# CS 10: Problem solving via Object Oriented Programming

## Graph Traversals

# Agenda

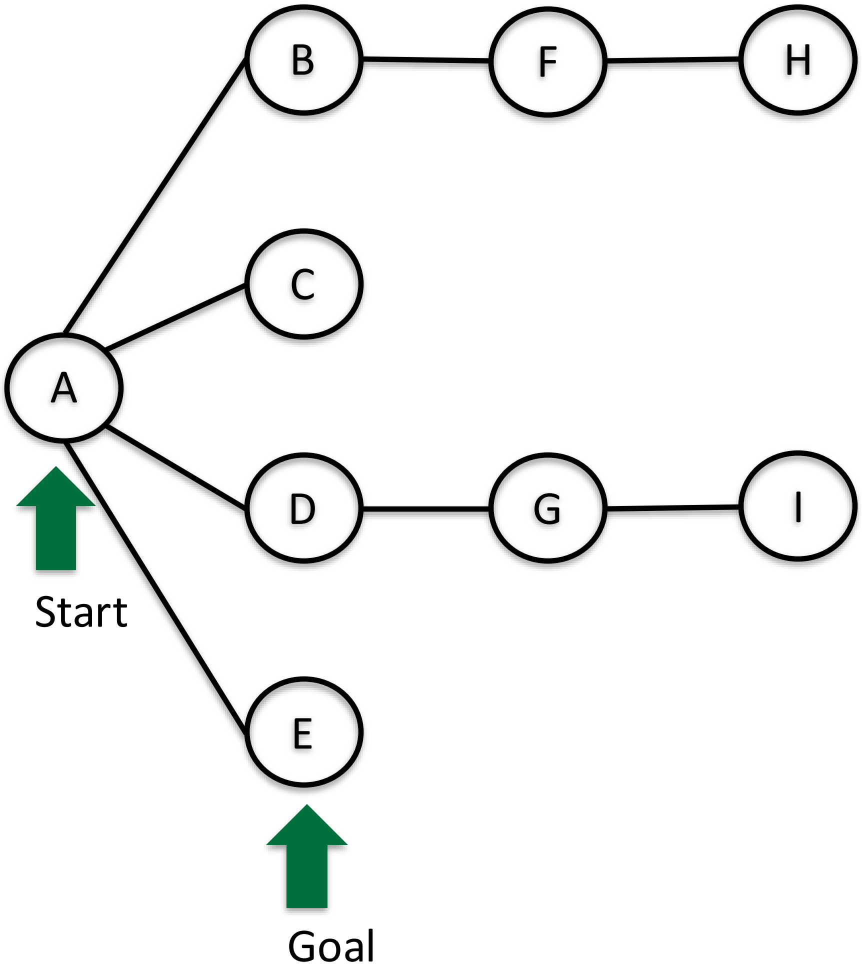
1. Depth first search (DFS)
2. Breadth first search (BFS)
3. Examples from last class and today

# Graph traversals are useful to answer questions about relationships

**Some Graph traversals uses**

- Uses are often around *reachability*
- Computing *path* from vertex `u` to vertex `v`
- Given start vertex `s` on Graph `G`, compute a path with the minimum number of edges between `s` and all other vertices (or report no such path exists)
- Testing whether `G` is fully connected (e.g., all vertices reachable)
- Identifying *cycles* in `G` (or reporting no cycle exists)
- Today's examples have ~~no~~ few cycles (cycles next class)

# Depth First Search (DFS) uses a stack to explore as if in a maze




**Goal: compute path from *start* to *goal (or to all other nodes)***

**DFS basic idea**

- Keep going until you can't go any further, then back track
- Relies on a **Stack** (implicit or explicit) to keep track of where you've been



Graph structure from http://stackoverflow.com/questions/687731/breadth-first-vs-depth-first

# Some of you did Depth First Search on Problem Set 1

**RegionFinder pseudo code**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

# Some of you did Depth First Search on Problem Set 1

**RegionFinder pseudo code**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

If you added to end of list…

# Some of you did Depth First Search on Problem Set 1

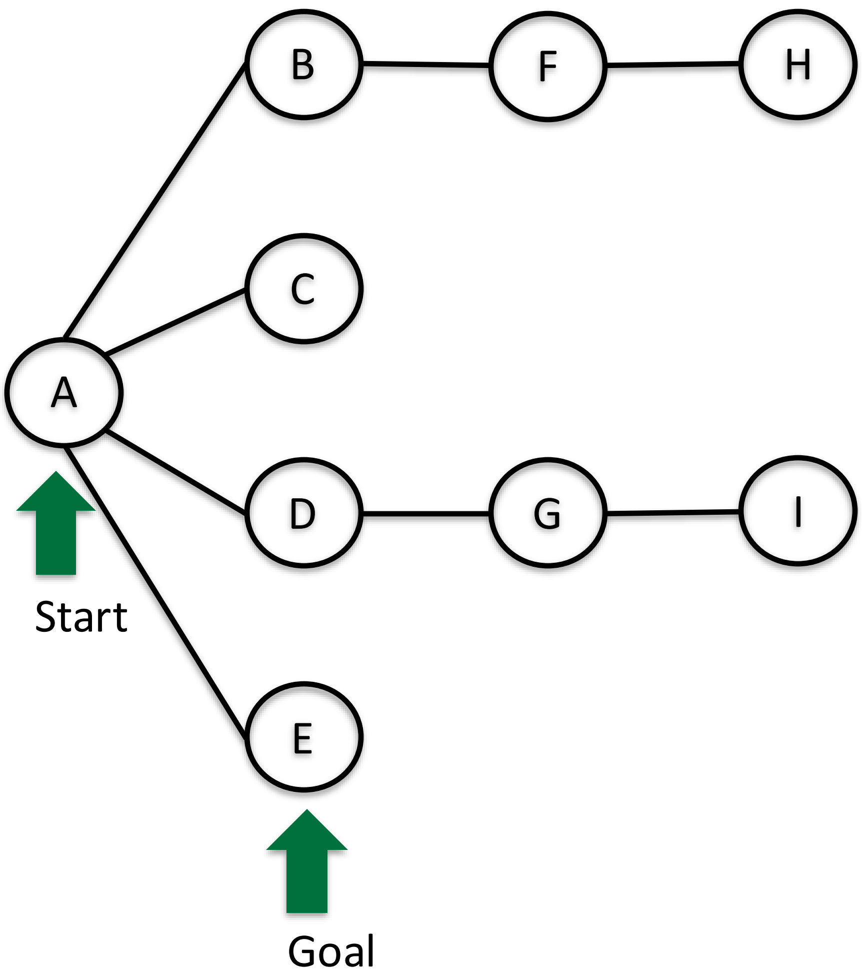
**RegionFinder pseudo code**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

And if you get a pixel from end of list, you implemented a stack

If you added to end of list…

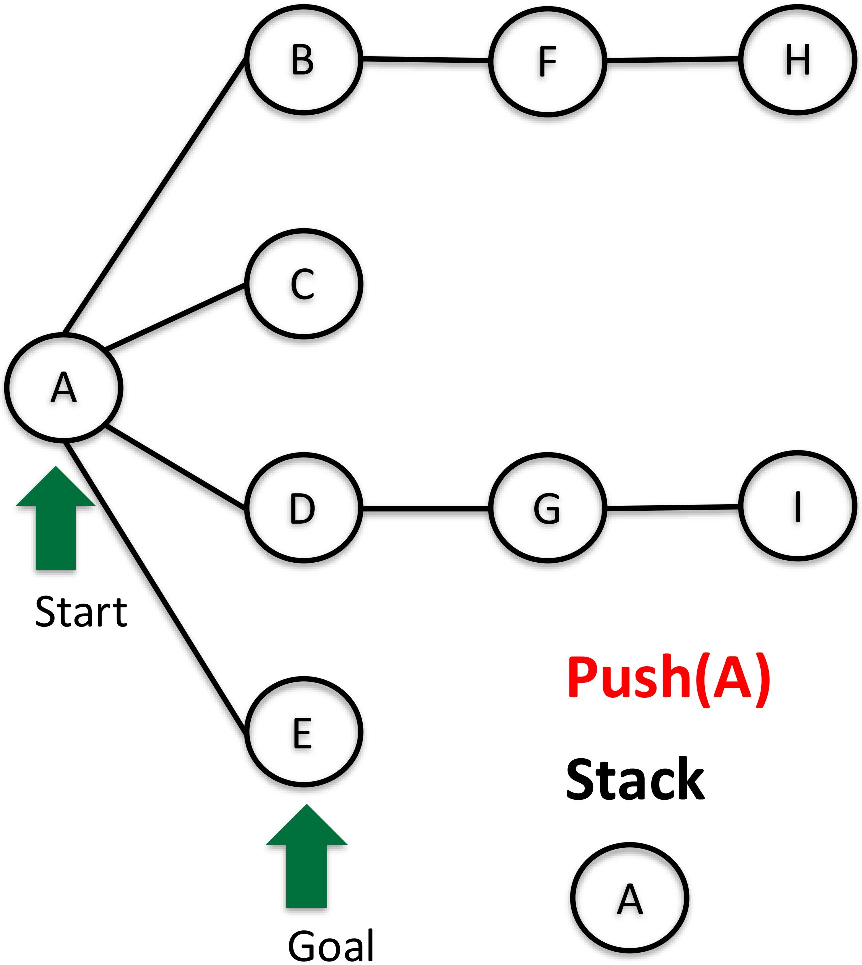
# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




**DFS algorithm**

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




**DFS algorithm**

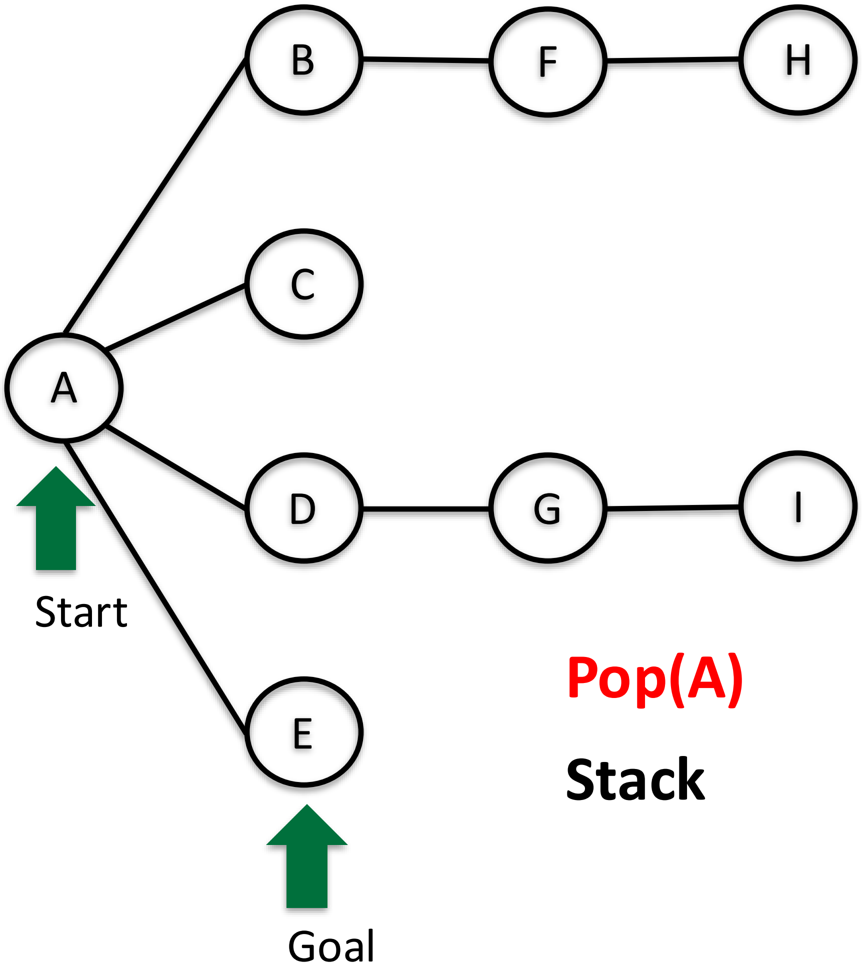
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Push(A)**

**Stack**

A

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)



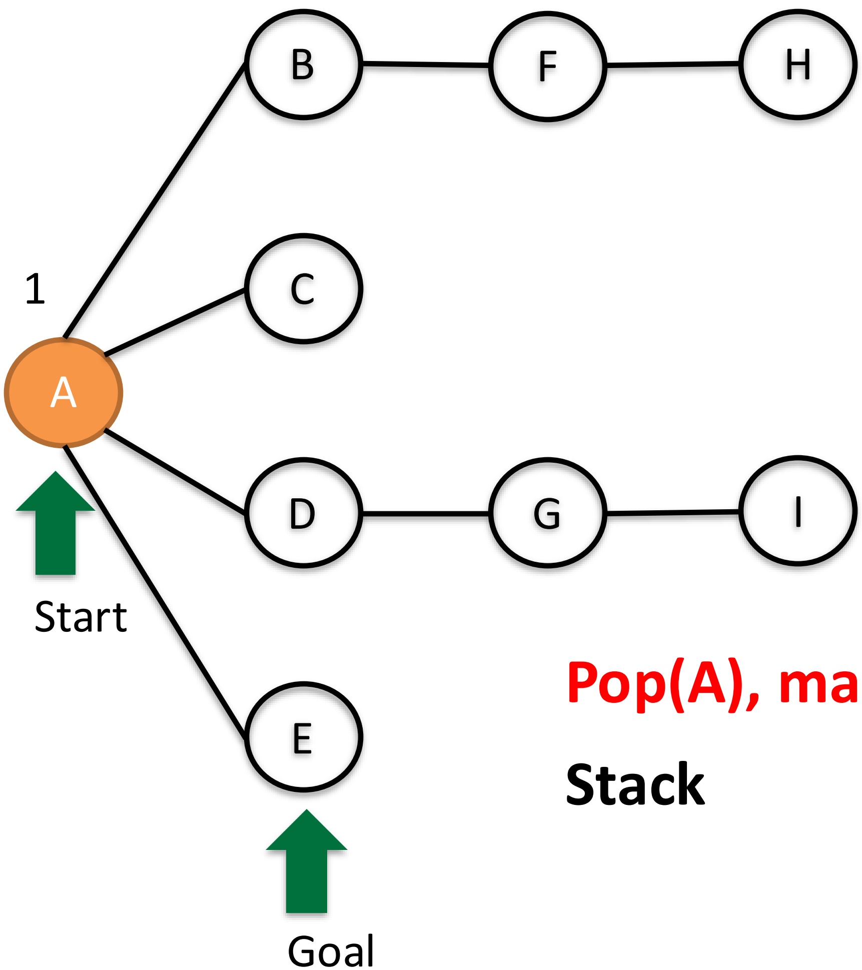

**DFS algorithm**

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
   u = stack.pop()
   if !u.visited
      u.visited = true
      (do something while here)
      for v ∈ u.adjacent
           if !v.visited
            stack.push(v)
```

**Pop(A)**

**Stack**

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




**DFS algorithm**

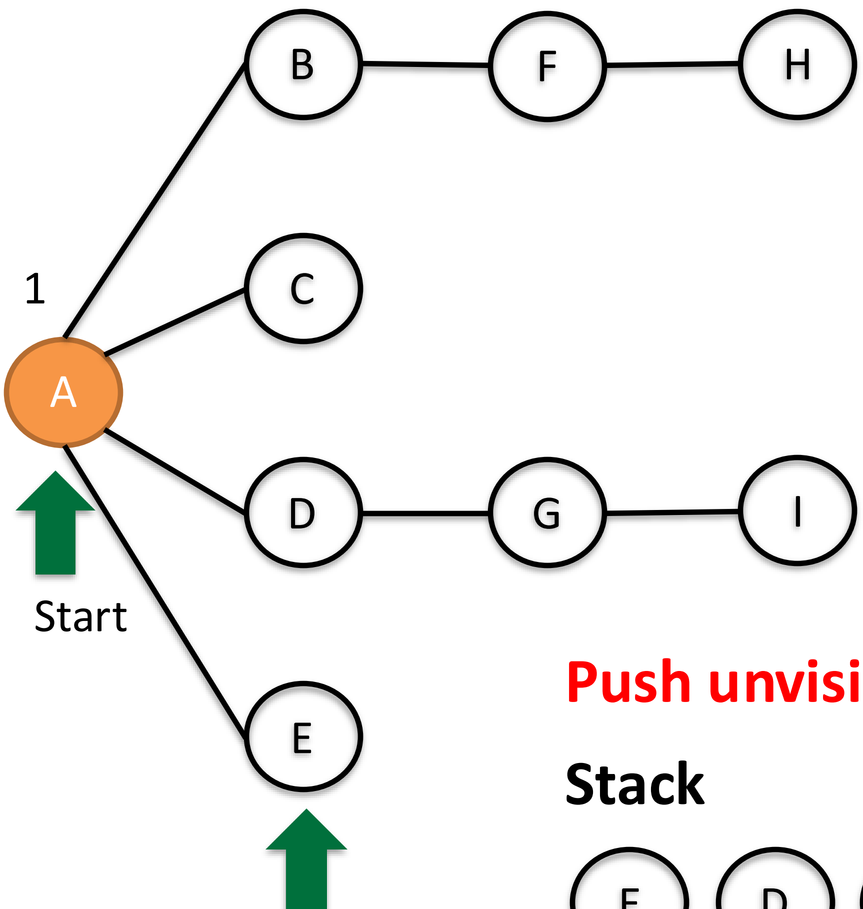
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
       u.visited = true
       (do something while here)
       for v ∈ u.adjacent
             if !v.visited
               stack.push(v)
```

**"Visit" the node**

**Pop(A), mark visited**

**Stack**

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)

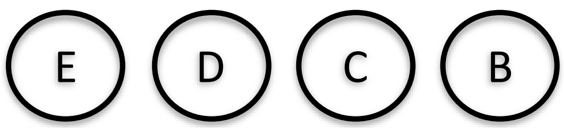



**DFS algorithm**

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

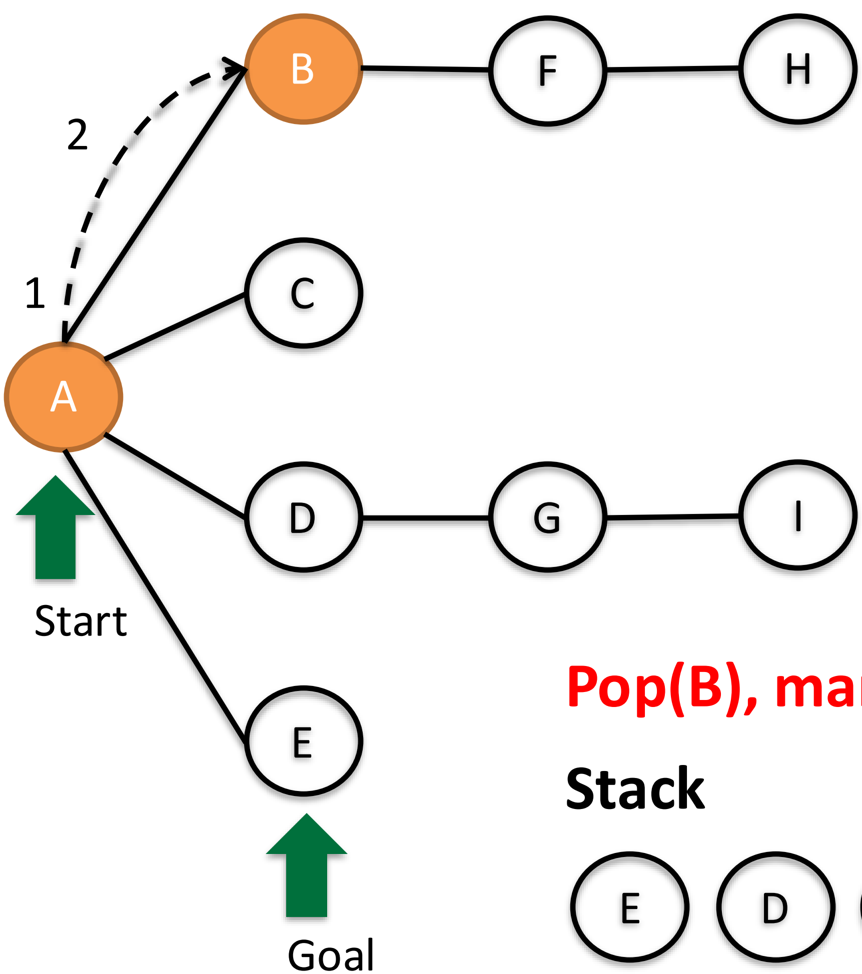
**Push unvisited adjacent**

**Stack**




- **What method would we use on our AdjacencyMapGraph?**
- ***graph.outNieghbors(u)***
- **Order pushed onto stack depends on order of nodes from outNeighbors iterator**

