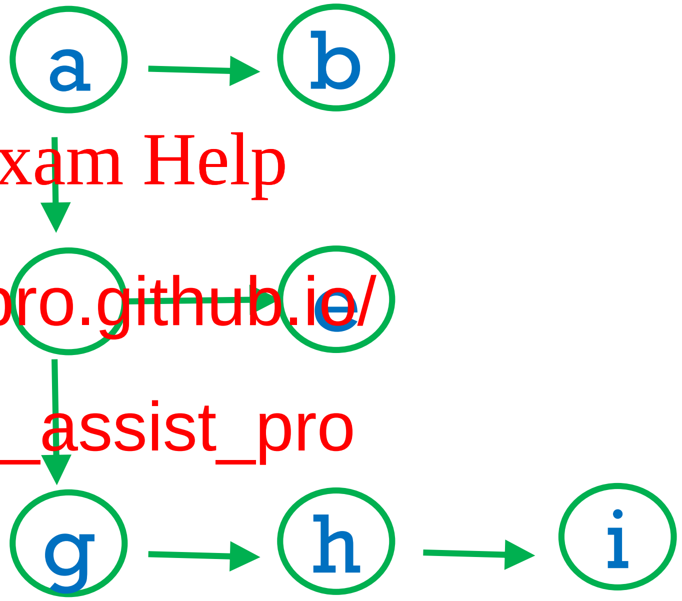




EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adgh**

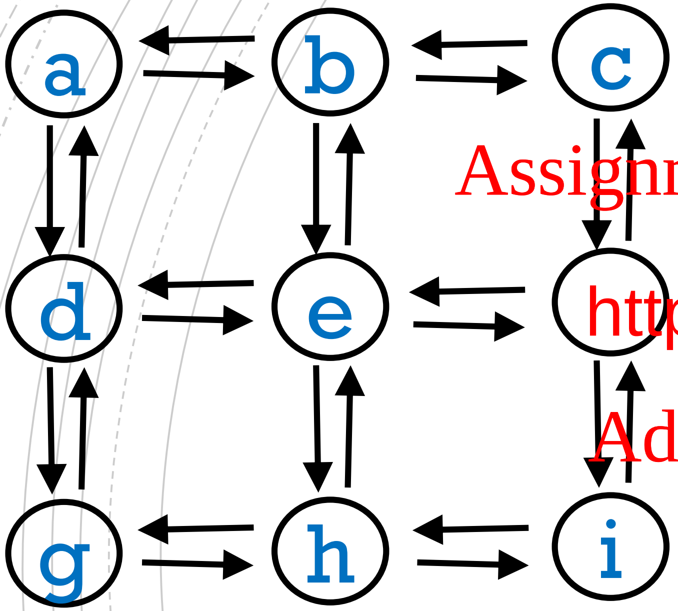


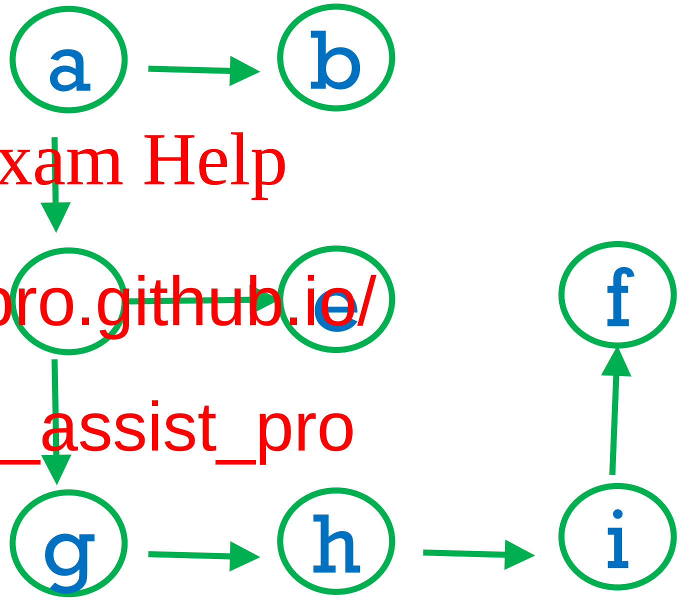


					g		h		i
			d		e	e	e	e	e
a	_	b	b	b	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghi**

