Graphs
Implementation and Traversals

CS 112 Spring 2020 – Apr 2

Graph Implementation

```
class Neighbor {
    int vertexNum;
    Neighbor next;
    Neighbor(int vnum, Neighbor nbr) {
        this.vertexNum = vnum;
        next = nbr;
    }
}
```

```
class Vertex {
    String name;
    Neighbor adjList;
    Vertex(String name, Neighbor neighbors) {
        this.name = name;
        this.adjList = neighbors;
    }
}
```

```
public class Graph {

   Vertex[] adjLists;
   boolean undirected=true;

   public Graph(String file)
   throws FileNotFoundException {
       Scanner sc = new Scanner(new File(file));
       String graphType = sc.next();
       if (graphType.equals("directed")) {
          undirected=false;
       }
       ...
   }
   ...
}
```

friendship.txt

```
undirected
10
Sara
Sam
Sean
Ajay
Mira
Jane
Maria
Rahul
Sapna
Rohit
Sara Sam
Sara Ajay
Sam Sean
Sam Mira
Mira Jane
Jane Maria
Rahul Sapna
Sapna Rohit
```

website.txt

```
directed
6
PageA
PageB
PageC
PageD
PageE
PageF
PageA PageB
PageA PageD
PageA PageE
PageB PageD
PageC PageA
PageD PageB
PageE PageF
PageF PageD
```

```
public class Graph {
    Vertex[] adjLists;
    boolean undirected=true;
    public Graph(String file)
    throws FileNotFoundException {
        Scanner sc = new Scanner(new File(file));
        String graphType = sc.next();
        if (graphType.equals("directed")) {
           undirected=false;
        }
        adjLists = new Vertex[sc.nextInt()];

        // read vertices
        for (int v=0; v < adjLists.length; v++) {
           adjLists[v] = new Vertex(sc.next(), null);
        }

        // read edges
        while (sc.hasNext()) {
            // read vertex names and translate to vertex numbers
            int v1 = indexForName(sc.next());
            int v2 = indexForName(sc.next());
            // add v2 to front of v1's adjacency list and
            adjLists[v1].adjList = new Neighbor(v2, adjLists[v1].adjList);
            if (undirected) {
               // add v1 to front of v2's adjacency list
               adjLists[v2].adjList = new Neighbor(v1, adjLists[v2].adjList);
            }
        }
        sc.close();
    }

    int indexForName(String name) { ... }
    ...
}
```

website.txt

```
directed
6
PageA
PageB
PageC
PageD
PageE
PageF
PageA PageB
PageA PageD
PageA PageE
PageB PageD
PageC PageA
PageD PageB
PageE PageF
PageF PageD
```

friendship.txt

```
undirected
10
Sara
Sam
Sean
Ajay
Mira
Jane
Maria
Rahul
Sapna
Rohit
Sara Sam
Sara Ajay
Sam Sean
Sam Mira
Mira Jane
Jane Maria
Rahul Sapna
Sapna Rohit
```

```
for (int v=0; v < adjLists.length; v++) {
    if (adjLists[v].name.equals(name)) {
       return v;
    }
}
return -1;
```

Graph Traversals

Traversals are a means to explore the topology of the graph

There are two standard graph traversal methods:

Depth-first search (DFS) Breadth-first search (BFS)

(The word ”search” does not mean searching for a specific vertex or edge - it actually means “scan”)

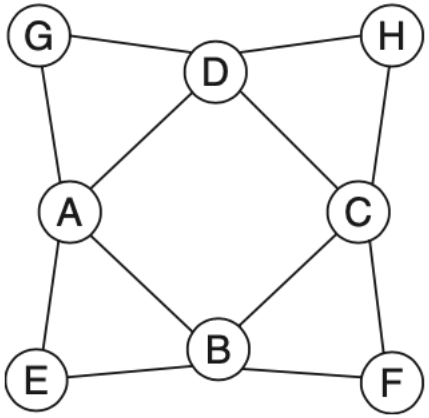
Neither of these traversals answers a specific question – typically an actual graph application might use one or the other as a building block to solve a practical problem

For example, LinkedIn might use BFS to tell how many links separate you from someone else e.g. 3rd connection