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
   u = stack.pop()
   if !u.visited
      u.visited = true
      (do something while here)
      for v ∈ u.adjacent
           if !v.visited
             stack.push(v)
```

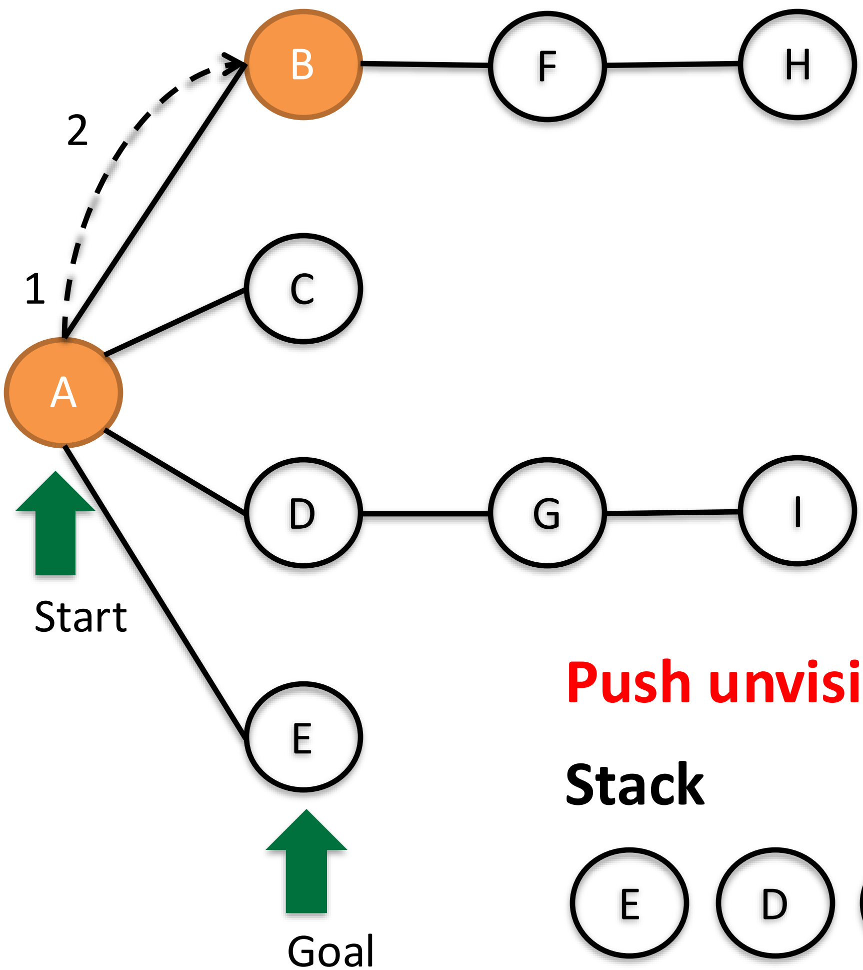
**Pop(B), mark visited**

**Stack**

E D C

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)

Order nodes visited




## DFS algorithm

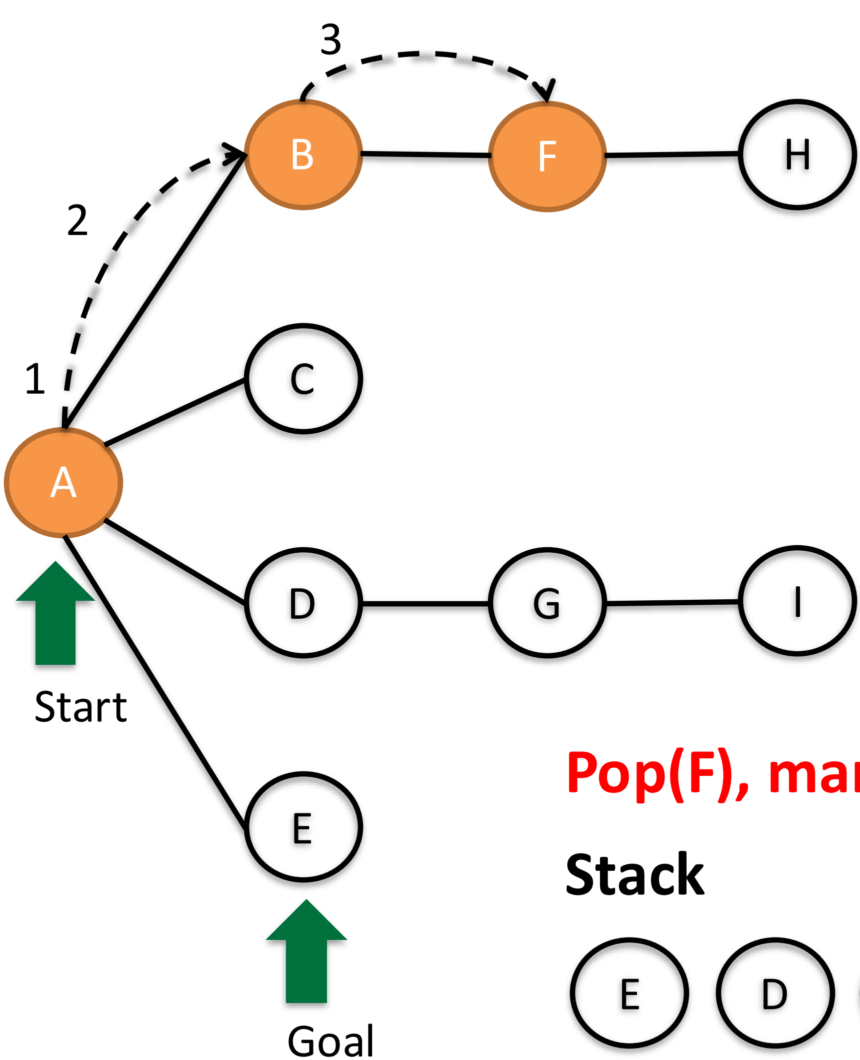
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Push unvisited adjacent (F, but not A)**

**Stack**

E D C F

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
   u = stack.pop()
   if !u.visited
       u.visited = true
       (do something while here)
       for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

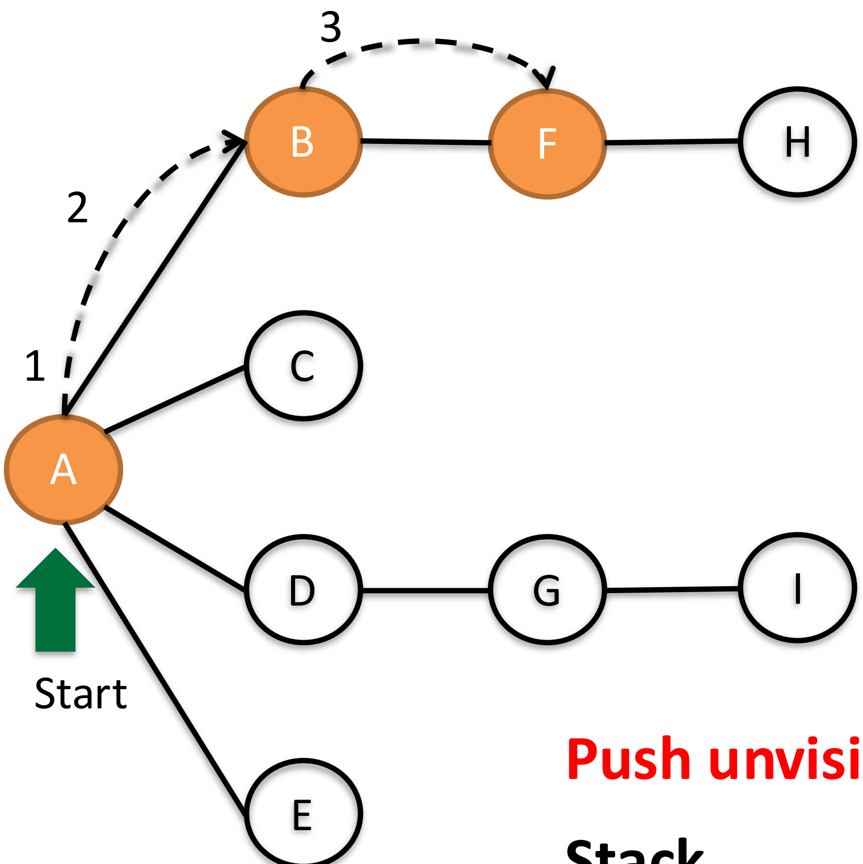
**Pop(F), mark visited**

**Stack**

E D C

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)

Order nodes visited



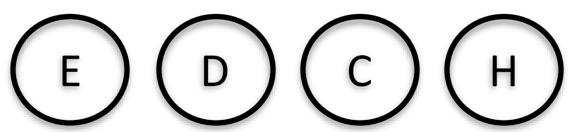


**DFS algorithm**

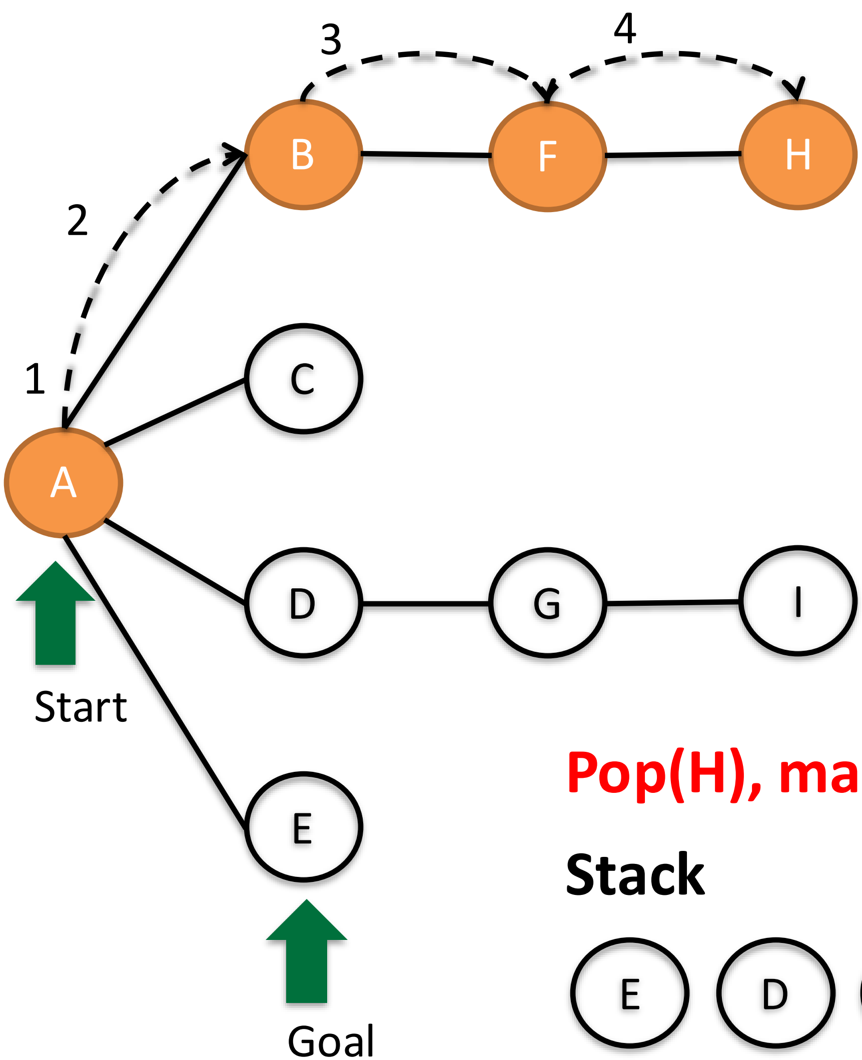
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Push unvisited adjacent (H, but not B)**

**Stack**



# Depth First Search (DFS) finds *__a path__* between *start* and other nodes (if exists)




Order nodes visited

## DFS algorithm

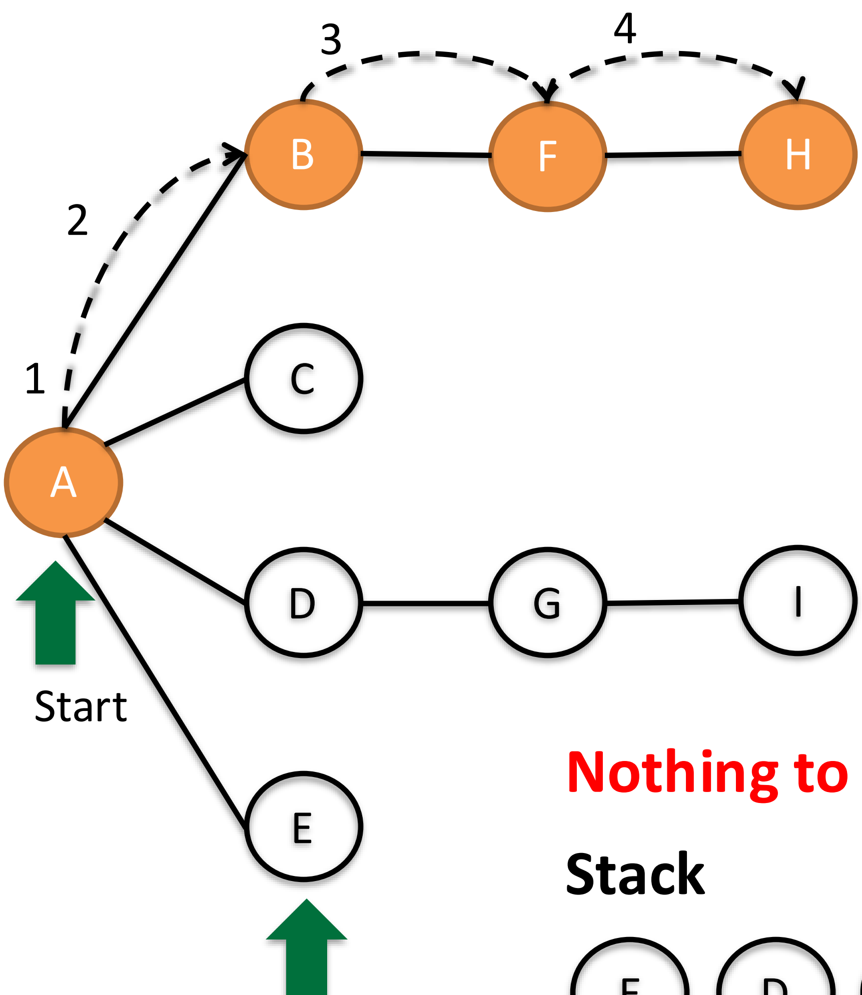
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(H), mark visited**

**Stack**

E D C

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)

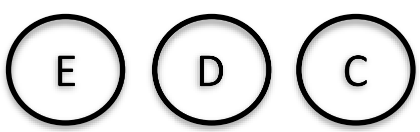



**DFS algorithm**

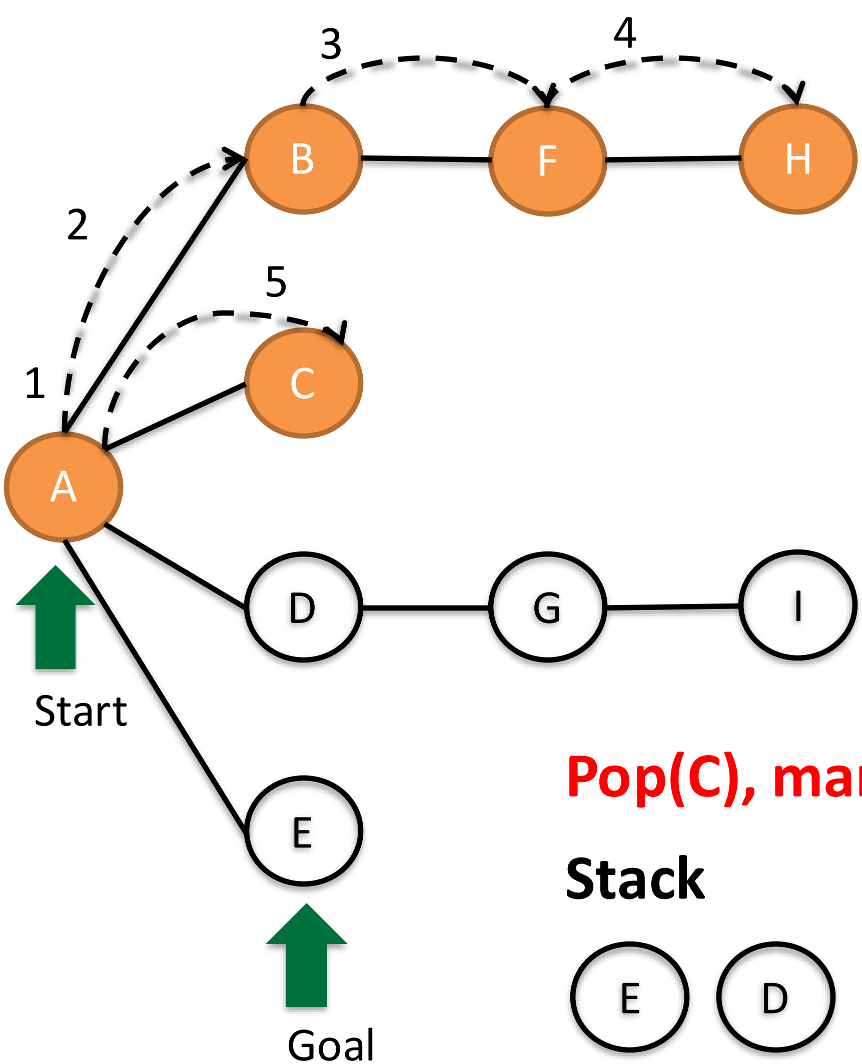
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
   u = stack.pop()
   if !u.visited
      u.visited = true
      (do something while here)
      for v ∈ u.adjacent
           if !v.visited
            stack.push(v)
```

**Nothing to push, back up by popping C**

**Stack**



# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




**DFS algorithm**

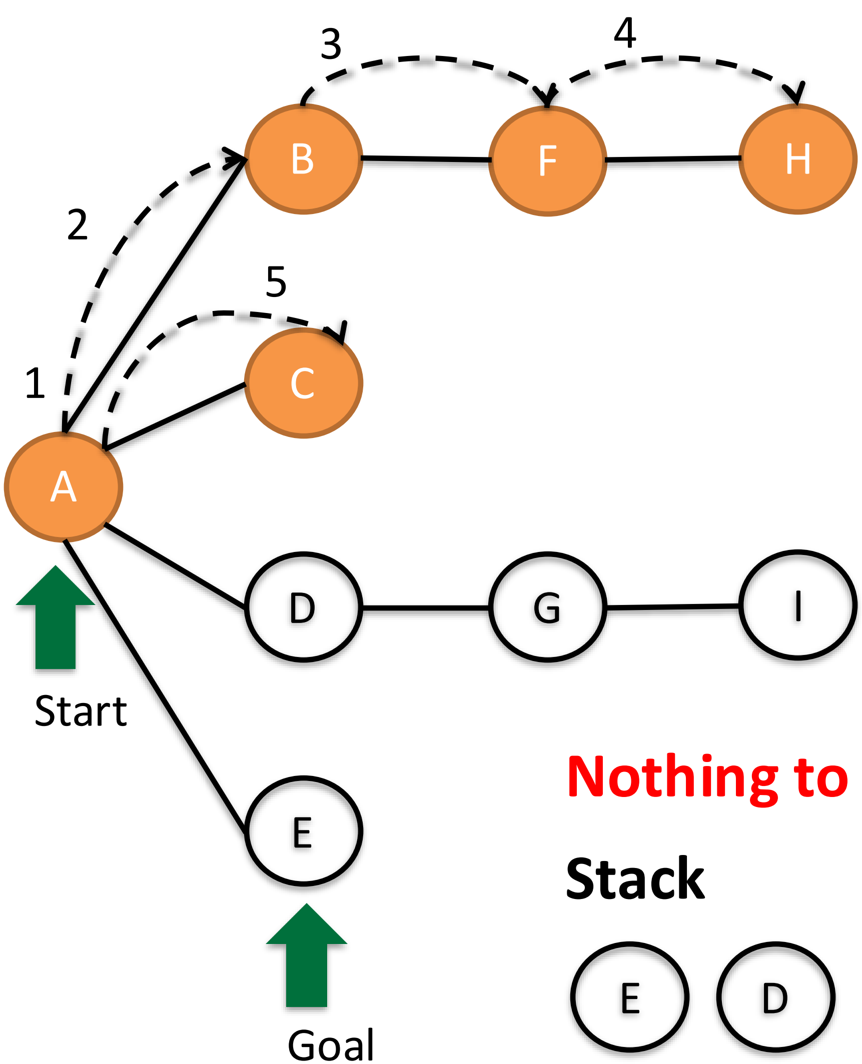
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(C), mark visited**

**Stack**

E D

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

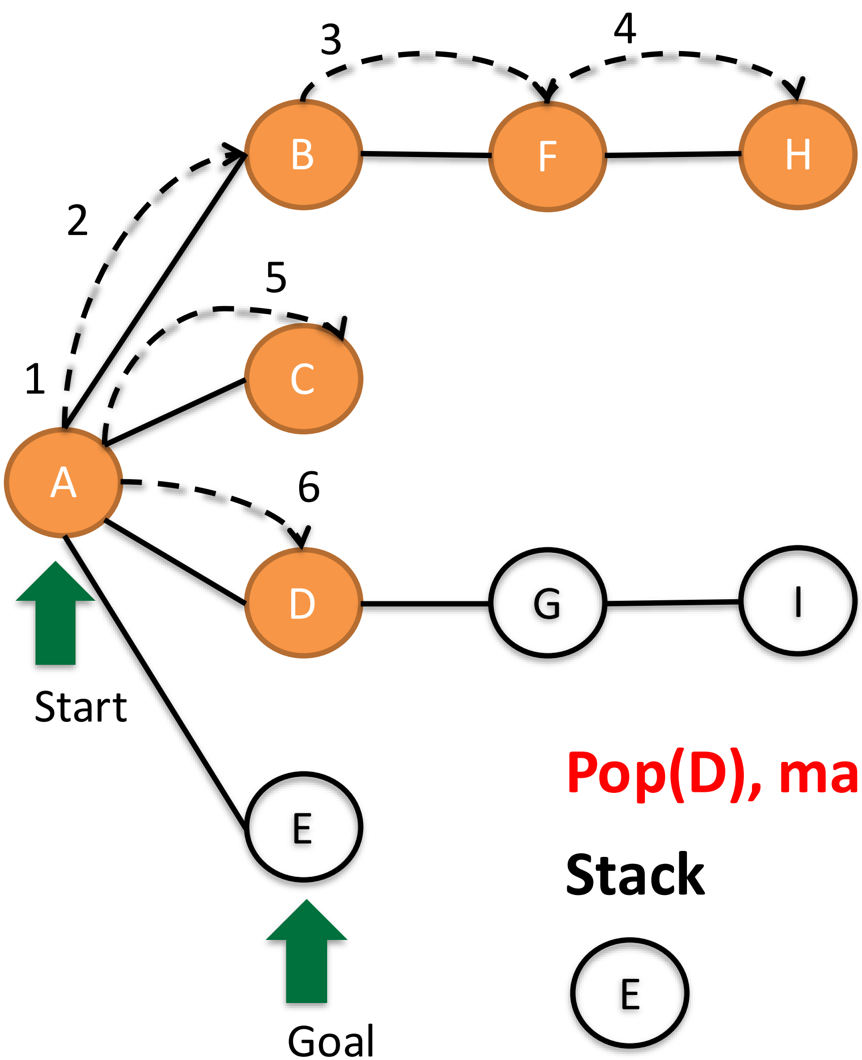
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Nothing to push, back up by popping D**

