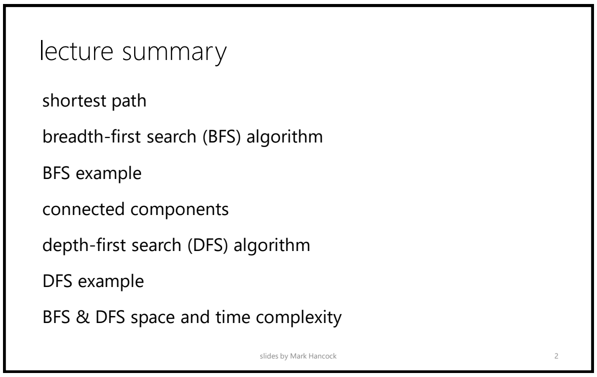
Breadth-First Search & Depth-First Search

MSCI 240: Algorithms & Data Structures



lecture summary

shortest path

breadth-first search (BFS) algorithm

BFS example

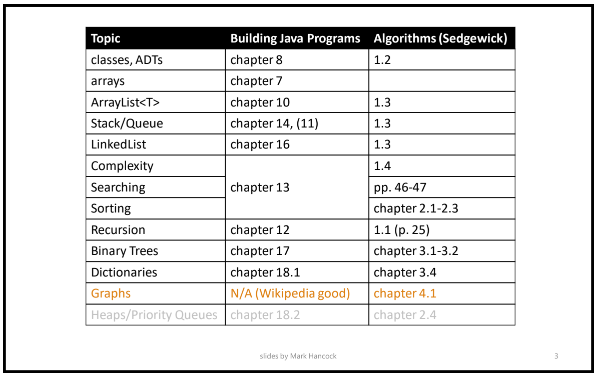
connected components

depth-first search (DFS) algorithm

DFS example

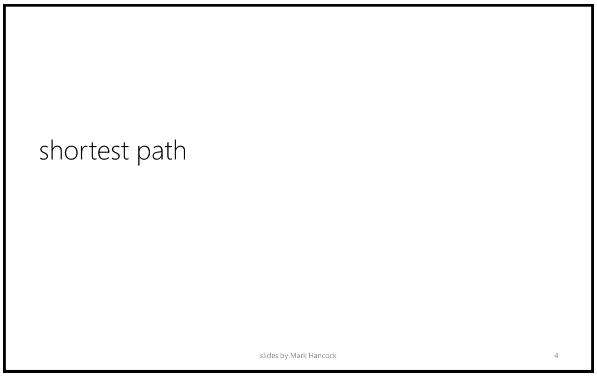
BFS & DFS space and time complexity

slides by Mark Hancock 2



Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity	chapter 13	1.4
Searching		pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
Binary Trees	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

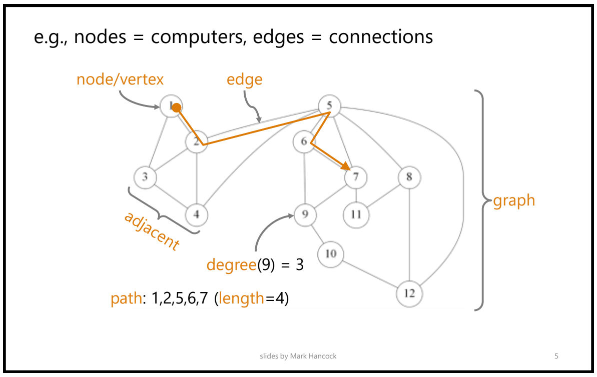
slides by Mark Hancock 3



shortest path

slides by Mark Hancock

4



e.g., nodes = computers, edges = connections

node/vertex

edge

graph

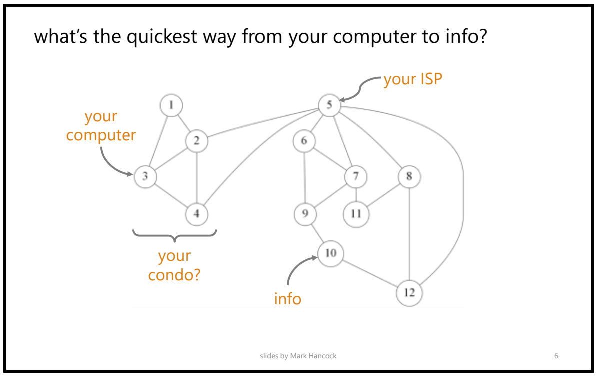
adjacent

degree(9) = 3

path: 1,2,5,6,7 (length=4)

slides by Mark Hancock

5



what's the quickest way from your computer to info?