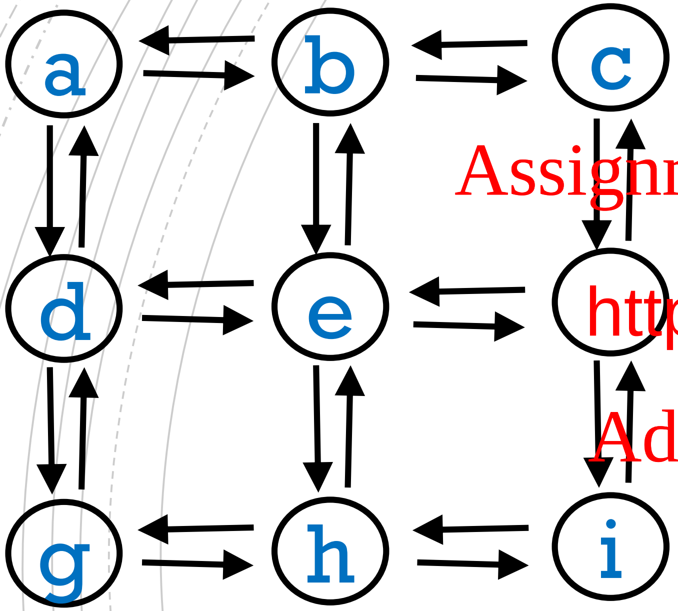


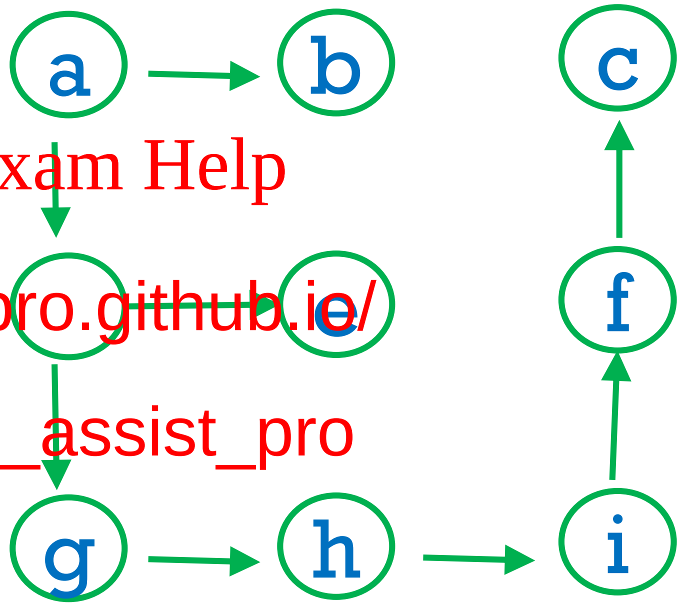


					g		h		i		f
			d		e	e	e	e	e	e	e
a	_	b	b	b	b	b	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghif**

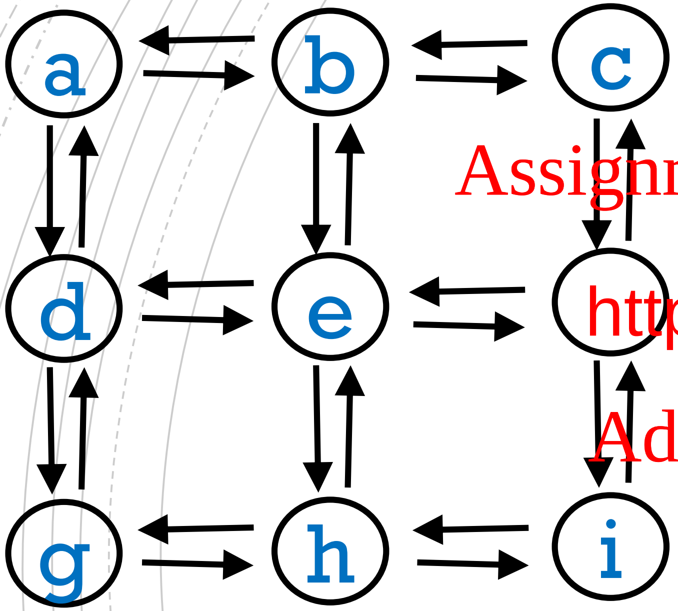


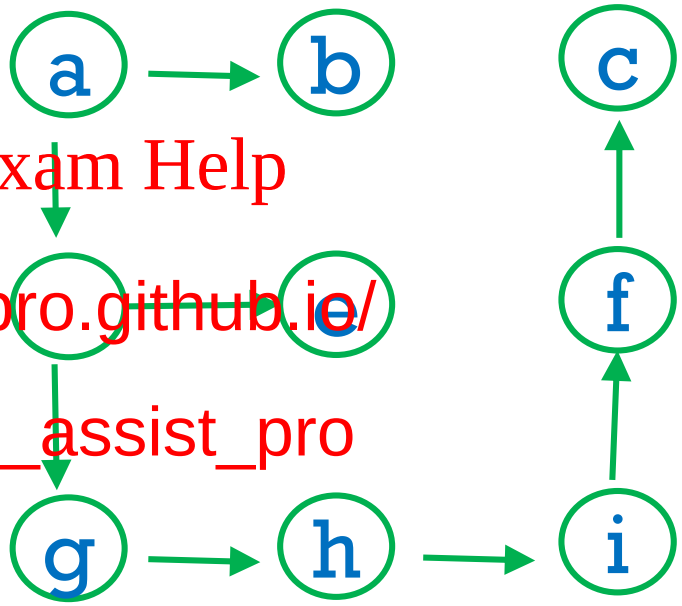


					g		h		i		f		c
			d		e	e	e	e	e	e	e	e	e
a	_	b	b	b	b	b	b	b	b	b	b	b	b

EXAMPLE: graphTraversalUsingStack(a)

