COMP 250

Lecture 31

graph traversal

Nov. 21, 2018

Today

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

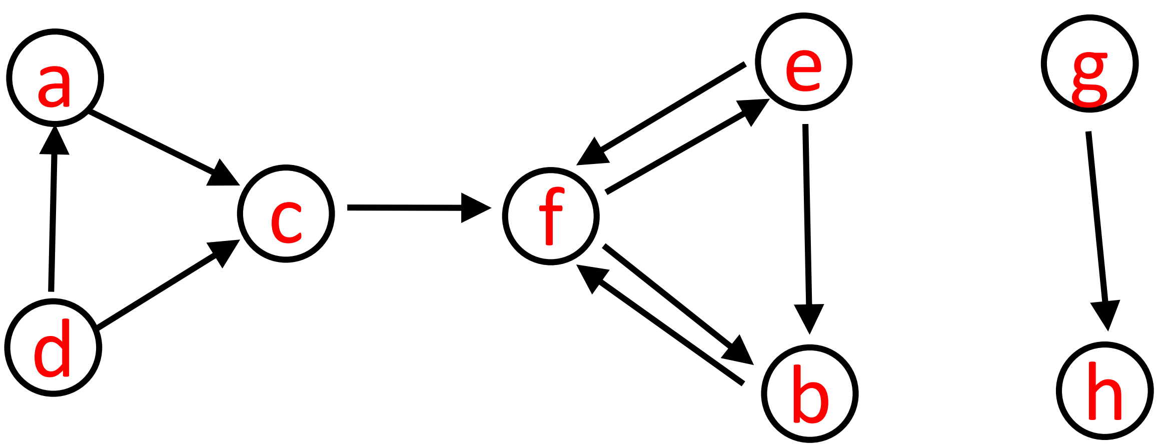
Recall: tree traversal (recursive)

```
depthfirst__Tree (root){
   if (root is not empty){
      root.visited = true                    //        "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.visited = true
   for each w such that (v,w) is in E   //  w in v.adjList
          ________?____________
}
```

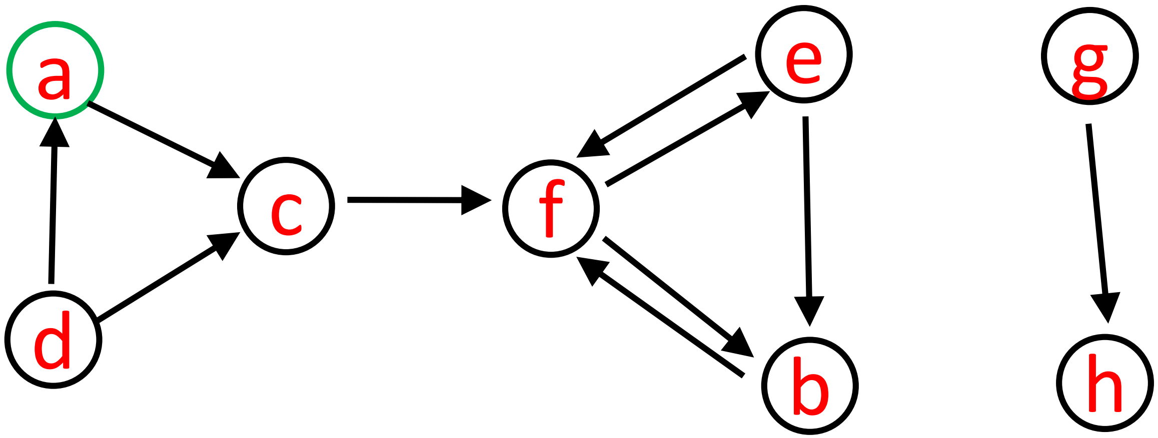
// Here "visiting" just means "reaching"

Graph traversal (recursive)

```
depthFirst_Graph(v){
   v.visited = true
   for each w such that (v,w) is in E   //  w in v.adjList
     if   ! (w.visited)                 //  avoids cycles
          depthFirst_Graph(w)
}
```

// Here “visiting” just means “reaching”

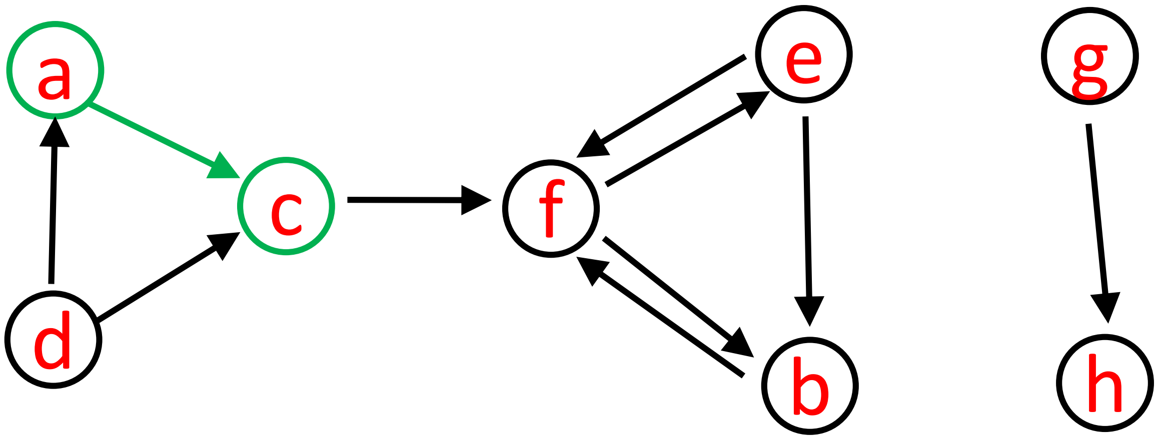
Call Stack for depthFirst(a)



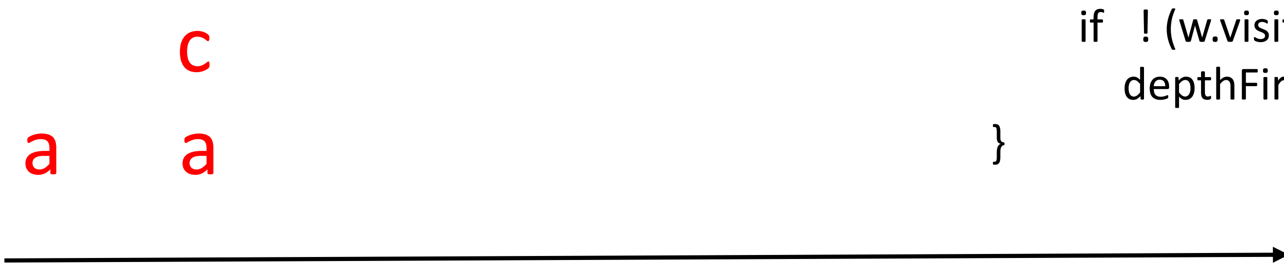

```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if  ! (w.visited)
        depthFirst_Graph(w)
}
```

a

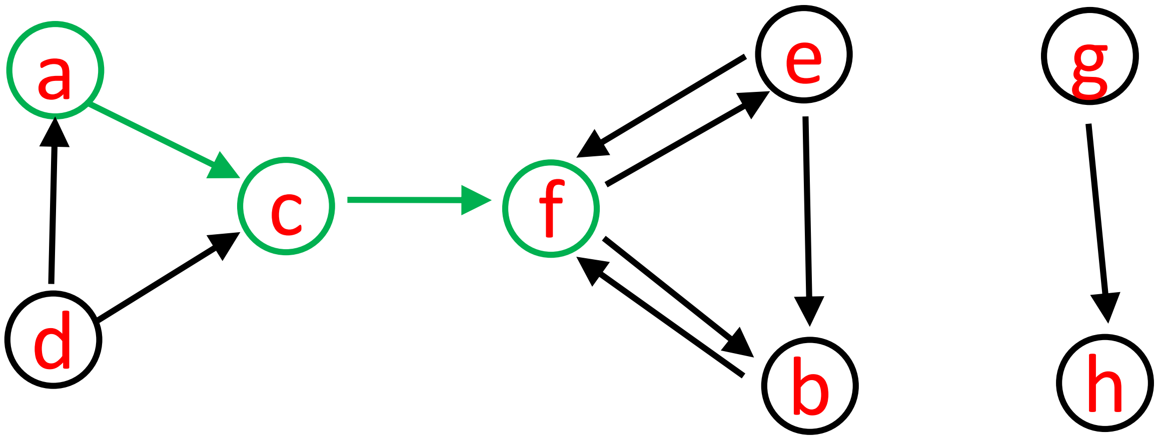
Call Stack for depthFirst(a)




```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if  ! (w.visited)
        depthFirst_Graph(w)
}
```