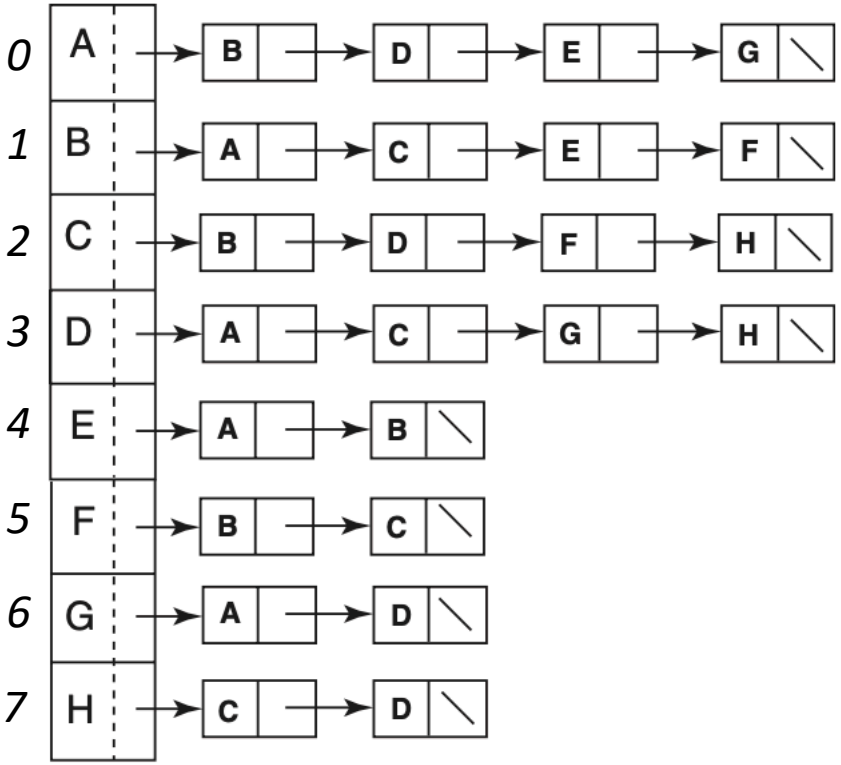
Or, a routing algorithm might use DFS to determine how many different paths there are to get from point A to point B

Depth-first Search (DFS)

Depth-first search (DFS)



Vertices have been numbered in alphabetical order for ease of exposition. In practice, they can be numbered (placed in the adjacency array) in any order

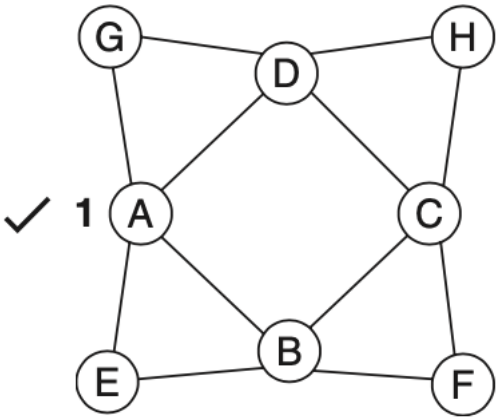


In the adjacency lir

1. Neighbors of a vertex are stored in alphabetical order only for ease of exposition. In practice, they could be in any order – it all depends on how they are stored in file
2. The linked list nodes show names of vertices. Again this is for ease of readability, in practice they would be vertex number. So, imagine 1 in place of B, 3 in place of D, etc.

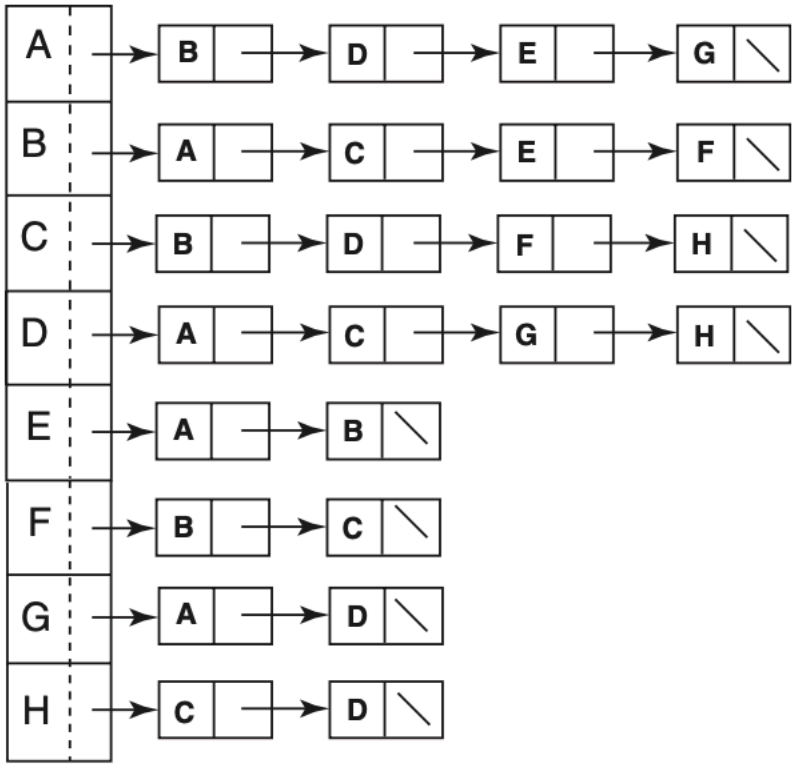
Depth-first search (DFS)

Suppose the traversal starts at vertex *A*. It visits *A* - *t*he traversal keeps track of which vertices have been visited by appropriately ‘‘marking’’ a vertex when it is visited (in the boolean visited array)