Order of visit: **adghifceb**

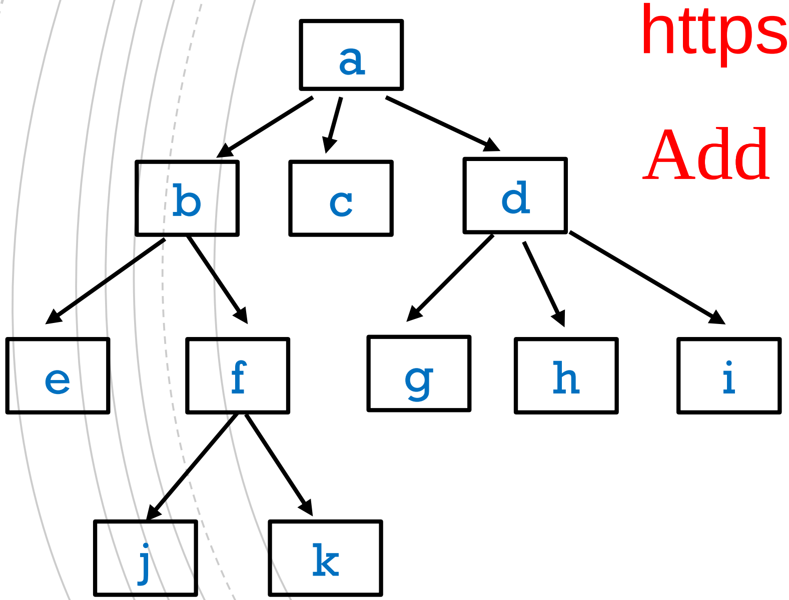




					g		h		i		f		c			
			d		e	e	e	e	e	e	e	e	e	e		
a	_	b	b	b	b	b	b	b	b	b	b	b	b	b	b	_

RECALL: BREADTH FIRST TREE TRAVERSAL

for each level i
    visit all nodes at level i



```
treeTraversalUsingQueue(root){
    initialize empty queue q
              eue(root)
          q is not empty {
              q.dequeue()
              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

BREADTH FIRST GRAPH TRAVERSAL

