L23: DFS & BFS

Alex Steiger

CompSci 201: Spring 2024

4/8/2024

Logistics, coming up

- Today, Monday, April 8
  - Project P5: Huffman due
  - Project P6: Route out by tomorrow

- This Wednesday, April 10
  - APT Quiz 2 due
  - Covers linked list and trees
    - Practice quiz from discussion is similar
  - No regular APTs due this week, just the quiz

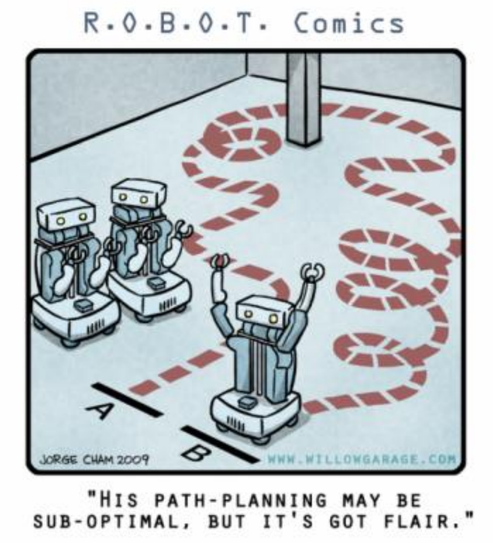
Today’s agenda

- General depth-first search (DFS)
  - Seen it on grid graphs, how about arbitrary graphs?

- Introduce breadth-first search (BFS)

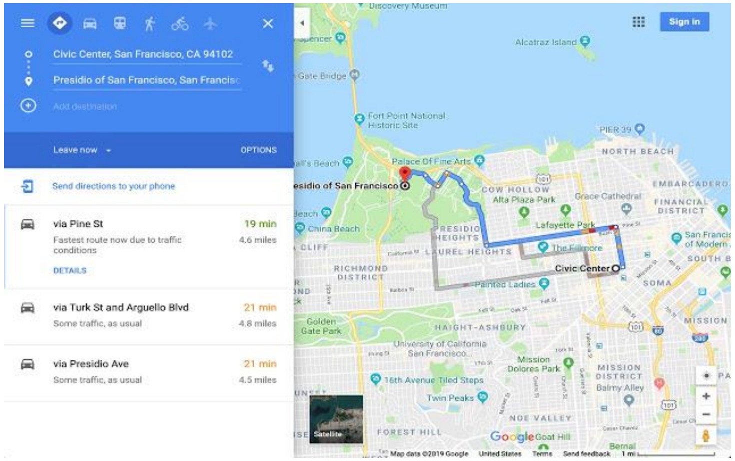
Depth-First Search in General Graphs

Pathfinding / Graph Search



Is there a way to get from point A to point B?

- Maps/directions
- Video games
- Robot motion planning
- Etc.



Civic Center, San Francisco, CA 94102

Presidio of San Francisco, San Francisc

Add destination

Leave now ▾ OPTIONS

Send directions to your phone

via Pine St — 19 min — 4.6 miles
Fastest route now due to traffic conditions
DETAILS

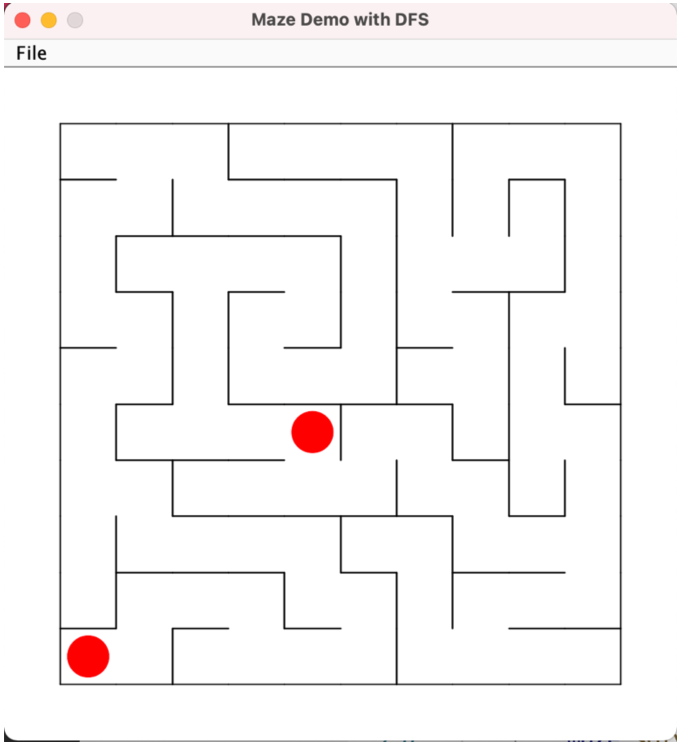
via Turk St and Arguello Blvd — 21 min — 4.8 miles
Some traffic, as usual

via Presidio Ave — 21 min — 4.5 miles
Some traffic, as usual

Recall: Grid Graph, Maze Example

```
public class MazeDemo {
    private int mySize;                 // dimension of maze
    private boolean[][] north;          // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
    private boolean[][] west;
```

- Example: 10 x 10 grid
- Edge = no wall, no edge = wall.
- Look for a path from start (lower left) to middle.

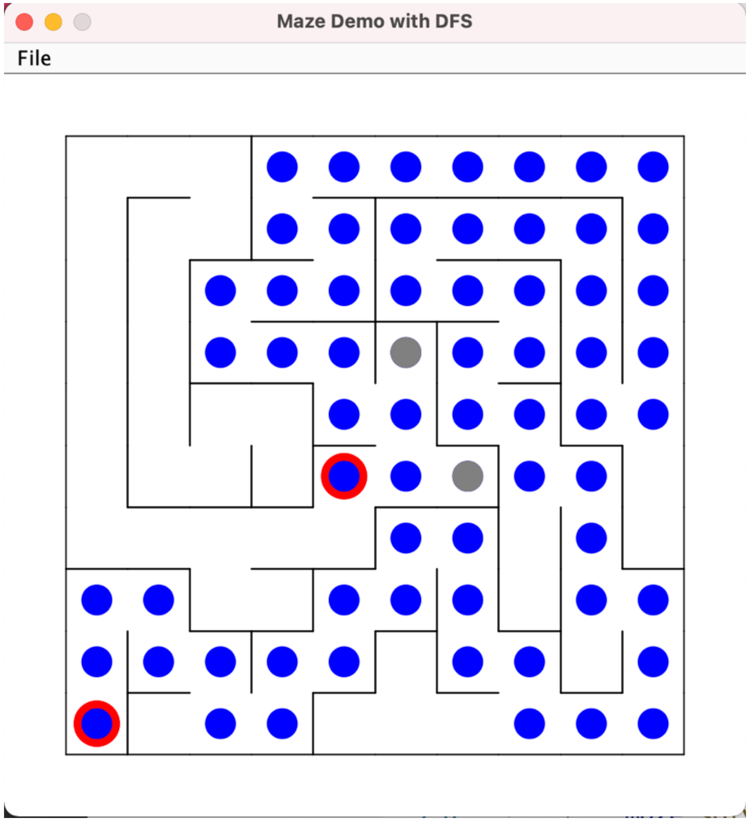


Depth-First Search for Solving Maze

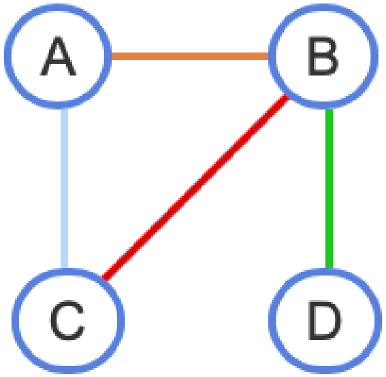
Always explore (recurse on) a new (unvisited) adjacent vertex if possible.

If nothing new (unvisited) vertex to explore:

- *backtrack* to the most recent vertex adjacent to an unvisited vertex, and then continue.
- if no such vertex, maze is unsolvable.



Representations for Arbitrary Graphs (not only Grid Graphs)



Adjacency List

Vertices	Adjacent vertices (edges)
A	B C
B	A C D
C	A B
D	B

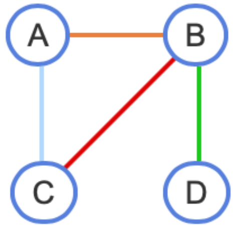
zyBook

Adjacency Matrix

$$\begin{array}{c|cccc|} & A & B & C & D \\ A & & 1 & 1 & \\ B & 1 & & 1 & 1 \\ C & 1 & 1 & & \\ D & & 1 & & \end{array}$$

Efficient Adjacency “List” Using Double Hashing

- `HashMap<Vertex, HashSet<Vertex>> aList`
  - `Vertex` type can be Integer, char, String, custom object, …, needs to have good hashCode() and equals().