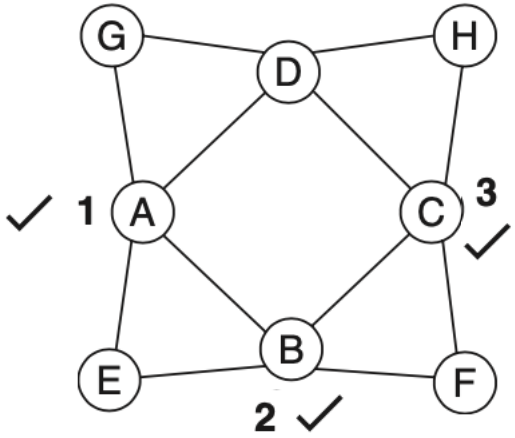
visited

T	F	F	F	F	F	F	F
A	B	C	D	E	F	G	H



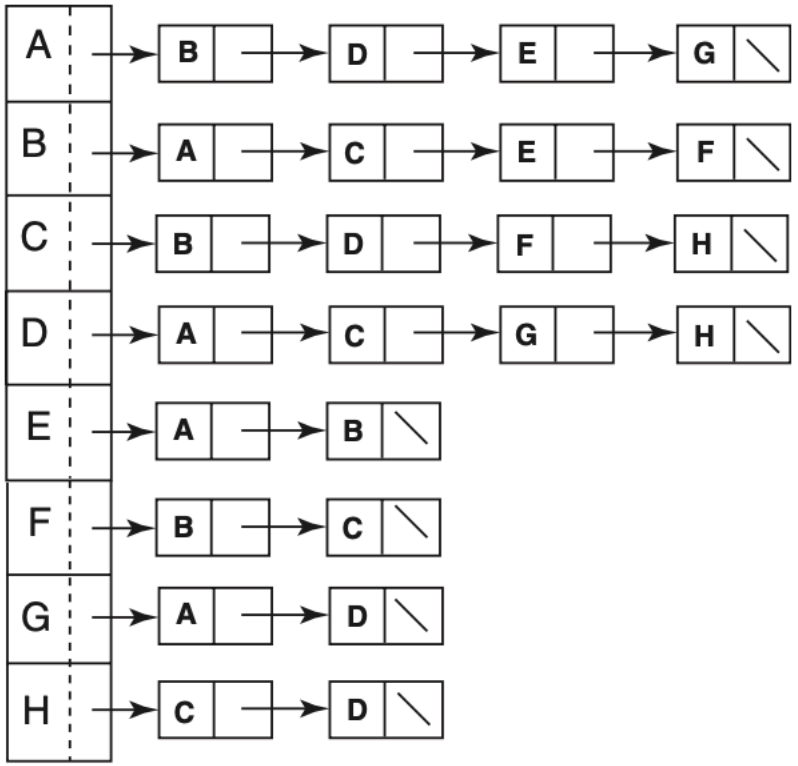
After visiting A, the traversal has to choose a vertex to visit next. *A* has four neighbors: *B*, *D*, *E*, and *G*. Any one of these can be picked as the next vertex to visit without affecting the correctness of the traversal. Which neighbor is picked next depends on the order in which the neighbors are stored. Here, B is the first vertex in A’s neighbor list, so it is picked next to visit

Depth-first search (DFS)