**Stack**

E D

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




Order nodes visited

## DFS algorithm

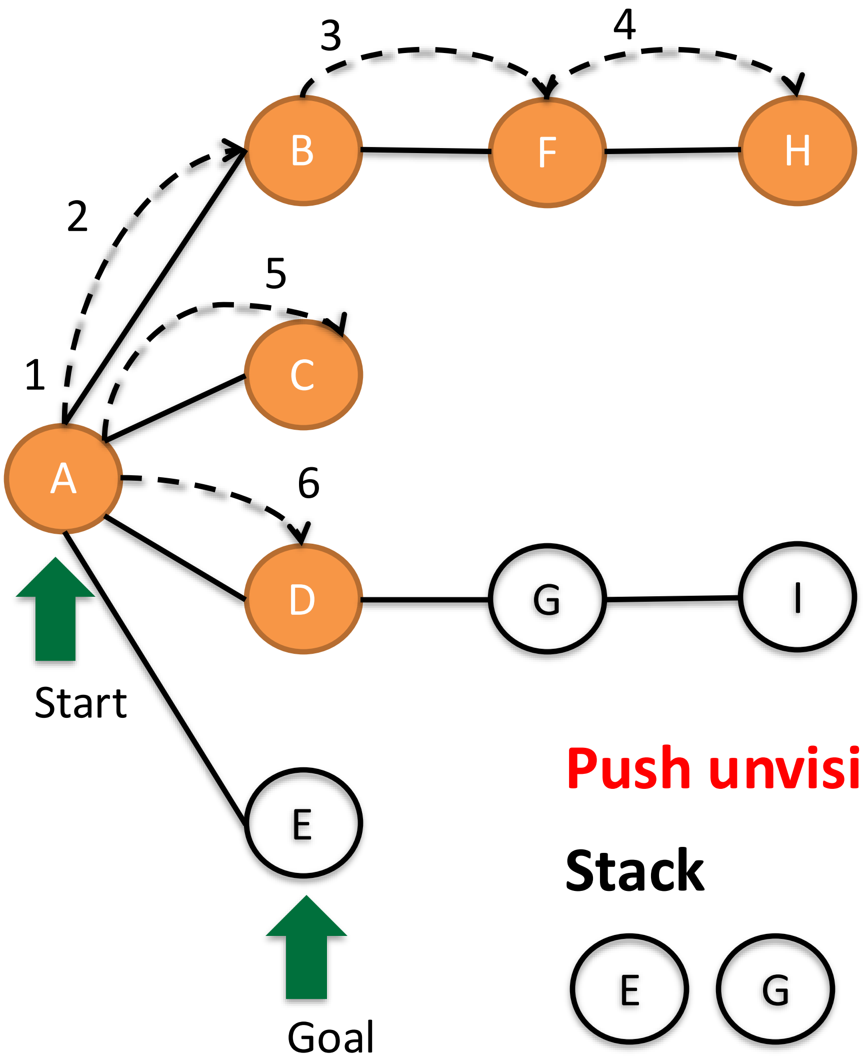
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(D), mark visited**

**Stack**

E

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




Order nodes visited

## DFS algorithm

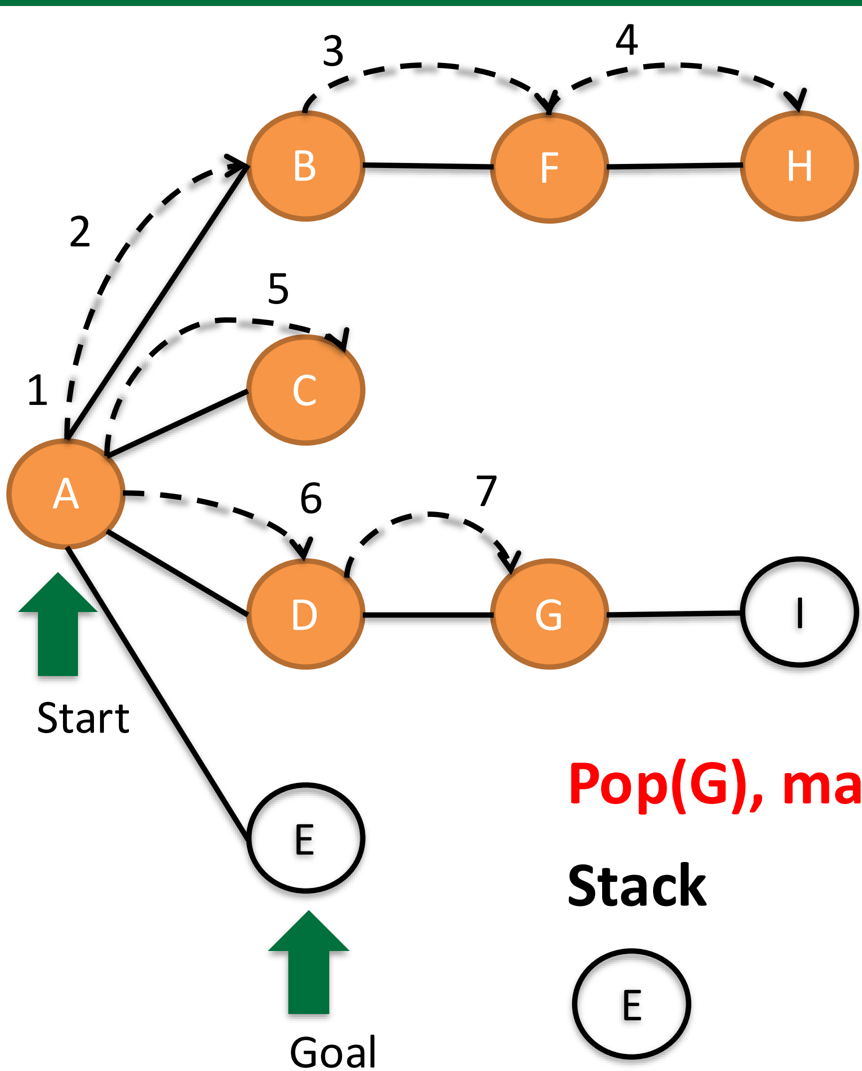
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Push unvisited adjacent (G, but not A)**

**Stack**

E G

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

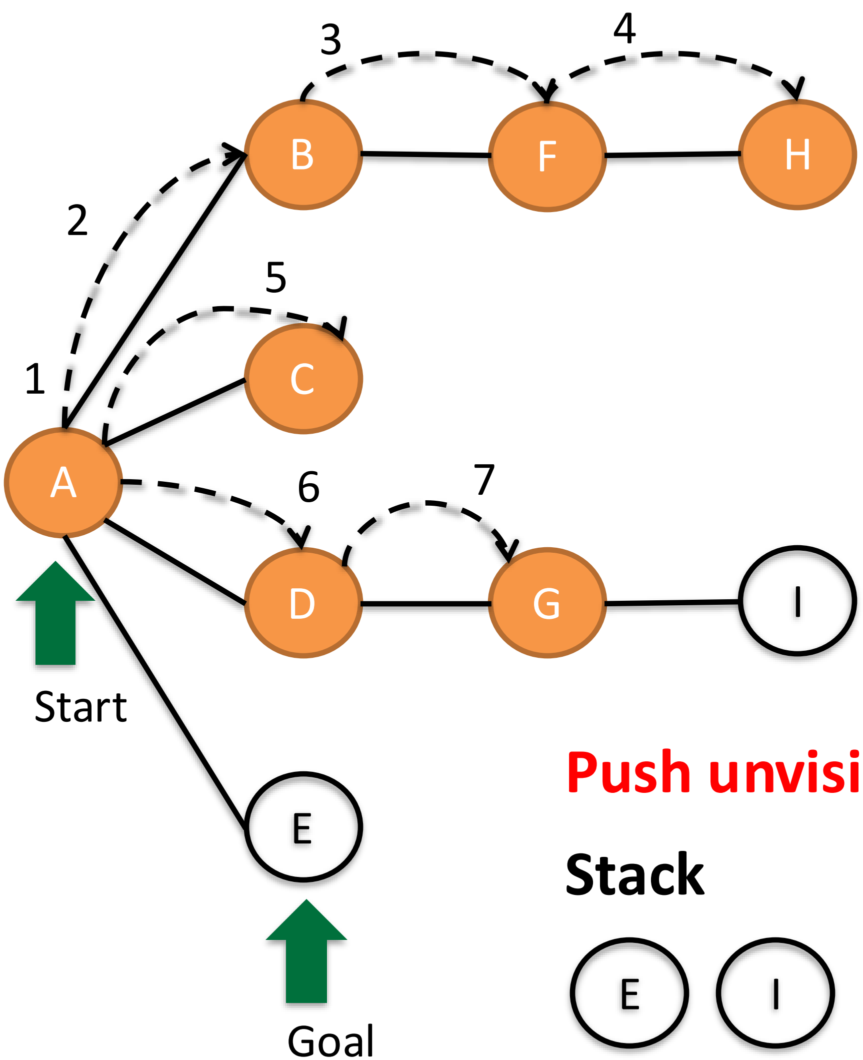
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(G), mark visited**

**Stack**

E

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




Order nodes visited

**DFS algorithm**

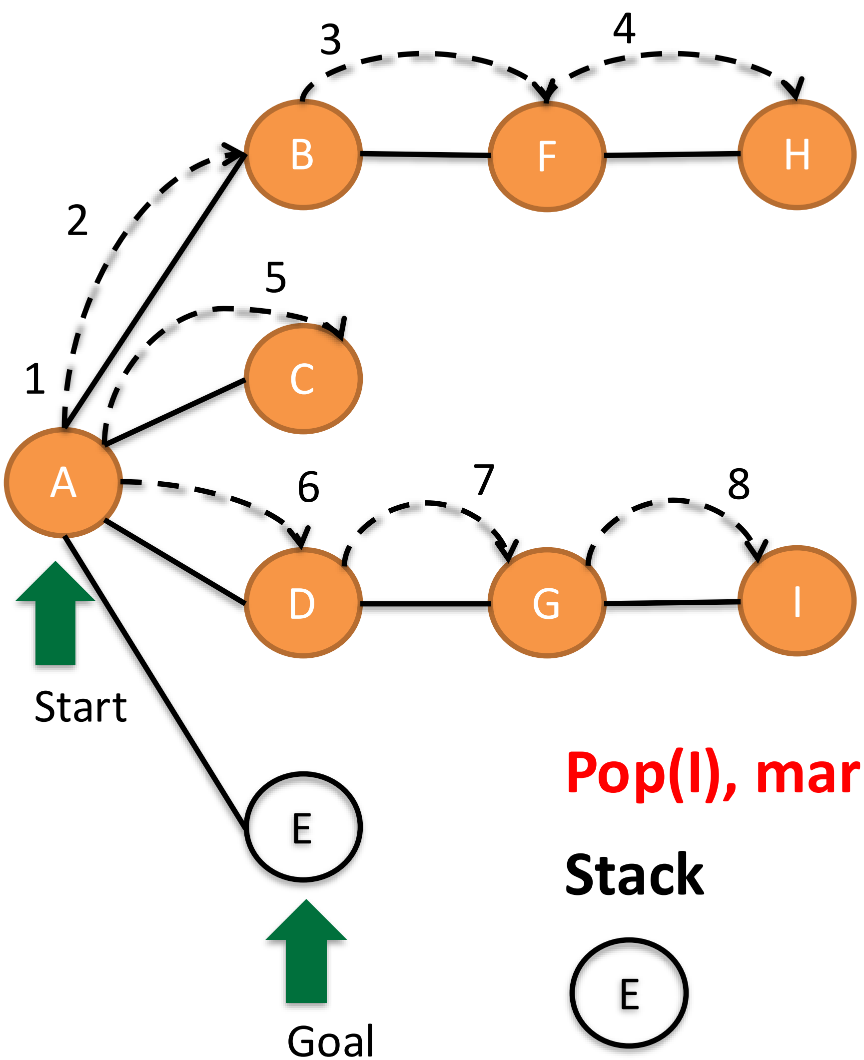
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Push unvisited adjacent (I, but not D)**

**Stack**

E I

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

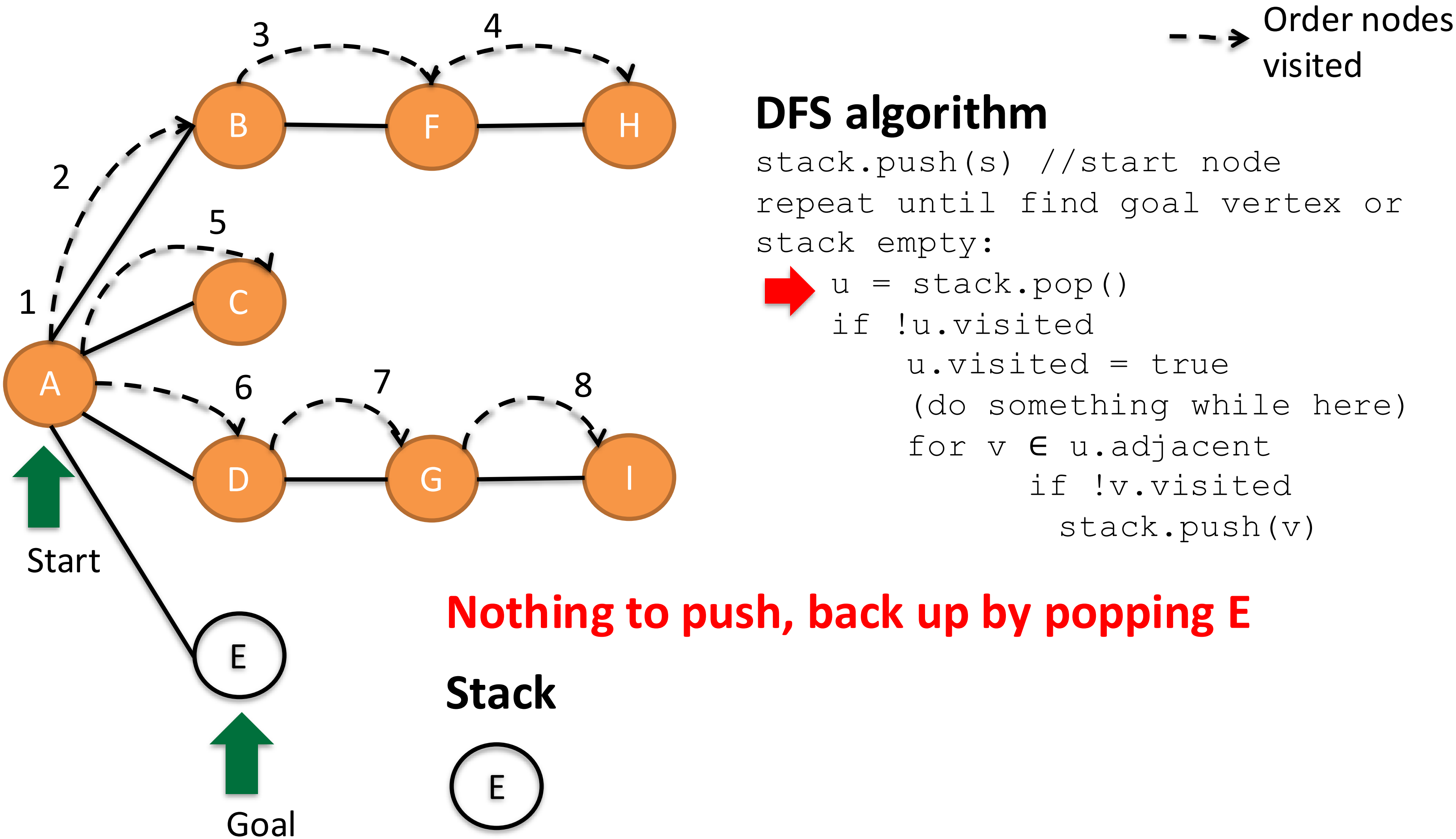
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(I), mark visited**

**Stack**

E

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




## DFS algorithm

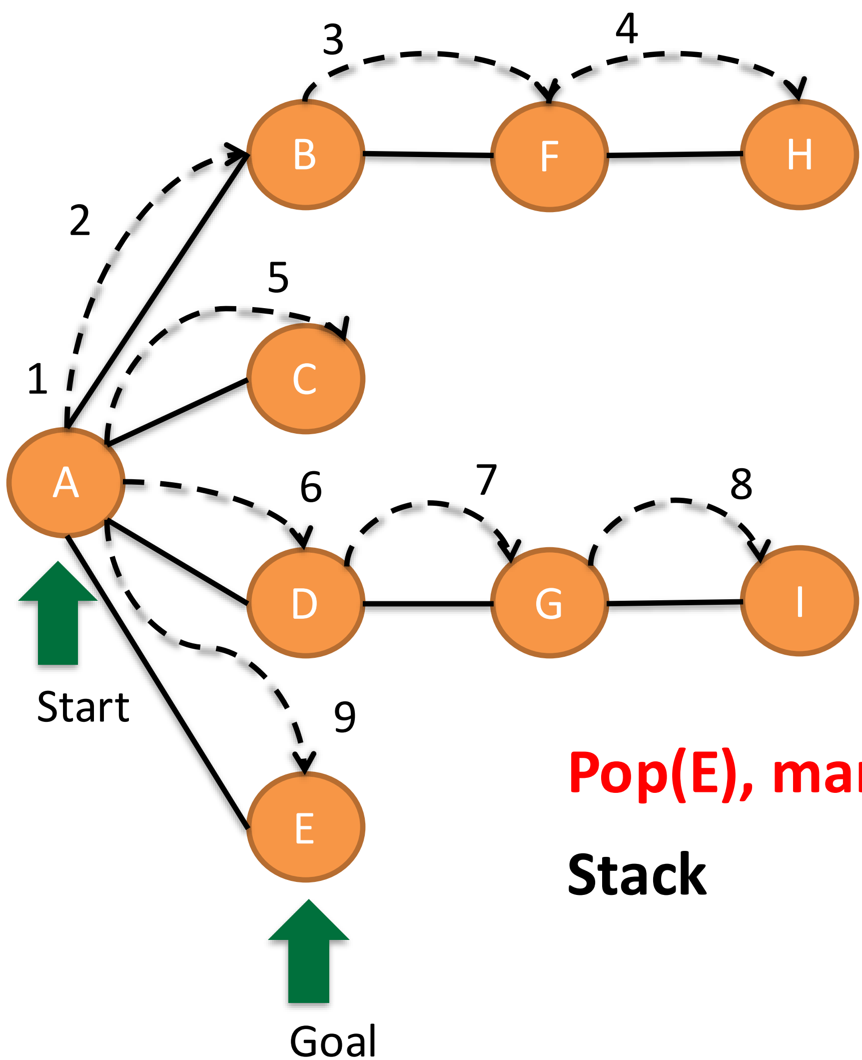
```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
   u = stack.pop()
   if !u.visited
      u.visited = true
      (do something while here)
      for v ∈ u.adjacent
           if !v.visited
            stack.push(v)
```

**Nothing to push, back up by popping E**

## Stack

E

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




Order nodes visited

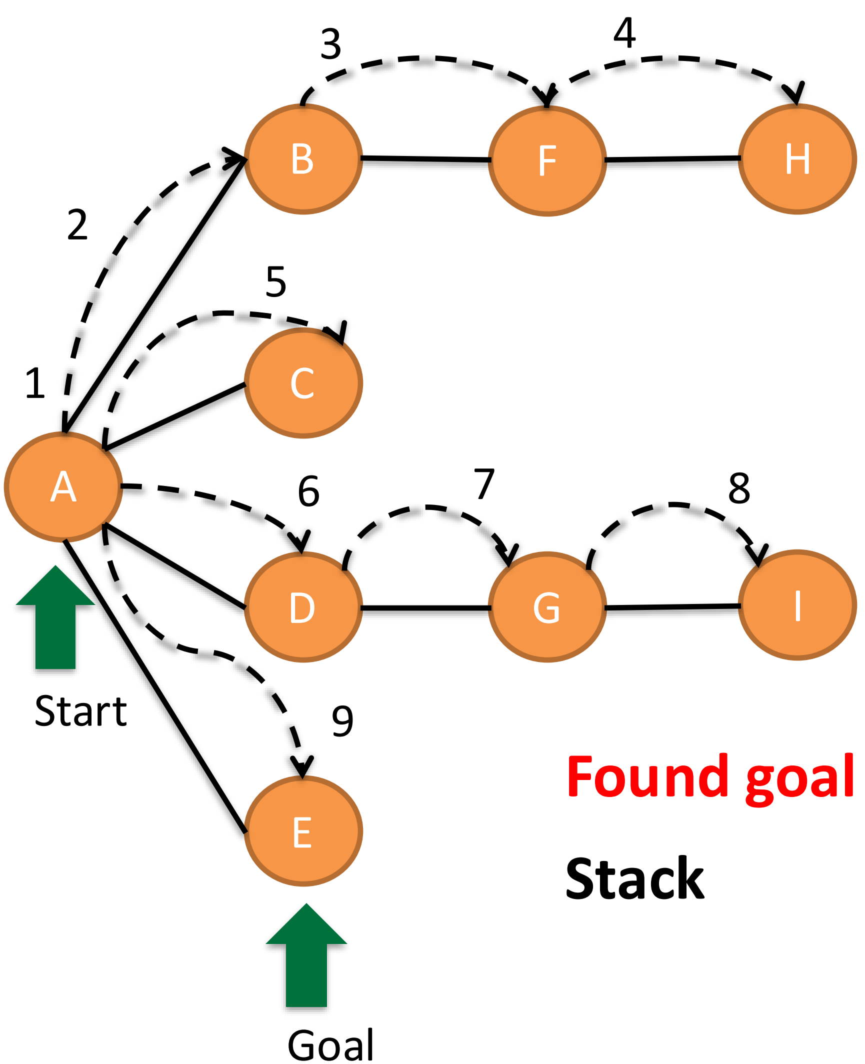
**DFS algorithm**

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

**Pop(E), mark visited**

**Stack**

# Depth First Search (DFS) finds *a path* between *start* and other nodes (if exists)




**Found goal**

**Stack**

## DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or
stack empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