Vertices	Adjacent vertices (edges)
A	B C
B	A C D
C	A B
D	B

- `aList.put(‘A’, new HashSet())`
- `aList.get(‘A’).add(‘B’)`
- `aList.get(‘A’).add(‘C’)`
- …

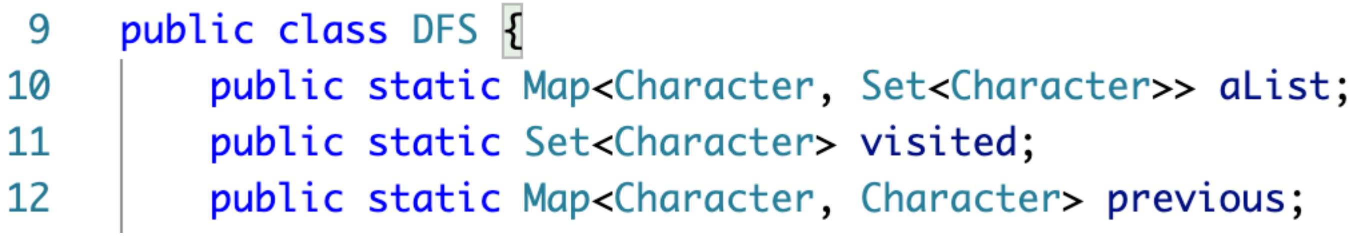
`O(1)` time to check if nodes are connected or get the neighbors of a node (assuming good hashCode)

Graph Search Data Structures

1) Have an adjacency list for the graph

2) Keep track of visited nodes in a set

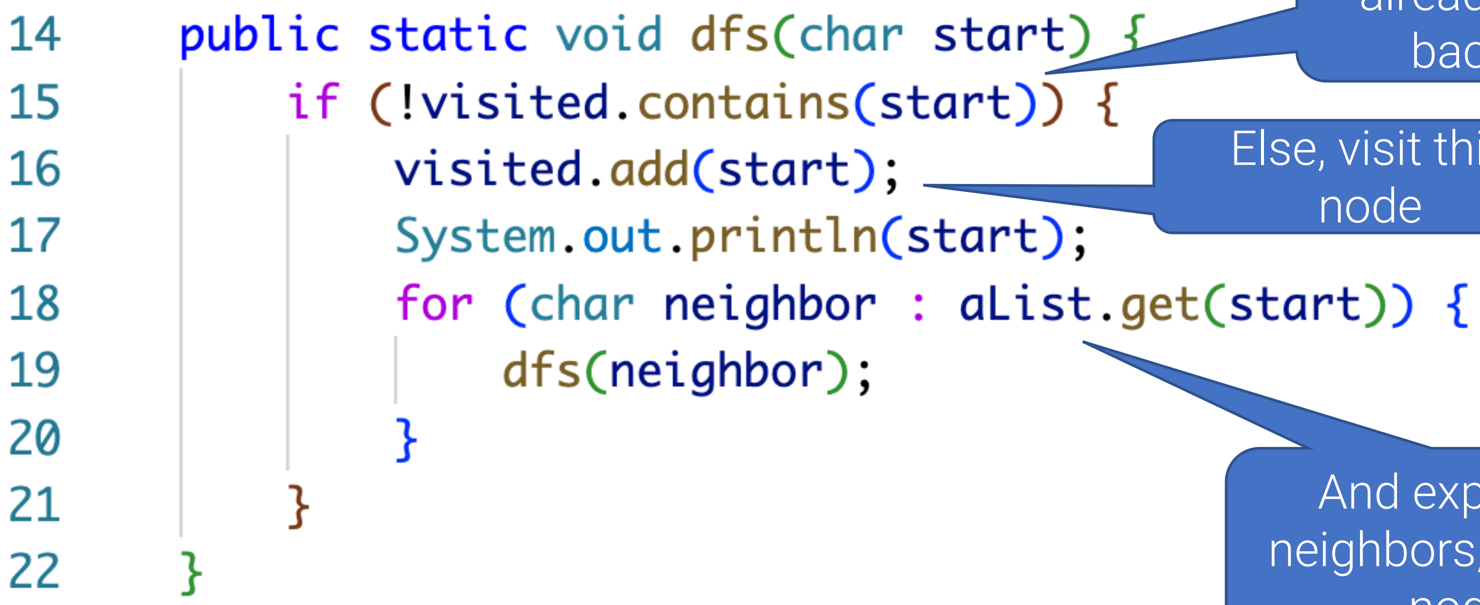
3) Keep track of the *previous* node: During search, how did I get to this node?



```
public class DFS {
    public static Map<Character, Set<Character>> aList;
    public static Set<Character> visited;
    public static Map<Character, Character> previous;
```

- Example has Character nodes, could be any label for the nodes.
- Storing as instance variables, accessible in methods.

Recursive DFS on a General Graph: Visiting all nodes



```java
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

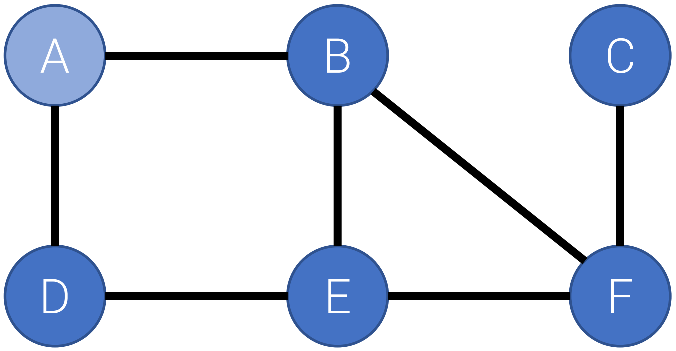
Base case: If already visited, backtrack

Else, visit this node

And explore its neighbors, adjacent nodes

Initialize search at A

start: A

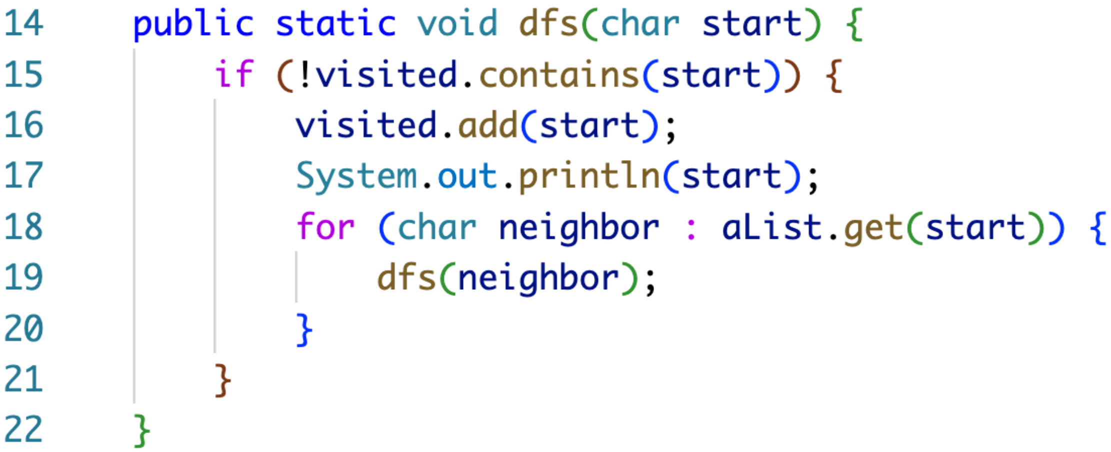


Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)

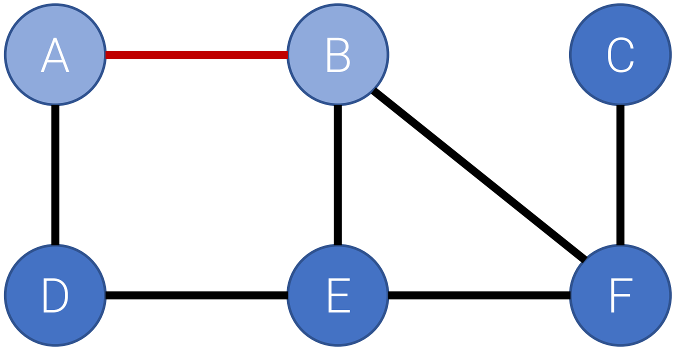
{A}



```java
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Recurse on B

start: A

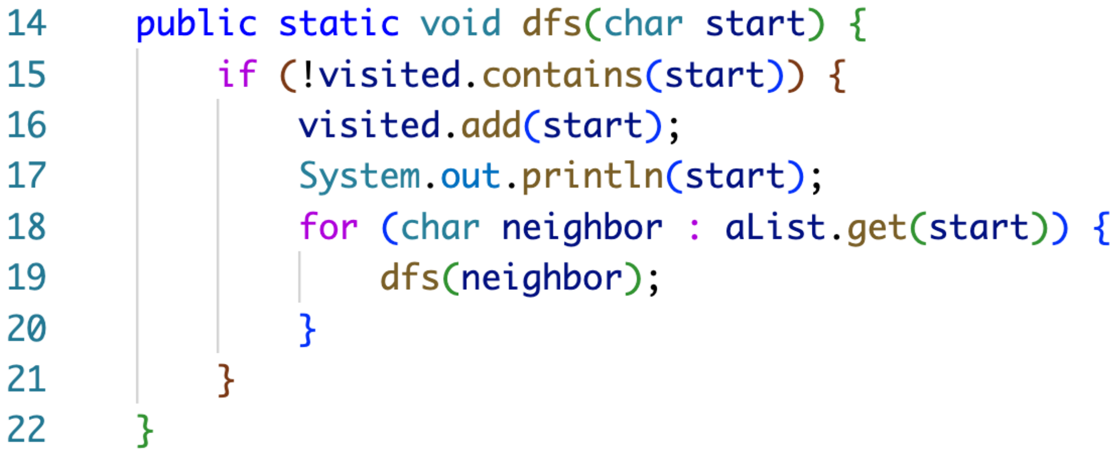


Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)