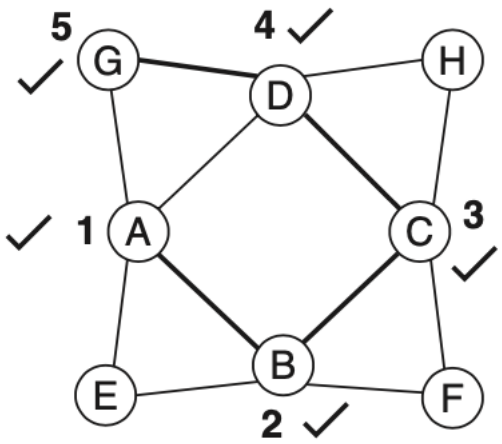
visited

A	B	C	D	E	F	G	H
T	T	T	F	F	F	F	F



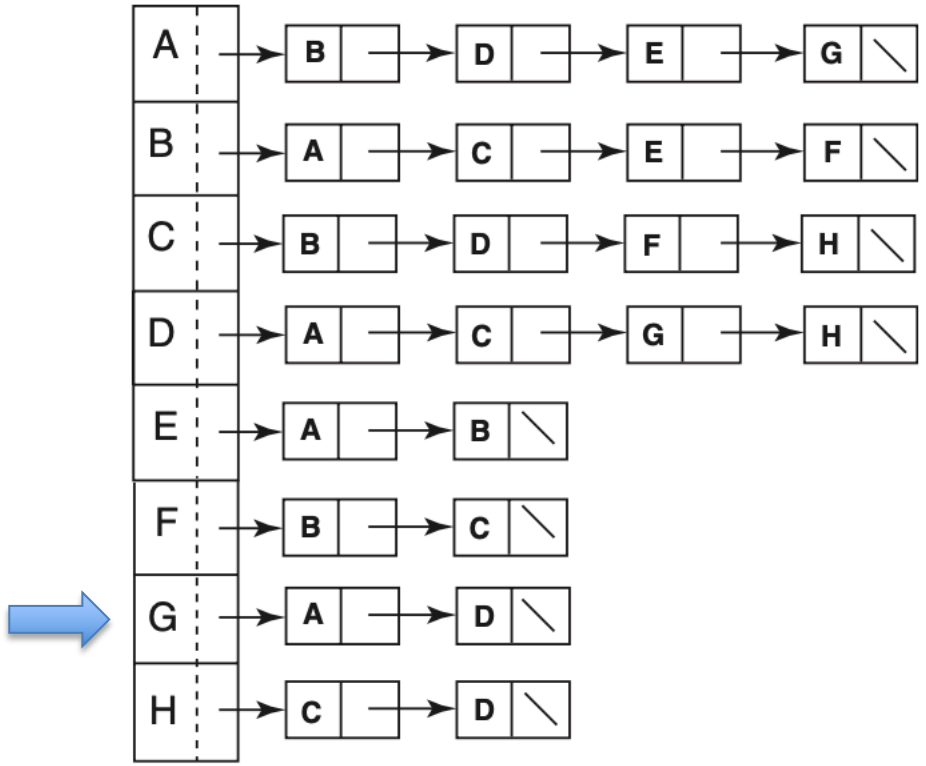
After visiting *B*, in order to decide which vertex to visit next, it first examines B's neighbor *A*. Since *A* has been visited, it is skipped. The next vertex in the neighbors list, C, has not yet been visited. So C is visited next.

Depth-first search (DFS)



visited

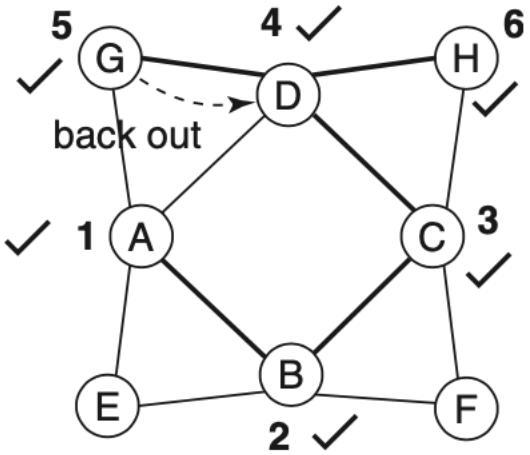
A	B	C	D	E	F	G	H
T	T	T	T	F	F	F	F



Continuing in this manner, the traversal next goes to D, then to G.

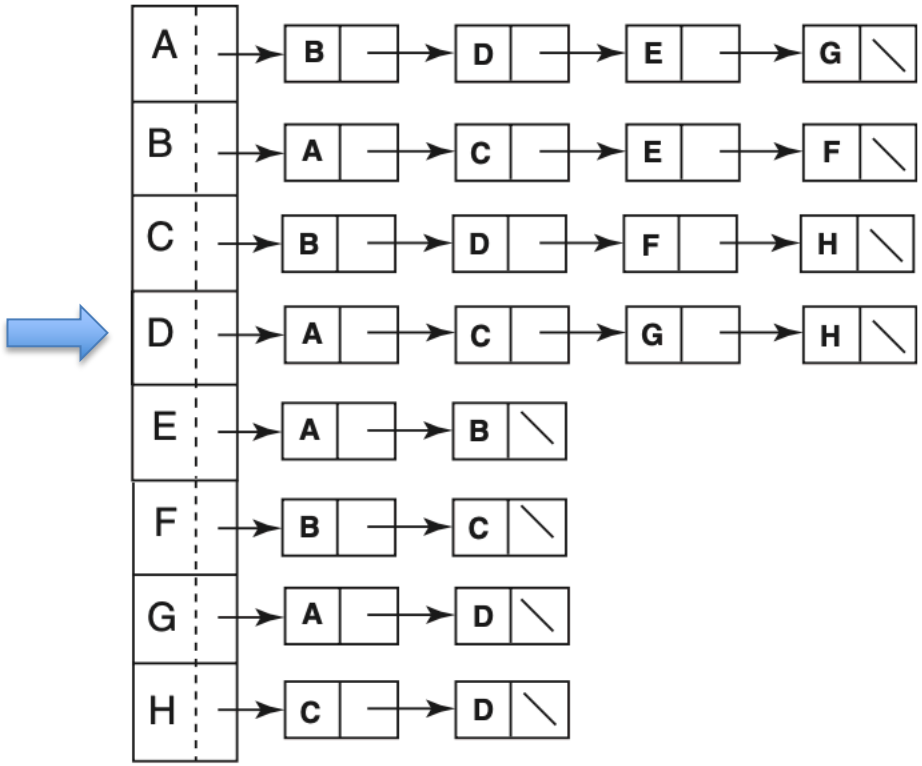
It examines the neighbors of *G* to select the next vertex to visit, but finds that both neighbors, *A* and *D,* have already been visited.
So the traversal has hit a dead end.