# After DFS, we can find a path from the *start* node to all other nodes in the Graph

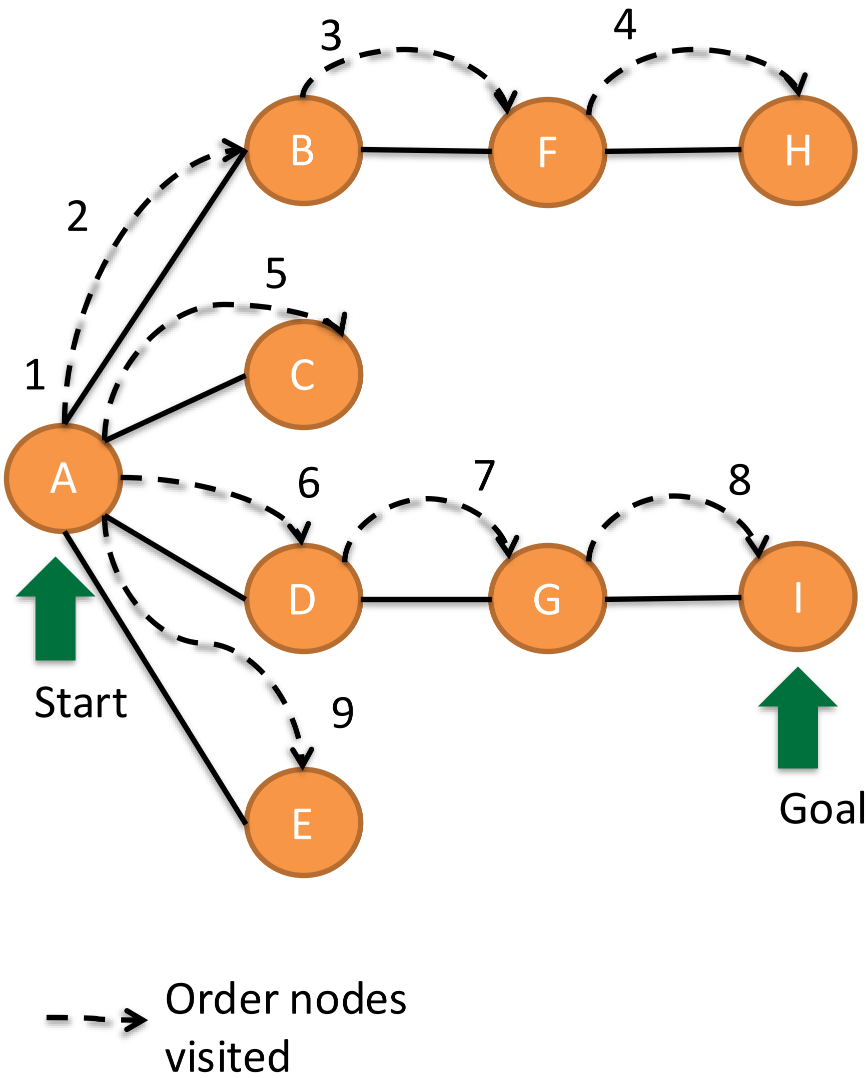
**Discovery edges**

- Edges that lead to unvisited nodes called *discovery edges*
- Discovery edges form a tree on the graph (root, no cycles)
- Can traverse from *start* to *goal* on tree (if *goal* reachable)
- Can tell us which nodes are not reachable from *start* (not on path formed by discovery edges)
- **With DFS, path <u>not</u> guaranteed to be shortest path!**

**Back, cross, and forward edges**

- Edges that lead to previously discovered nodes
- Back edges lead to ancestor nodes, forward edges to descendants, cross edges to non-ancestor or descendant
- Back edges indicate presence of a cycle in the Graph
- Today's focus on graphs without cycles

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered"



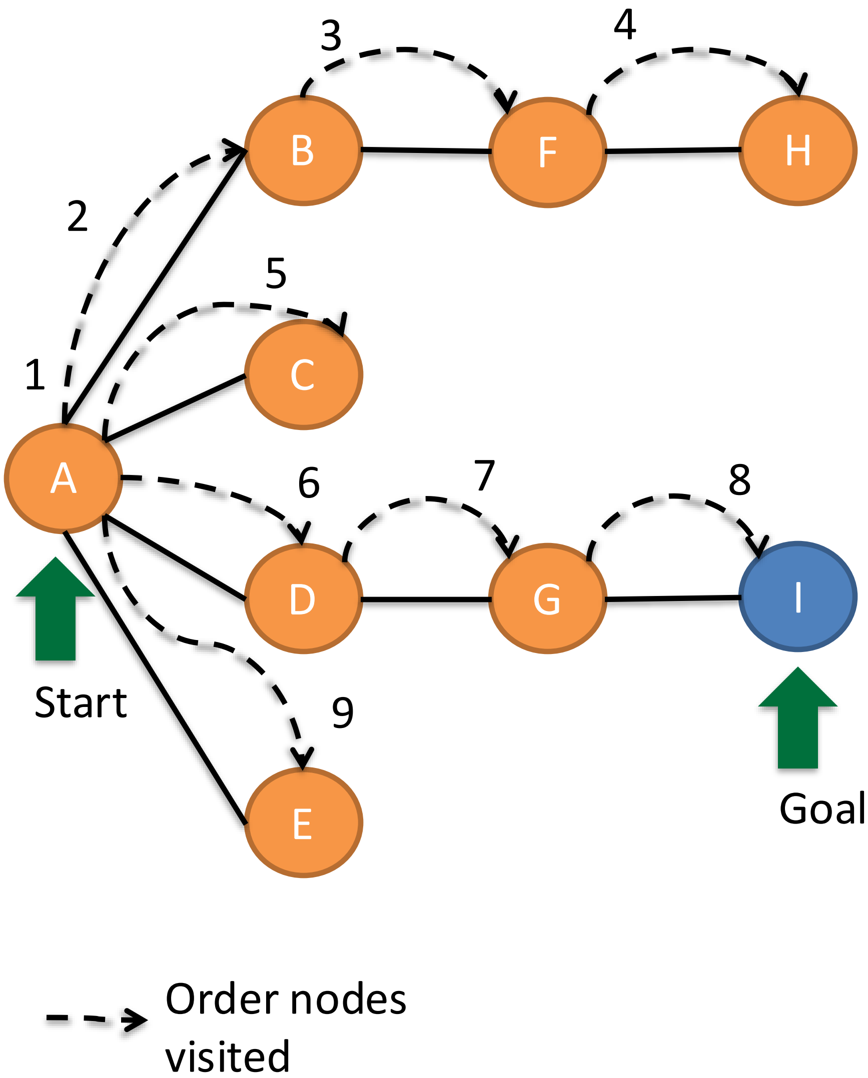

**Path from *start* to *goal***

1. Do DFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After DFS complete, find path using Map
   - Begin at *goal* node
   - Track <u>*backward*</u> on Map until find *start* node
   - Will find a path if it exists, but not necessarily the shortest path (wait for BFS)

**Path A to I**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| D | A |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered




**Path from *start* to *goal***

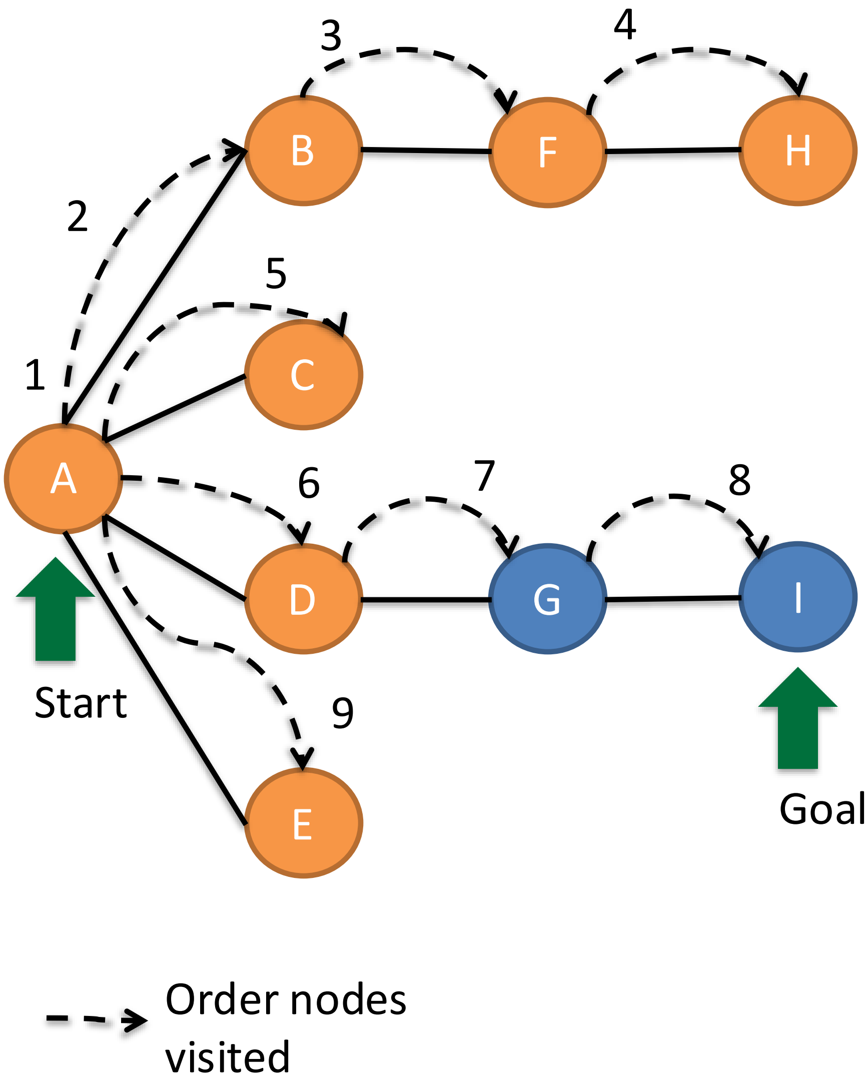
1. Do DFS from *start*
- When node *discovered*, record previous node
- Could keep Map with node as Key and previous as Value
2. After DFS complete, find path using Map
- Begin at *goal* node
- Track <u>*backward*</u> on Map until find *start* node
- Will find a path if it exists, but not necessarily the shortest path (wait for BFS)

**Path A to I**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| D | A |
| E | A |
| F | B |
| G | D |
| H | F |
| **I** | **G** |

**Path**
**G,I**

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered




**Path from *start* to *goal***

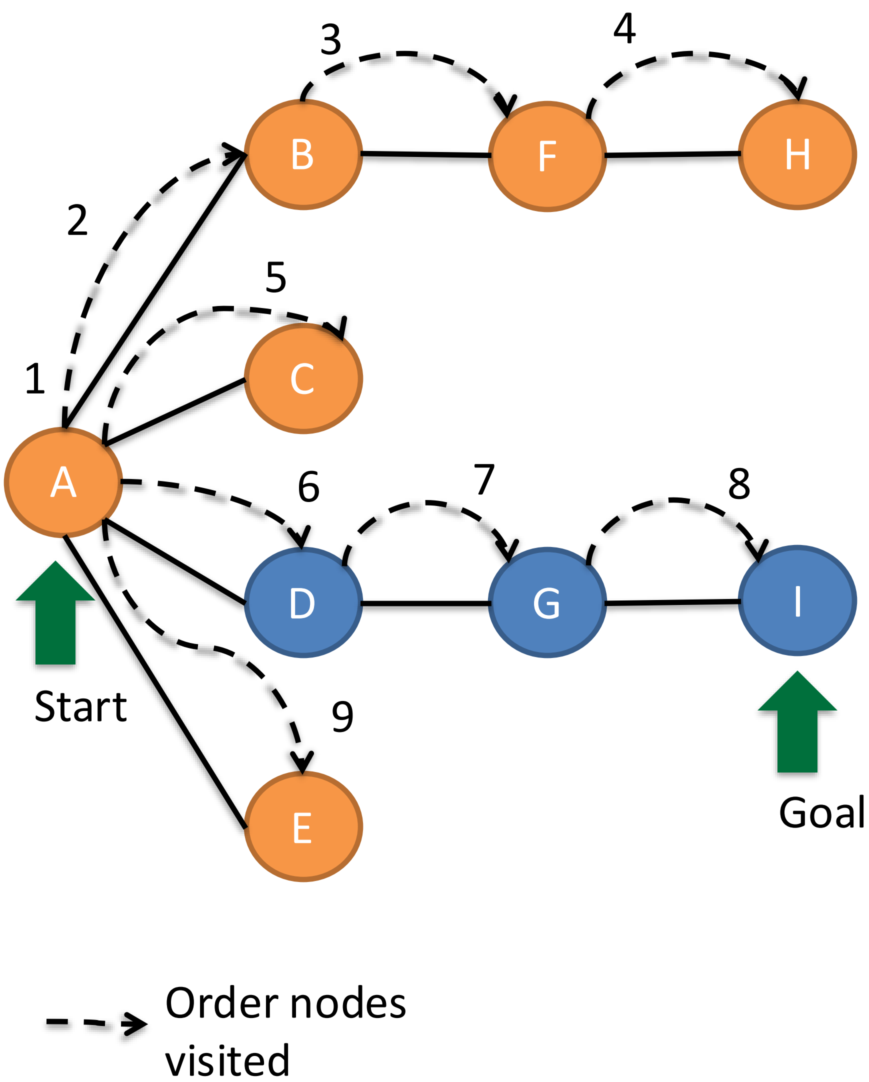
1. Do DFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After DFS complete, find path using Map
   - Begin at *goal* node
   - Track ***backward*** on Map until find *start* node
   - Will find a path if it exists, but not necessarily the shortest path (wait for BFS)

**Path A to I**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| D | A |
| E | A |
| F | B |
| **G** | **D** |
| H | F |
| I | G |

**Path D,G,I**

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered




**Path from *start* to *goal***

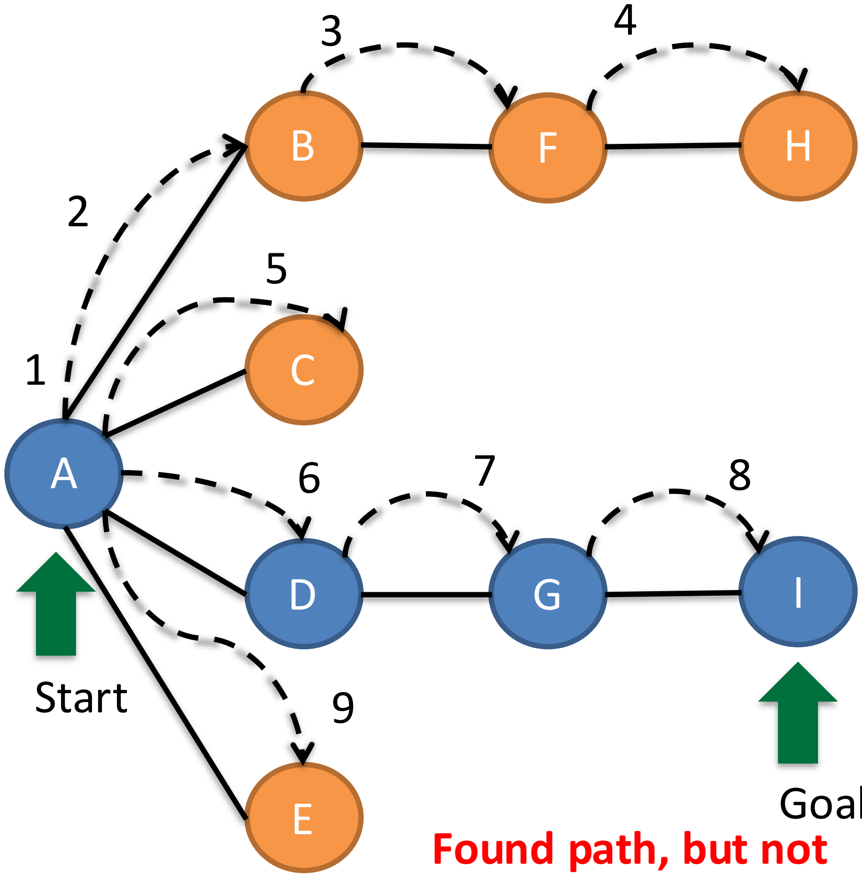
1. Do DFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After DFS complete, find path using Map
   - Begin at *goal* node
   - Track <u>*backward*</u> on Map until find *start* node
   - Will find a path if it exists, but not necessarily the shortest path (wait for BFS)

**Path A to I**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| **D** | **A** |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

**Path A,D,G,I**

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered




**Found path, but not necessarily shortest path**

Order nodes visited

**After DFS from *start* can find *a* path from *start* to any other reachable node**

**Path from *start* to *goal***

1. Do DFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After DFS complete, find path using Map
   - Begin at *goal* node
   - Track ***backward*** on Map until find *start* node
   - Will find a path if it exists, but not necessarily the shortest path (wait for BFS)

**Path A to I**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| **D** | **A** |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

**Path A,D,G,I**



**Could we start from node other than A? No!**

# GraphTraversal.java: DFS code

```java
public class GraphTraversal<V,E> {

    public Map<V,V> backTrack; //keep track of prior vertex when v

    /**
     * Constructor.  Initialize backTrack to new HashMap.
     */
    public GraphTraversal() {
        backTrack = new HashMap<V,V>();
    }

    /**
     * Depth First Search
     * @param G -- graph to search
     * @param start -- starting vertex
     */
    public void DFS(AdjacencyMapGraph<V,E> G, V start) {
        System.out.println("\nDepth First Search from " + start);
        backTrack = new HashMap<V,V>(); //initialize backTrack
        backTrack.put(start, null); //load start node with null pa
        Set<V> visited = new HashSet<V>(); //Set to track which ve
        Stack<V> stack = new Stack<V>(); //stack to implement DFS

        stack.push(start); //push start vertex
        while (!stack.isEmpty()) { //loop until no more vertices
            V u = stack.pop(); //get most recent vertex
            if (!visited.contains(u)) { //if not already visited
                visited.add(u); //add to visited Set
                for (V v : G.outNeighbors(u)) { //loop over out ne
                    if (!visited.contains(v)) { //if neighbor not
                        stack.push(v); //push non-visted neighbor
                        backTrack.put(v, u); //save that this vert
                    }
                }
            }
        }
    }
```

- When running DFS (or BFS), keep track of prior vertex when a vertex is discovered
- Map Key is current vertex, Value is prior vertex

DFS – given Graph *G* and *start* vertex

- Use *Set* to track visited vertices
- Use *Stack* to track vertices to visit
- Follow pseudo code from previous slides
- Add vertex to *backTrack* when discovered
- Only discovered vertices are added, non-reachable vertices not added to *backTrack*

After *DFS* can get from *start* to any reachable node in Graph using *backTrack*

# DFS run time is O(n+m)

**Run time**

- Assume graph with *n* nodes and *m* edges
- Visit each node at most one time due to visited indicator
- Examine each edge at most one time
- Run-time complexity is O(n+m)

# After DFS (or BFS) *findPath()* finds a path from start to end if it exists

**GraphTraversal.java**

**Make sure DFS or BFS has been previously run from *start* vertex**

```java
public List<V> findPath(V start, V end) {
    //check that DFS or BFS have already been run from start
    if (backTrack.isEmpty() || !backTrack.containsKey(start) ||
            (backTrack.containsKey(start) && backTrack.get(start) != null)) {
        System.out.println("Run DFS or BFS on " + start + " before trying to find a path");
        return new ArrayList<V>();
    }
    System.out.println("Path from " + start + " to " + end);
    //make sure end vertex in backTrack
    if (!backTrack.containsKey(end)) {
        System.out.println("\tNo path found");
        return new ArrayList<V>();
    }
    //start from end vertex and work backward to start vertex
    List<V> path = new LinkedList<V>(); //this will hold the path from start to end vertex
    V current = end; //start at end vertex
    //loop from end vertex back to start vertex
    while (current != null) {
        path.add( index: 0,current); //add this vertex to front of arraylist path
        current = backTrack.get(current); //get vertex that discovered this vertex
    }
    System.out.println(path);
    return path;
}
```

**Make sure *end* vertex reachable**

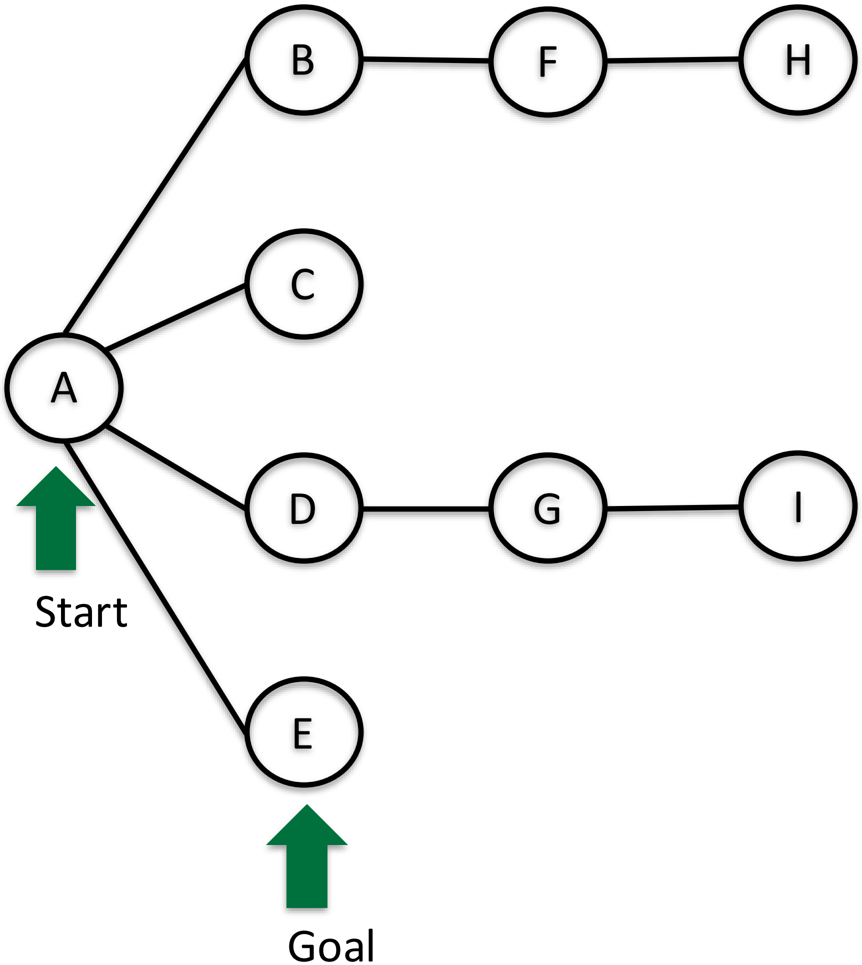
- **Loop backward from *end* to *start* (parent null)**
- **Add new vertices to front of *path***
- **Return *path* when done**