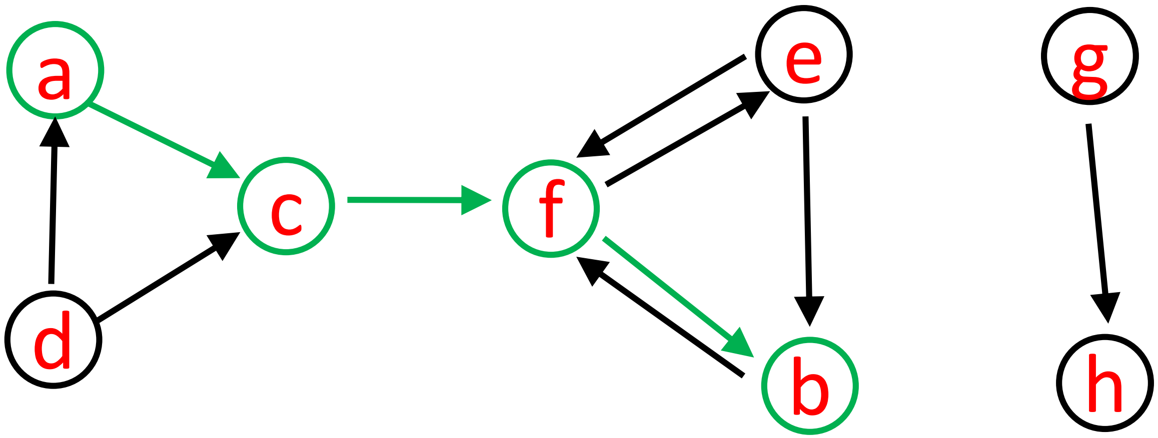
Call Stack for depthFirst(a)



		f
	c	c
a	a	a

```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if   ! (w.visited)
         depthFirst_Graph(w)
}
```

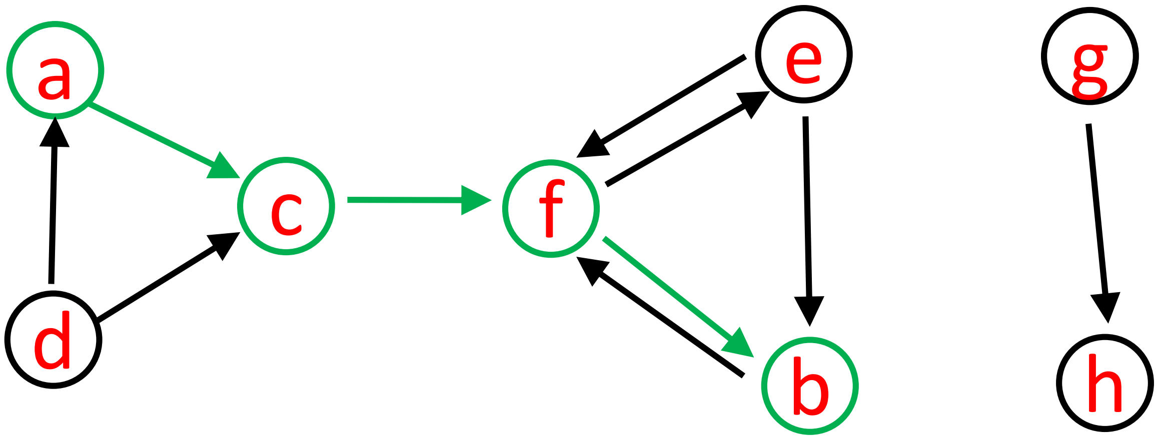
Call Stack for depthFirst(a)



			b
		f	f
	c	c	c
a	a	a	a

```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if   ! (w.visited)
         depthFirst_Graph(w)
}
```

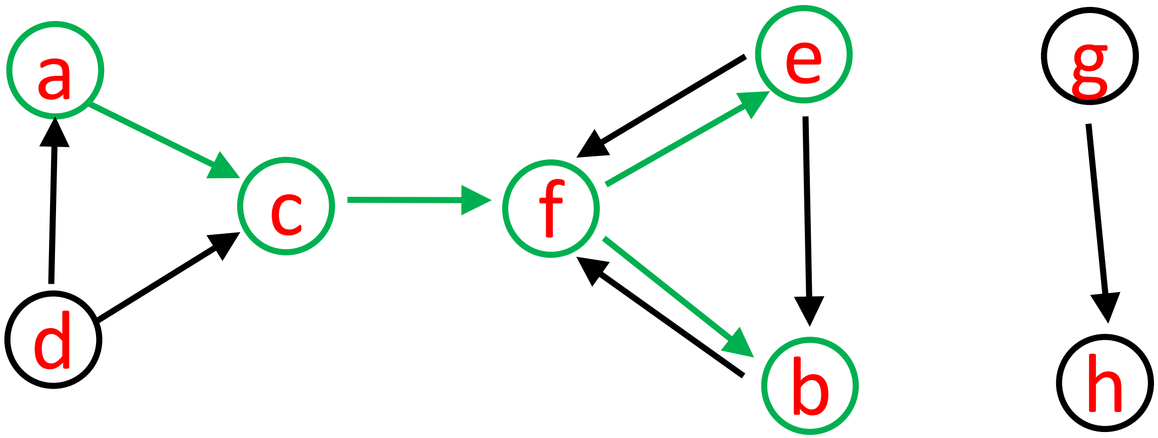
Call Stack for depthFirst(a)



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

```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if  ! (w.visited)
         depthFirst_Graph(w)
}
```

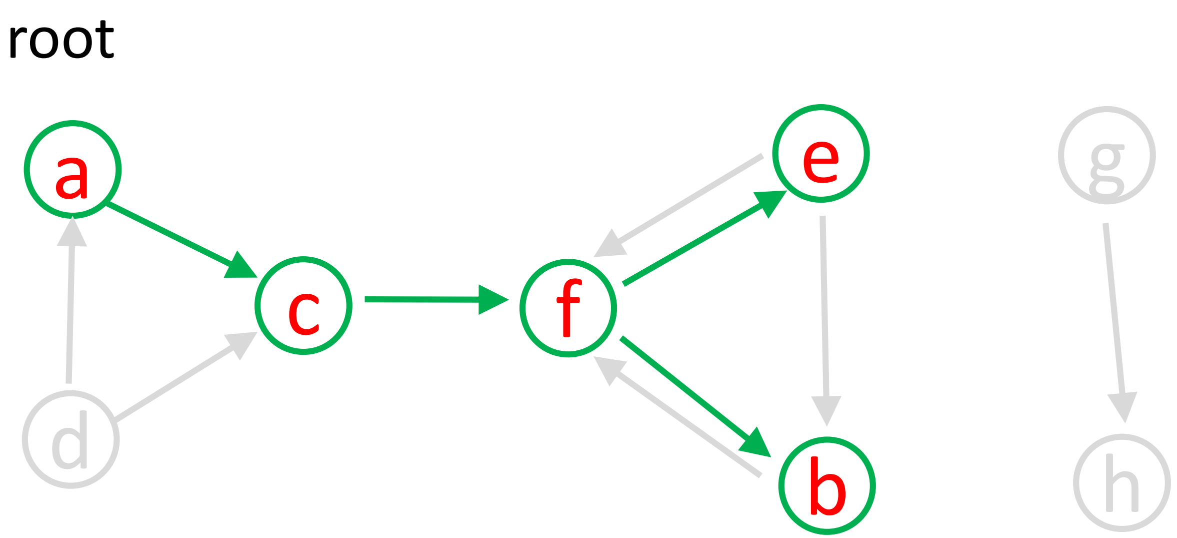
Call Stack for depthFirst(a)



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

```
depthFirst_Graph(v){
  v.visited = true
  for each w such that (v,w) is in E
     if  ! (w.visited)
        depthFirst_Graph(w)
}
```