your computer

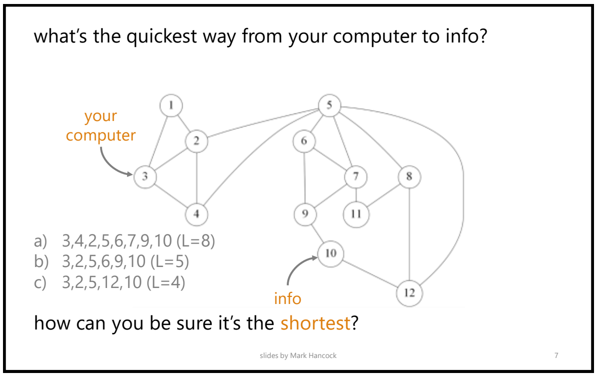
your ISP

your condo?

info

slides by Mark Hancock

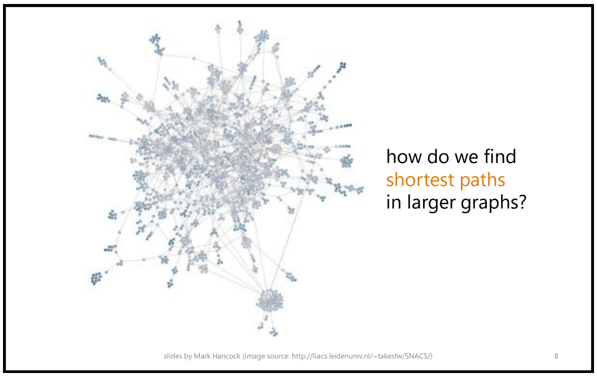
6



what's the quickest way from your computer to info?

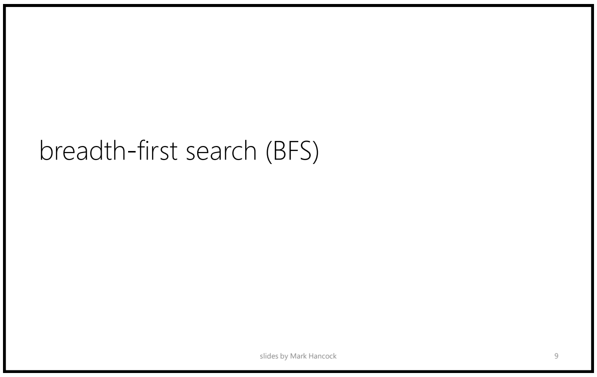
a) 3,4,2,5,6,7,9,10 (L=8)
b) 3,2,5,6,9,10 (L=5)
c) 3,2,5,12,10 (L=4)

how can you be sure it's the shortest?

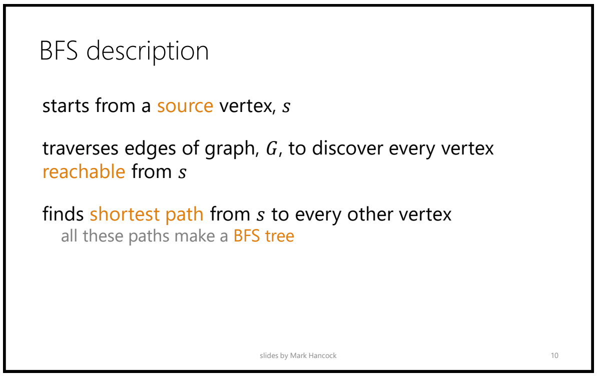


how do we find
shortest paths
in larger graphs?

slides by Mark Hancock (image source: http://liacs.leidenuniv.nl/~takesfw/SNACS/)



breadth-first search (BFS)