- **Run time performance?**
- **Dependent on length of path from *start* to *end***

# Agenda

1. Depth first search (DFS)
2. Breadth first search (BFS)
3. Examples from last class and today

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**Goal: compute path from *start* to *goal (or to all other nodes)***

**BFS basic idea**

- Explore outward in "ripples"
- Look at all nodes 1 step away, then all nodes 2 steps away…
- Relies on a **Queue** (implicit or explicit) implementation
- Path from `s` to any other vertex is **shortest**

# Some of you did Breadth First Search on Problem Set 1

**RegionFinder**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

# Some of you did Breadth First Search on Problem Set 1

**RegionFinder**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

If you added to end of list…

# Some of you did Breadth First Search on Problem Set 1

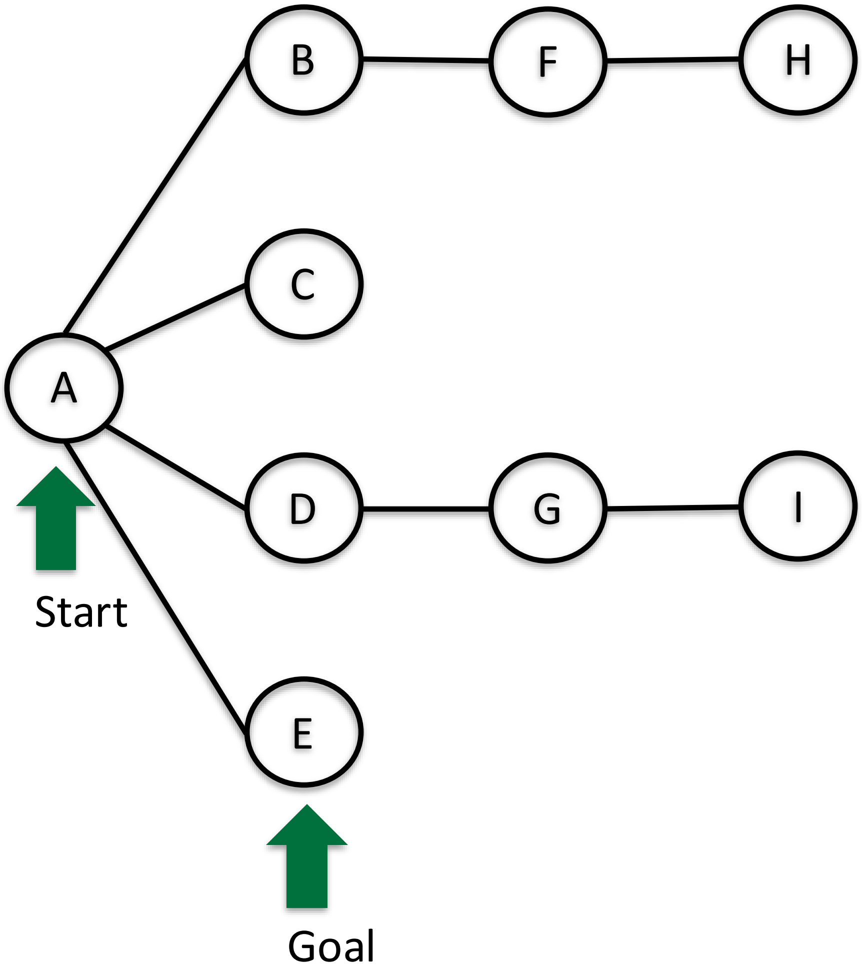
**RegionFinder**

```
Loop over all the pixels
    If a pixel is unvisited and of the correct color
        Start a new region
        Keep track of pixels need to be visited, initially just one
        As long as there's some pixel that needs to be visited
            Get one to visit
            Add it to the region
            Mark it as visited
            Loop over all its neighbors
                If the neighbor is of the correct color
                    Add it to the list of pixels to be visited
        If the region is big enough to be worth keeping, do so
```

And if you get a pixel from front of list, you implemented a Queue

If you added to end of list…

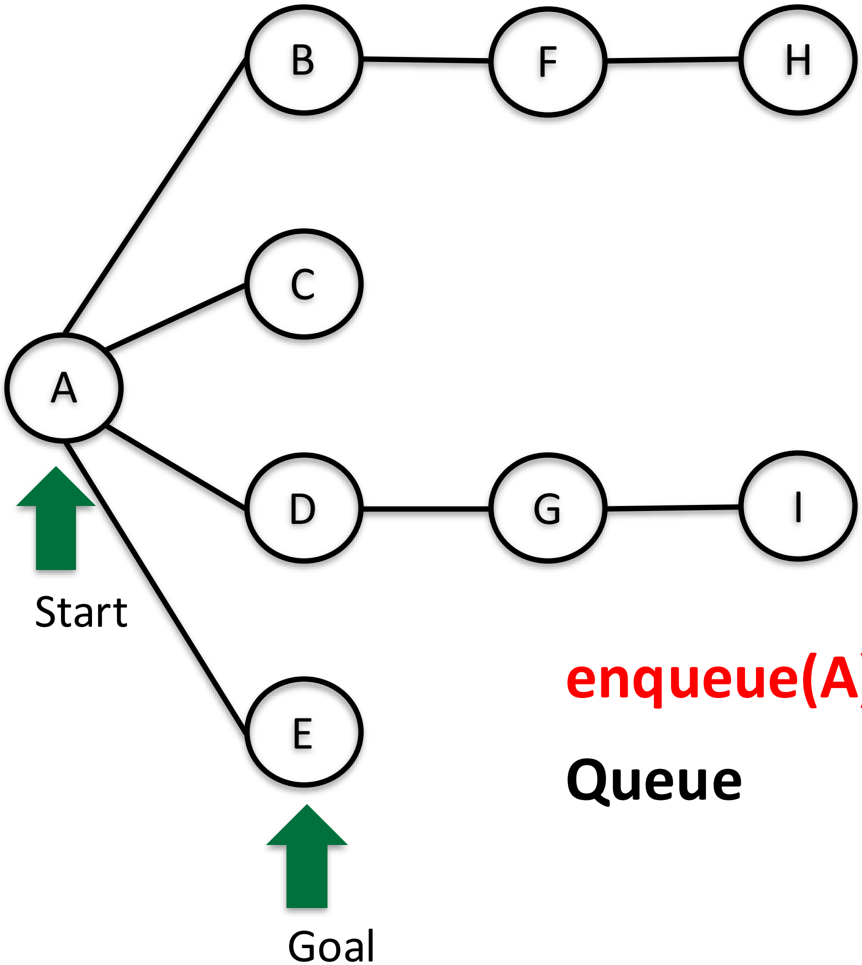
# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes



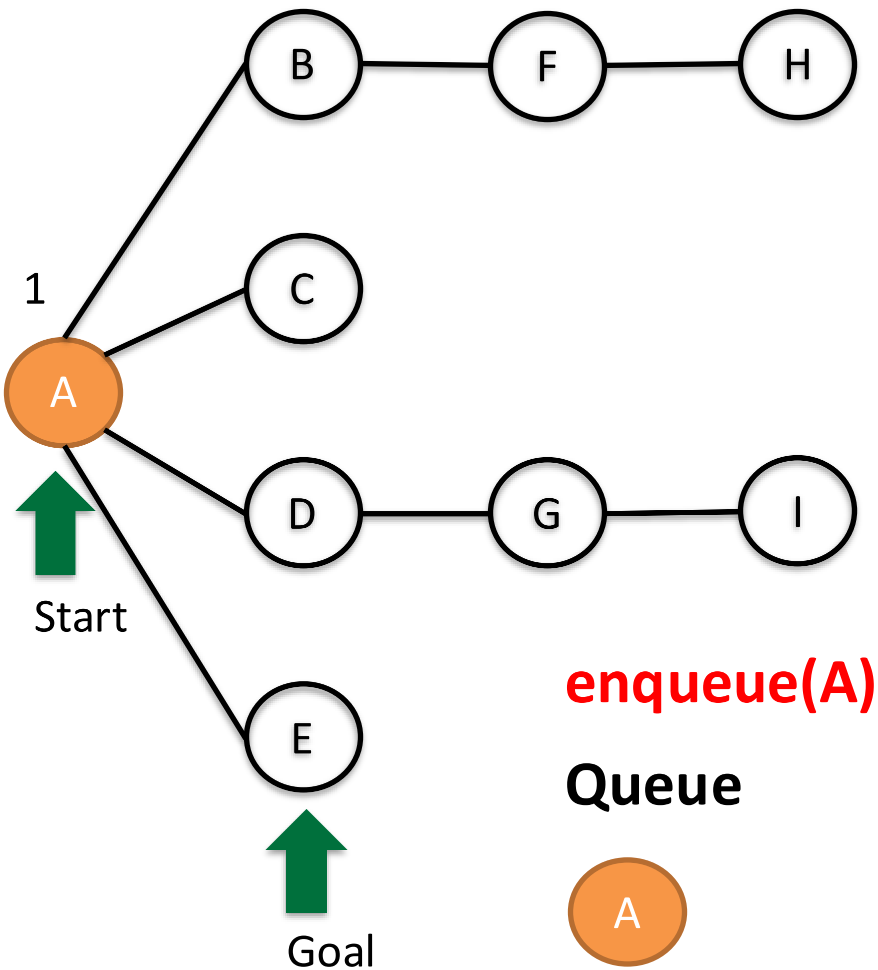

**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
     u = dequeue()
     (do something here)
     for v ∈ u.adjacent
          if !v.visited
               v.visited = true
               enqueue(v)
```

**enqueue(A)**

**Queue**

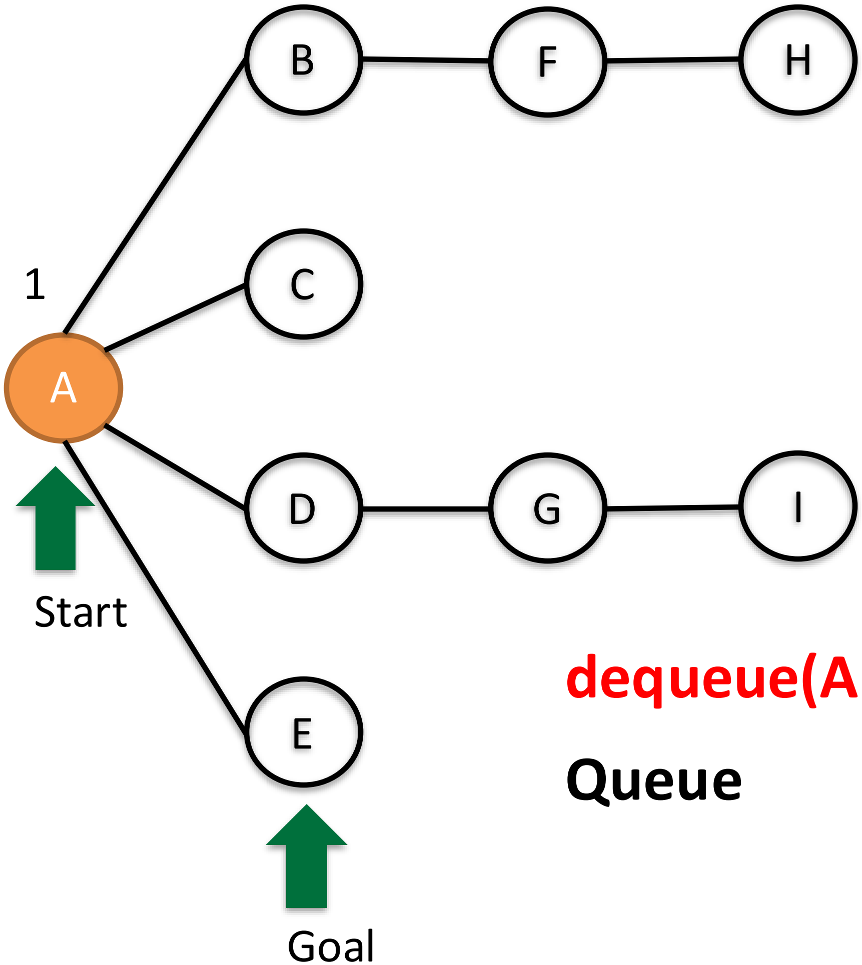
# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**"Visit" node *u***

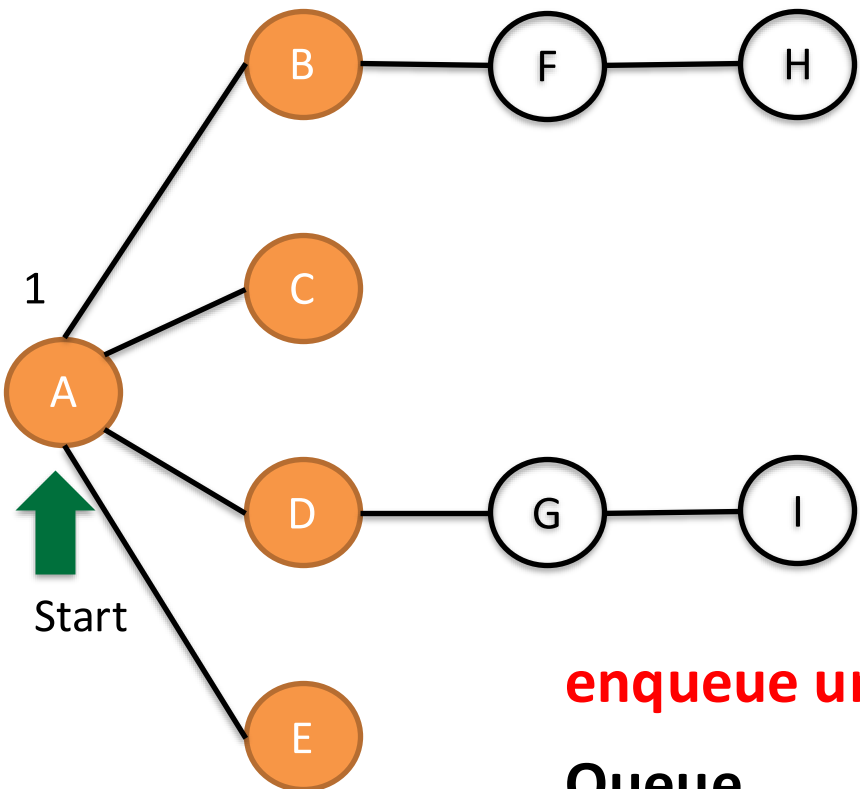
## BFS algorithm

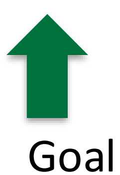
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(A), visit A**

**Queue**

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**"Visit" node *u***

**BFS algorithm**

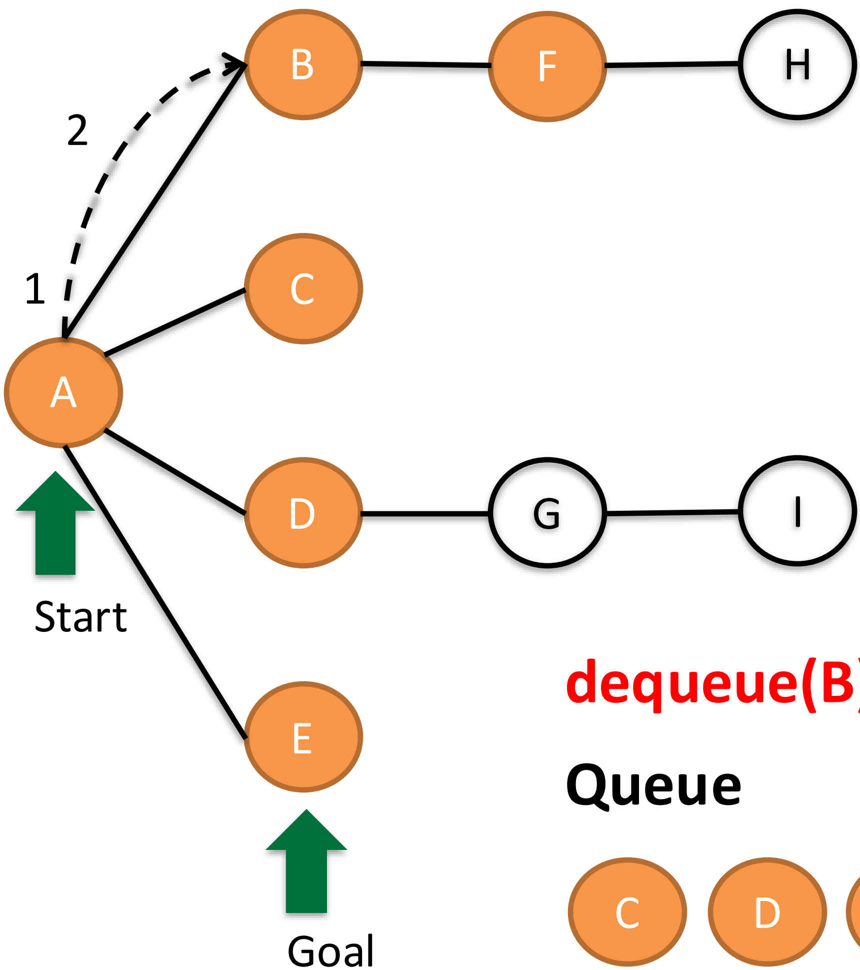
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**enqueue unvisited adjacent**

**Queue**

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited




## BFS algorithm

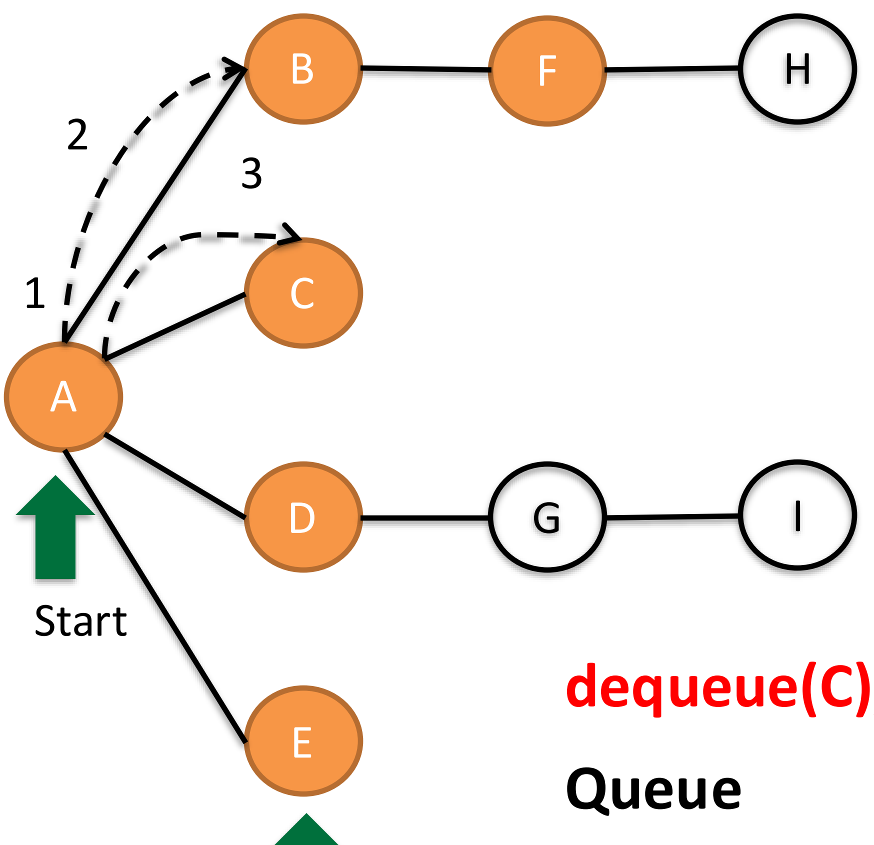
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(B), enqueue unvisited adjacent F (not A)**

## Queue

C D E F

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

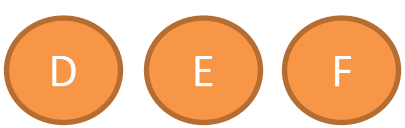



**BFS algorithm**

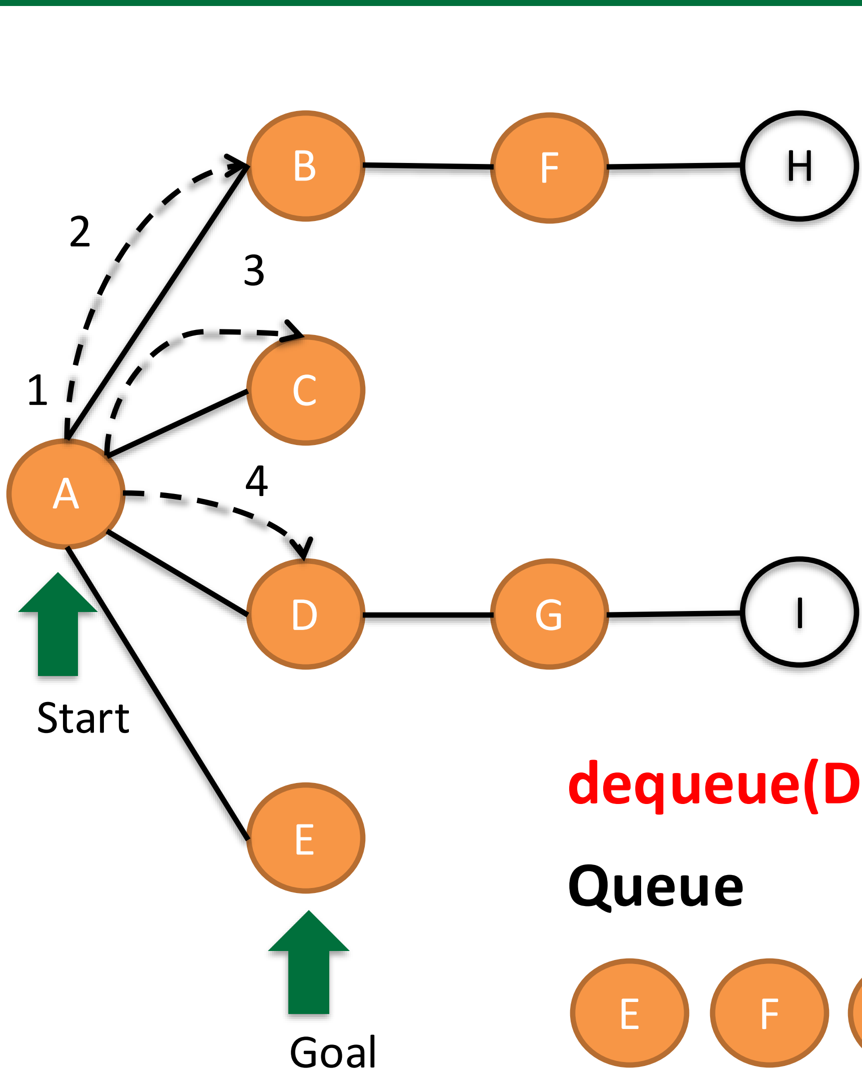
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(C), enqueue unvisited adjacent (none)**

**Queue**



# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

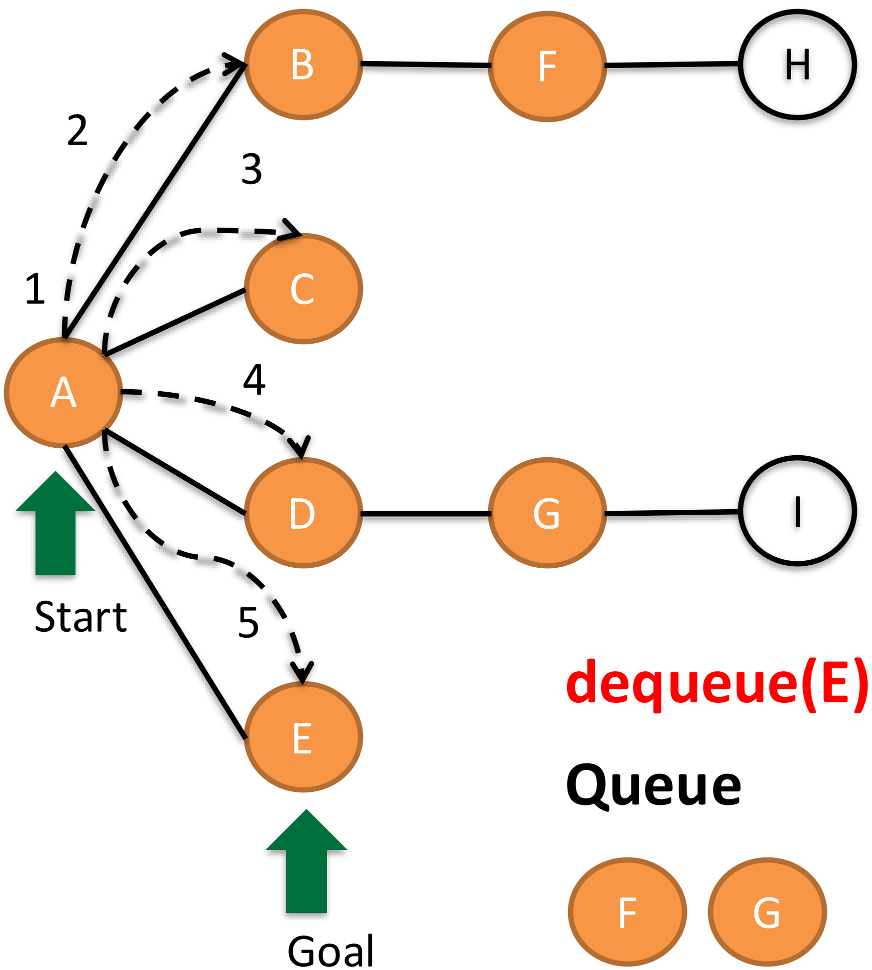
**dequeue(D), enqueue unvisited adjacent G (not A)**

**Queue**

E F G

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited




**BFS algorithm**

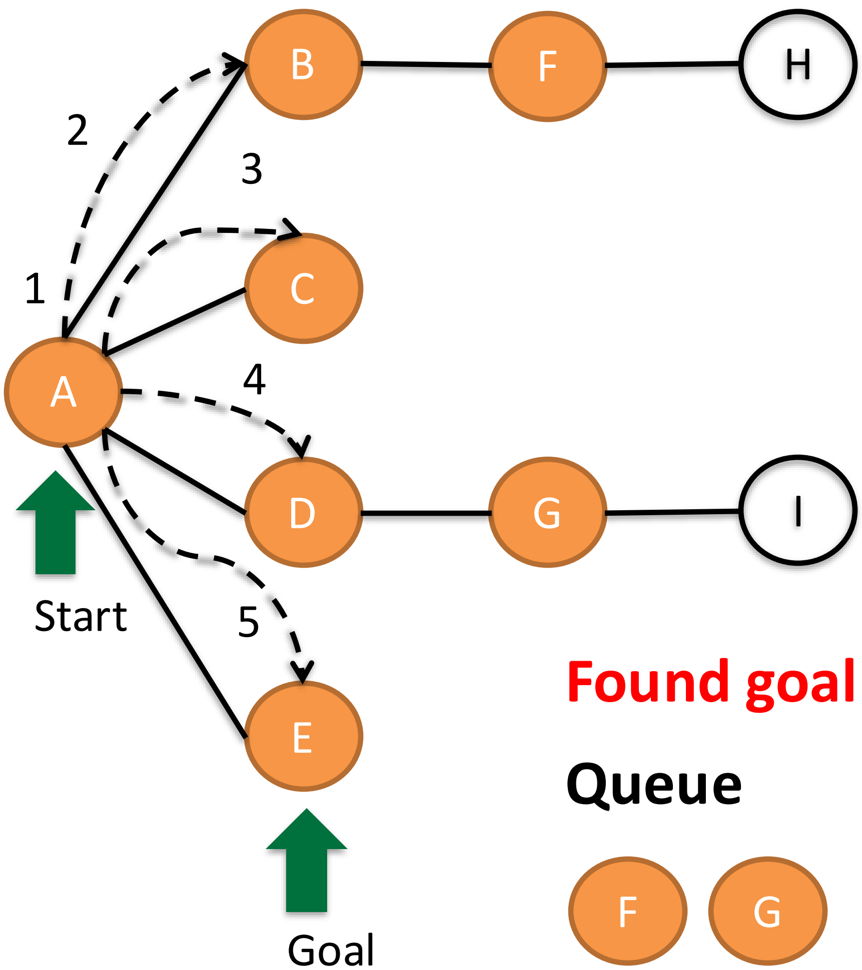
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(E), enqueue unvisited adjacent (none)**

**Queue**

F G

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes


**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
     u = dequeue()
     (do something here)
     for v ∈ u.adjacent
          if !v.visited
               v.visited = true
               enqueue(v)
```