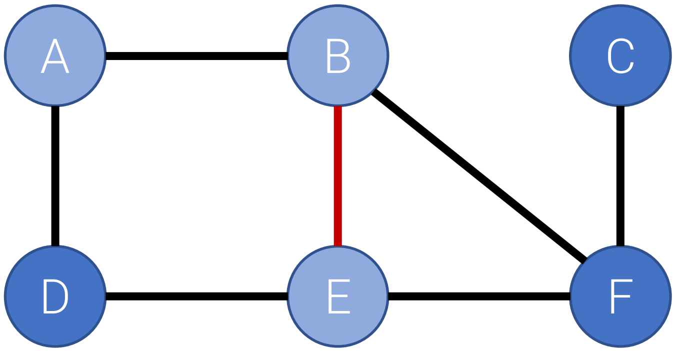
{A, B}



```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Recurse on E

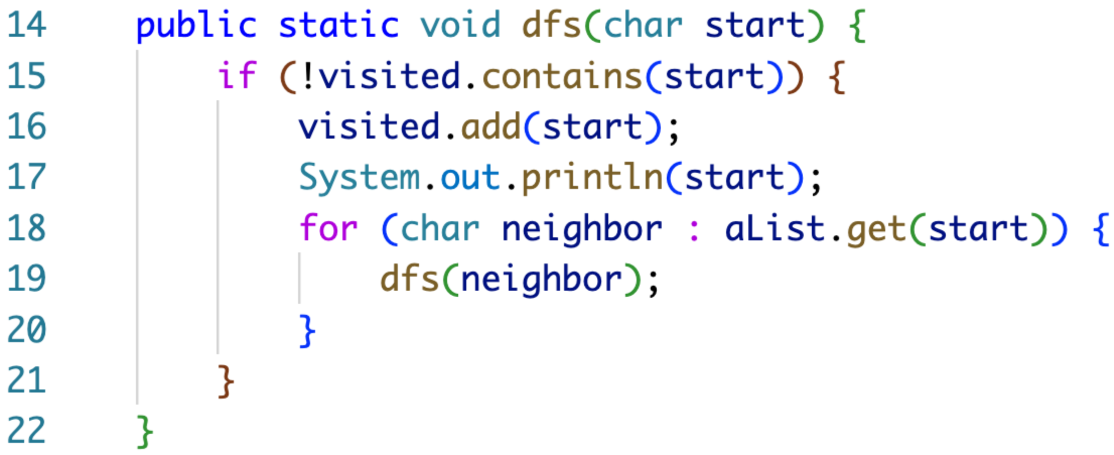
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)

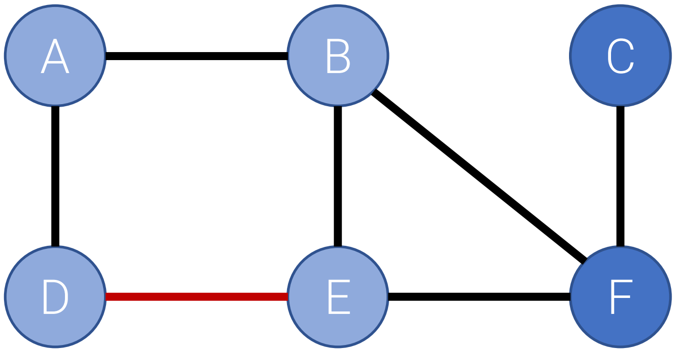
{A, B, E}



```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Recurse on D

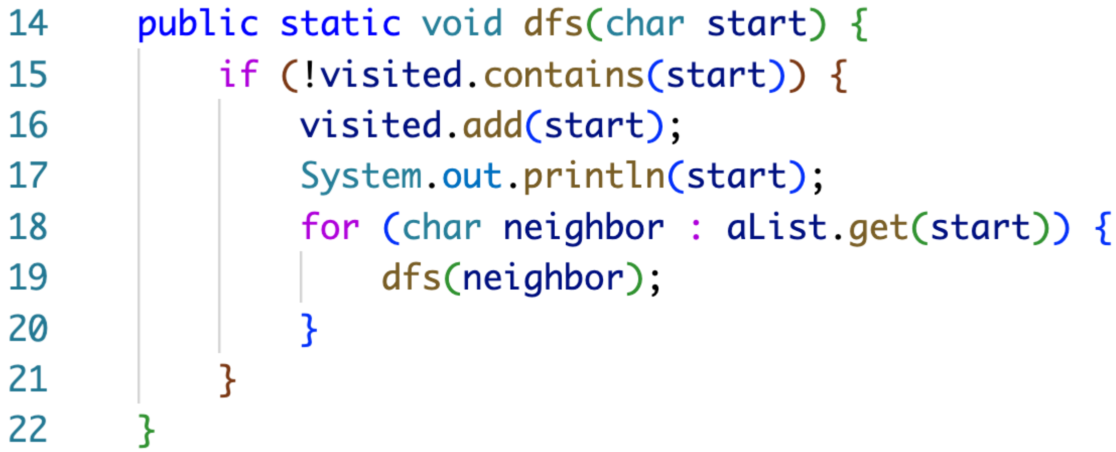
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)

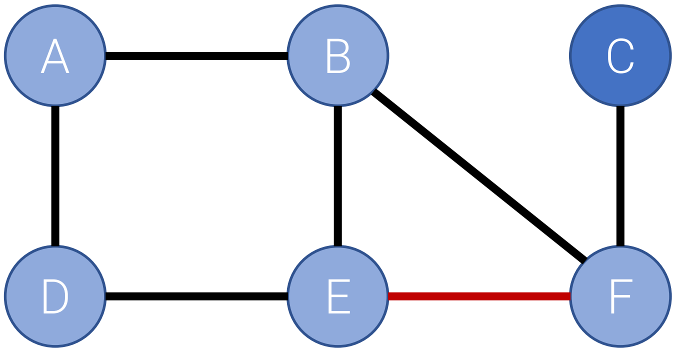
{A, B, E, D}



```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Backtrack to E, recurse on F

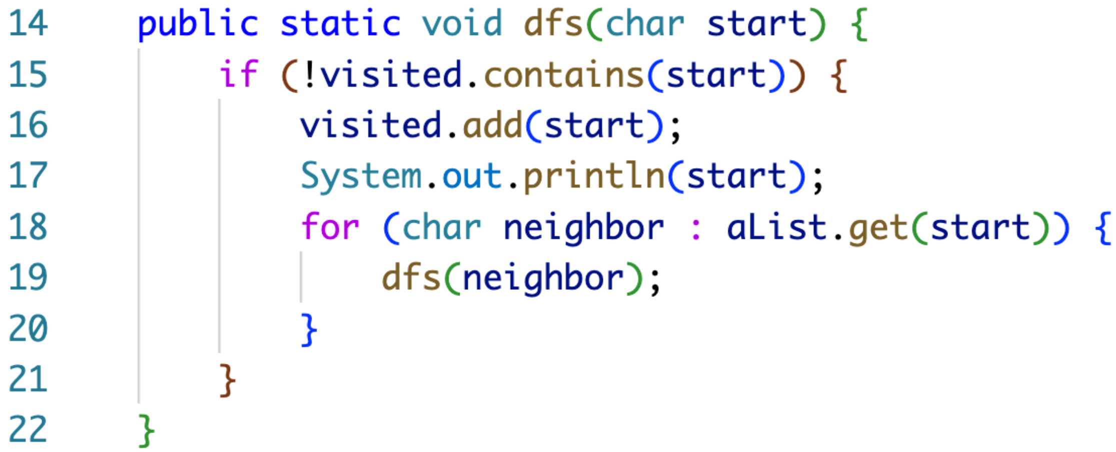
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)

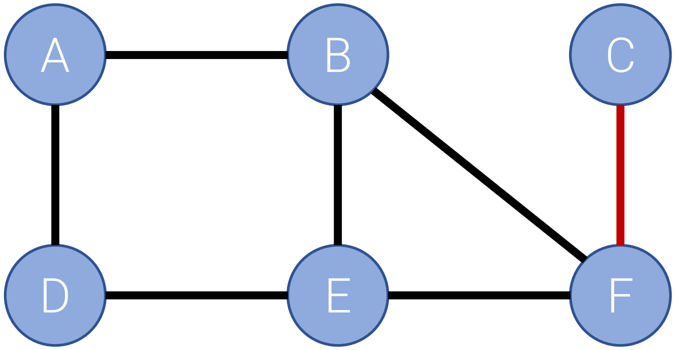
{A, B, E, D, F}



```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Recurse on C

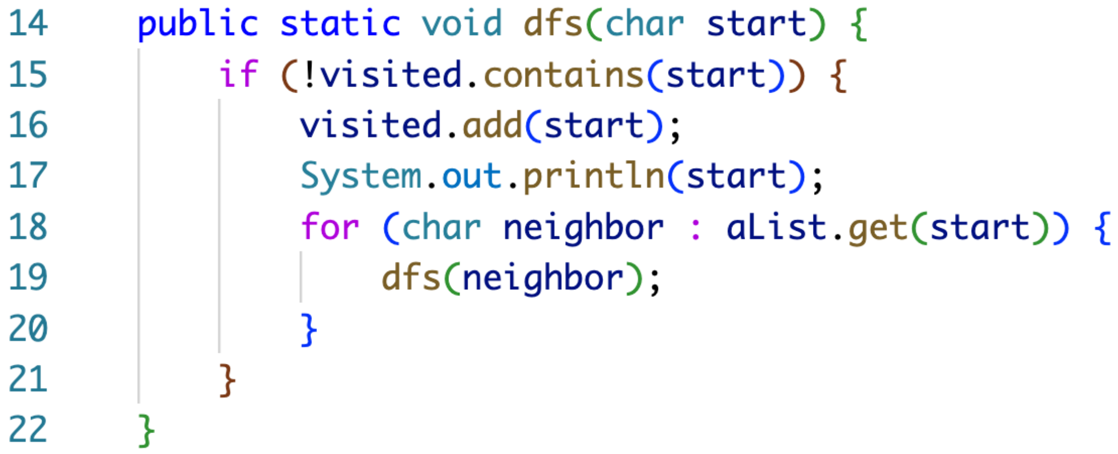
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

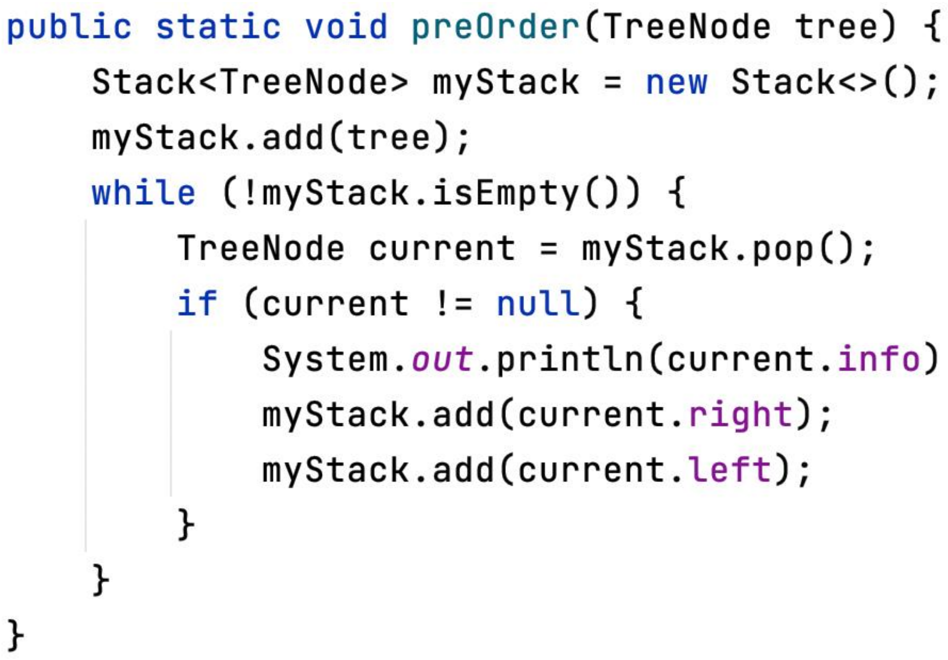
Visited (set)