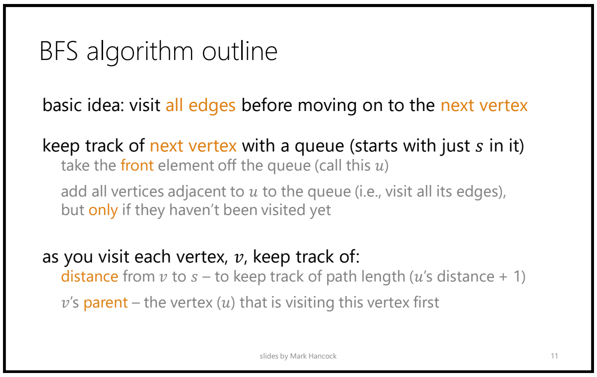
BFS description

starts from a source vertex, s

traverses edges of graph, G, to discover every vertex reachable from s

finds shortest path from s to every other vertex

- all these paths make a BFS tree



BFS algorithm outline

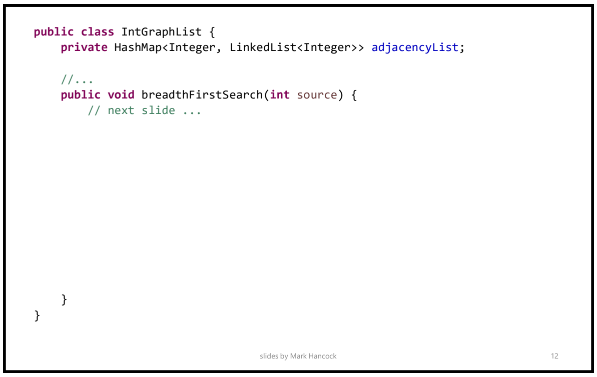
basic idea: visit all edges before moving on to the next vertex

keep track of next vertex with a queue (starts with just s in it)

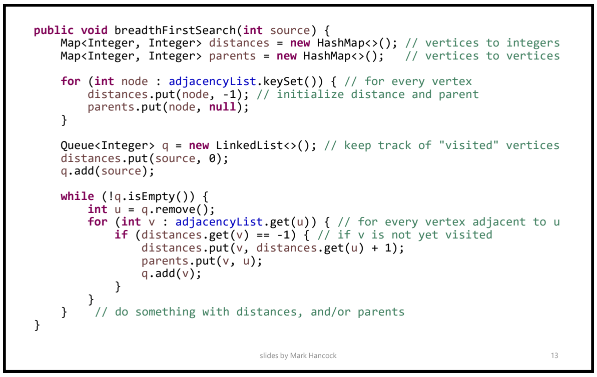
- take the front element off the queue (call this u)
- add all vertices adjacent to u to the queue (i.e., visit all its edges), but only if they haven't been visited yet

as you visit each vertex, v, keep track of:

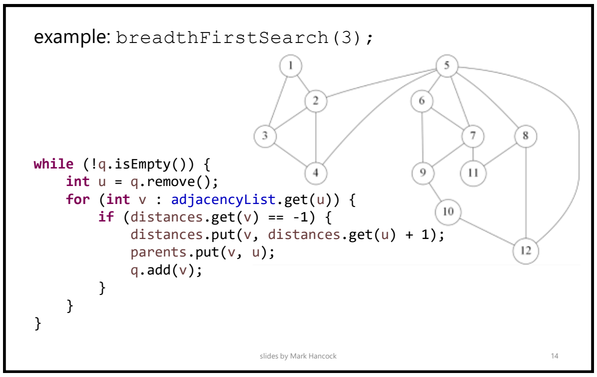
- distance from v to s – to keep track of path length (u's distance + 1)
- v's parent – the vertex (u) that is visiting this vertex first



```
public class IntGraphList {
    private HashMap<Integer, LinkedList<Integer>> adjacencyList;

    //...
    public void breadthFirstSearch(int source) {
        // next slide ...








    }
}
```

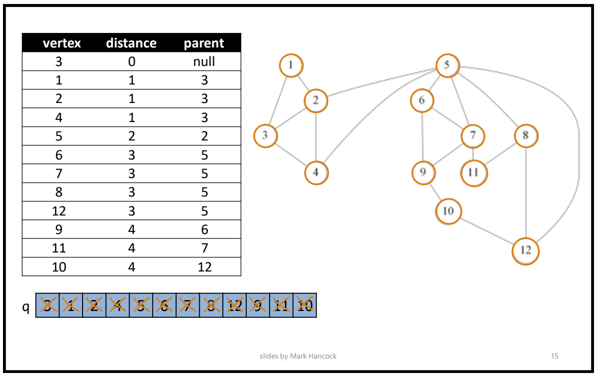


```
public void breadthFirstSearch(int source) {
    Map<Integer, Integer> distances = new HashMap<>(); // vertices to integers
    Map<Integer, Integer> parents = new HashMap<>();   // vertices to vertices

    for (int node : adjacencyList.keySet()) { // for every vertex
        distances.put(node, -1); // initialize distance and parent
        parents.put(node, null);
    }

    Queue<Integer> q = new LinkedList<>(); // keep track of "visited" vertices
    distances.put(source, 0);
    q.add(source);

    while (!q.isEmpty()) {
        int u = q.remove();
        for (int v : adjacencyList.get(u)) { // for every vertex adjacent to u
            if (distances.get(v) == -1) { // if v is not yet visited
                distances.put(v, distances.get(u) + 1);
                parents.put(v, u);
                q.add(v);
            }
        }
    }    // do something with distances, and/or parents
}
```