Call Tree



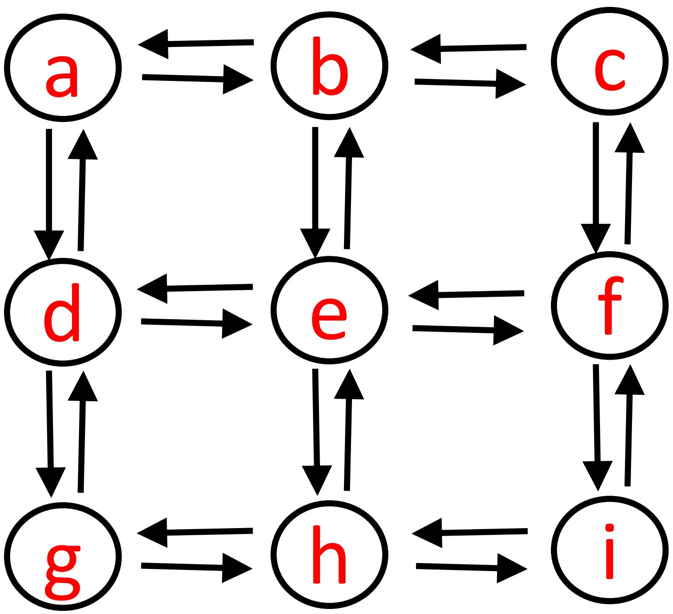
			b		e			
		f	f	f	f	f		
	c	c	c	c	c	c	c	
a	a	a	a	a	a	a	a	a

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 list.

Example 2



Adjacency List

a - (b,d)

b - (a,c,e)

c - (b,f)

d - (a,e,g)

e - (b,d,f,h)

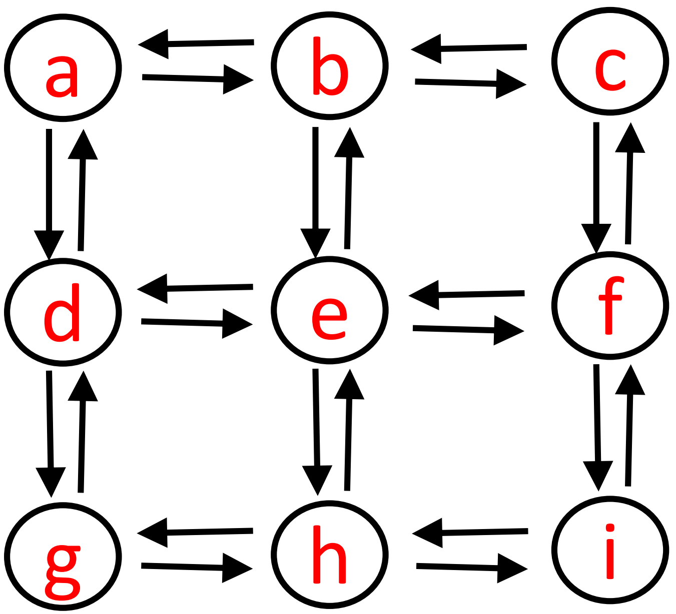
f - (c,e,i)

g - (d,h)

h - (e,g,i)

i - (f,h)

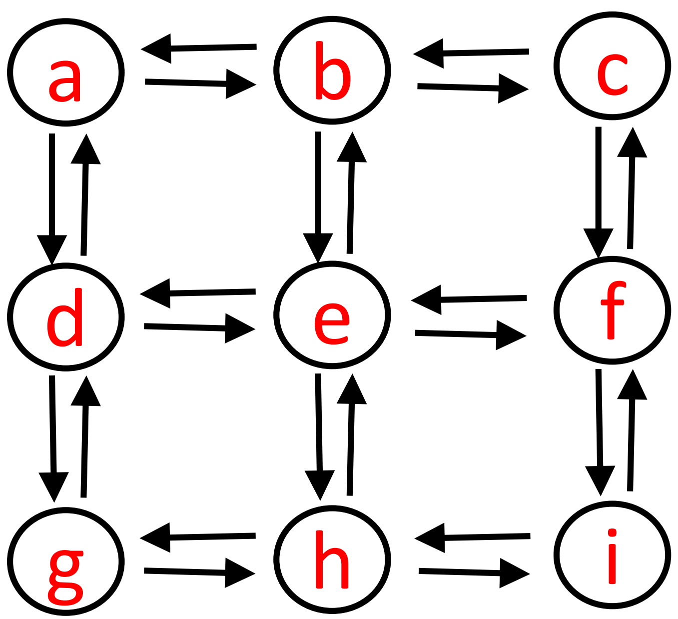
Example 2

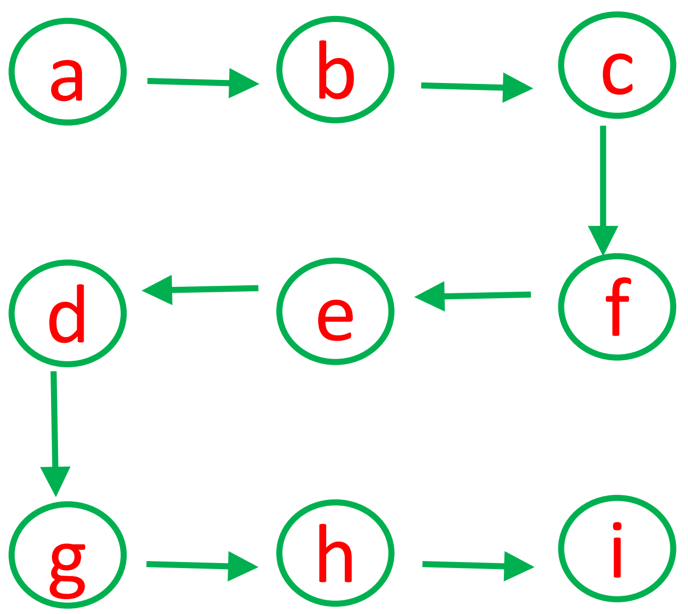


What is the call tree for depthFirst(a) *?*

(Do it in your head.)

Example 2





call tree for depthFirst(a)

Q: Can we do non-recursive graph traversal ?

Q: Can we do non-recursive graph traversal ?

A: Yes, similar to tree traversal: Use a stack or queue.

Recall: depth first tree traversal
(with a slight variation)

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

Visit a node *before pushing* it onto the stack. (Preorder)

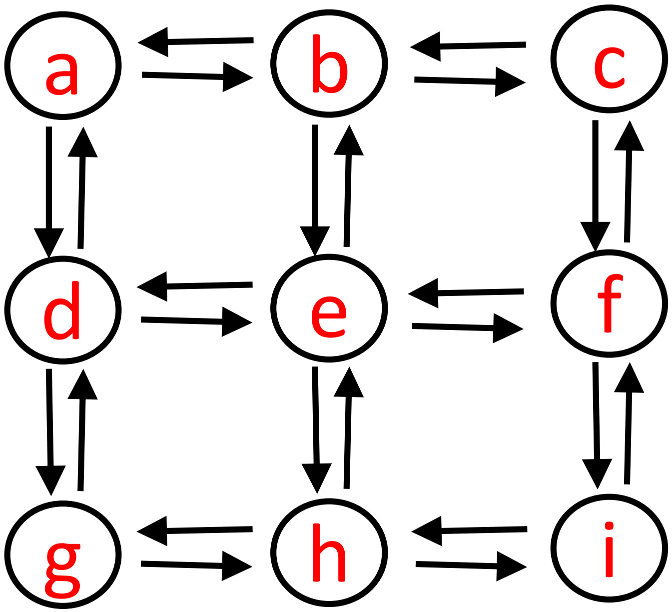
Every node in the tree gets visited, pushed, and then popped.

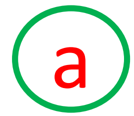
Recall that visits occur down right side first.