{A, B, E, D, F, C}

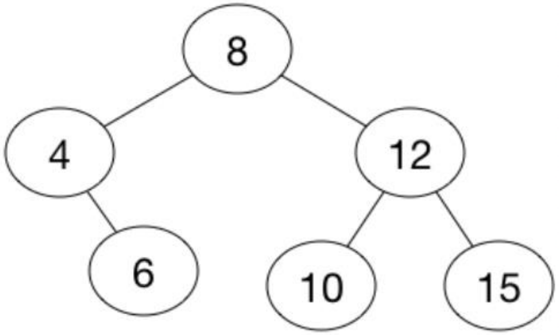


```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```

Did we really need recursion? preOrder Tree Traversal with Stack



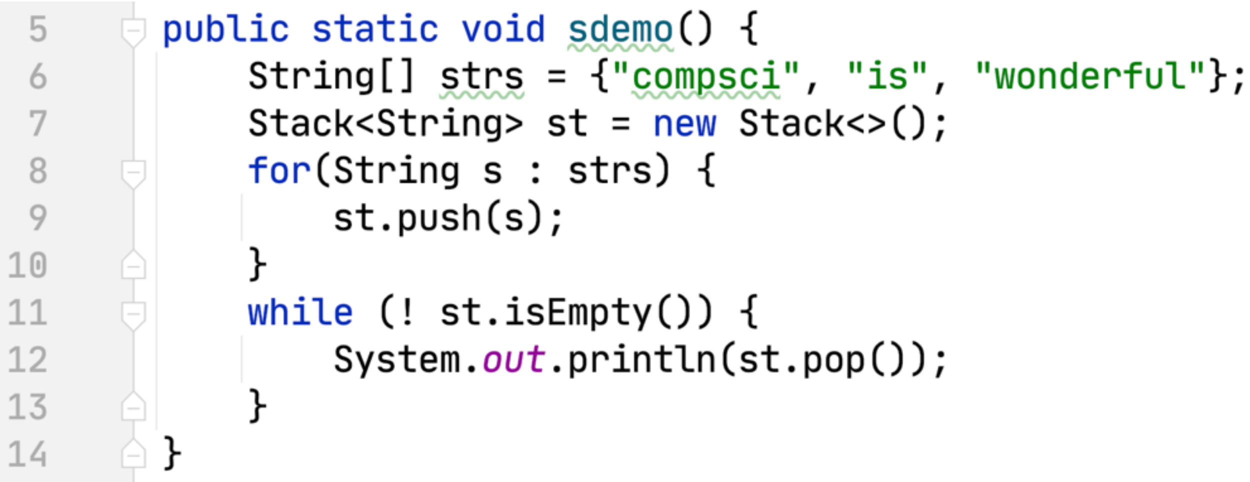
```
public static void preOrder(TreeNode tree) {
    Stack<TreeNode> myStack = new Stack<>();
    myStack.add(tree);
    while (!myStack.isEmpty()) {
        TreeNode current = myStack.pop();
        if (current != null) {
            System.out.println(current.info)
            myStack.add(current.right);
            myStack.add(current.left);
        }
    }
}
```



8
4
6
12
10
15

Recursion uses the call stack to keep track of nodes
Could also explicitly use a stack, can do the same for DFS

Stack Abstract Data Structure: LIFO List



```
public static void sdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Stack<String> st = new Stack<>();
    for(String s : strs) {
        st.push(s);
    }
    while (! st.isEmpty()) {
        System.out.println(st.pop());
    }
}
```

```
wonderful
is
compsci
```

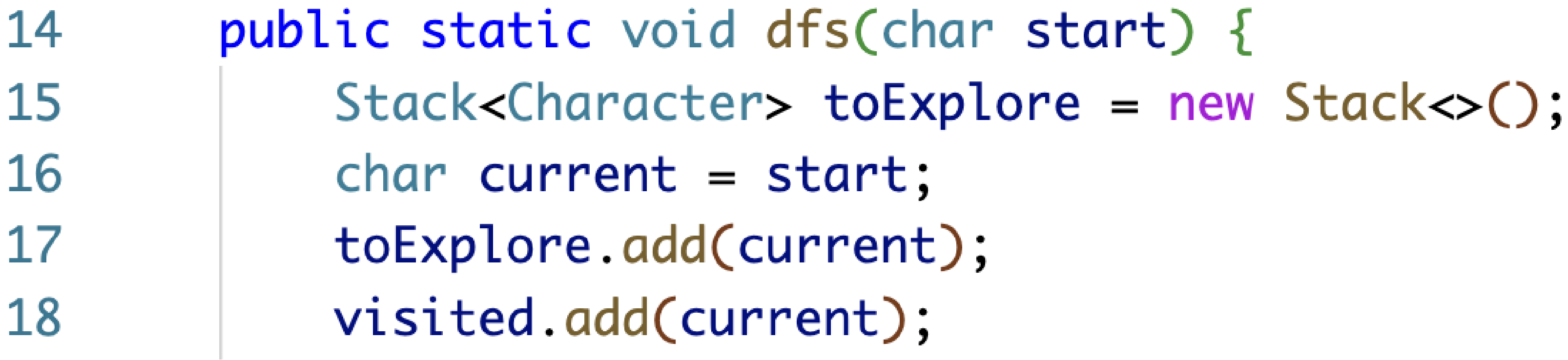
LIFO = Last In First Out

Push: Add element to stack

Pop: Get last element in

Initializing Iterative DFS

- **Stack** stores nodes we have *visited/discovered*, but not explored from yet.
- Explore from one *current* node at a time.



```
    public static void dfs(char start) {
        Stack<Character> toExplore = new Stack<>();
        char current = start;
        toExplore.add(current);
        visited.add(current);
```

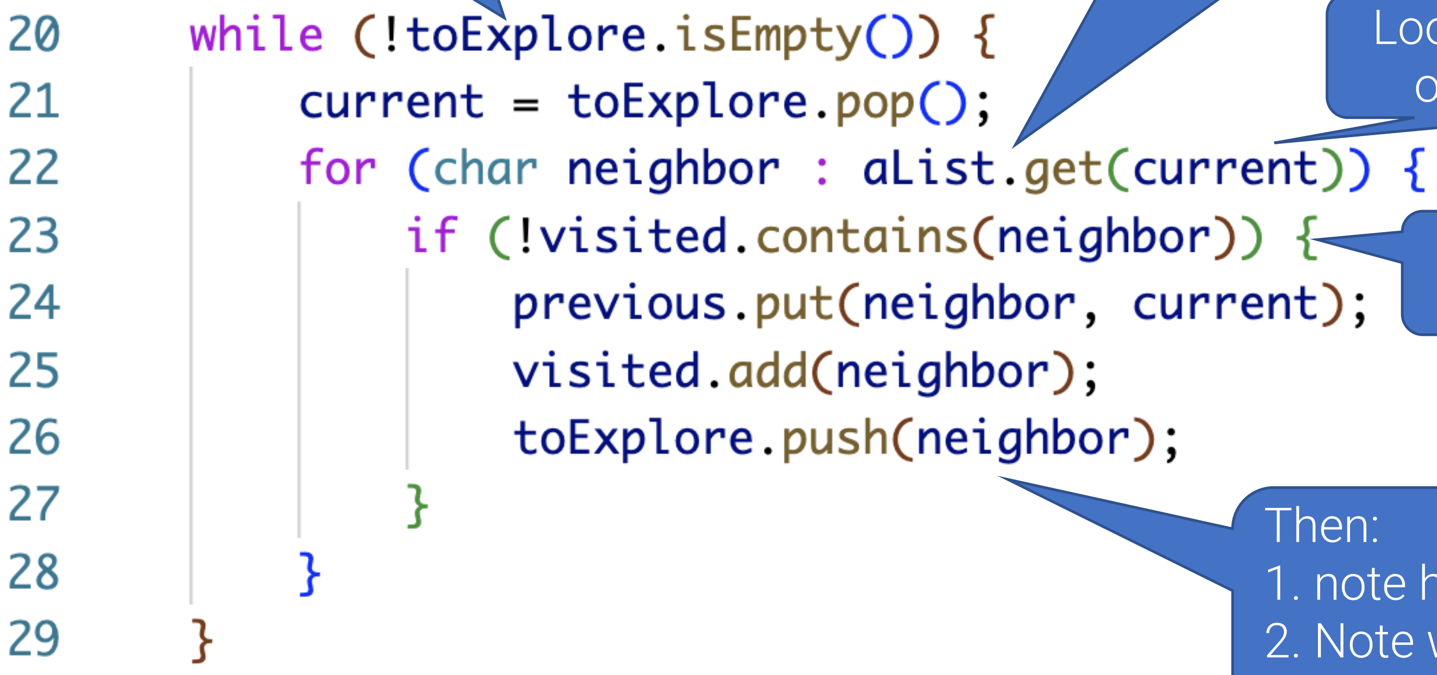
- Stack is LIFO (last-in first-out), so we always explore from the *last node we discovered*, **depth-first**!

Iterative DFS Loop

While there are nodes we have not explored from…

Explore from the most recently discovered node…

Look at all neighbors of current node…

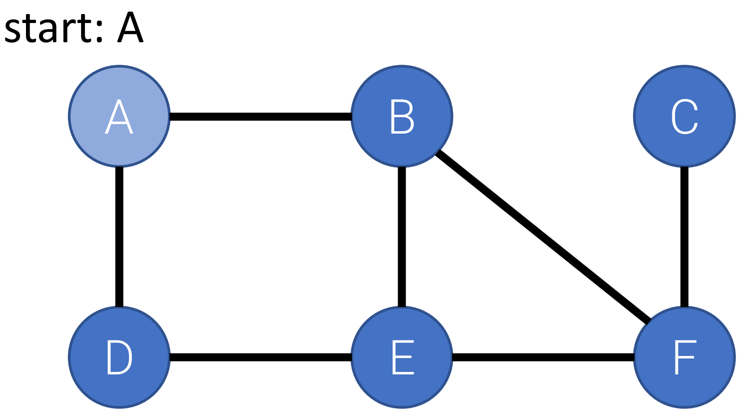