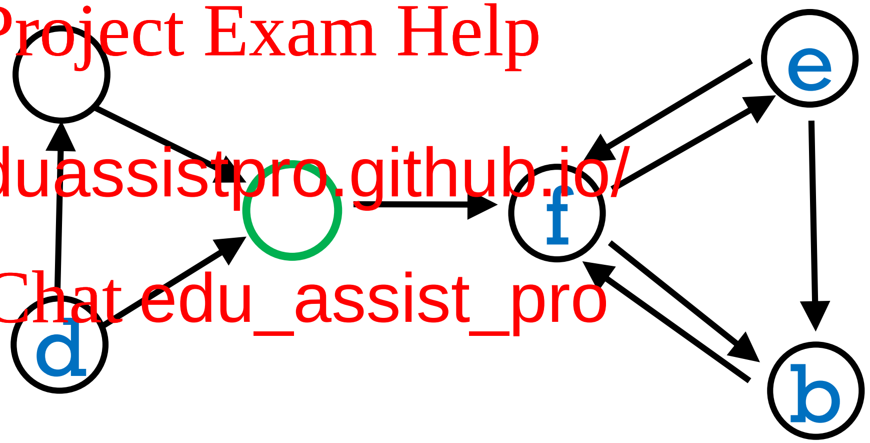
```
graphTraversalUsingQueue(v){
    initialize empty queue q
    v.visited = true
    q.enqueue(v)
    while
        cu
        for each w in           ist {
            if(!w.visit
                w.visited = true
                q.enqueue(w)
            }
        }
    }
}
```

EXAMPLE

graphTraversalUsingQueue(c)

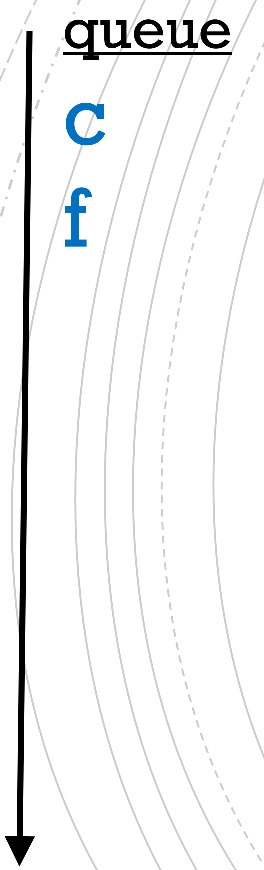
queue

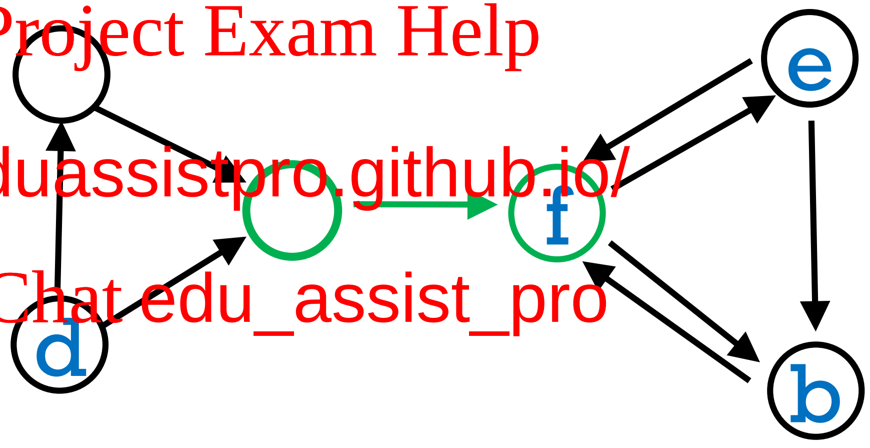
c



EXAMPLE