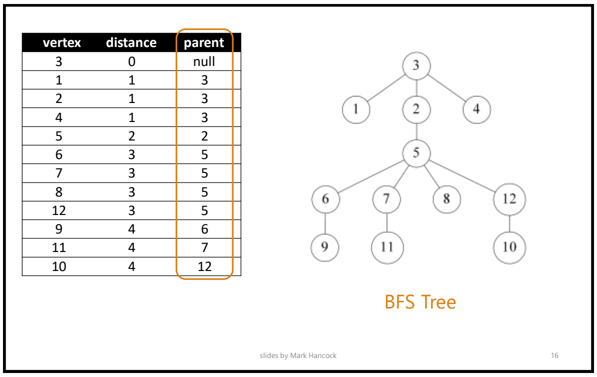
example: `breadthFirstSearch(3);`

```
while (!q.isEmpty()) {
    int u = q.remove();
    for (int v : adjacencyList.get(u)) {
        if (distances.get(v) == -1) {
            distances.put(v, distances.get(u) + 1);
            parents.put(v, u);
            q.add(v);
        }
    }
}
```



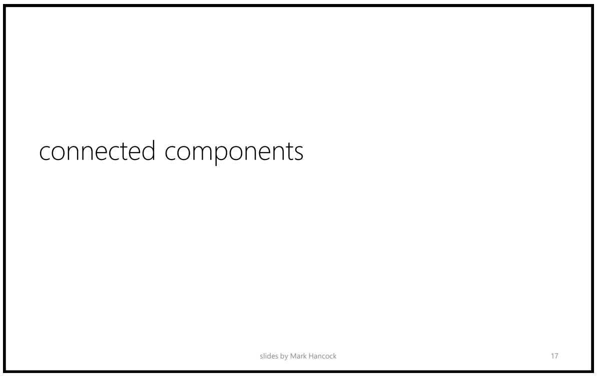
vertex	distance	parent
3	0	null
1	1	3
2	1	3
4	1	3
5	2	2
6	3	5
7	3	5
8	3	5
12	3	5
9	4	6
11	4	7
10	4	12

q ~~3~~ ~~1~~ ~~2~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ ~~8~~ ~~12~~ ~~9~~ ~~11~~ ~~10~~

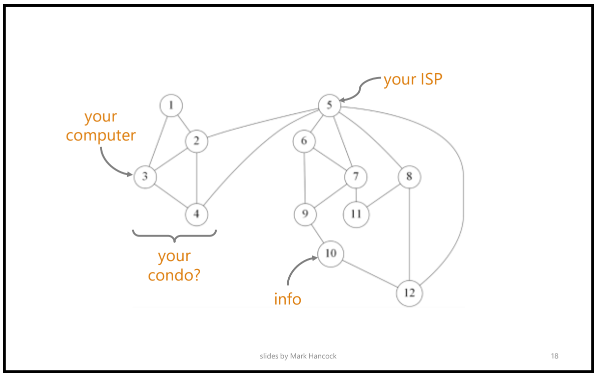


vertex	distance	parent
3	0	null
1	1	3
2	1	3
4	1	3
5	2	2
6	3	5
7	3	5
8	3	5
12	3	5
9	4	6
11	4	7
10	4	12

BFS Tree



connected components

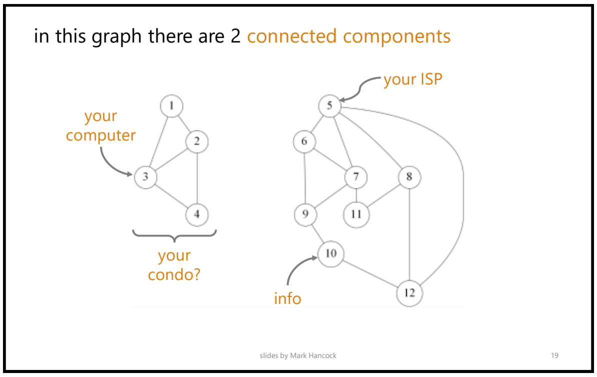


your computer

your condo?