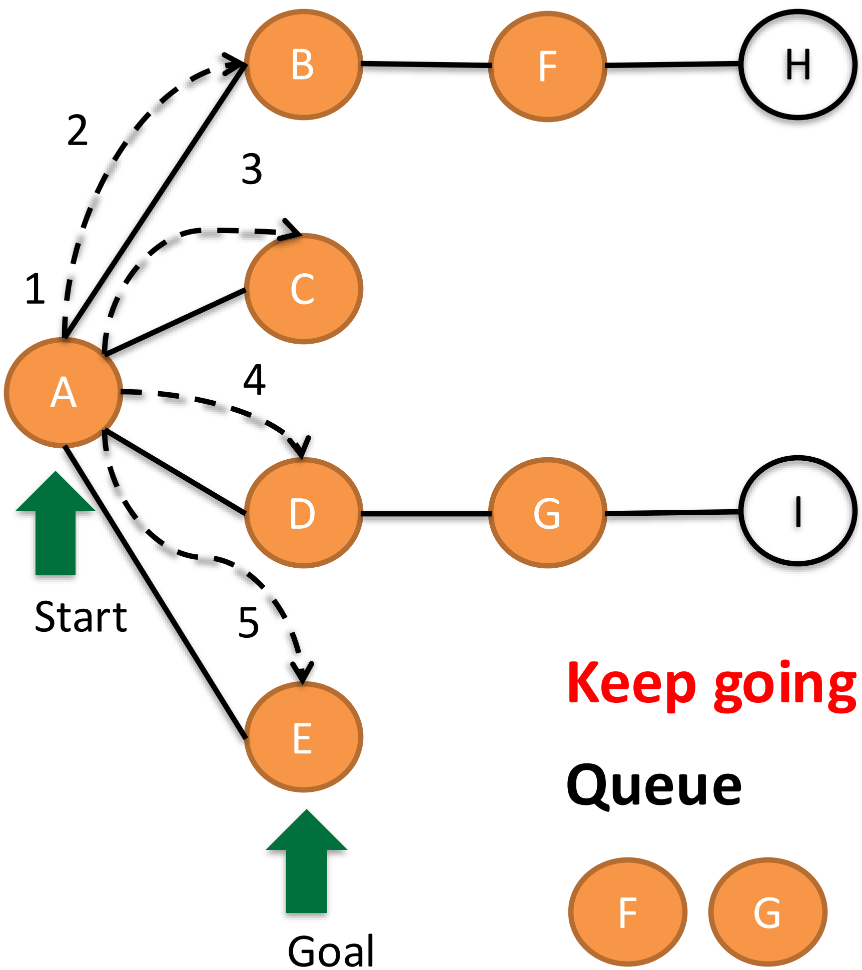
**Found goal! Can stop now**

**Queue**

F G

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited




**BFS algorithm**

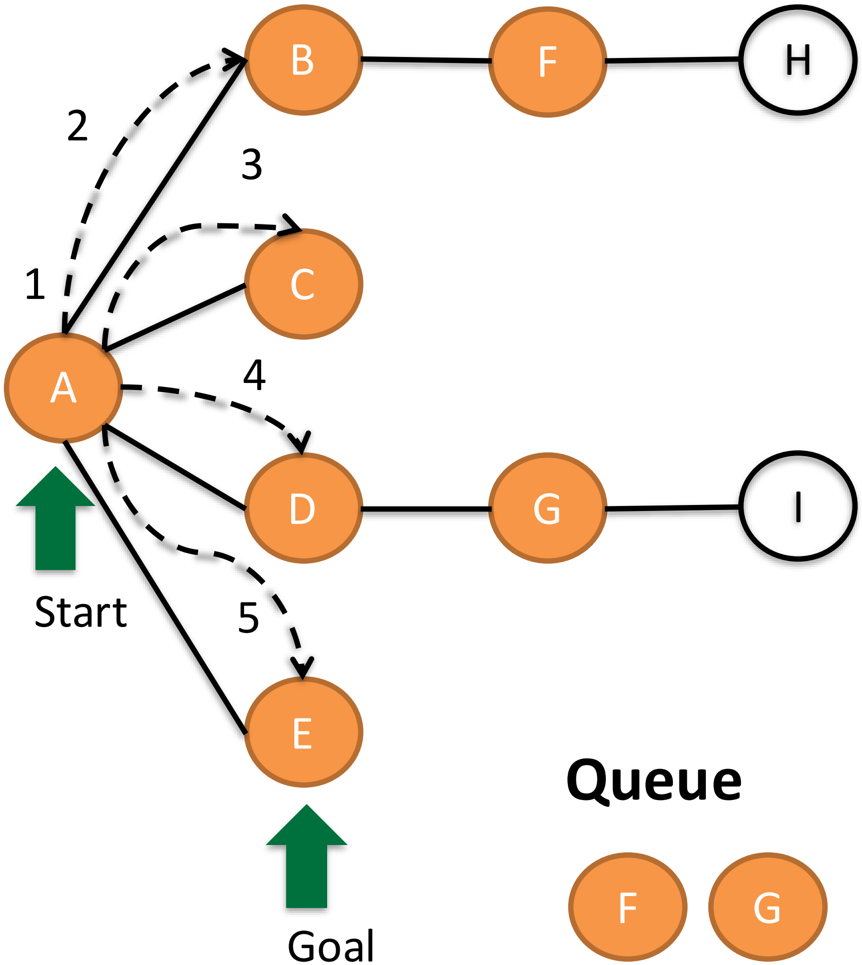
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**Keep going to further illustrate BFS process**

**Queue**

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited



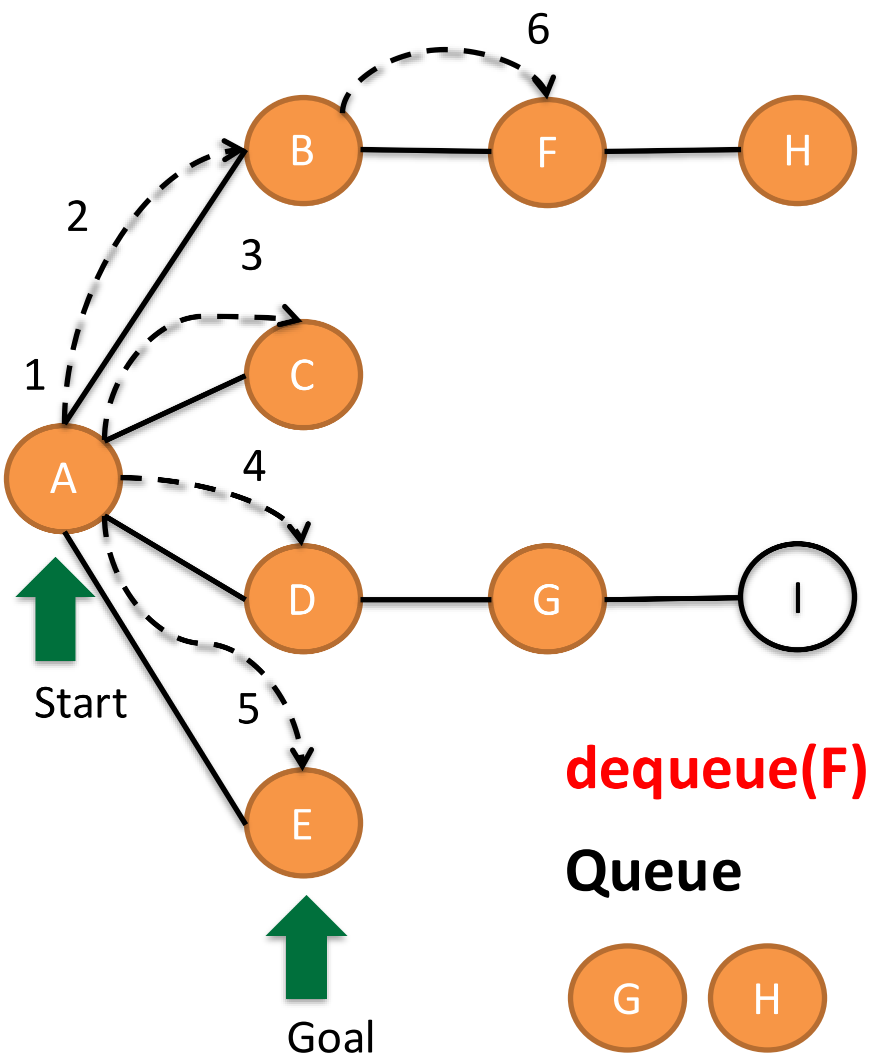

**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**Queue**

**Notice that *all* nodes 1 step away are visited before *any* node 2 steps away is visited**

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




Order nodes visited

## BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

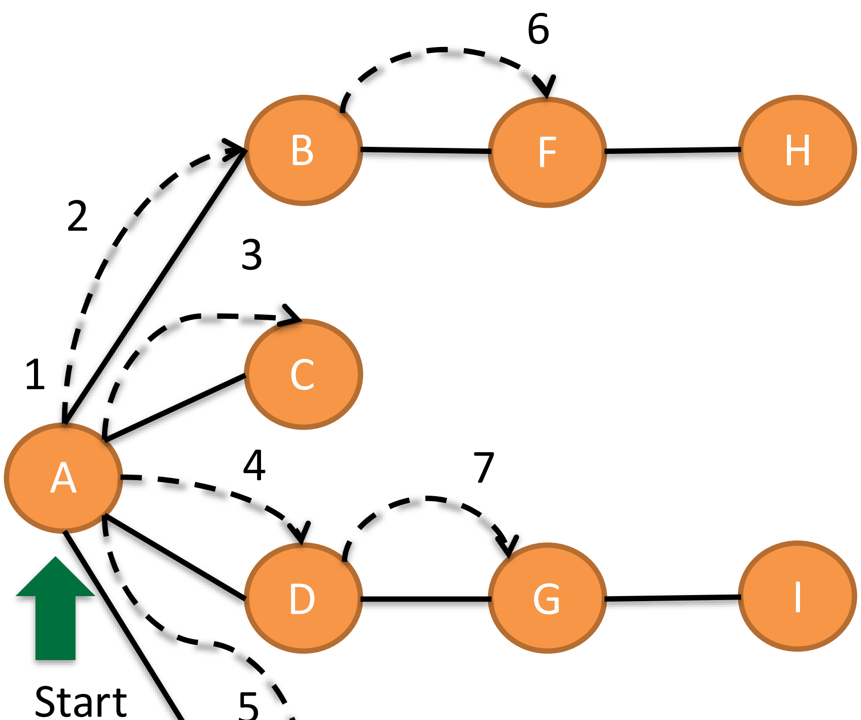
**dequeue(F), enqueue unvisited adjacent H**

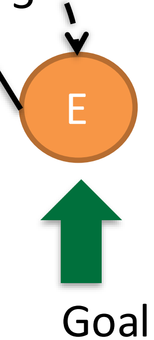
**Queue**

G H

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited



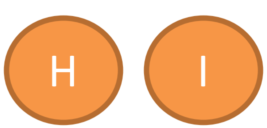




**BFS algorithm**

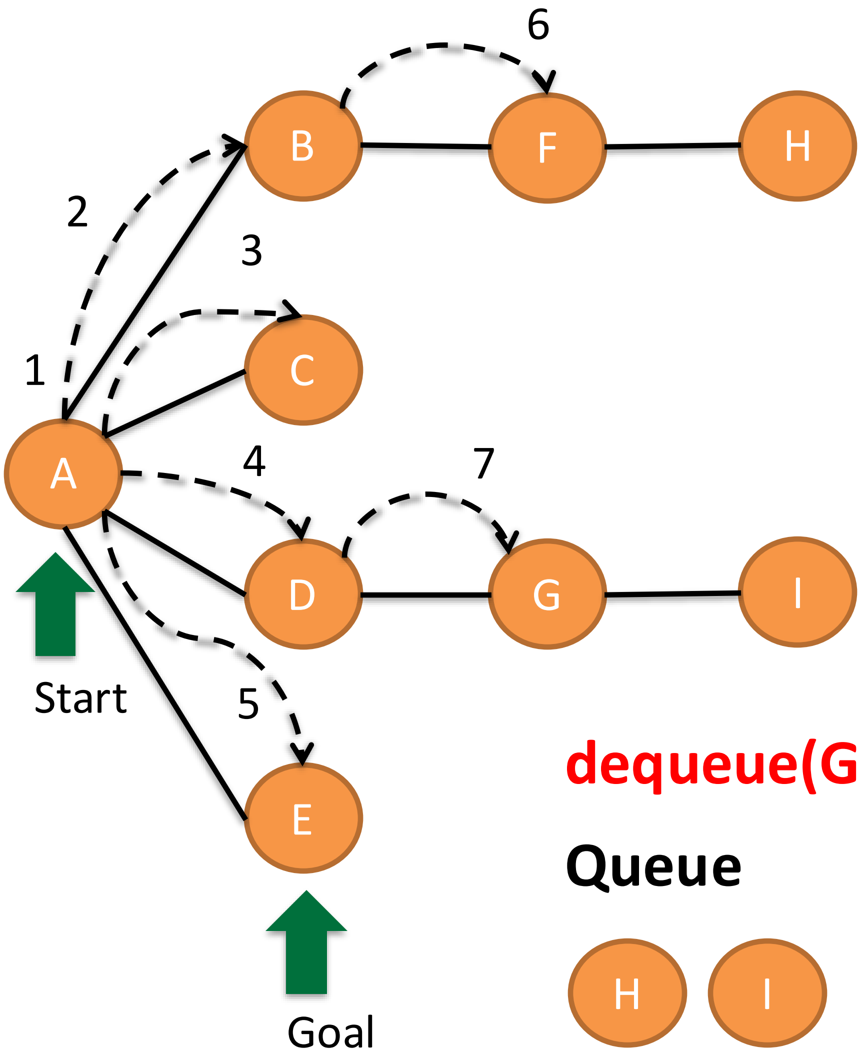
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(G), enqueue unvisited adjacent I**

**Queue**



# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




Order nodes visited

**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeque()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(G), enqueue unvisited adjacent I**