graphTraversalUsingQueue(c)





EXAMPLE

graphTraversalUsingQueue(c)

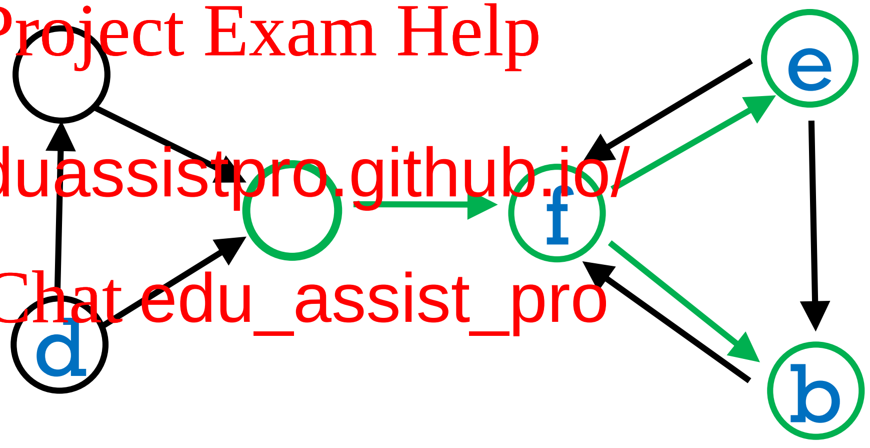
queue

c

f

be

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



EXAMPLE

graphTraversalUsingQueue(c)

queue

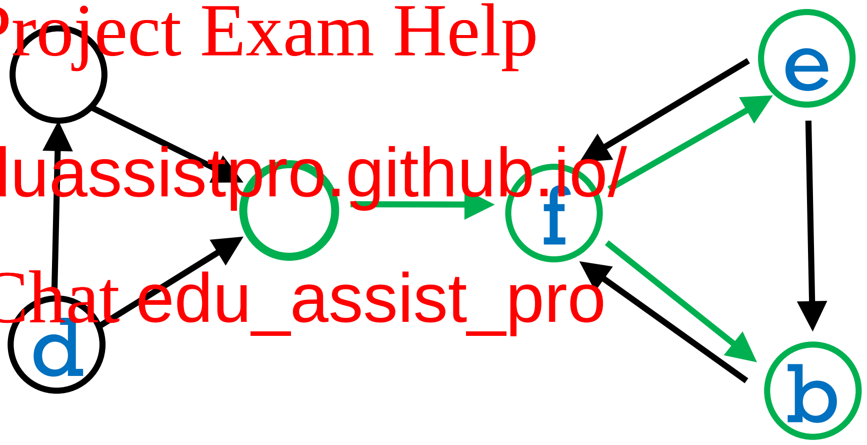
c

f

be

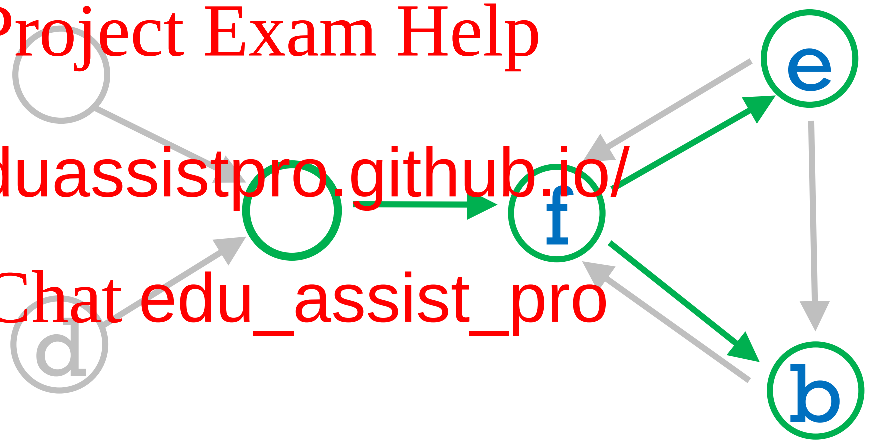
e

_



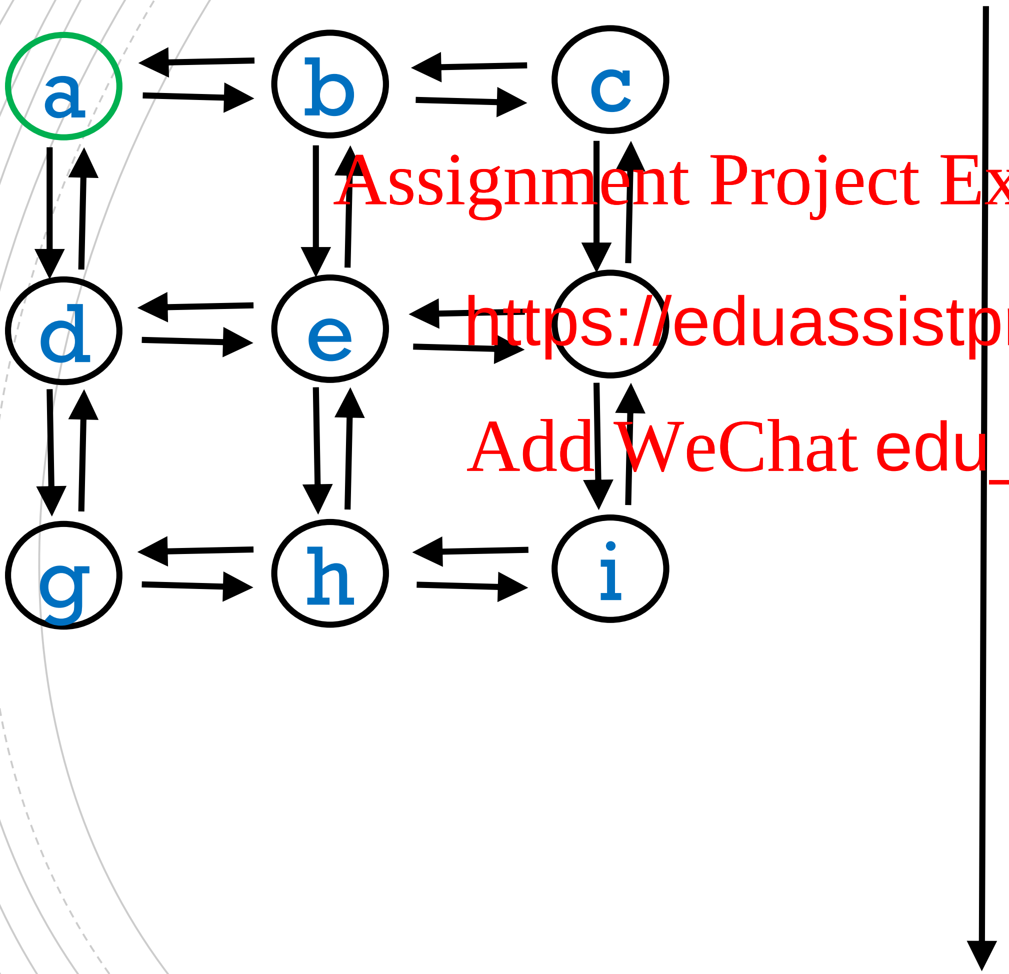
EXAMPLE

graphTraversalUsingQueue(c)



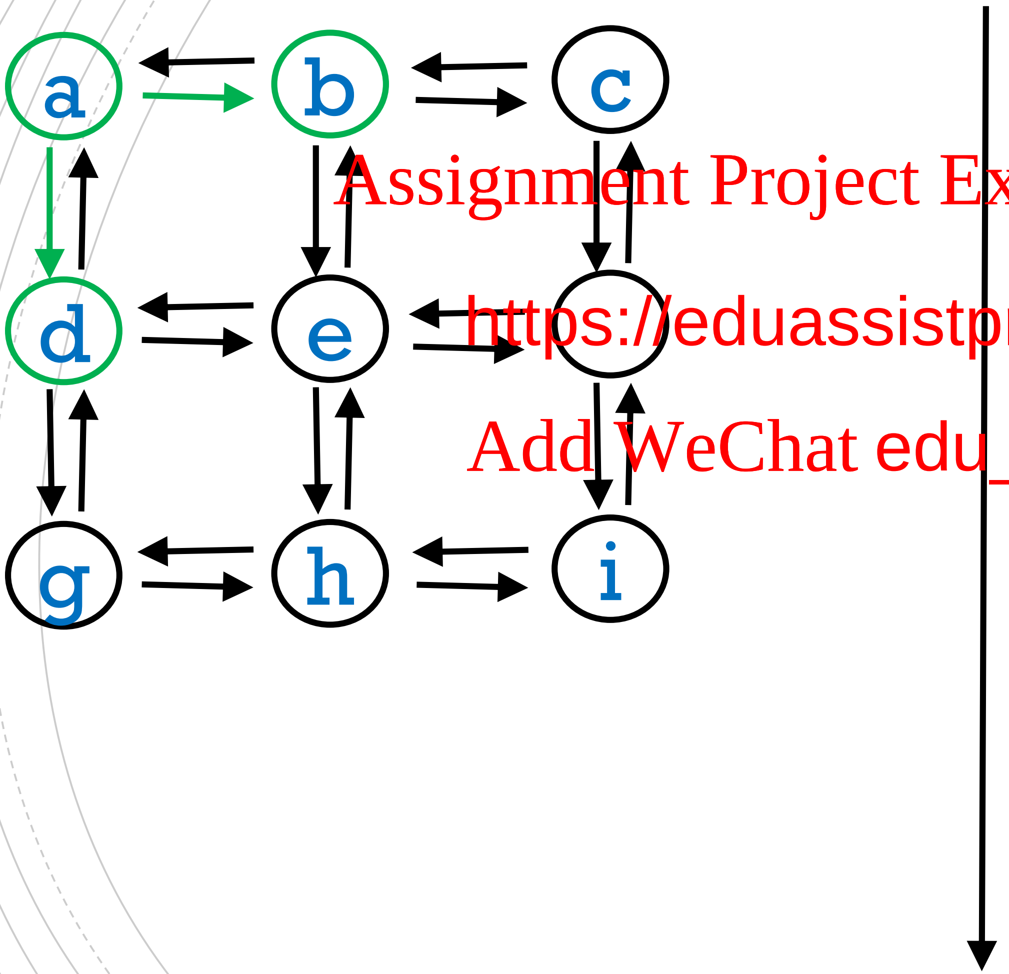
It defines a tree whose root is the starting vertex.