Generalize to graphs...

```
graphTraversalUsingStack(v){
   initialize empty stack s
   v.visited = true
   s.push(v)
   while (!s.empty) {
      u =  s.pop()
      for each w in u.adjList{
         if (!w.visited){                    //  The only new part.  Why?
            w.visited = true
            s.push(w)
         }
      }
   }
}
```

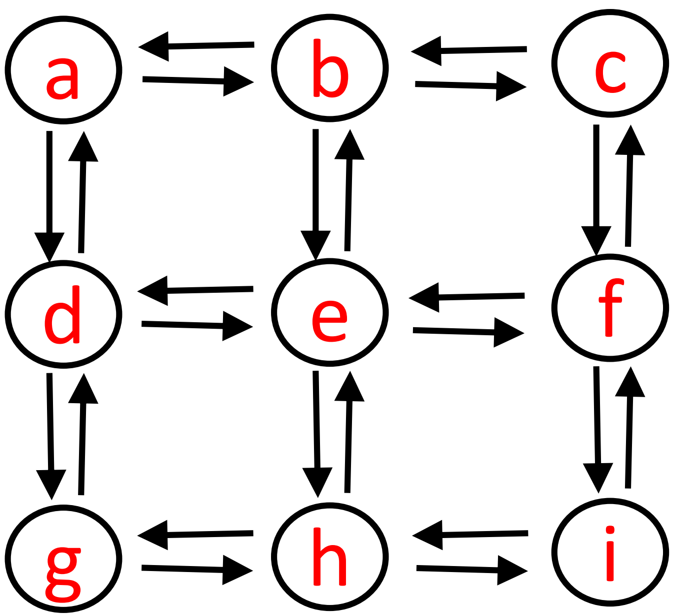
Example: graphTraversalUsingStack(a)

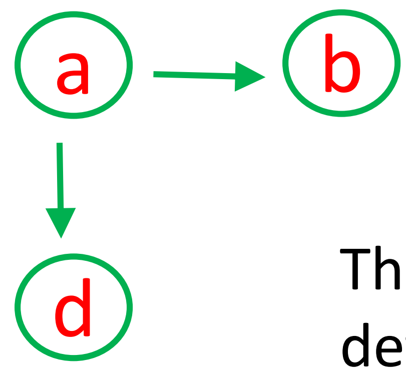




a

Example: graphTraversalUsingStack(a)



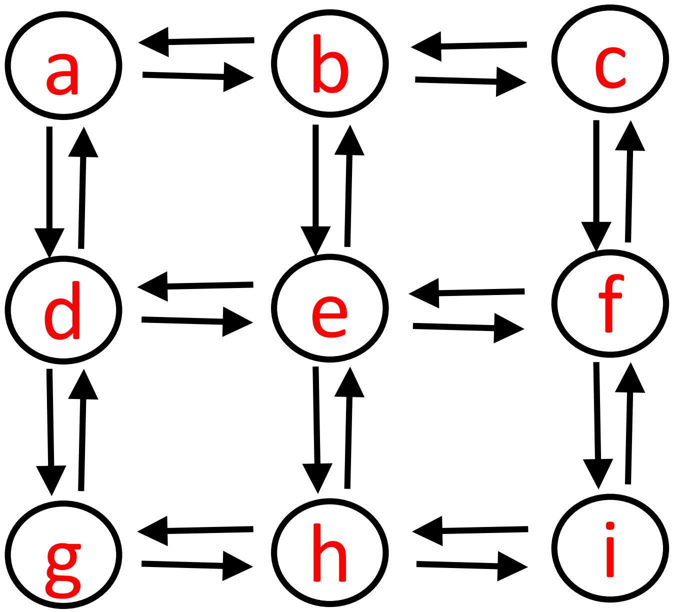


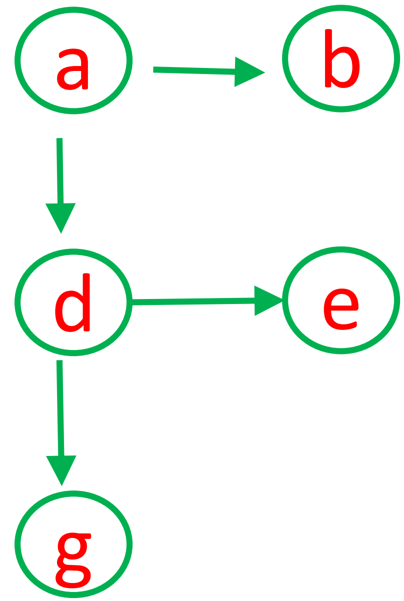
The traversal defines a rooted tree, but it is not a "call tree". (non-recursive)

d

a b 'a' is popped. 'b' and 'd' are pushed.

Example: graphTraversalUsingStack(a)



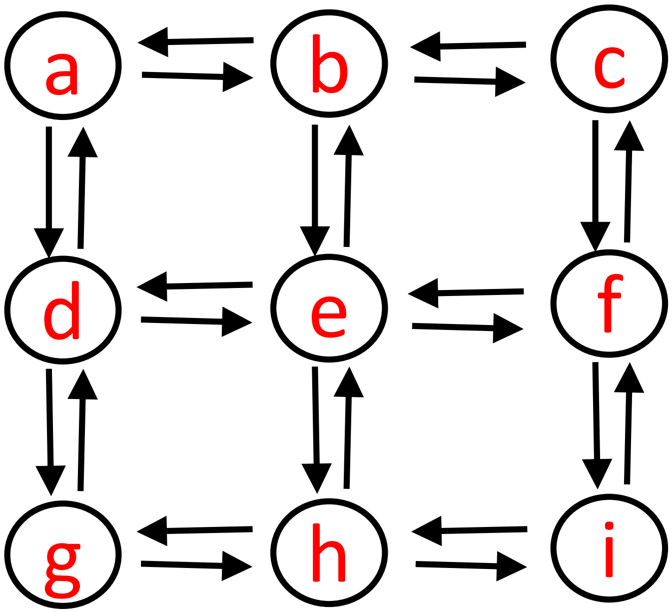


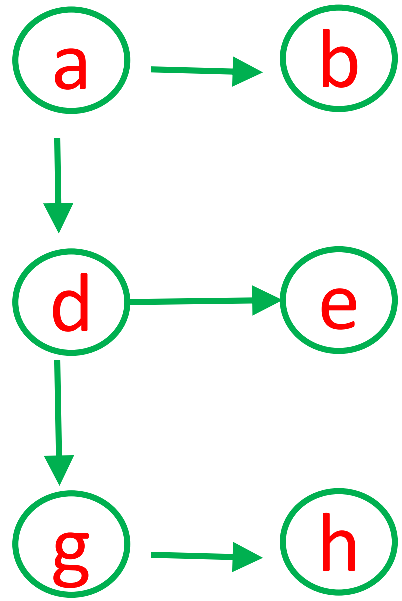
g

d e ‘d’ is popped. ‘e’ and ‘g’ are pushed.

a b b

Example: graphTraversalUsingStack(a)