Depth-first search (DFS)



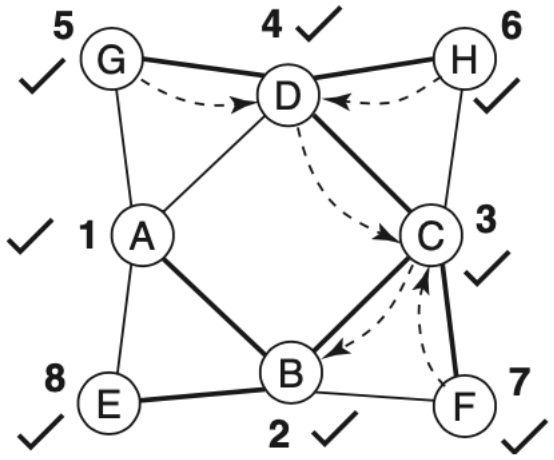
visited

A	B	C	D	E	F	G	H
T	T	T	T	F	F	T	F



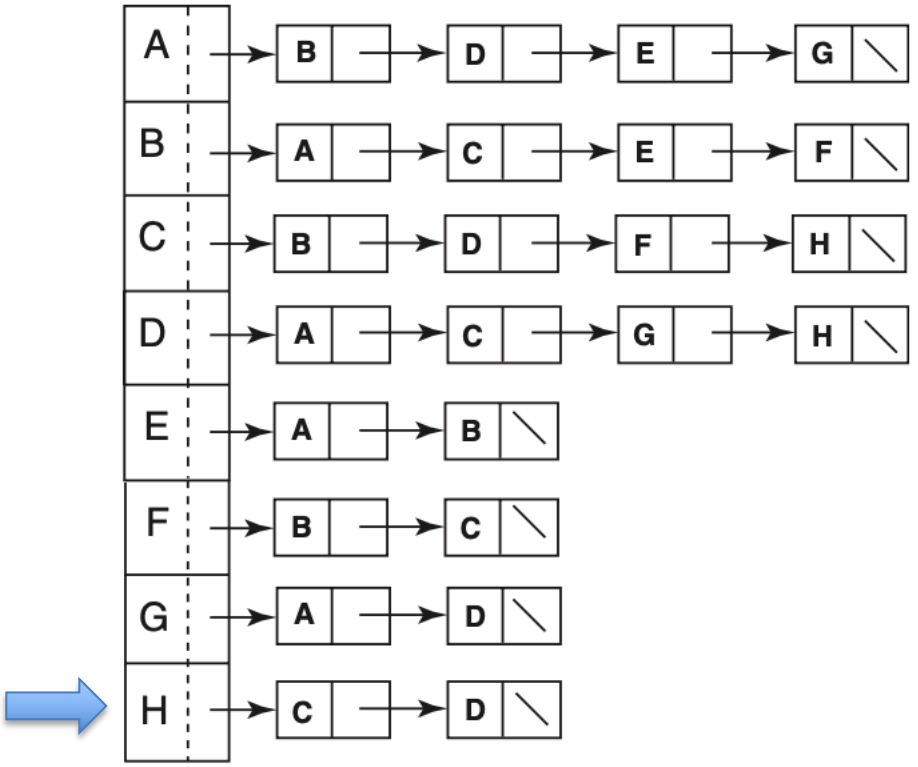
With no way forward from G, the traversal backs up to the previously visited vertex D. This is shown by the dashed arc with an arrow – this is basically a return to the previous step in the recursion, not an edge in the graph.

Depth-first search (DFS)