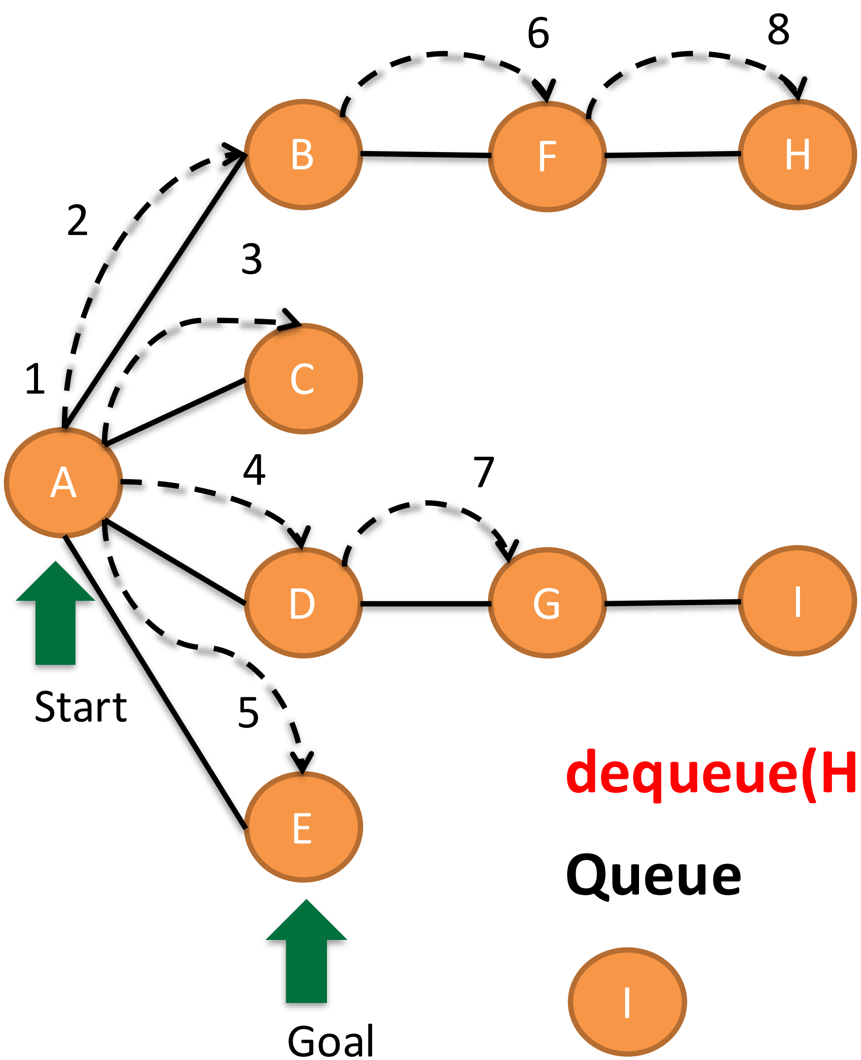
**Queue**

H I

**Notice that *all* nodes 2 steps away are visited before *any* node 3 steps away is visited**

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes

Order nodes visited




## BFS algorithm

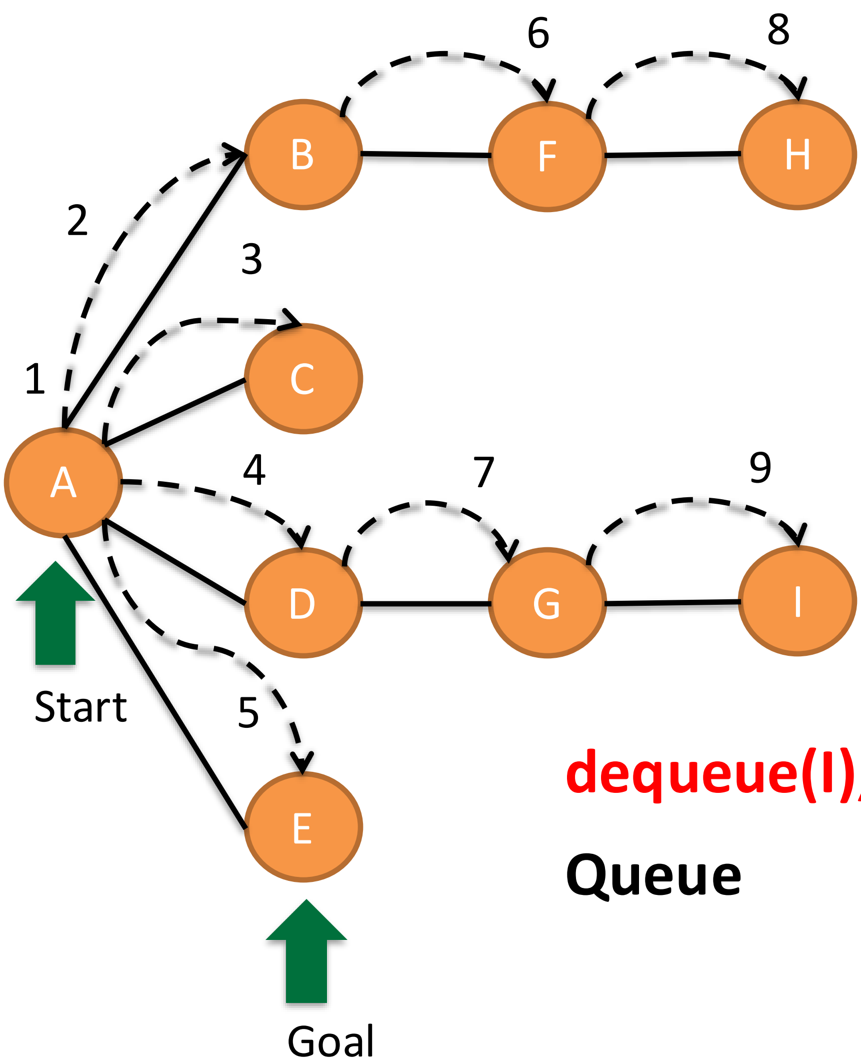
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(H), enqueue unvisited adjacent (none)**

**Queue**

I

# Breadth First Search (BFS) finds *shortest* path between *start* and other nodes




Order nodes visited

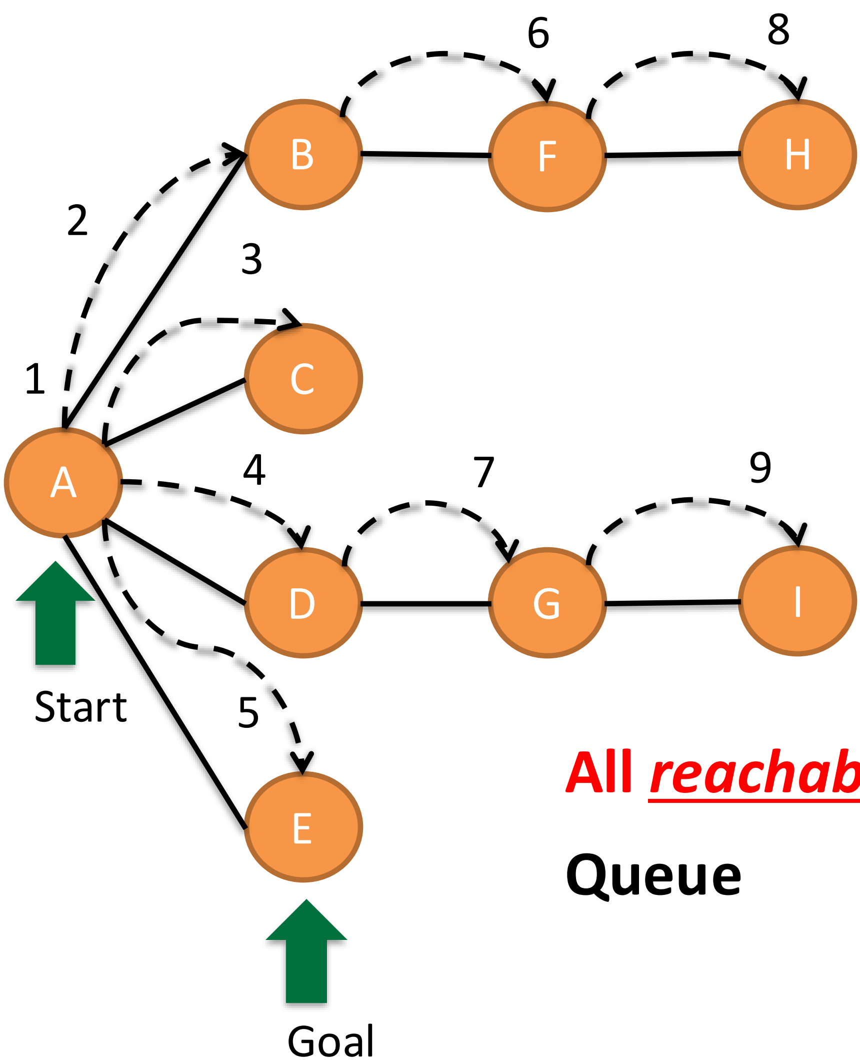
**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**dequeue(I), enqueue unvisited adjacent (none)**

**Queue**

# Breadth First Search (BFS) finds <u>*shortest*</u> path between *start* and other nodes




Order nodes visited

**BFS algorithm**

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    (do something here)
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

**All *<u>reachable</u>* nodes from *start* explored**

**Queue**

# Node discovery tells us something about the graph

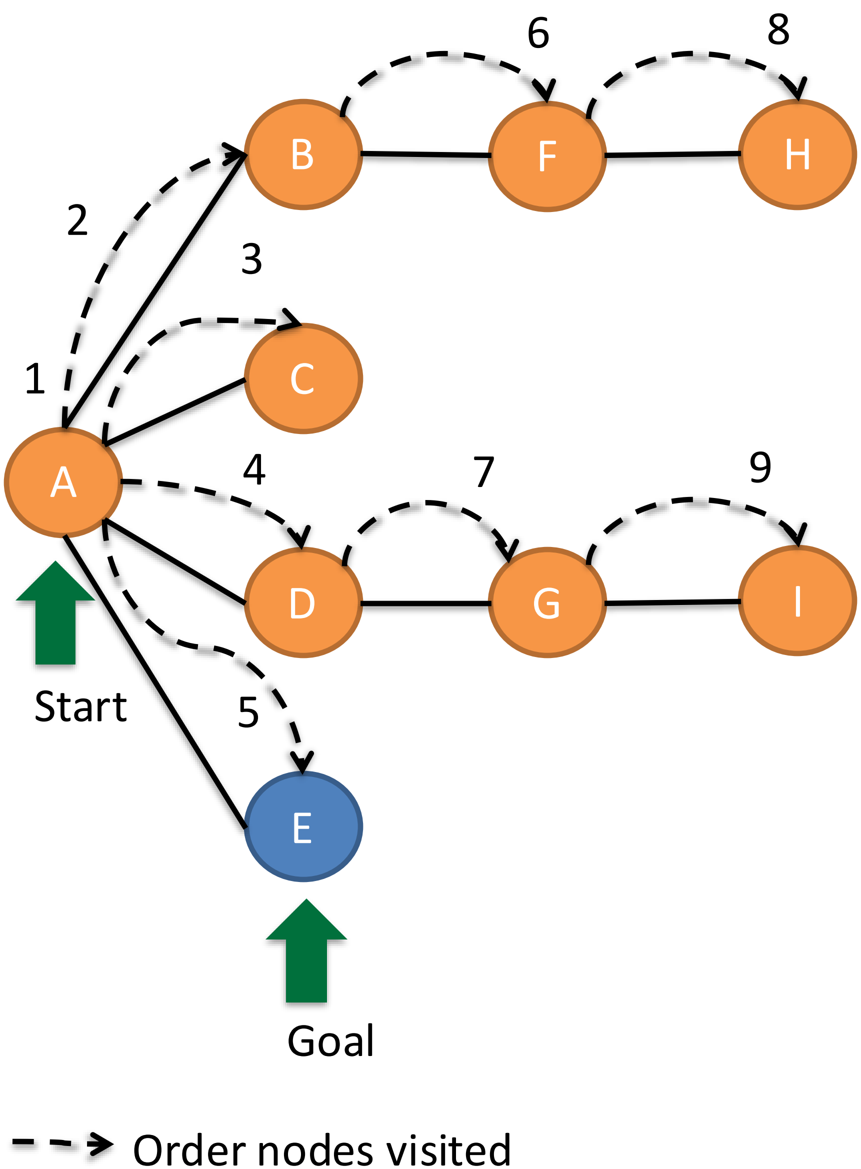
**Discovery edges**

- Edges that lead to unvisited nodes called *discovery edges*
- Discovery edges form a tree on the graph (root, no cycles)
- Can traverse from *start* to *goal* on tree (if goal reachable)
- Can tell us which nodes are not reachable (not on path formed by discovery edges)
- **Path guaranteed to have smallest number of edges**

**Can track how we got to node to find shortest path**

- Keep track of parent vertex
- Parent of each vertex is vertex that discovered it
- Parent is unique because we don't visit vertices twice

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered"



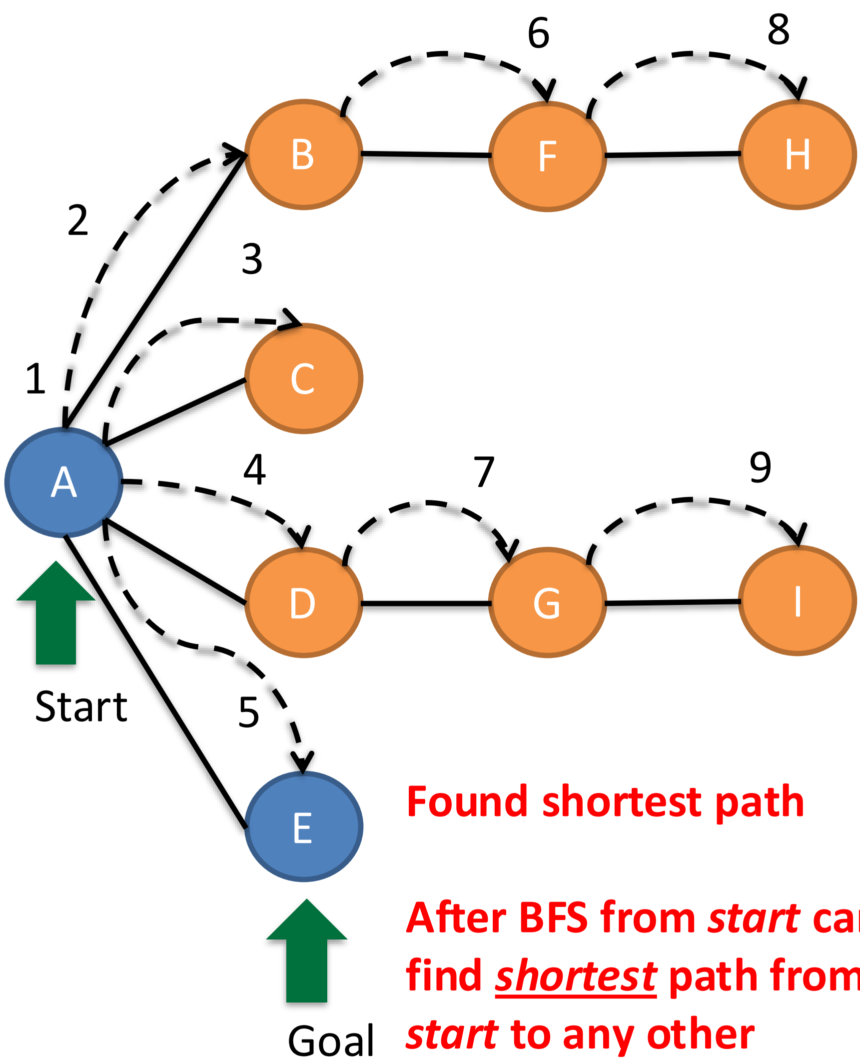

**Path from *start* to *goal***

1. Do BFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After BFS complete, find path on Map
   - Begin at *goal* node
   - Track <u>*backward*</u> on Map until find *start* node
   - **Will find shortest path if it exists**

**Path A to E**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| D | A |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

# To find path from *start* to *goal*, keep track of previous node as nodes are "discovered"




**Found shortest path**

**After BFS from *start* can find *shortest* path from *start* to any other reachable node**

Order nodes visited

## Path from *start* to *goal*

1. Do BFS from *start*
   - When node *discovered*, record previous node
   - Could keep Map with node as Key and previous as Value
2. After BFS complete, find path on Map
   - Begin at *goal* node
   - Track <u>*backward*</u> on Map until find *start* node
   - **Will find shortest path if it exists**

**Path A to E**

| Key | Value |
|---|---|
| A | Null |
| B | A |
| C | A |
| D | A |
| **E** | **A** |
| F | B |
| G | D |
| H | F |
| I | G |

**Path A,E**

# BFS run-time complexity is O(n+m)

**Run time**

- Assume graph with *n* nodes and *m* edges
- Visit each node at most one time due to visited indicator
- Visit each edge at most one time
- Run-time complexity O(n+m)
- Useful for the Kevin Bacon game (PS-4)!

# GraphTraversal.java: BFS code

**GraphTraversal.java**

- **When running BFS, keep track of prior vertex when a vertex is discovered**
- ***backTrack* Map Key is current vertex, Value is prior (parent) vertex**

```
public void BFS(AdjacencyMapGraph<V,E> G, V start) {
    System.out.println("\nBreadth First Search from " + start);
    backTrack = new HashMap<V,V>(); //initialize backTrack
    backTrack.put(start, null); //load start vertex with null parent
    Set<V> visited = new HashSet<V>(); //Set to track which vertices have already been visited
    Queue<V> queue = new LinkedList<V>(); //queue to implement BFS

    queue.add(start); //enqueue start vertex
    visited.add(start); //add start to visited Set
    while (!queue.isEmpty()) { //loop until no more vertices
        V u = queue.remove(); //dequeue
        for (V v : G.outNeighbors(u)) { //loop over out neighbors
            if (!visited.contains(v)) { //if neighbor not visited, then neighbor is discovered
                visited.add(v); //add neighbor to visited Set
                queue.add(v); //enqueue neighbor
                backTrack.put(v, u); //save that this vertex was discovered from prior vertex
            }
        }
    }
}
```

**BFS – given Graph G and start vertex**

- **Use Set to track visited vertices**
- **Use Queue to track which vertices to visit**
- **Follow pseudo code**
- **Add vertex to *backTrack* when discovered**
- **Use same *findPath()* method**

# Agenda

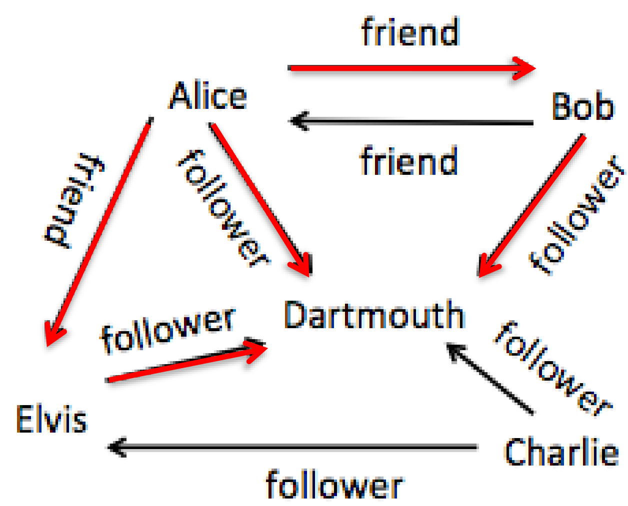
1. Depth first search (DFS)
2. Breadth first search (BFS)
3. Examples from last class and today

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
	public static void main(String[] args) {
		//set up graph from class introducing Graphs
		GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
		AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
		g.insertVertex("Alice");
		g.insertVertex("Bob");
		g.insertVertex("Charlie");
		g.insertVertex("Dartmouth");
		g.insertVertex("Elvis");
		g.insertDirected("Alice", "Dartmouth", "follower");
		g.insertDirected("Bob", "Dartmouth", "follower");
		g.insertDirected("Charlie", "Dartmouth", "follower");
		g.insertDirected("Elvis", "Dartmouth", "follower");
		g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
		g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
		g.insertDirected("Charlie", "Elvis", "follower");

		//run DFS from Alice
		GT.DFS(g,"Alice");
		//find path from start to end
		GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
		GT.findPath("Alice","Dartmouth");
		GT.findPath("Alice","Charlie");
		GT.findPath("Alice","Alice");

		//run BFS
		GT.BFS(g,"Alice");

		//find path from start to end
		GT.findPath("Alice","Dartmouth");
		GT.findPath("Alice","Charlie");
```

**Create graph**

- **Run DFS with start=Alice**
- **Find paths from Alice**




| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Elvis |
| Elvis | Alice |

```
Depth First Search from Alice
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
	No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
	No path found
```

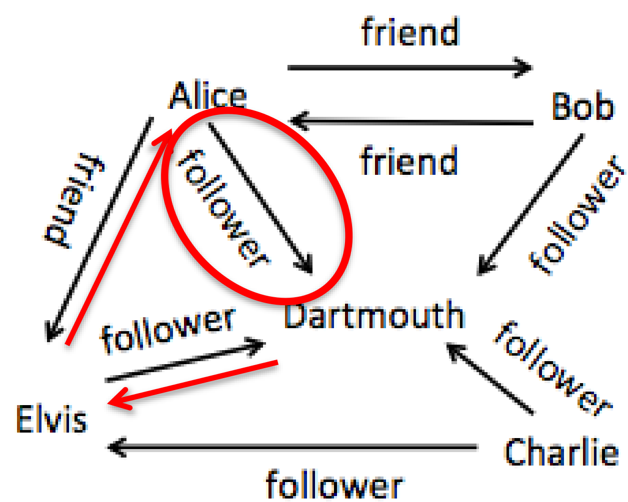
**findPath("Bob", "Dartmouth")**
**DFS not run from Bob**
**DFS was run from Alice**
**So do not do this search (check *findPath* code)**

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
    public static void main(String[] args) {
        //set up graph from class introducing Graphs
        GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
        AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
        g.insertVertex("Alice");
        g.insertVertex("Bob");
        g.insertVertex("Charlie");
        g.insertVertex("Dartmouth");
        g.insertVertex("Elvis");
        g.insertDirected("Alice", "Dartmouth", "follower");
        g.insertDirected("Bob", "Dartmouth", "follower");
        g.insertDirected("Charlie", "Dartmouth", "follower");
        g.insertDirected("Elvis", "Dartmouth", "follower");
        g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
        g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
        g.insertDirected("Charlie", "Elvis", "follower");

        //run DFS from Alice
        GT.DFS(g,"Alice");
        //find path from start to end
        GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
        GT.findPath("Alice","Alice");

        //run BFS
        GT.BFS(g,"Alice");

        //find path from start to end
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
```