```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

If we haven't seen them before…

Then:
1. note how we got here
2. Note we have seen
3. Mark to explore later

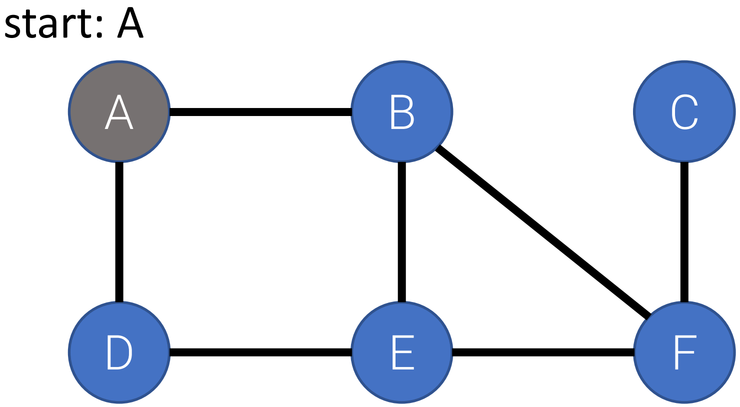
Initialize search at A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
A		{A}

Pop A off the stack



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

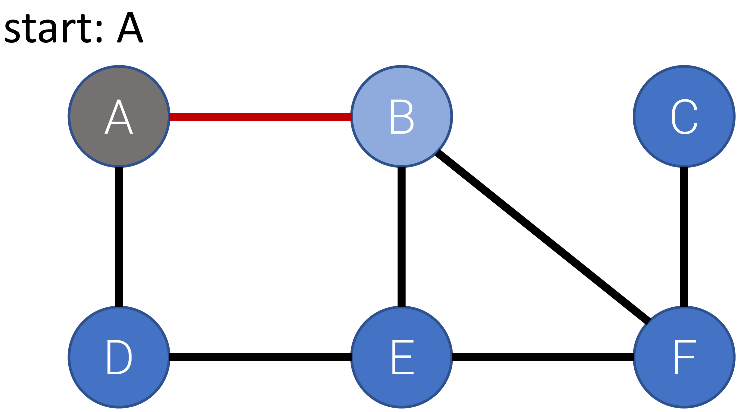
toExplore (stack)

previous (map)

Visited (set)

{A}

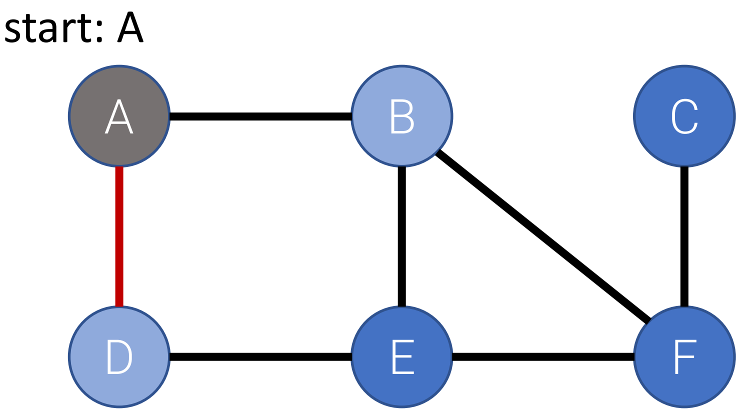
Find B from A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
B	B <- A	{A, B}

Find D from A

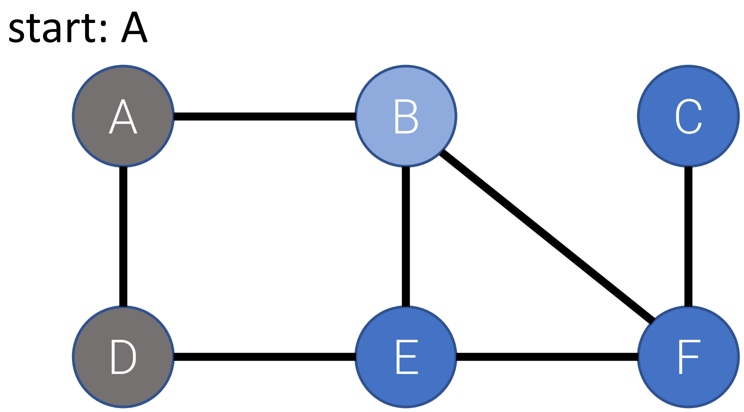


Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
D	B <- A	{A, B, D}
B	D <- A	

Pop D off the stack

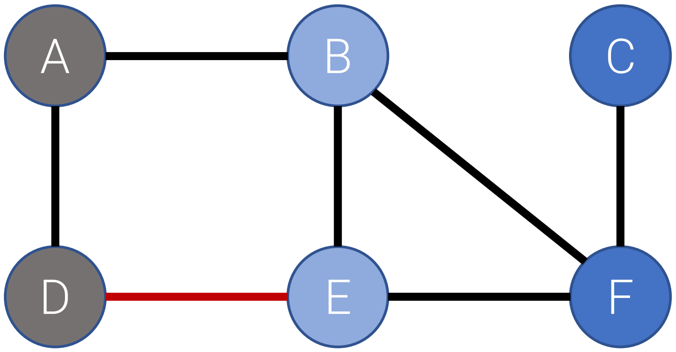


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
B	B <- A	{A, B, D}
	D <- A	

Find E from D

start: A

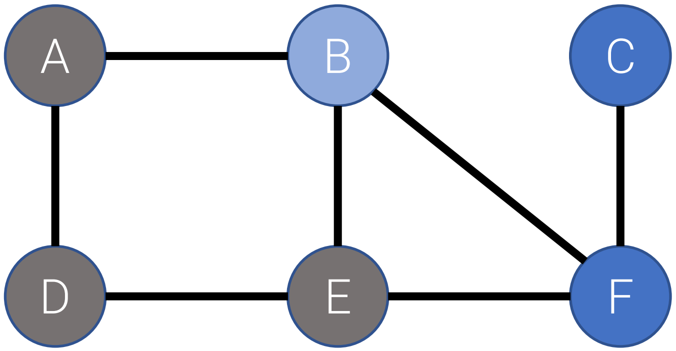


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
E	B <- A	{A, B, D, E}
B	D <- A	
	E <- D	

Pop E off the stack

start: A

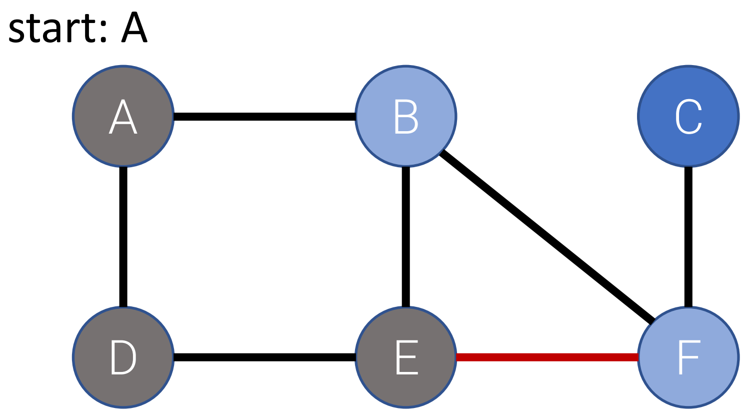


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)

B	B <- A
	D <- A
	E <- D
		{A, B, D, E}

Find F from E

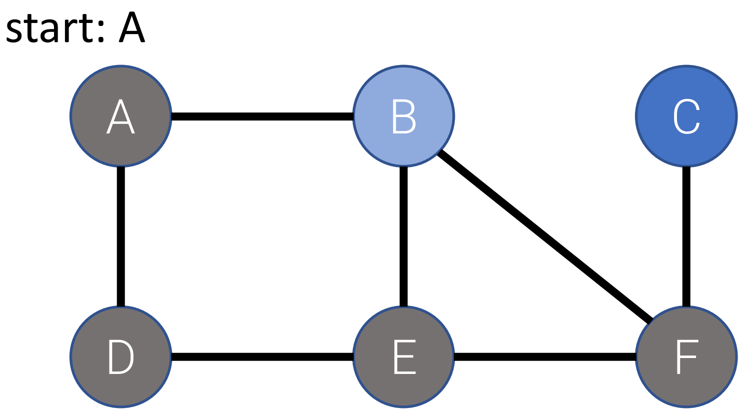


Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
F	B <- A	{A, B, D, E, F}
B	D <- A	
	E <- D	
	F <- E	

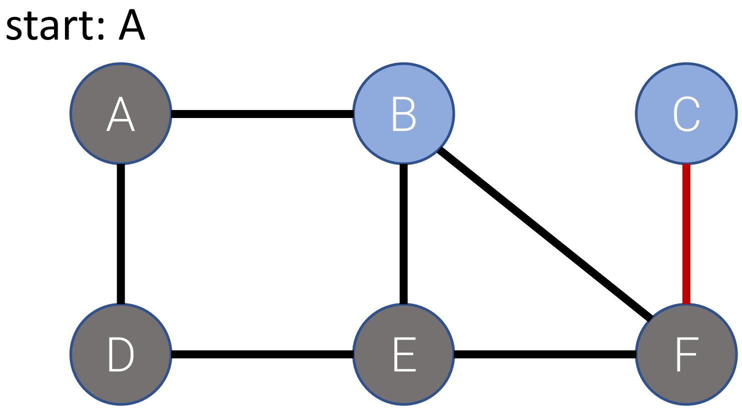
Pop F off the stack



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
B	B <- A	{A, B, D, E, F}
	D <- A	
	E <- D	
	F <- E	