visited

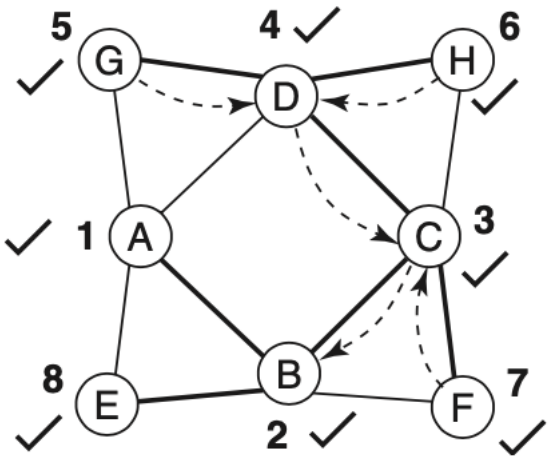
A	B	C	D	E	F	G	H
T	T	T	T	F	F	T	T

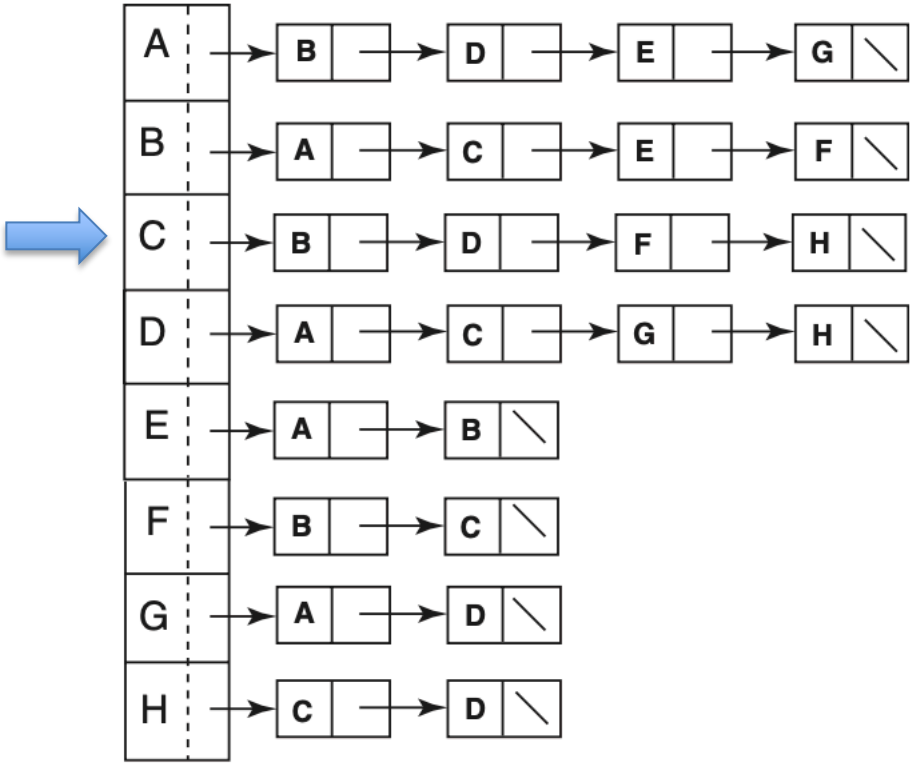


At *D*, the traversal now examines *H*. Since *H* has not been visited, the traversal moves forward and visits *H*.

After visiting *H* it finds that both neighbors of H, C and D, have been visited. So the traversal is at a dead-end again, and it backs out to *D* only to come to a dead end once more, since all the neighbors of *D* have been visited.

Depth-first search (DFS)





visited

A	B	C	D	E	F	G	H
T	T	T	T	T	T	T	T

The traversals backs up from D to *C*, from where it visits *F*, and then finds itself at a dead end yet again – both of F's neighbors, B and C, have already been visited. So from F it backs up to *C,* and then to *B*, from where it moves forward to visit *E*. It then backs up to *B* and then to *A*, upon which it terminates.

The final sequence of visits is: *A*, *B*, *C*, *D*, *G*, *H*, *F*, *E*

Depth-first search (DFS)

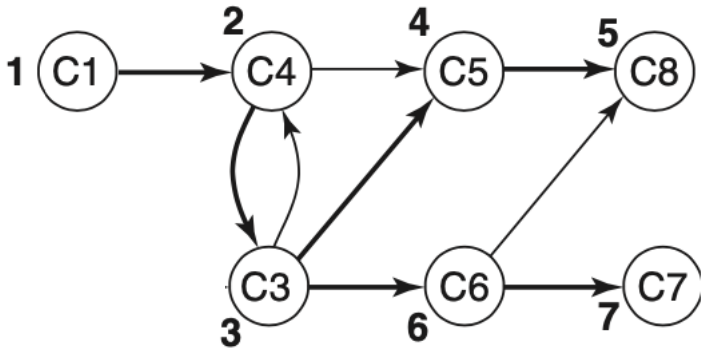
```
// recursive dfs
private void dfs(int v, boolean[] visited) {
    visited[v] = true;
    for (Neighbor e=adjLists[v].adjList; e != null; e=e.next) {
        if (!visited[e.vertexNum]) {
            dfs(e.vertexNum, visited);
        }
    }
}
```

Starting at some vertex, depth-first search follows a path that goes as deep or far down the graph as possible, visiting each vertex along this path.

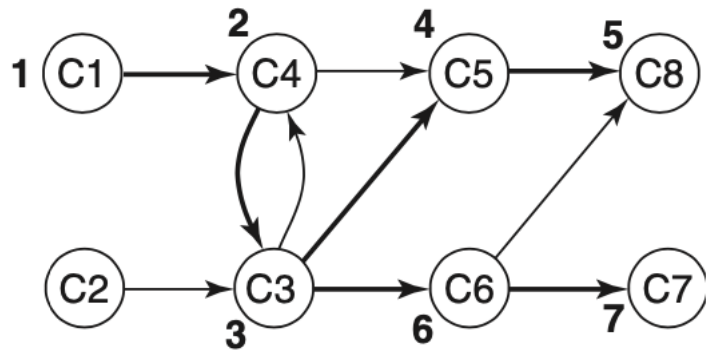
When the traversal cannot proceed any farther along the current direction, it backs up and tries to take another direction for traversal, again going forward as deep as possible before backing up and repeating.

Depth-first search (DFS)

The process (and code) is the exactly same for a directed graph.



What if there was another vertex, C2, in the graph, with an edge to C3?