It finds the shortest path (number of edges) to all vertices reachable from the starting vertex.

EXAMPLE: graphTraversalUsingQueue(a)



a

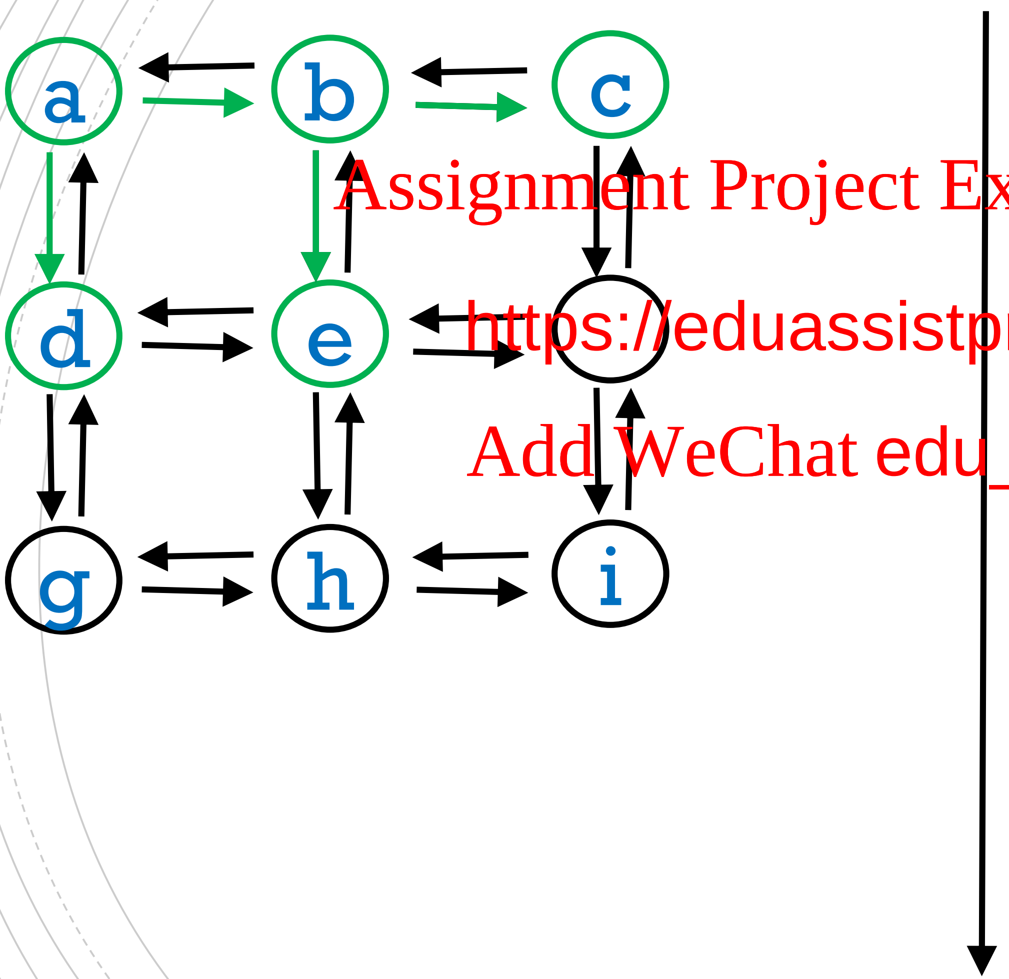
EXAMPLE: graphTraversalUsingQueue(a)



a

bd

EXAMPLE: graphTraversalUsingQueue(a)

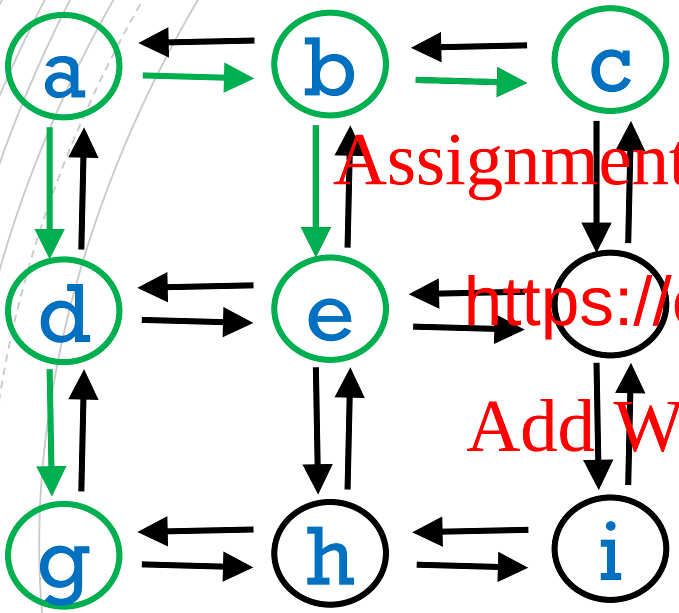


a

bd

ce

EXAMPLE: graphTraversalUsingQueue(a)



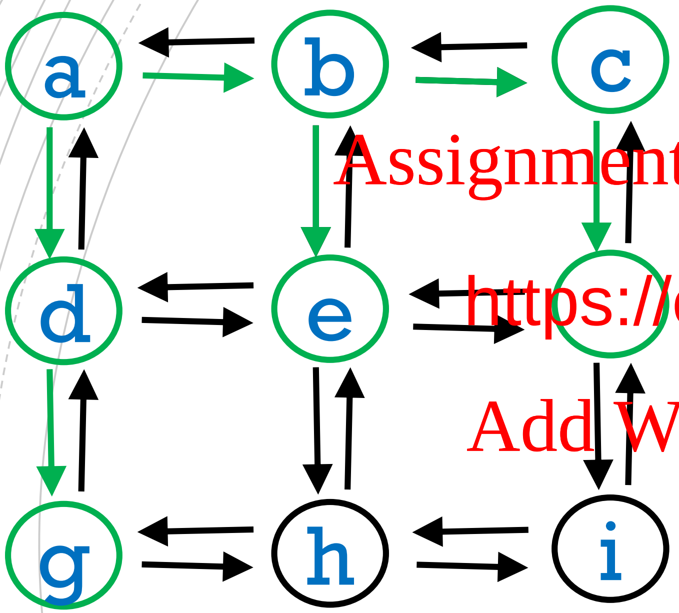
a

bd

ce

eg

EXAMPLE: graphTraversalUsingQueue(a)