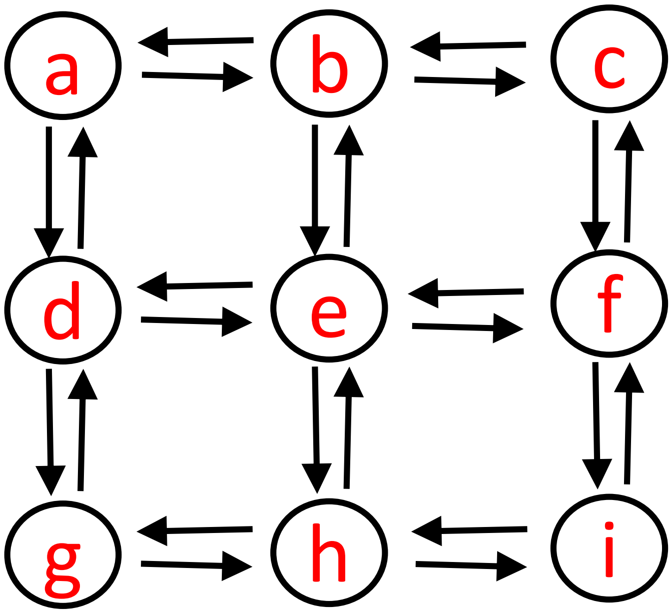


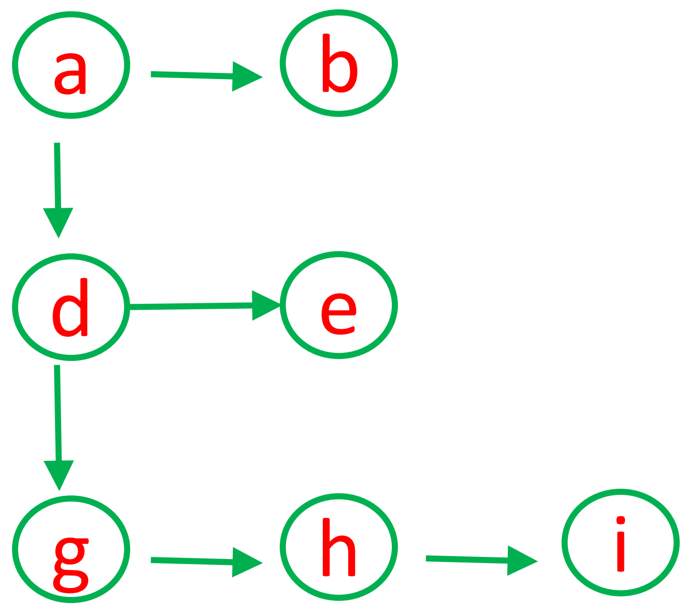


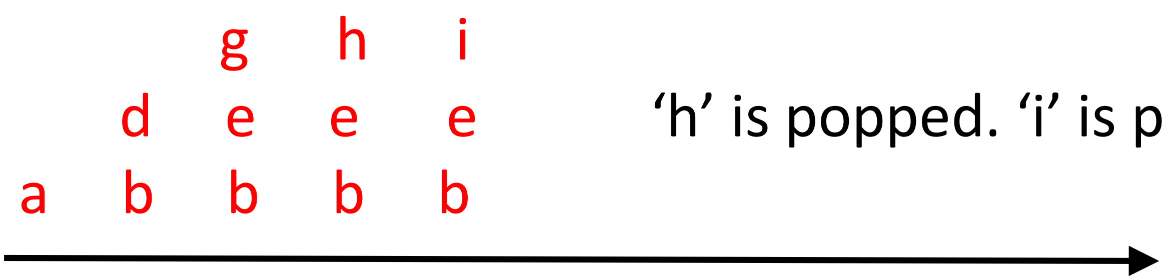
		g	h
	d	e	e
a	b	b	b

‘g’ is popped. ‘h’ is pushed.

Example: graphTraversalUsingStack(a)

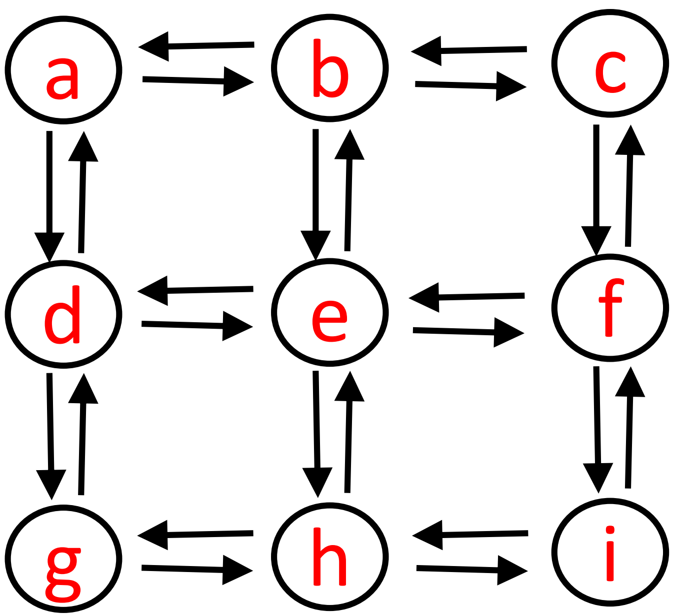


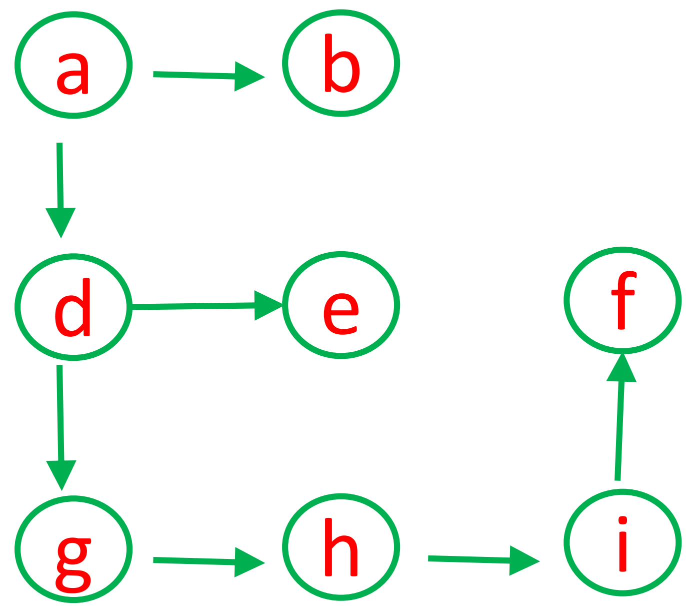




‘h’ is popped. ‘i’ is pushed.

Example: graphTraversalUsingStack(a)

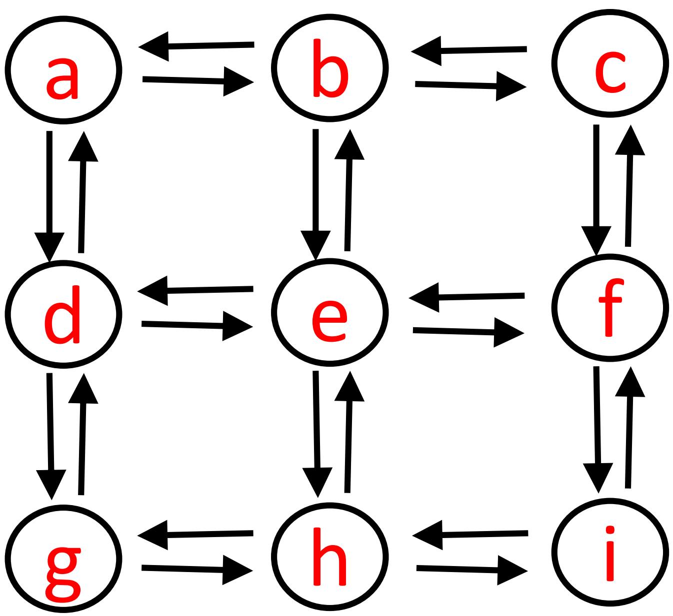


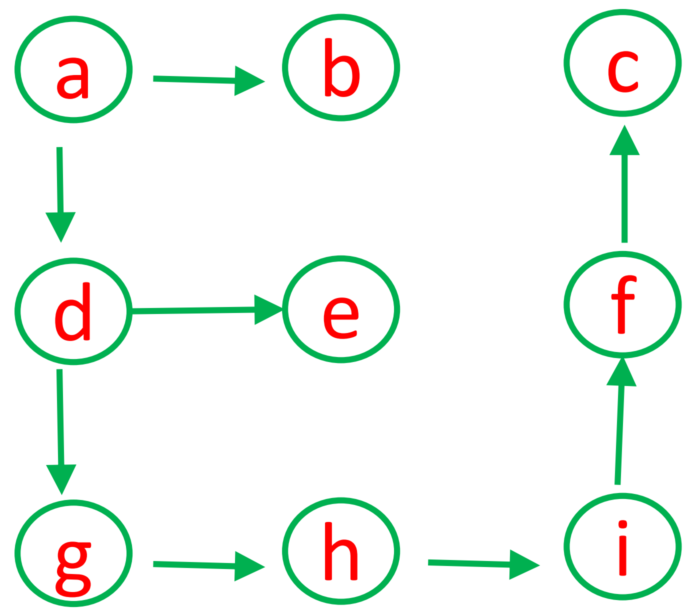


		g	h	i	f
	d	e	e	e	e
a	b	b	b	b	b

'i' is popped. 'f' is pushed.