**Create graph**

- **Run DFS with start=Alice**
- **Find paths from Alice**




| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Elvis |
| Elvis | Alice |

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

```
<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)
Depth First Search from Alice
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

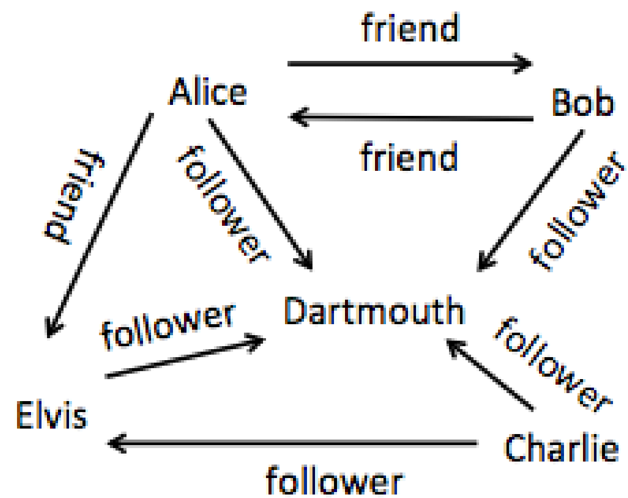
**findPath("Alice", "Dartmouth") finds path**
**Alice->Elvis->Dartmouth**
**Path yes, but not shortest path**
**Shortest is Alice->Dartmouth**

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
public static void main(String[] args) {
    //set up graph from class introducing Graphs
    GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
    AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
    g.insertVertex("Alice");
    g.insertVertex("Bob");
    g.insertVertex("Charlie");
    g.insertVertex("Dartmouth");
    g.insertVertex("Elvis");
    g.insertDirected("Alice", "Dartmouth", "follower");
    g.insertDirected("Bob", "Dartmouth", "follower");
    g.insertDirected("Charlie", "Dartmouth", "follower");
    g.insertDirected("Elvis", "Dartmouth", "follower");
    g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
    g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
    g.insertDirected("Charlie", "Elvis", "follower");

    //run DFS from Alice
    GT.DFS(g,"Alice");
    //find path from start to end
    GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
    GT.findPath("Alice","Dartmouth");
    GT.findPath("Alice","Charlie");
    GT.findPath("Alice","Alice");

    //run BFS
    GT.BFS(g,"Alice");

    //find path from start to end
    GT.findPath("Alice","Dartmouth");
    GT.findPath("Alice","Charlie");
```

**Create graph**

- **Run DFS with start=Alice**
- **Find paths from Alice**




| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Elvis |
| Elvis | Alice |

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)

```
Depth First Search from Alice
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

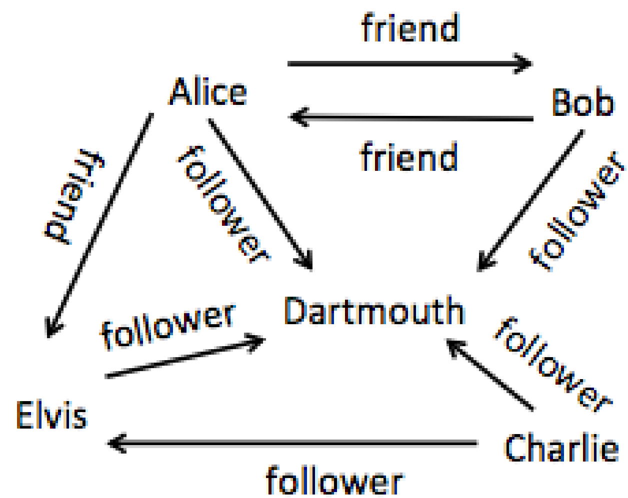
**Alice can't reach Charlie in this graph**
**Charlie is not in *backTrack***

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
    public static void main(String[] args) {
        //set up graph from class introducing Graphs
        GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
        AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
        g.insertVertex("Alice");
        g.insertVertex("Bob");
        g.insertVertex("Charlie");
        g.insertVertex("Dartmouth");
        g.insertVertex("Elvis");
        g.insertDirected("Alice", "Dartmouth", "follower");
        g.insertDirected("Bob", "Dartmouth", "follower");
        g.insertDirected("Charlie", "Dartmouth", "follower");
        g.insertDirected("Elvis", "Dartmouth", "follower");
        g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
        g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
        g.insertDirected("Charlie", "Elvis", "follower");

        //run DFS from Alice
        GT.DFS(g,"Alice");
        //find path from start to end
        GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
        GT.findPath("Alice","Alice");

        //run BFS
        GT.BFS(g,"Alice");

        //find path from start to end
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
```

**Create graph**

- **Run DFS with start=Alice**
- **Find paths from Alice**




| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Elvis |
| Elvis | Alice |

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)

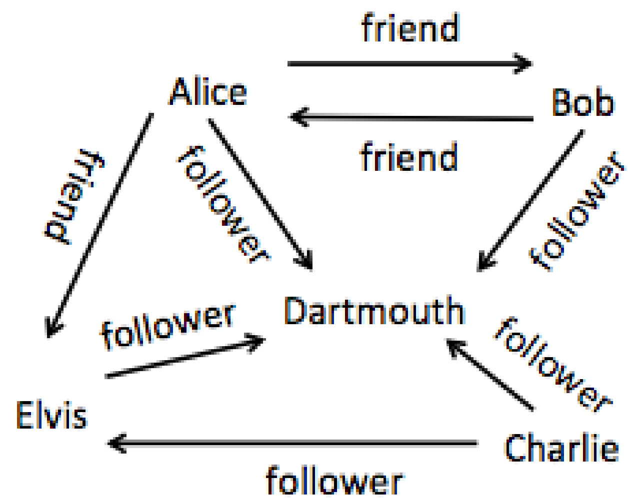
```
Depth First Search from Alice
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

**Alice can reach herself**

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
    public static void main(String[] args) {
        //set up graph from class introducing Graphs
        GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
        AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
        g.insertVertex("Alice");
        g.insertVertex("Bob");
        g.insertVertex("Charlie");
        g.insertVertex("Dartmouth");
        g.insertVertex("Elvis");
        g.insertDirected("Alice", "Dartmouth", "follower");
        g.insertDirected("Bob", "Dartmouth", "follower");
        g.insertDirected("Charlie", "Dartmouth", "follower");
        g.insertDirected("Elvis", "Dartmouth", "follower");
        g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
        g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
        g.insertDirected("Charlie", "Elvis", "follower");

        //run DFS from Alice
        GT.DFS(g,"Alice");
        //find path from start to end
        GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
        GT.findPath("Alice","Alice");

        //run BFS
        GT.BFS(g,"Alice");

        //find path from start to end
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
```

- **Run BFS start=Alice**




Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)

```
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

# GraphTraversal.java: DFS and BFS on graph we looked at last class

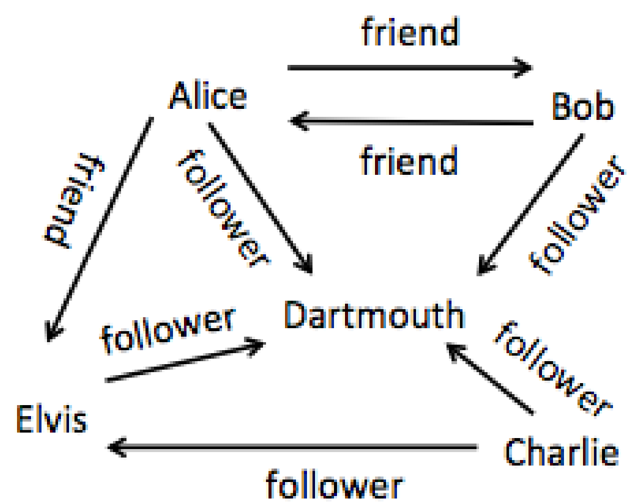
```java
    public static void main(String[] args) {
        //set up graph from class introducing Graphs
        GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
        AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
        g.insertVertex("Alice");
        g.insertVertex("Bob");
        g.insertVertex("Charlie");
        g.insertVertex("Dartmouth");
        g.insertVertex("Elvis");
        g.insertDirected("Alice", "Dartmouth", "follower");
        g.insertDirected("Bob", "Dartmouth", "follower");
        g.insertDirected("Charlie", "Dartmouth", "follower");
        g.insertDirected("Elvis", "Dartmouth", "follower");
        g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
        g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
        g.insertDirected("Charlie", "Elvis", "follower");

        //run DFS from Alice
        GT.DFS(g,"Alice");
        //find path from start to end
        GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
        GT.findPath("Alice","Alice");

        //run BFS
        GT.BFS(g,"Alice");

        //find path from start to end
        GT.findPath("Alice","Dartmouth");
        GT.findPath("Alice","Charlie");
```

- **Run BFS start=Alice**

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)

```
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

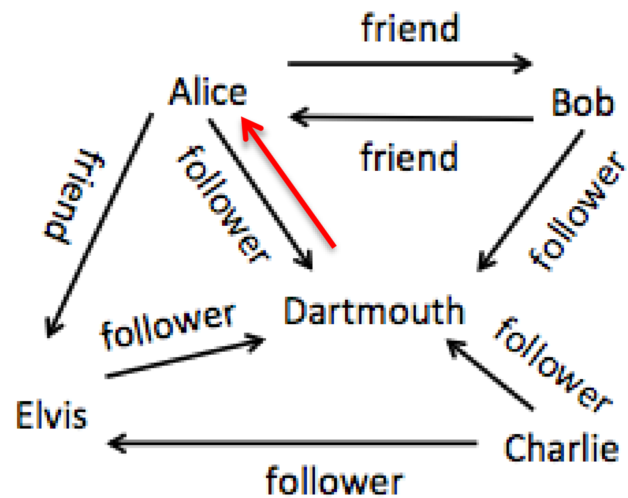



| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Alice |
| Elvis | Alice |

# GraphTraversal.java: DFS and BFS on graph we looked at last class

```java
public static void main(String[] args) {
    //set up graph from class introducing Graphs
    GraphTraversal<String,String> GT = new GraphTraversal<String,String>();
    AdjacencyMapGraph<String,String> g = new AdjacencyMapGraph<String,String>();
    g.insertVertex("Alice");
    g.insertVertex("Bob");
    g.insertVertex("Charlie");
    g.insertVertex("Dartmouth");
    g.insertVertex("Elvis");
    g.insertDirected("Alice", "Dartmouth", "follower");
    g.insertDirected("Bob", "Dartmouth", "follower");
    g.insertDirected("Charlie", "Dartmouth", "follower");
    g.insertDirected("Elvis", "Dartmouth", "follower");
    g.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edge
    g.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed edge!
    g.insertDirected("Charlie", "Elvis", "follower");

    //run DFS from Alice
    GT.DFS(g,"Alice");
    //find path from start to end
    GT.findPath("Bob", "Dartmouth"); //DFS wasn't run from Bob, should reject this
    GT.findPath("Alice","Dartmouth");
    GT.findPath("Alice","Charlie");
    GT.findPath("Alice","Alice");

    //run BFS
    GT.BFS(g,"Alice");

    //find path from start to end
    GT.findPath("Alice","Dartmouth");
    GT.findPath("Alice","Charlie");
```

- **Run BFS start=Alice**
- **Find paths from Alice**




| Key | Value |
|---|---|
| Alice | Null |
| Bob | Alice |
| Dartmouth | Alice |
| Elvis | Alice |

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 4:31:06 PM)

```
Run DFS or BFS on Bob before trying to find a path
Path from Alice to Dartmouth
[Alice, Elvis, Dartmouth]
Path from Alice to Charlie
        No path found
Path from Alice to Alice
[Alice]

Breadth First Search from Alice
Path from Alice to Dartmouth
[Alice, Dartmouth]
Path from Alice to Charlie
        No path found
```

**BFS**

**findPath("Alice", "Dartmouth") finds shortest path Alice->Dartmouth (DFS went through Elvis before Dartmouth)**

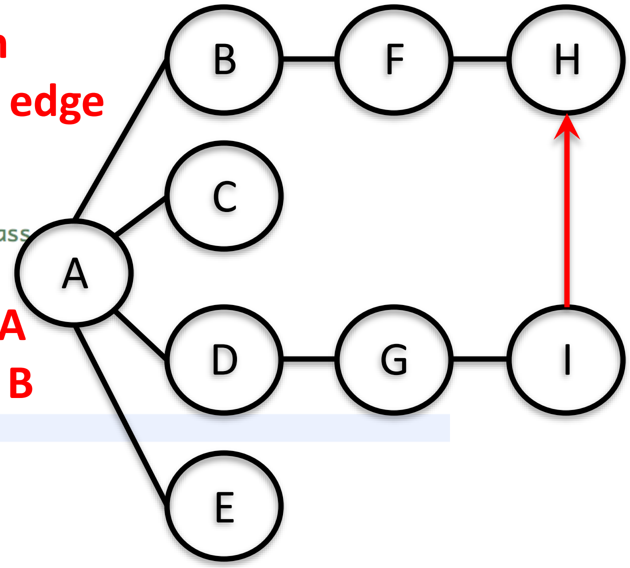
# DFS on today’s graph

**GraphTraversal.java**

```java
        //set up graph from Graph Traversal class
        AdjacencyMapGraph<String,String> g2 = new AdjacencyMapGraph<String,String>();
        g2.insertVertex("A"); g2.insertVertex("B"); g2.insertVertex("C"); g2.insertVertex("D");
        g2.insertVertex("E"); g2.insertVertex("F"); g2.insertVertex("G"); g2.insertVertex("H"); g2.insert
        g2.insertUndirected("A", "B", "");
        g2.insertUndirected("B", "F", "");
        g2.insertUndirected("F", "H", "");
        g2.insertUndirected("A", "C", "");
        g2.insertUndirected("A", "D", "");
        g2.insertUndirected("D", "G", "");
        g2.insertUndirected("G", "I", "");
        g2.insertUndirected("A", "E", "");
        g2.insertDirected("I", "H", ""); //directed edge not from class

        //run DFS from A and find path to H
        GT.DFS(g2,"A");
        GT.findPath("A", "B");

        //run BFS from A and find path to H
        GT.BFS(g2,"A");
        GT.findPath("A", "B");

    }
}
```

- Create graph
- Added extra edge

- Run DFS from A
- Find path A to B




Problems  Javadoc  Declaration  Console  Debug  Expressions  Error Log  Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 5:14:16 PM)

```
Depth First Search from A
Path from A to B
[A, D, G, I, H, F, B]

Breadth First Search from A
Path from A to B
[A, B]
```

**DFS**

| Key | Value |
|---|---|
| A | Null |
| B | F |
| C | A |
| D | A |
| E | A |
| F | H |
| G | D |
| H | I |
| I | G |

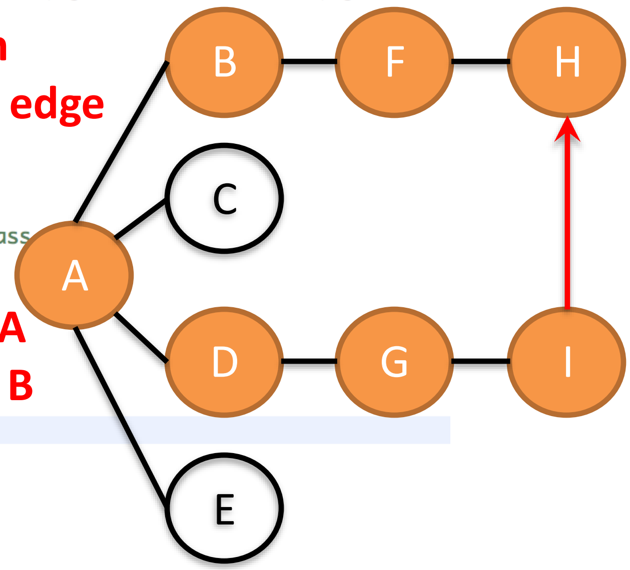
# DFS on today's graph

## GraphTraversal.java

```java
        //set up graph from Graph Traversal class
        AdjacencyMapGraph<String,String> g2 = new AdjacencyMapGraph<String,String>();
        g2.insertVertex("A"); g2.insertVertex("B"); g2.insertVertex("C"); g2.insertVertex("D");
        g2.insertVertex("E"); g2.insertVertex("F"); g2.insertVertex("G"); g2.insertVertex("H"); g2.insert
        g2.insertUndirected("A", "B", "");
        g2.insertUndirected("B", "F", "");
        g2.insertUndirected("F", "H", "");
        g2.insertUndirected("A", "C", "");
        g2.insertUndirected("A", "D", "");
        g2.insertUndirected("D", "G", "");
        g2.insertUndirected("G", "I", "");
        g2.insertUndirected("A", "E", "");
        g2.insertDirected("I", "H", ""); //directed edge not from class

        //run DFS from A and find path to H
        GT.DFS(g2,"A");
        GT.findPath("A", "B");

        //run BFS from A and find path to H
        GT.BFS(g2,"A");
        GT.findPath("A", "B");

    }

}
```

- Create graph
- Added extra edge

- Run DFS from A
- Find path A to B




**DFS**

| Key | Value |
|---|---|
| A | Null |
| **B** | **F** |
| C | A |
| **D** | **A** |
| E | A |
| **F** | **H** |
| **G** | **D** |
| **H** | **I** |
| **I** | **G** |

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 5:14:16 PM)

```
Depth First Search from A
Path from A to B
[A, D, G, I, H, F, B]

Breadth First Search from A
Path from A to B
[A, B]
```

**DFS findPath("A", "B") finds path, but not shortest path!**

**A->D->G->I->H->F->B**

**Why?**

**DFS explores as in a maze, as far as it can go before backing up**

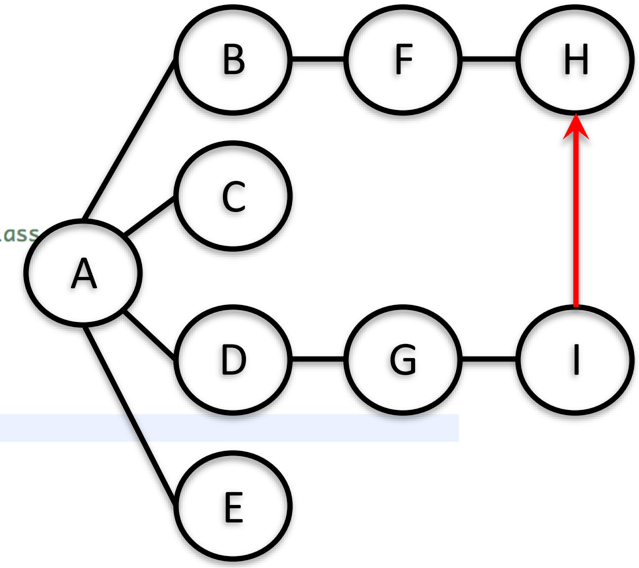
**Here DFS popped D from Stack before it popped B and explored until B found**

# BFS on today's graph

## GraphTraversal.java

```java
        //set up graph from Graph Traversal class
        AdjacencyMapGraph<String,String> g2 = new AdjacencyMapGraph<String,String>();
        g2.insertVertex("A"); g2.insertVertex("B"); g2.insertVertex("C"); g2.insertVertex("D");
        g2.insertVertex("E"); g2.insertVertex("F"); g2.insertVertex("G"); g2.insertVertex("H"); g2.insert
        g2.insertUndirected("A", "B", "");
        g2.insertUndirected("B", "F", "");
        g2.insertUndirected("F", "H", "");
        g2.insertUndirected("A", "C", "");
        g2.insertUndirected("A", "D", "");
        g2.insertUndirected("D", "G", "");
        g2.insertUndirected("G", "I", "");
        g2.insertUndirected("A", "E", "");
        g2.insertDirected("I", "H", ""); //directed edge not from class

        //run DFS from A and find path to H
        GT.DFS(g2,"A");
        GT.findPath("A", "B");

        //run BFS from A and find path to H
        GT.BFS(g2,"A");
        GT.findPath("A", "B");

    }
}
```

- **Run BFS**
- **Find path A to B**




Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 5:14:16 PM)

```
Depth First Search from A
Path from A to B
[A, D, G, I, H, F, B]

Breadth First Search from A
Path from A to B
[A, B]
```

**BFS**

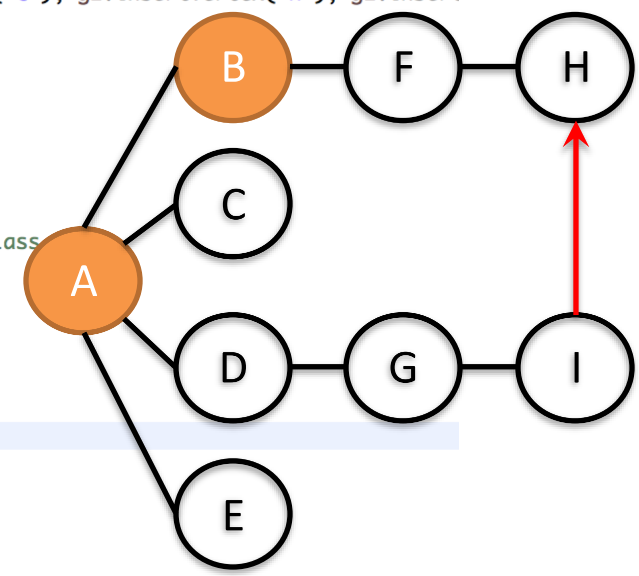
| Key | Value |
|---|---|
| A | Null |
| **B** | **A** |
| C | A |
| D | A |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

# BFS on today's graph

**GraphTraversal.java**

```java
        //set up graph from Graph Traversal class
        AdjacencyMapGraph<String,String> g2 = new AdjacencyMapGraph<String,String>();
        g2.insertVertex("A"); g2.insertVertex("B"); g2.insertVertex("C"); g2.insertVertex("D");
        g2.insertVertex("E"); g2.insertVertex("F"); g2.insertVertex("G"); g2.insertVertex("H"); g2.insert
        g2.insertUndirected("A", "B", "");
        g2.insertUndirected("B", "F", "");
        g2.insertUndirected("F", "H", "");
        g2.insertUndirected("A", "C", "");
        g2.insertUndirected("A", "D", "");
        g2.insertUndirected("D", "G", "");
        g2.insertUndirected("G", "I", "");
        g2.insertUndirected("A", "E", "");
        g2.insertDirected("I", "H", ""); //directed edge not from class

        //run DFS from A and find path to H
        GT.DFS(g2,"A");
        GT.findPath("A", "B");

        //run BFS from A and find path to H
        GT.BFS(g2,"A");
        GT.findPath("A", "B");

    }
}
```

- **Run BFS**
- **Find path A to B**




**BFS**

| Key | Value |
|---|---|
| A | Null |
| **B** | **A** |
| C | A |
| D | A |
| E | A |
| F | B |
| G | D |
| H | F |
| I | G |

Problems Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> GraphTraversal [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 12, 2018, 5:14:16 PM)

```
Depth First Search from A
Path from A to B
[A, D, G, I, H, F, B]

Breadth First Search from A
Path from A to B
[A, B]
```

**BFS findPath("A", "B") finds shortest path**

**A->B**

**Why?**

**BFS explores outward in ripples**