a

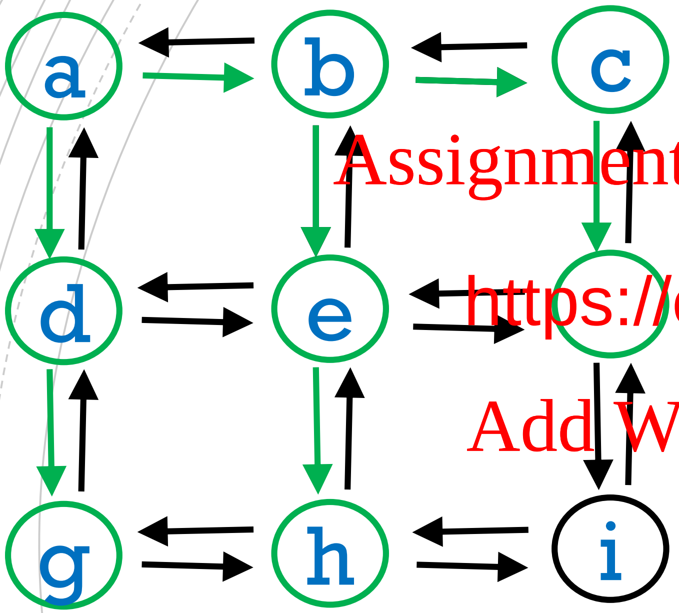
bd

ce

eg

f

EXAMPLE: graphTraversalUsingQueue(a)



a

bd

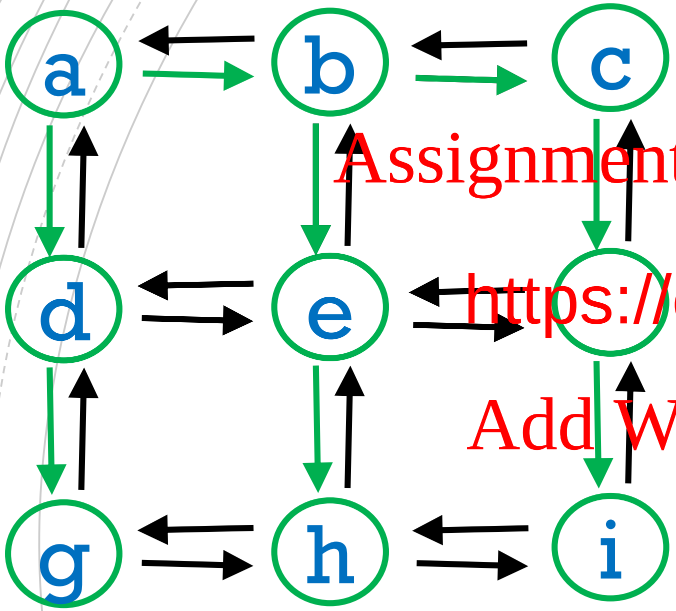
ce

eg

f

gfh

EXAMPLE: graphTraversalUsingQueue(a)



a

bd

ce

eg

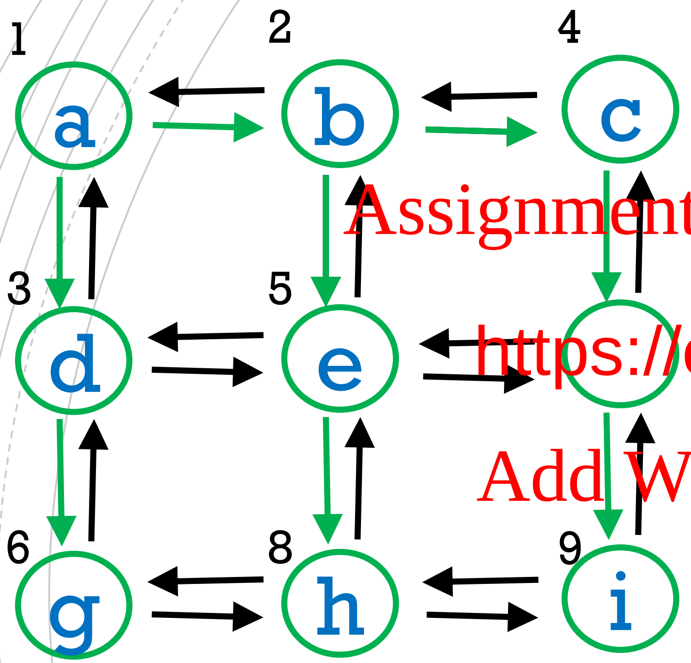
f

gfh

fh

hi

EXAMPLE: graphTraversalUsingQueue(a)



a

bd

ce

eg

f

gfh

fh

hi

i

_

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> {
      Arra                                   ;
      T el
      boolean visited;
   }

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

PRIOR TO TRAVERSAL!

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

How should we impl

PRIOR TO TRAVERSAL!

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

```
class
   ArrayList<Verte             exList;
   :
   public void resetVisited() {



   }
}
```

PRIOR TO TRAVERSAL!

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

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