Find C from F

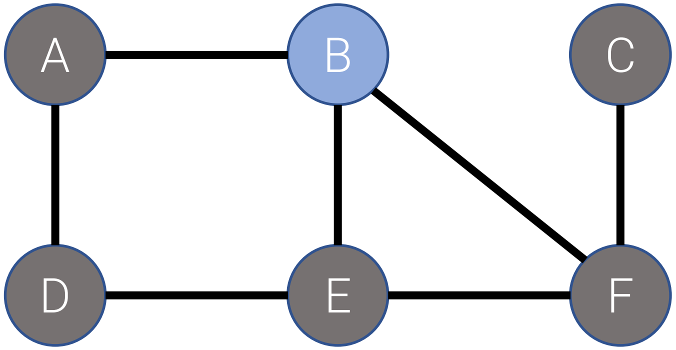


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
C	B <- A	{A, B, D, E, F, C}
B	D <- A	
	E <- D	
	F <- E	
	C <- F	

Pop C off the stack

start: A



Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)

B

previous (map)

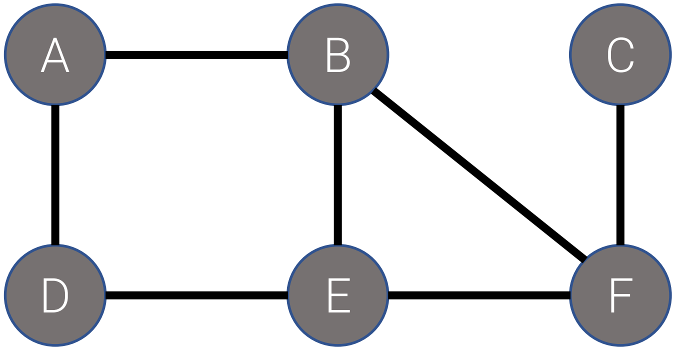
B <- A
D <- A
E <- D
F <- E
C <- F

Visited (set)

{A, B, D, E, F, C}

Pop B off the stack

start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)

previous (map)

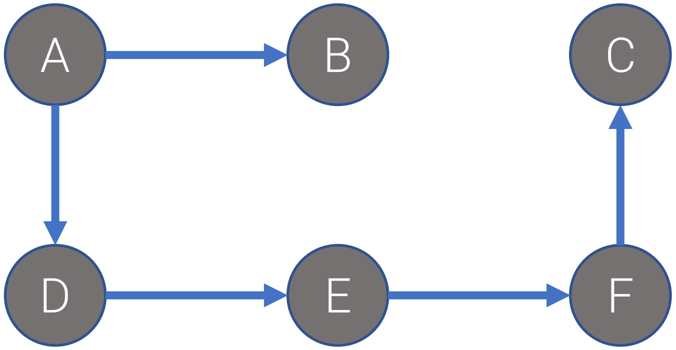
B <- A
D <- A
E <- D
F <- E
C <- F

Visited (set)

{A, B, D, E, F, C}

DFS Search Tree

start: A



Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)

previous (map)

B <- A
D <- A
E <- D
F <- E
C <- F

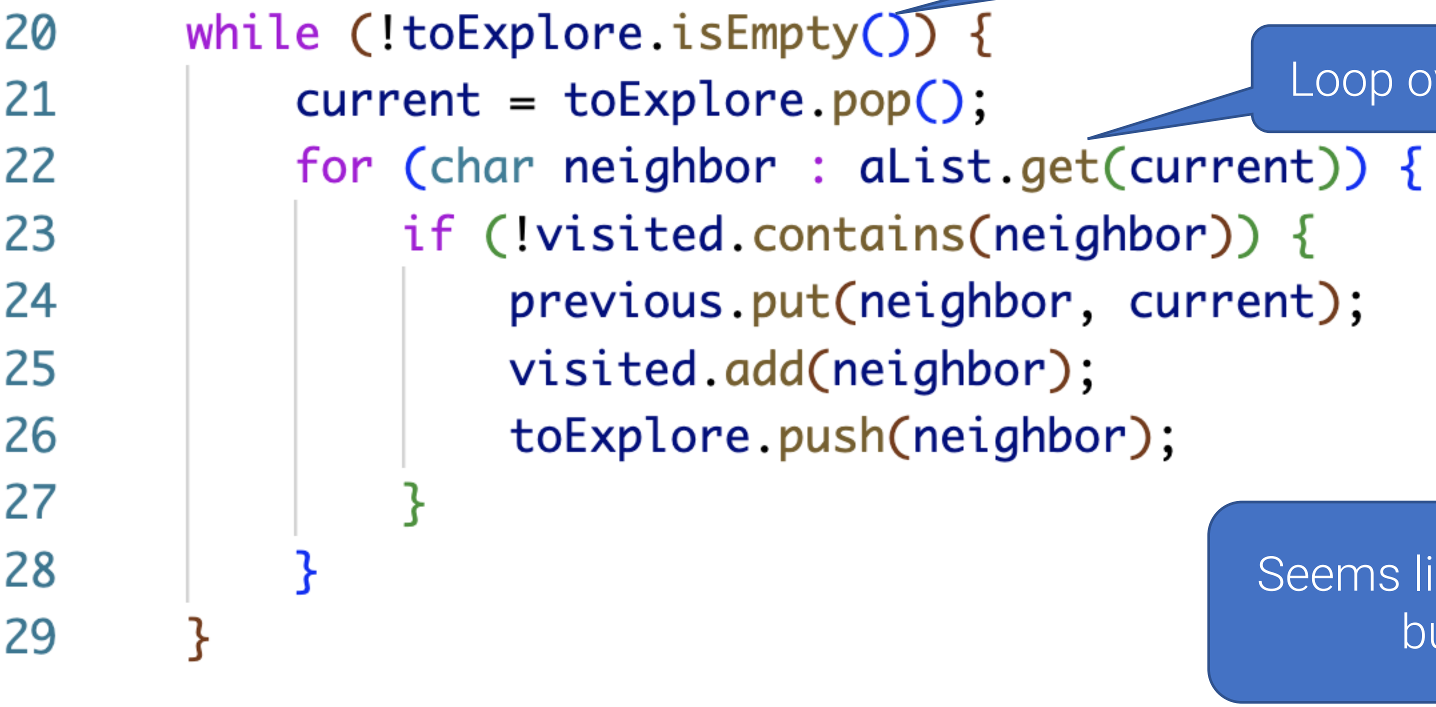
Visited (set)

{A, B, D, E, F, C}

Can find paths from A to X by following previous backwards from X

Path from A to C:
C <- F <- E <- D <- A

DFS Complexity?