Example: graphTraversalUsingStack(a)

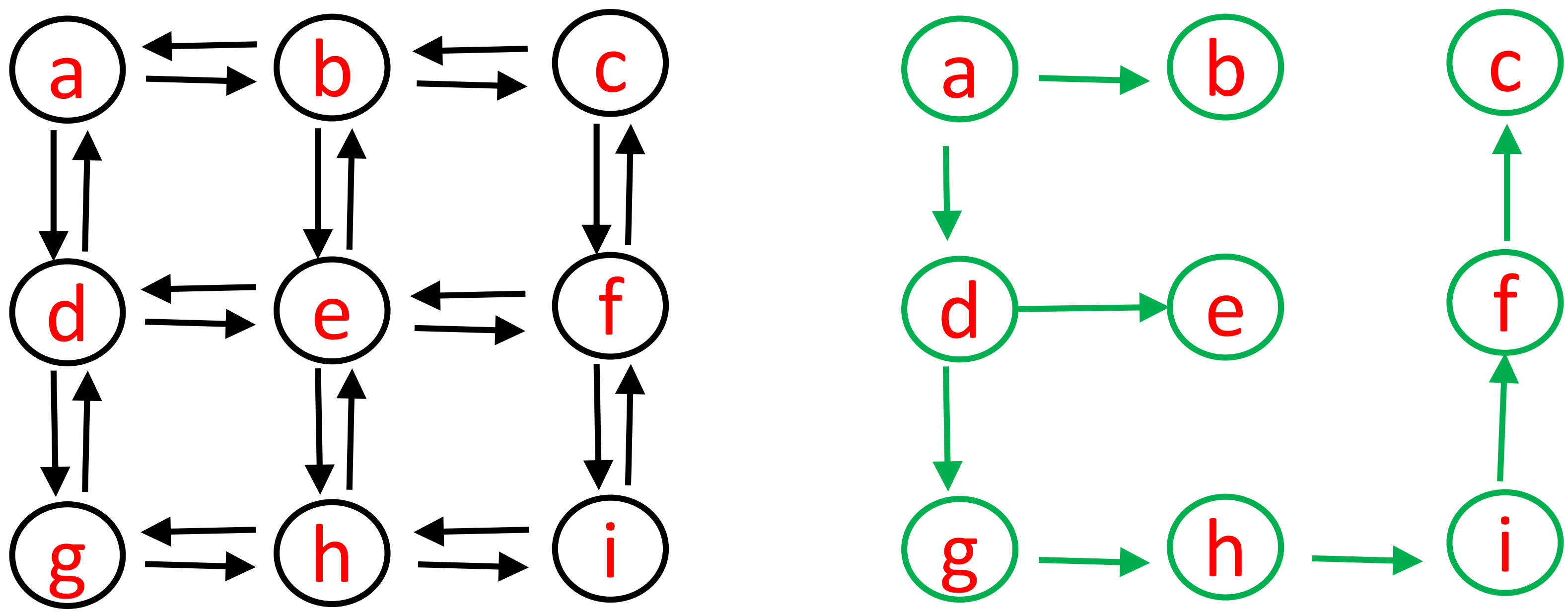


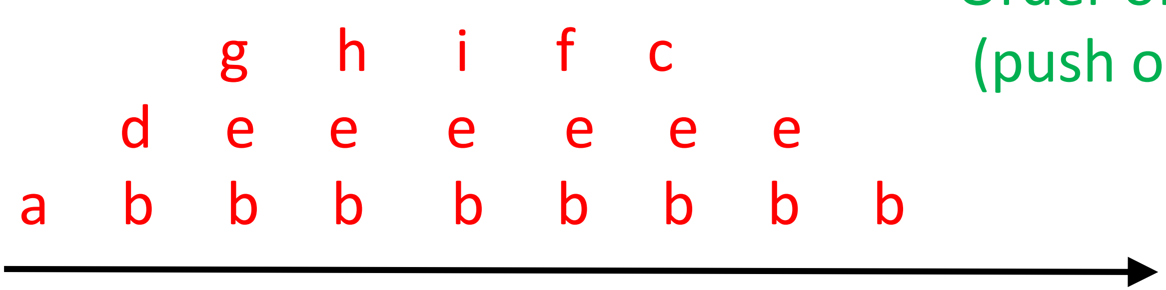


		g	h	i	f	c
	d	e	e	e	e	e
a	b	b	b	b	b	b

‘f’ is popped. ‘c’ is pushed.

Example: graphTraversalUsingStack(a)

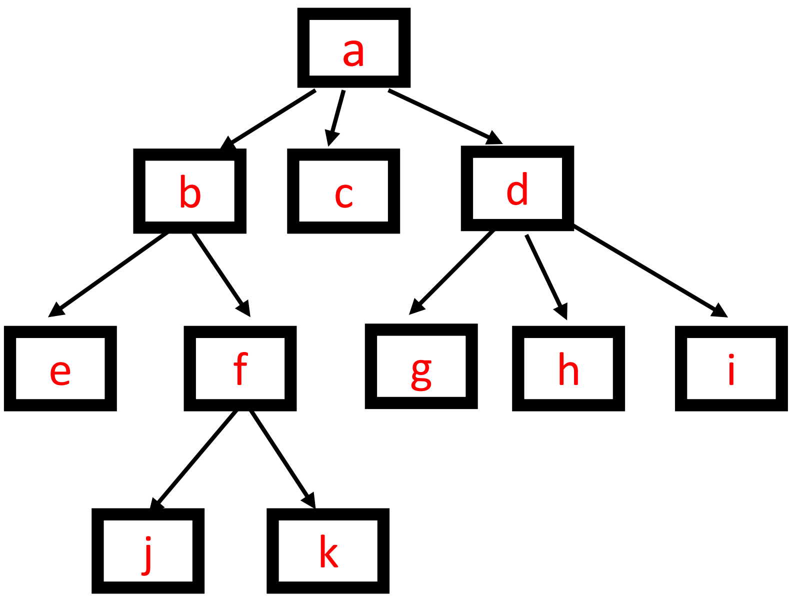




Order of nodes visited
(push order) : abdeghifc

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

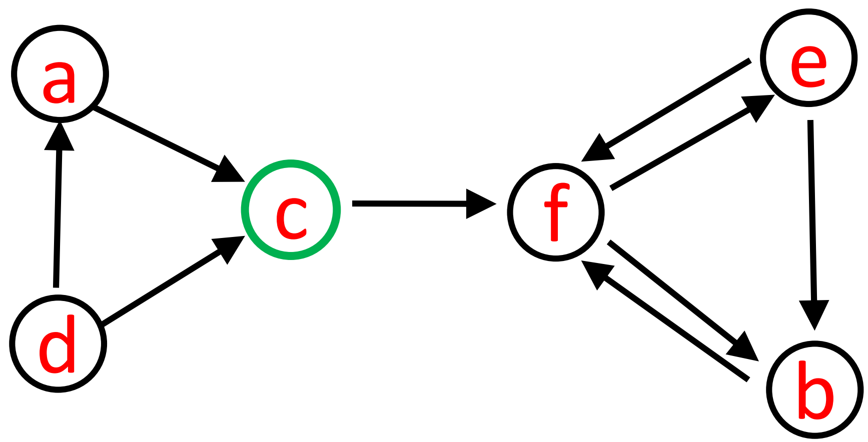
```
graphTraversalUsingQueue(v){
    initialize empty queue q
    v.visited = true
    q.enqueue(v)
    while (! q.empty) {
        u =  q.dequeue()
        for each w in u.adjList{
            if (!w.visited){
                w.visited = true
                q.enqueue(w)
            }
        }
    }
}
```