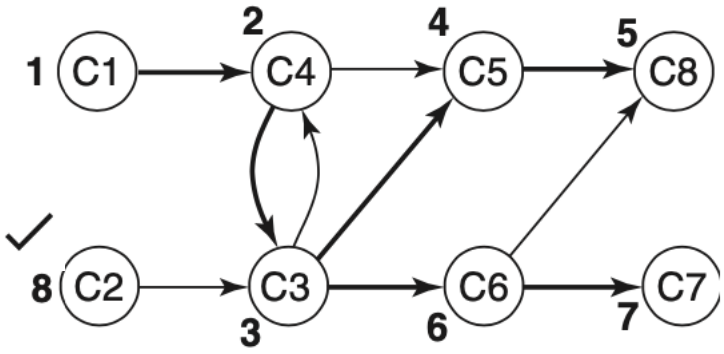
If the traversal starts at C1, there is no way to reach C2. (So DFS can be used to find out all vertices reachable from a given vertex.)

However, if we want to do a complete traversal, and get to every single vertex, the only way to do this is to restart the traversal.

And the restart is done at the first unvisited vertex in the visited array

visited

T	F	T	T	T	T	T	T
C1	C2	C3	C4	C5	C6	C7	C8



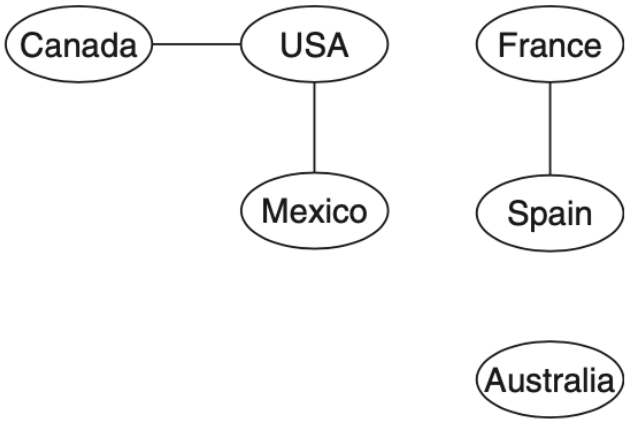
Depth-first search (DFS)

We need a ”driver” to restart the traversal every time it stalls, and we haven’t finished visiting all vertices.

```
// driver (this is the method called from any application
public void dfs() {
    boolean[] visited = new boolean[adjLists.length];
    for (int v=0; v < visited.length; v++) {
        visited[v] = false;
    }
    for (int v=0; v < visited.length; v++) {
        if (!visited[v]) { // start/restart at v
            dfs(v, visited);
        }
    }
}
// recursive dfs
private void dfs(int v, boolean[] visited) {
    visited[v] = true;
    for (Neighbor e=adjLists[v].adjList; e != null; e=e.next) {
        if (!visited[e.vertexNum]) {
            dfs(e.vertexNum, visited);
        }
    }
}
```

Depth-first search (DFS)

Restarts can happen with undirected graphs as well



This graph has islands – it is "unconnected"

No matter where we start the traversal, 2 restarts would be needed

Start at US

F	F	F	F	F	F
US	SP	AU	CA	MX	FR

Restart at SP

T	F	F	T	T	F
US	SP	AU	CA	MX	FR

Restart at AU

T	T	F	T	T	T
US	SP	AU	CA	MX	FR

Done

T	T	T	T	T	T
US	SP	AU	CA	MX	FR

DFS big O Running Time

What to count towards running time?

$O(n+e)$ Recursive dfs

n - Visiting a vertex : O(1)

(The O(1) includes marking cell in visited array as true, plus whatever is done in visiting, e.g. printing the vertex name)

e (directed)
$2e$ (undirected) - Checking whether a neighbor is visited: O(1)

(The O(1) includes indexing into the visited array, as well as pushing pointer to next neighbor in adjacency linked list)

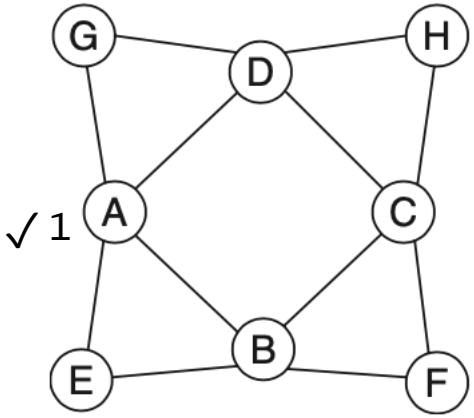
$O(n)$ Driver
- Scanning the visited array

Total: $O(n+e)$

There is no worst case/best case separation of running times since the traversal runs unconditionally through the entire graph

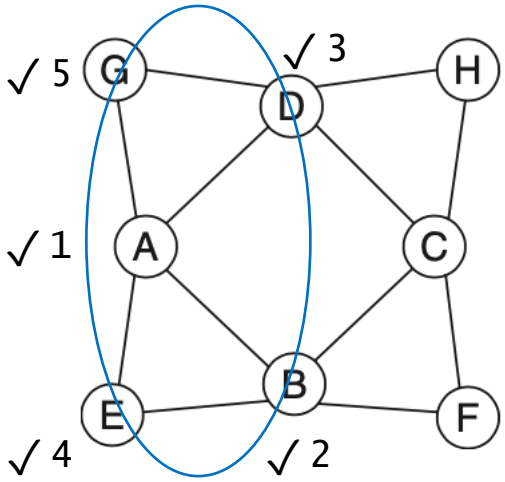
Breadth-first Search (BFS)

Breadth-first search (BFS)



Suppose the traversal starts at vertex *A*. It visits *A*.