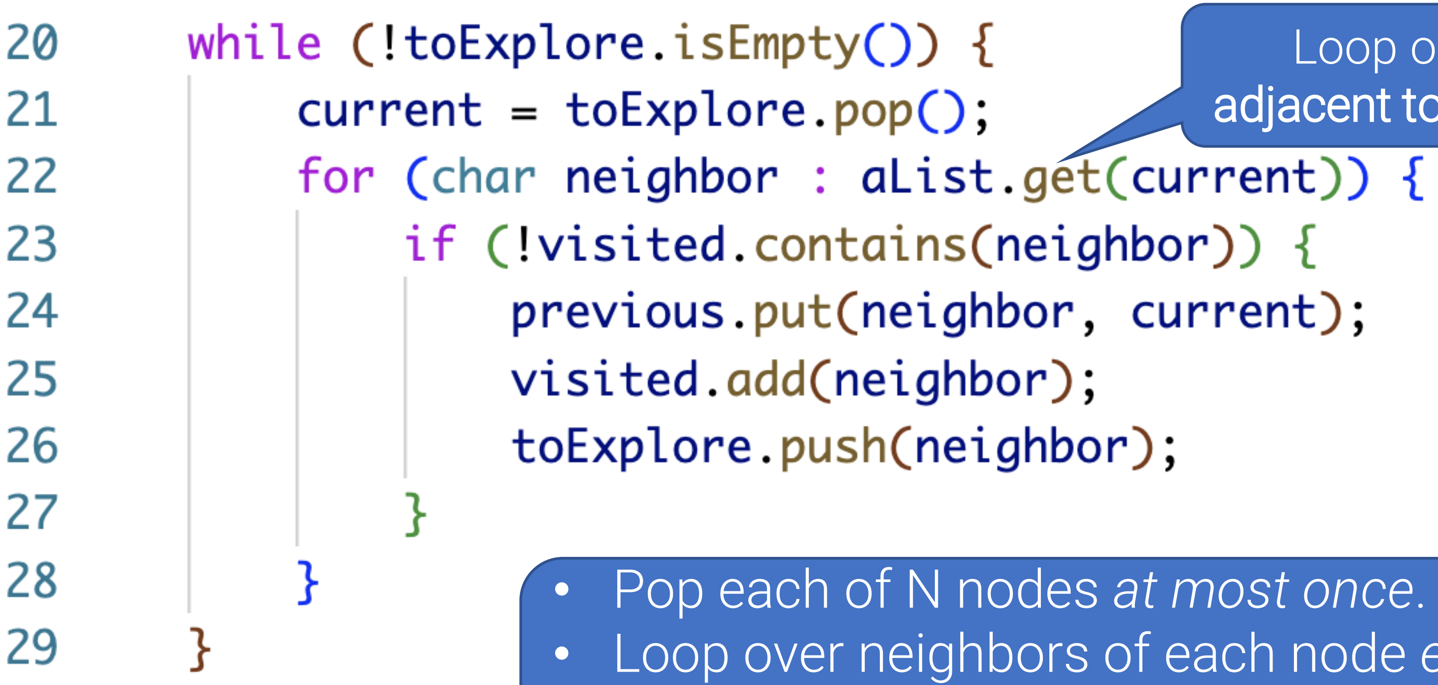
```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

While loop over all nodes (N), potentially?

Loop over edges (M)

Seems like O(NM), but…

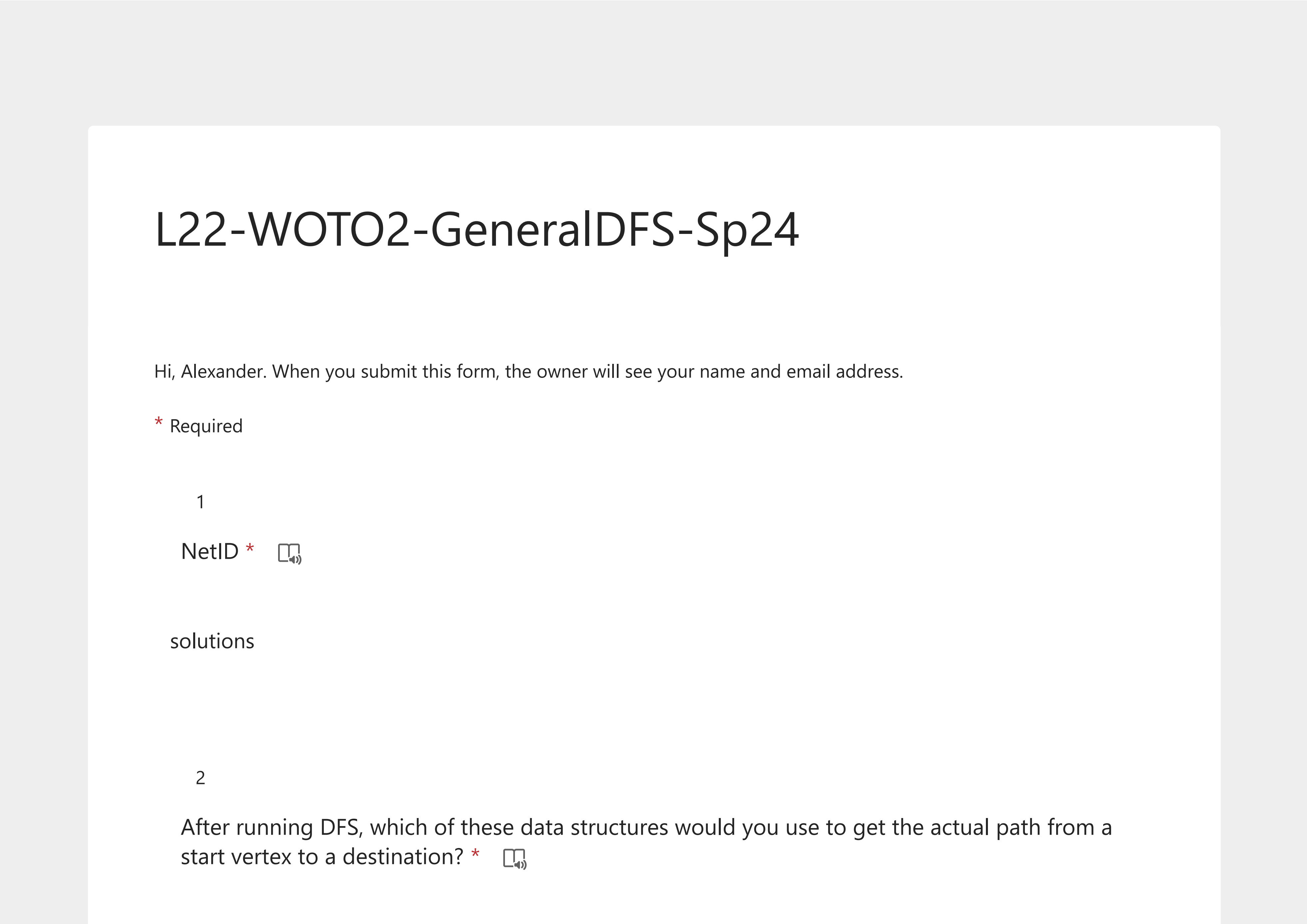
DFS Complexity?



```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

Loop over edges
adjacent to current node

- Pop each of N nodes *at most once*.
- Loop over neighbors of each node *exactly once*, considers each edge twice.
- N+2M is O(N+M).



L22-WOTO2-GeneralDFS-Sp24

Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

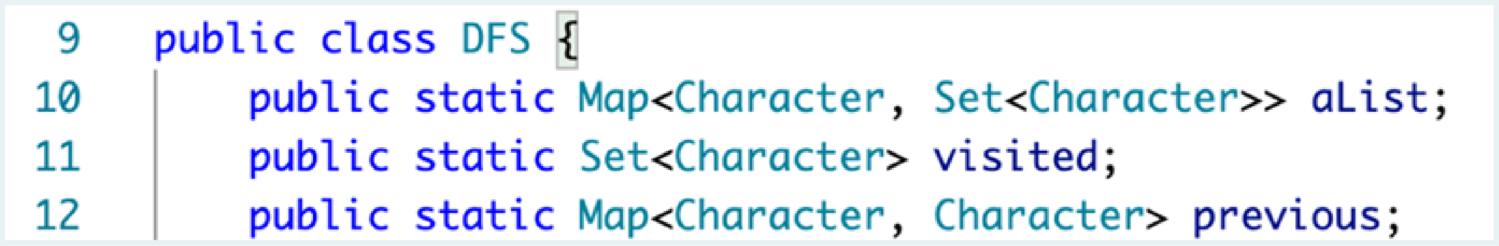
1

NetID *

solutions

2

After running DFS, which of these data structures would you use to get the actual path from a start vertex to a destination? *



```
 9   public class DFS {
10       public static Map<Character, Set<Character>> aList;
11       public static Set<Character> visited;
12       public static Map<Character, Character> previous;
```

○ aList

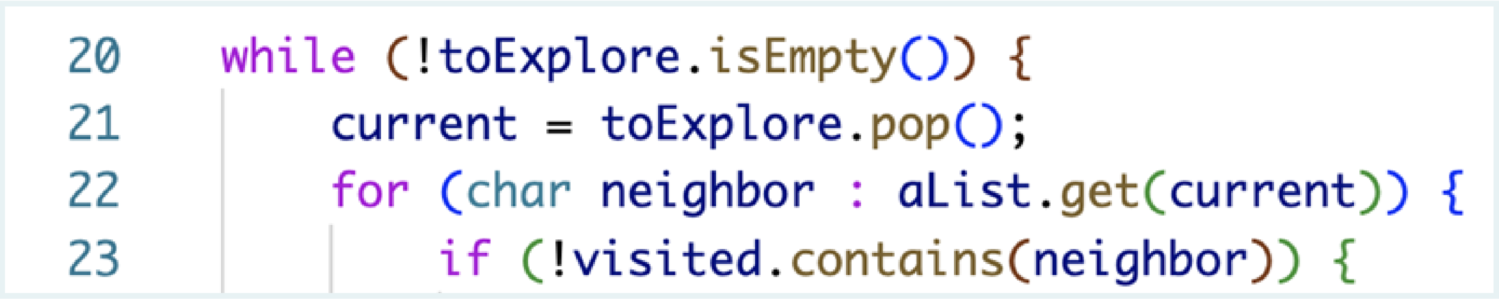
○ visited

● previous

○ none of the above

3

The best explanation of the loop on line 22 is... *



```
20   while (!toExplore.isEmpty()) {
21       current = toExplore.pop();
22       for (char neighbor : aList.get(current)) {
23           if (!visited.contains(neighbor)) {
```

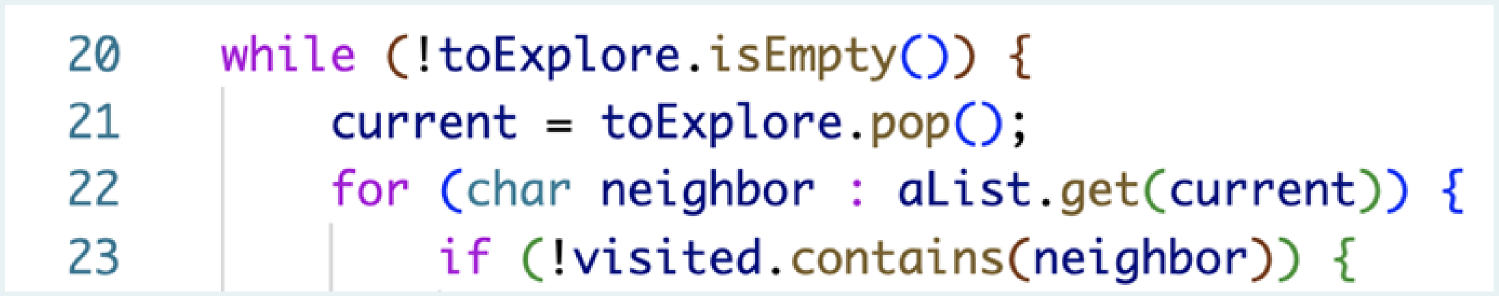
○ Check all nodes reachable by one edge from any visited nodes

● Check all nodes reachable by one edge from the node we are exploring

○ Check all of the unvisited nodes

4

Same code. The while loop on line 20 might have fewer than N iterations (when there are N nodes in the graph) when... *



```
20    while (!toExplore.isEmpty()) {
21        current = toExplore.pop();
22        for (char neighbor : aList.get(current)) {
23            if (!visited.contains(neighbor)) {
```