Example

graphTraversalUsingQueue(c)

queue

c



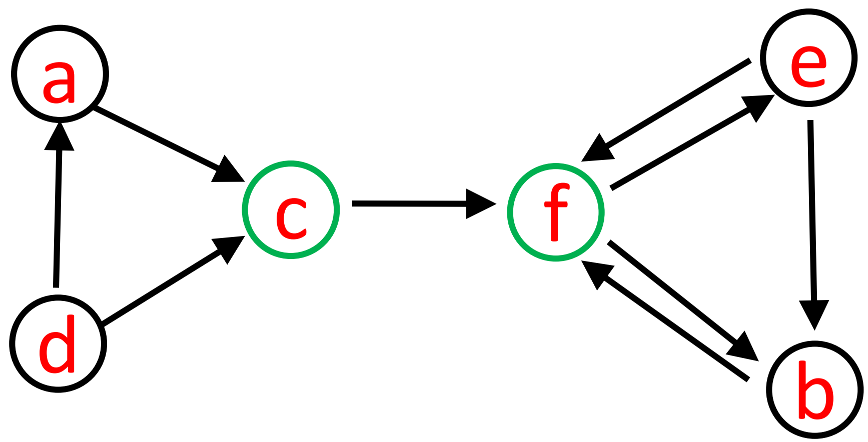
Example

graphTraversalUsingQueue(c)

queue

c

f



Example

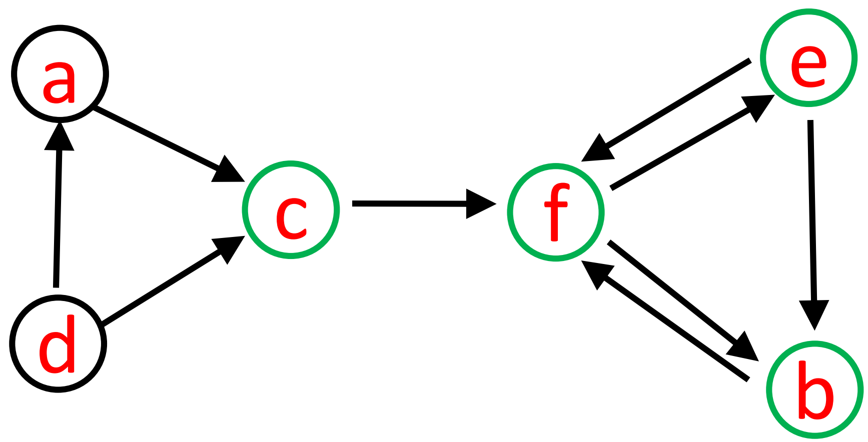
graphTraversalUsingQueue(c)

queue

c

f

be



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

Example

graphTraversalUsingQueue(c)

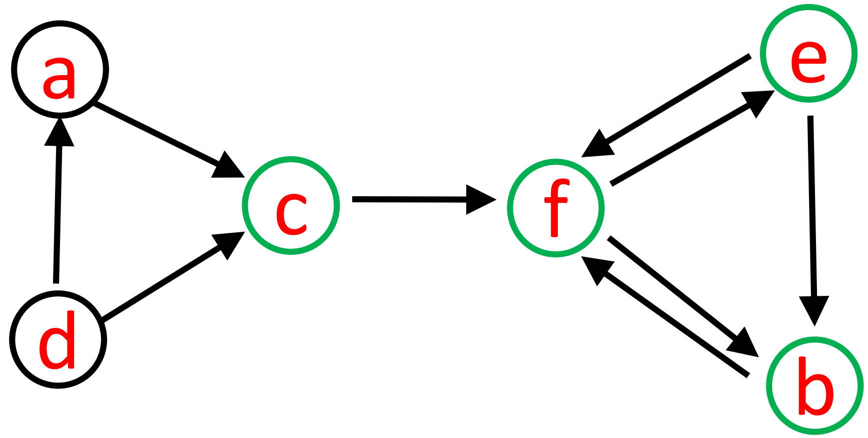
queue

c

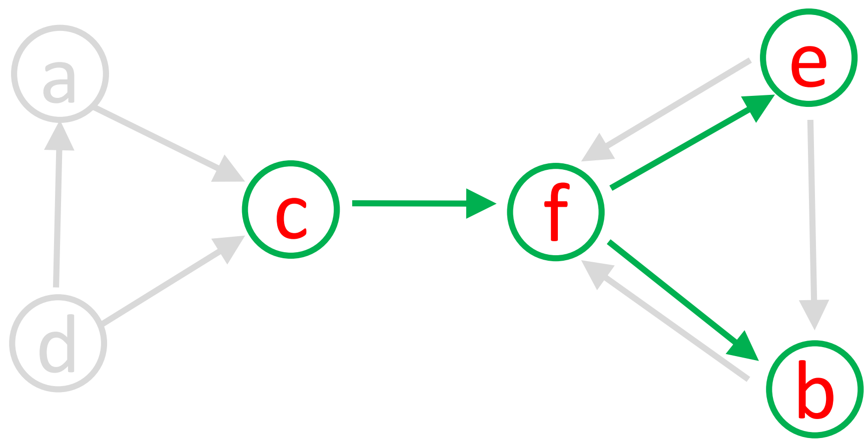
f

be

e

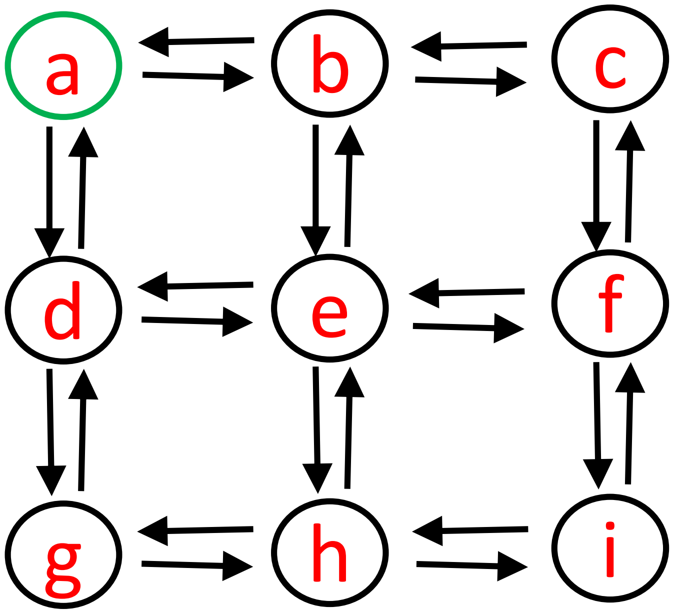


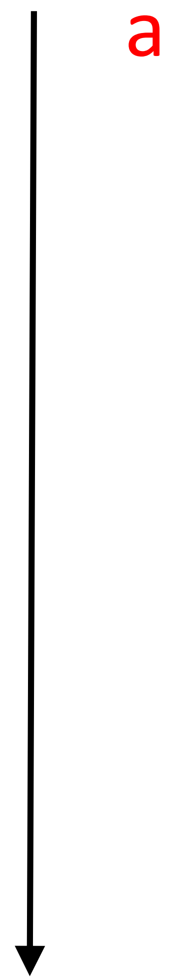
graphTraversalUsingQueue(c)



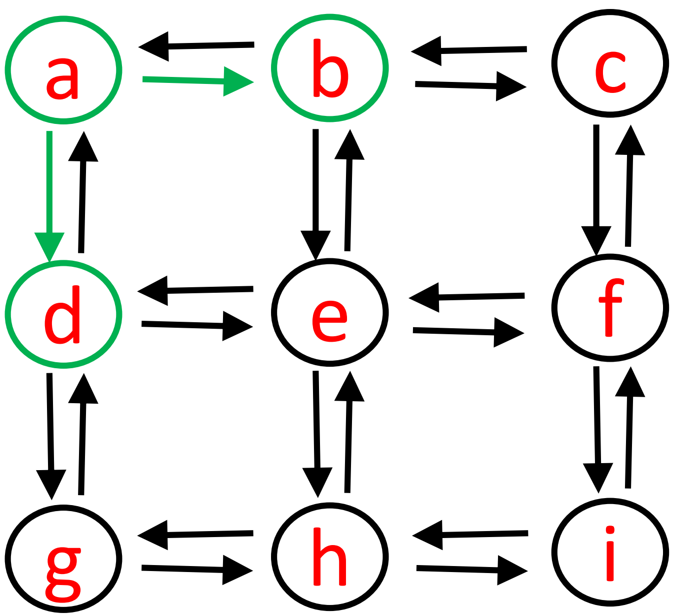
It defines a tree whose root is the starting vertex. It finds the shortest path (number of vertices) to all vertices reachable from starting vertex.