your ISP

info



in this graph there are 2 connected components

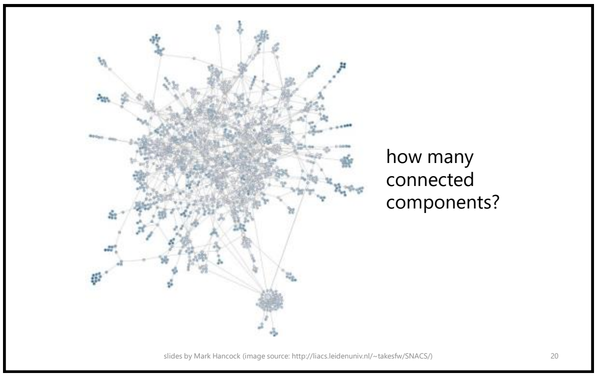
your ISP

your computer

your condo?

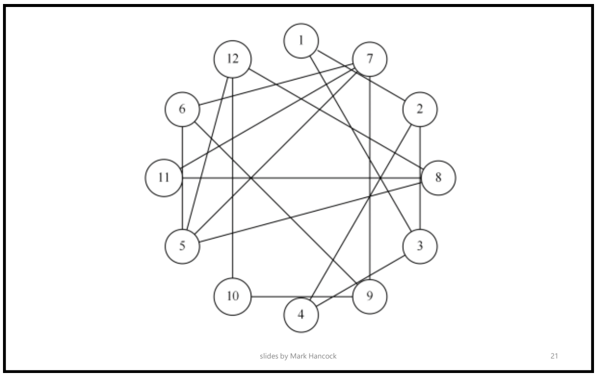
info

slides by Mark Hancock



how many connected components?

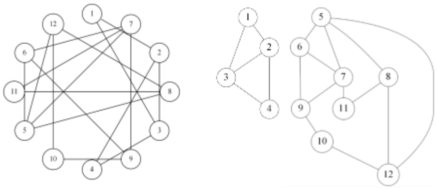
slides by Mark Hancock (image source: http://liacs.leidenuniv.nl/~takesfw/SNACS/)



slides by Mark Hancock

same graph!

∴ 2 connected components each



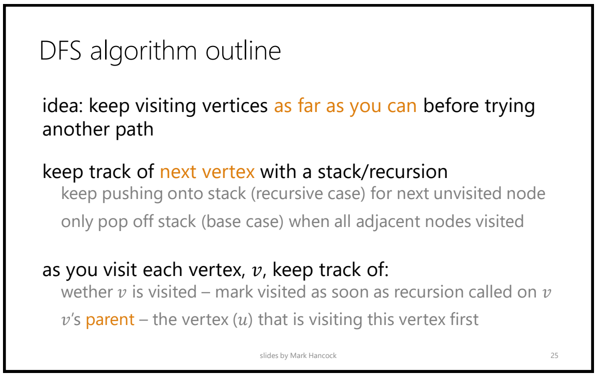
depth-first search (DFS)

DFS description

does not require a source vertex
but can have a starting point

traverses edges of graph, G, depth-first to discover connected components
every vertex reachable from every other vertex

all these paths make a DFS tree



DFS algorithm outline

idea: keep visiting vertices as far as you can before trying another path

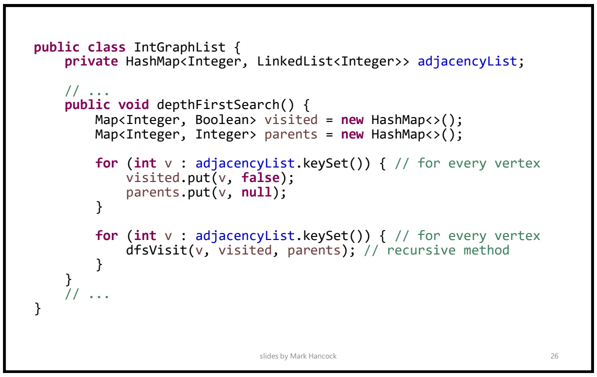
keep track of next vertex with a stack/recursion

- keep pushing onto stack (recursive case) for next unvisited node
- only pop off stack (base case) when all adjacent nodes visited

as you visit each vertex, v, keep track of:

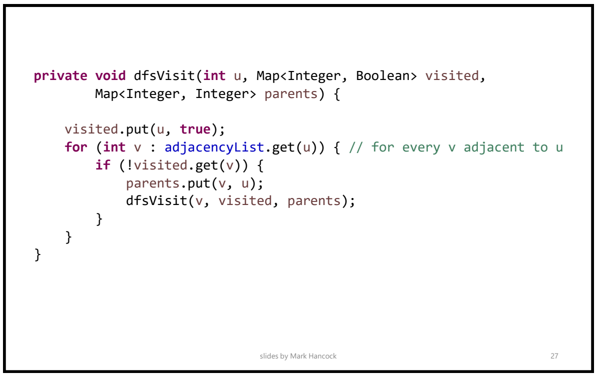
- wether v is visited – mark visited as soon as recursion called on v
- v's parent – the vertex (u) that is visiting this vertex first

slides by Mark Hancock 25



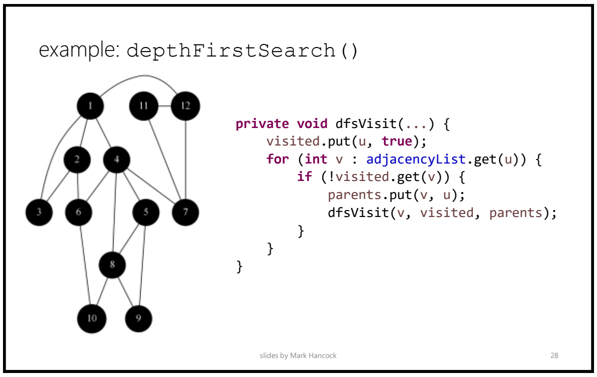
```
public class IntGraphList {
    private HashMap<Integer, LinkedList<Integer>> adjacencyList;

    // ...
    public void depthFirstSearch() {
        Map<Integer, Boolean> visited = new HashMap<>();
        Map<Integer, Integer> parents = new HashMap<>();

        for (int v : adjacencyList.keySet()) { // for every vertex
            visited.put(v, false);
            parents.put(v, null);
        }

        for (int v : adjacencyList.keySet()) { // for every vertex
            dfsVisit(v, visited, parents); // recursive method
        }
    }
    // ...
}
```

slides by Mark Hancock 26



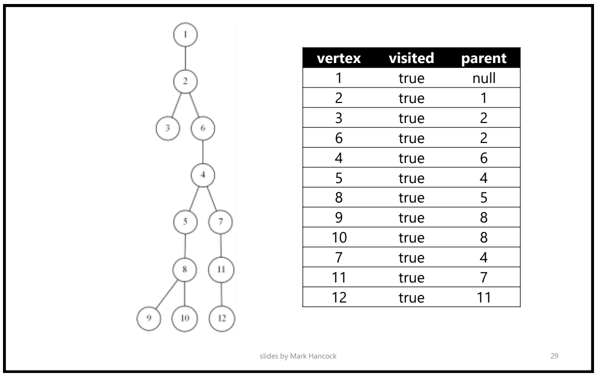
```
private void dfsVisit(int u, Map<Integer, Boolean> visited,
        Map<Integer, Integer> parents) {

    visited.put(u, true);
    for (int v : adjacencyList.get(u)) { // for every v adjacent to u
        if (!visited.get(v)) {
            parents.put(v, u);
            dfsVisit(v, visited, parents);
        }
    }
}
```

slides by Mark Hancock 27



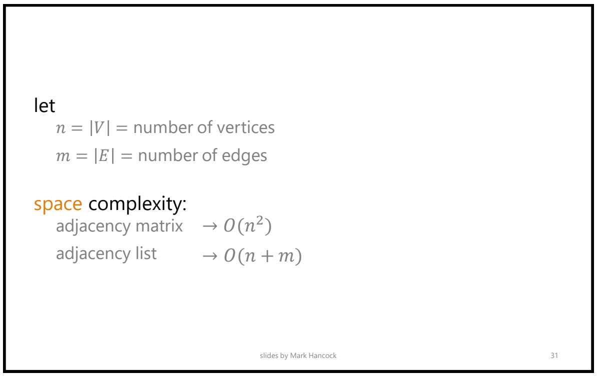
example: `depthFirstSearch()`

```
private void dfsVisit(...) {
    visited.put(u, true);
    for (int v : adjacencyList.get(u)) {
        if (!visited.get(v)) {
            parents.put(v, u);
            dfsVisit(v, visited, parents);
        }
    }
}
```



vertex	visited	parent
1	true	null
2	true	1
3	true	2
6	true	2
4	true	6
5	true	4
8	true	5
9	true	8
10	true	8
7	true	4
11	true	7
12	true	11