T	F	F	F	F	F	F	F
A	B	C	D	E	F	G	H

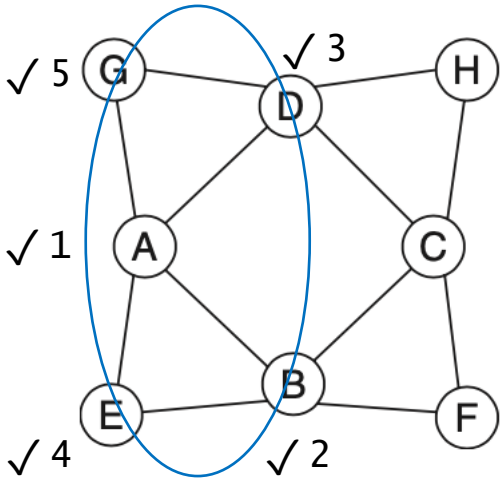


After visiting A, in the next phase the traversal visits ALL neighbors of A, in the order in which they appear in the neighbors list: B, D, E, and G

T	T	F	T	T	F	T	F
A	B	C	D	E	F	G	H

This is the first ”wave” of progress through the graph.

Breadth-first search (BFS)



The next wave would be all vertices adjacent to B, then all vertices adjacent to D, and so on.

In other words, the subsequent waves will fan out from vertices in the order in which they were seen in the first wave, i.e. in first-in first-out sequence.

Initial step

A ←

To implement this sequence, all vertices visited in a wave are added to a queue.
The queue is jump started with the very first vertex, A.

First iteration:
-- Dequeue A
-- Visit and enqueue B,D,E,G

B D E G ←

The BFS algorithm then iterates until the queue is empty: in every iteration, it dequeues a vertex, and visits all its neighbors

Breadth-first search (BFS)

Algorithm BFS(v)

```
visit v and mark v as visited
add v to the queue
while the queue is not empty do
        w ← vertex at the front of the queue
        delete w from the queue
        for each neighbor p of w do
            if (p is not visited) then
               visit p and mark p as visited
               add p to the queue
            endif
        endfor
endwhile
```

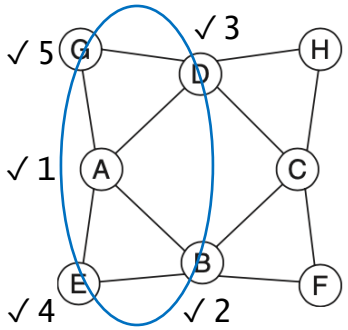
After the third iteration, every iteration dequeues a vertex.
But no new vertices are found to be unvisited, so nothing is added to the queue, and the queue eventually empties out.

Initial step

A

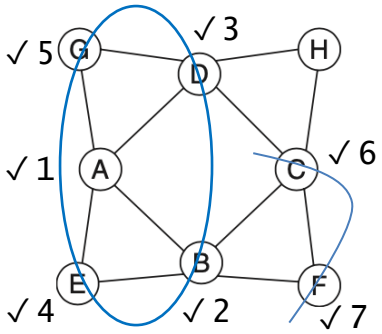
First iteration:
-- Dequeue A
-- Visit and enqueue B,D,E,G

B	D	E	G



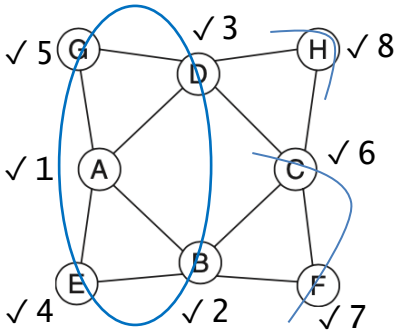
Second iteration:
-- Dequeue B
-- Visit and enqueue C,F

D	E	G	C	F



Third iteration:
-- Dequeue D
-- Visit and enqueue H

E	G	C	F	H



BFS big O Running Time

As with DFS, there will be a driver to start/restart the BFS process.

Operations to count towards the running time:

n - Visiting a vertex : O(1)
(The O(1) includes marking cell in visited array as true, plus whatever is done in visiting, e.g. printing the vertex name)

e (directed)
$2e$ (undirected) - Checking whether a neighbor is visited: O(1)
(The O(1) includes indexing into the visited array, as well as pushing pointer to next neighbor in adjacency linked list)

$2n$ - Vertex Enqueue/Vertex Dequeue: O(1)

$O(n)$ - Driver
Scanning the visited array

Total: $O(n+e)$

There is no worst case/best case separation of running times since the traversal runs unconditionally through the entire graph