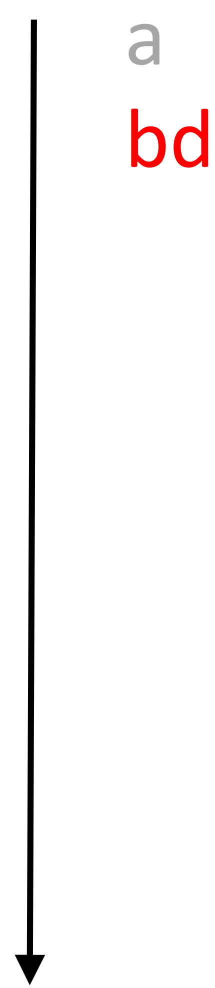
Example: graphTraversalUsingQueue(a)



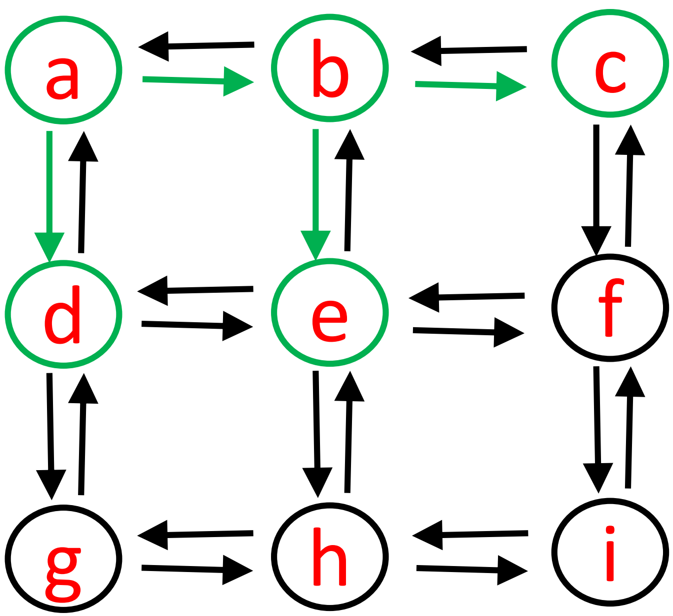


Example: graphTraversalUsingQueue(a)

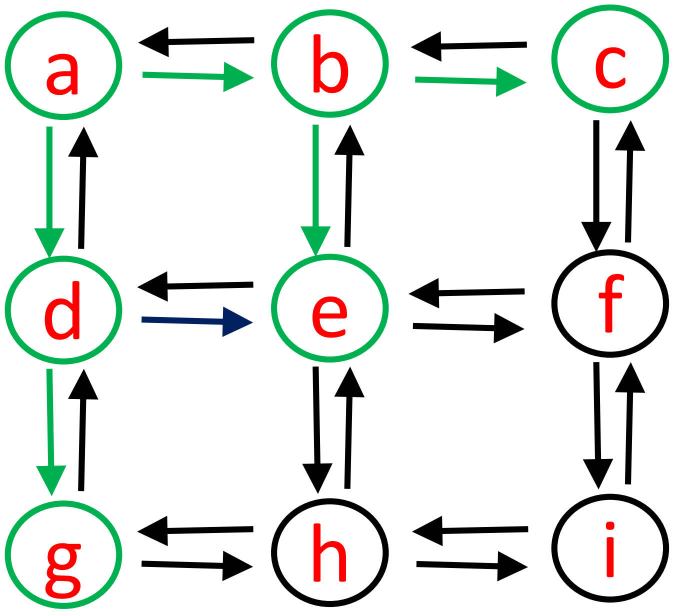


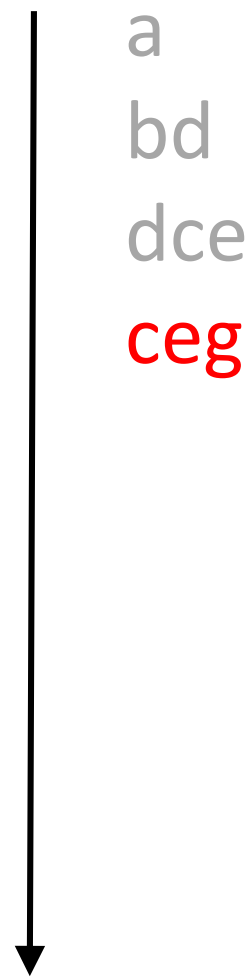


Example: graphTraversalUsingQueue(a)

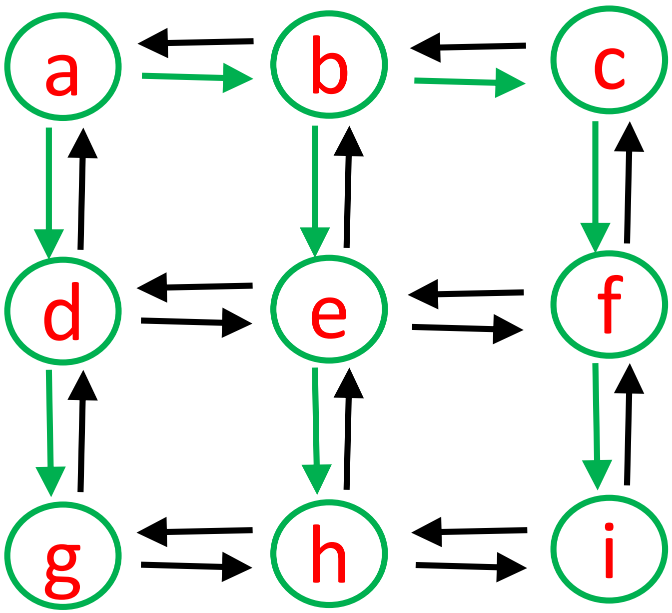


Example: graphTraversalUsingQueue(a)



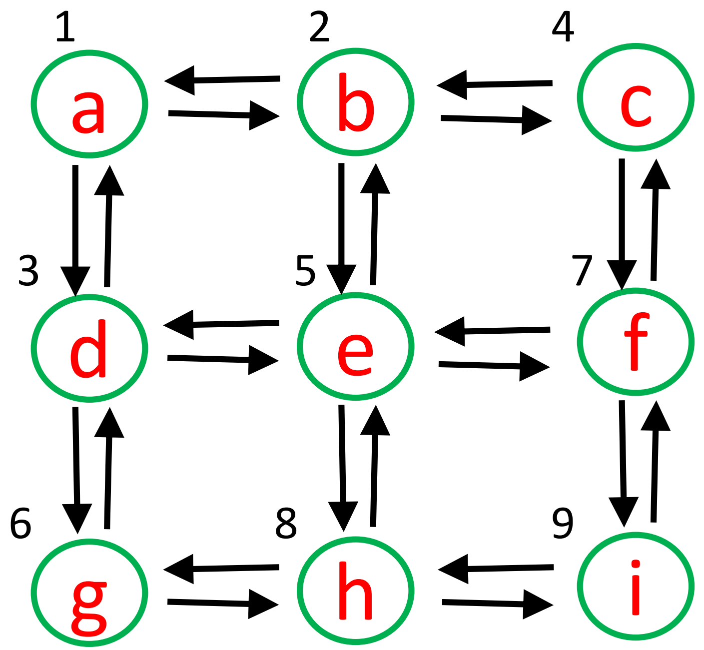


Example: graphTraversalUsingQueue(a)



a
bd
dce
ceg
egf
gfh
fh
hi
i

Example: graphTraversalUsingQueue(a)



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

Recall: How to implement a Graph class in Java?

```
class Graph<T>  {
    HashMap< String, Vertex<T> >    vertexMap;

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

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

HEADS UP ! Prior to traversal, ...

for each w in V
    w.visited = false

How to implement this ?

HEADS UP ! Prior to traversal, ...

for each w in V
    w.visited = false

How to implement this ?

```
class Graph<T> {
    HashMap< String, Vertex<T> >    vertexMap;
        :
    public void resetVisited() {



    }
}
```

HEADS UP ! Prior to traversal, ...

for each w in V
    w.visited = false

How to implement this ?

```
class Graph<T> {
    HashMap< String, Vertex<T> >   vertexMap;
      :
    public void resetVisited() {
        for( Vertex<T>   v  :   vertexMap.values() ){
            v.visited = false;
        }
    }
}
```

TODO

next lecture

Non-linear Data Structures

22. rooted trees
23. tree traversal
24. binary trees e.g. expression trees
25. binary search trees
26. priority queue, heaps 1
27. heaps 2
28. maps, hash codes
29. hash maps
30. graphs
31. graph traversal (breadth vs depth first)
32. graph applications: Google page rank, garbage collection

Mathematical Tools for Analysis of Algorithms

33. recurrences 1 *back substitution method, examples*
34. recurrences 2 *mergesort, quicksort*
35. big O 1 *formal big O definition*
36. big O 2 *rules for big O, big Omega, some incorrect proofs*
37. big O 3 *big Theta, best and worst case, limits rules*