BFS & DFS complexity



let

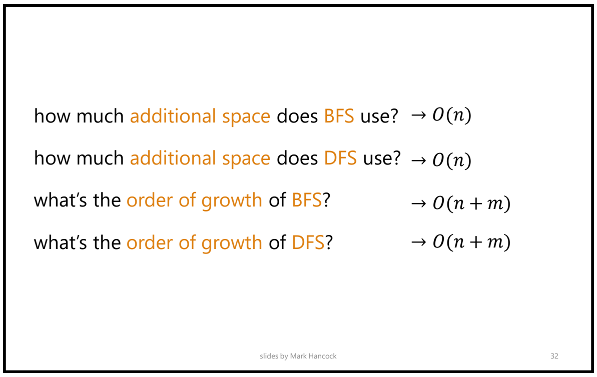
$n = |V| =$ number of vertices

$m = |E| =$ number of edges

space complexity:

adjacency matrix → $O(n^2)$

adjacency list → $O(n + m)$

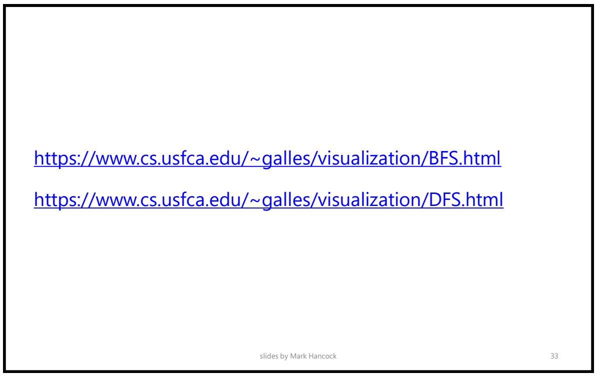


how much additional space does BFS use? → $O(n)$

how much additional space does DFS use? → $O(n)$

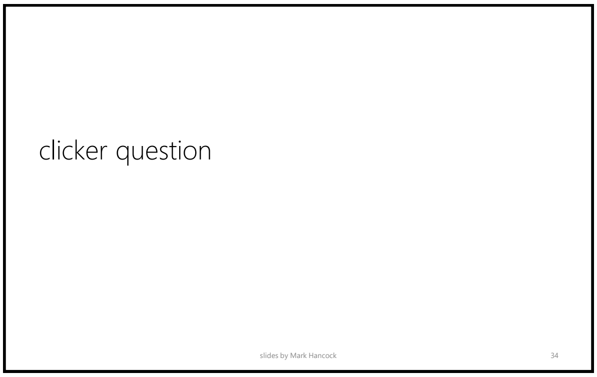
what's the order of growth of BFS? → $O(n + m)$

what's the order of growth of DFS? → $O(n + m)$



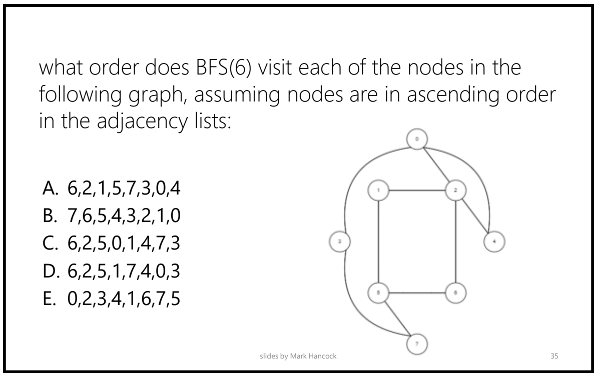
https://www.cs.usfca.edu/~galles/visualization/BFS.html

https://www.cs.usfca.edu/~galles/visualization/DFS.html



clicker question

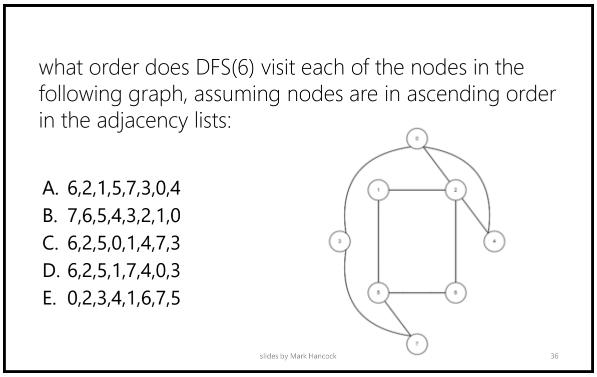
slides by Mark Hancock 34



what order does BFS(6) visit each of the nodes in the following graph, assuming nodes are in ascending order in the adjacency lists:

A. 6,2,1,5,7,3,0,4
B. 7,6,5,4,3,2,1,0
C. 6,2,5,0,1,4,7,3
D. 6,2,5,1,7,4,0,3
E. 0,2,3,4,1,6,7,5

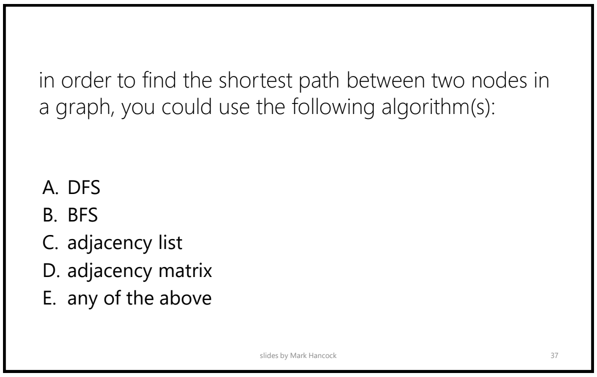
slides by Mark Hancock 35



what order does DFS(6) visit each of the nodes in the following graph, assuming nodes are in ascending order in the adjacency lists:

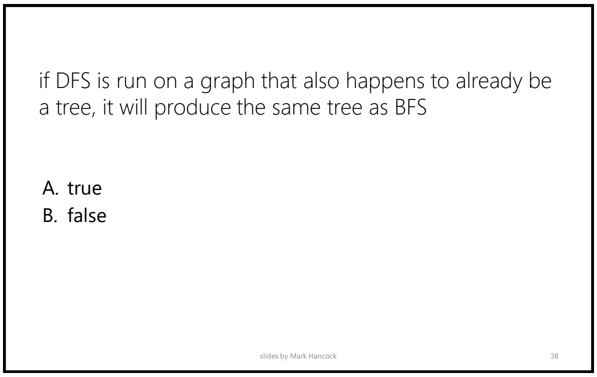
A. 6,2,1,5,7,3,0,4
B. 7,6,5,4,3,2,1,0
C. 6,2,5,0,1,4,7,3
D. 6,2,5,1,7,4,0,3
E. 0,2,3,4,1,6,7,5

slides by Mark Hancock 36



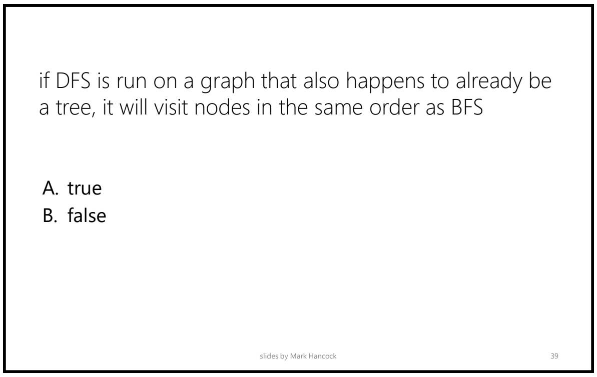
in order to find the shortest path between two nodes in a graph, you could use the following algorithm(s):

A. DFS
B. BFS
C. adjacency list
D. adjacency matrix
E. any of the above



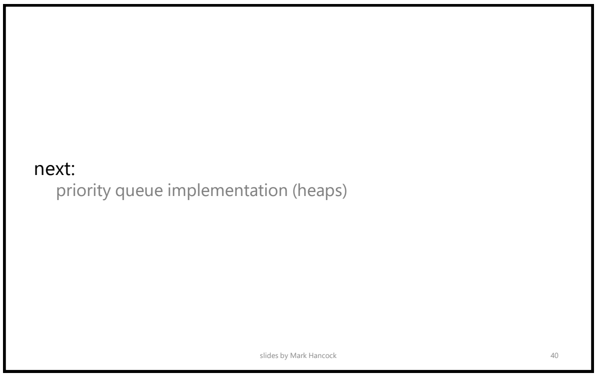
if DFS is run on a graph that also happens to already be a tree, it will produce the same tree as BFS

A. true
B. false



if DFS is run on a graph that also happens to already be a tree, it will visit nodes in the same order as BFS

A. true
B. false



next:

- priority queue implementation (heaps)

slides by Mark Hancock