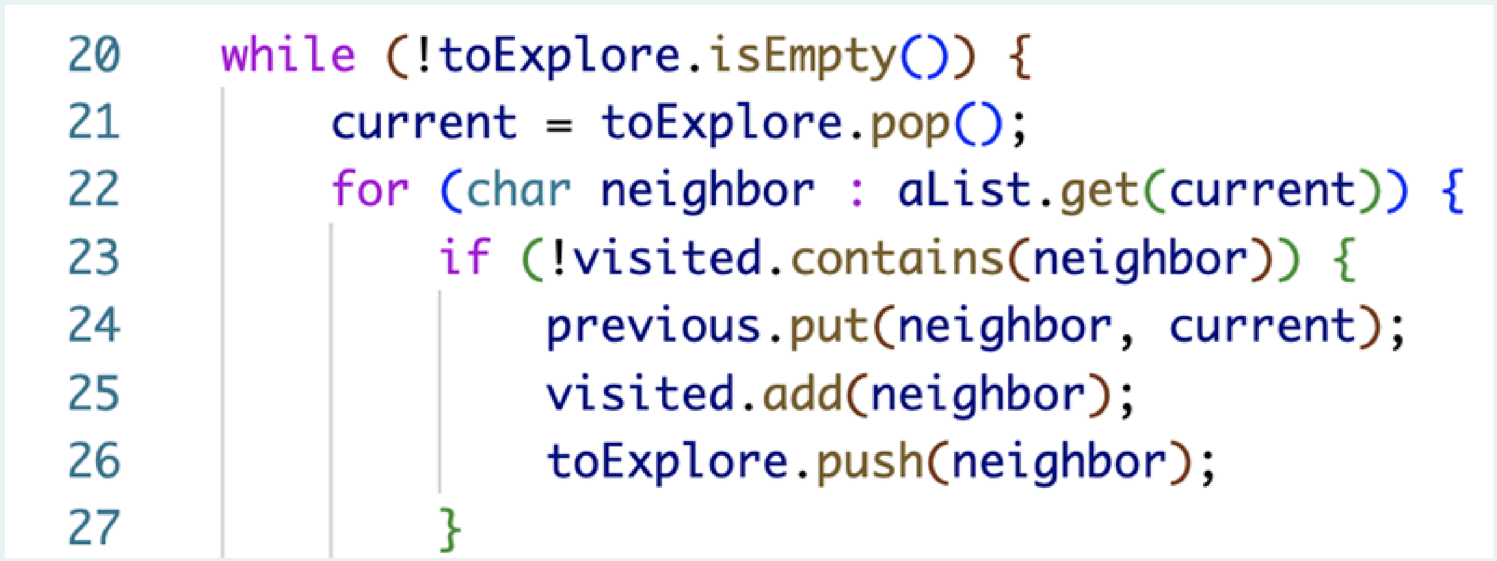
○ Some nodes are connected to many other nodes in the graph

● Some nodes are not reachable from others

○ Never, the while loop should always have N iterations

5

What best describes the runtime complexity of DFS using a stack and hash-based data structures? Let N be the number of vertices and M be the number of edges. *



```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
```

○ O(N)

● O(N+M)

○ O(NM)

6

True or false: This dfs algorithm will always find the shortest path from the start node to other nodes *

○ True

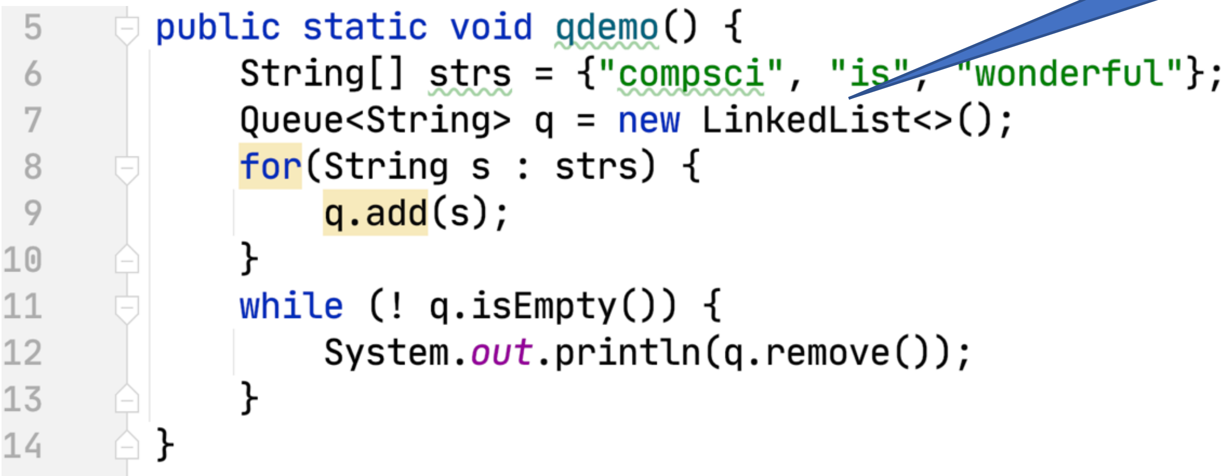
● False

Iterative Breadth-First Search (BFS)

Queue: A FIFO List

- Both add and remove are O(1)
  - Add at end of LinkedList
  - Remove from front of LinkedList

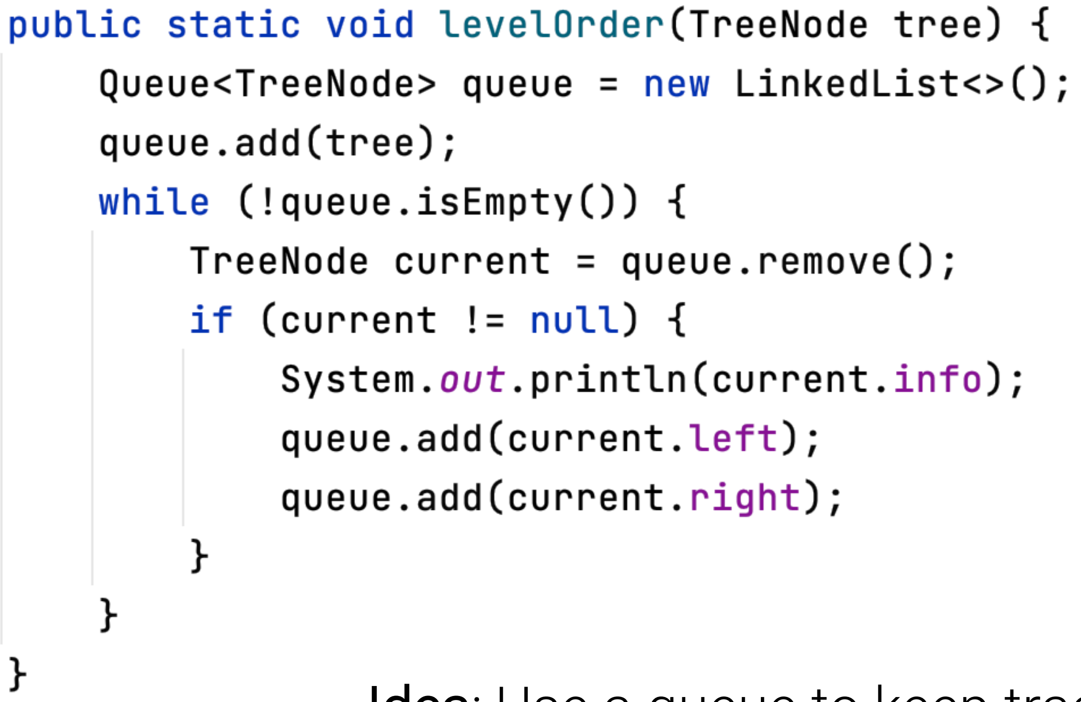
LinkedList implements the Queue interface.



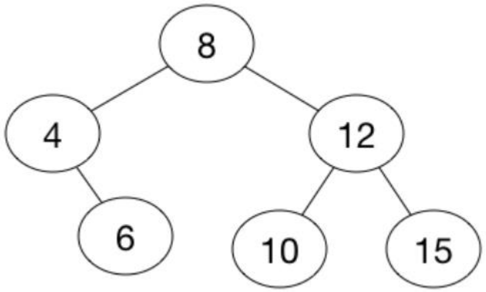
```java
public static void qdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Queue<String> q = new LinkedList<>();
    for(String s : strs) {
        q.add(s);
    }
    while (! q.isEmpty()) {
        System.out.println(q.remove());
    }
}
```

```
compsci
is
wonderful
```

Level Order Tree Traversal using a Queue



```java
public static void levelOrder(TreeNode tree) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(tree);
    while (!queue.isEmpty()) {
        TreeNode current = queue.remove();
        if (current != null) {
            System.out.println(current.info);
            queue.add(current.left);
            queue.add(current.right);
        }
    }
}
```



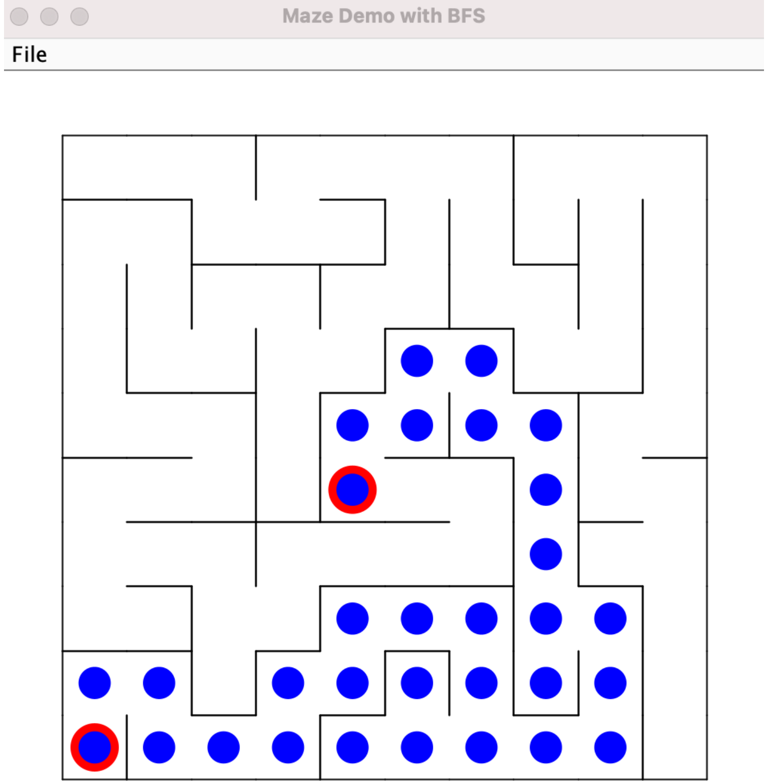
```
8
4
12
6
10
15
```

<u>Idea</u>: Use a queue to keep track of nodes.
First-in first-out, nodes visited in *level order*

Depth-First Search for Solving Maze

Always explore (recurse on) a new (unvisited) adjacent vertex if possible.

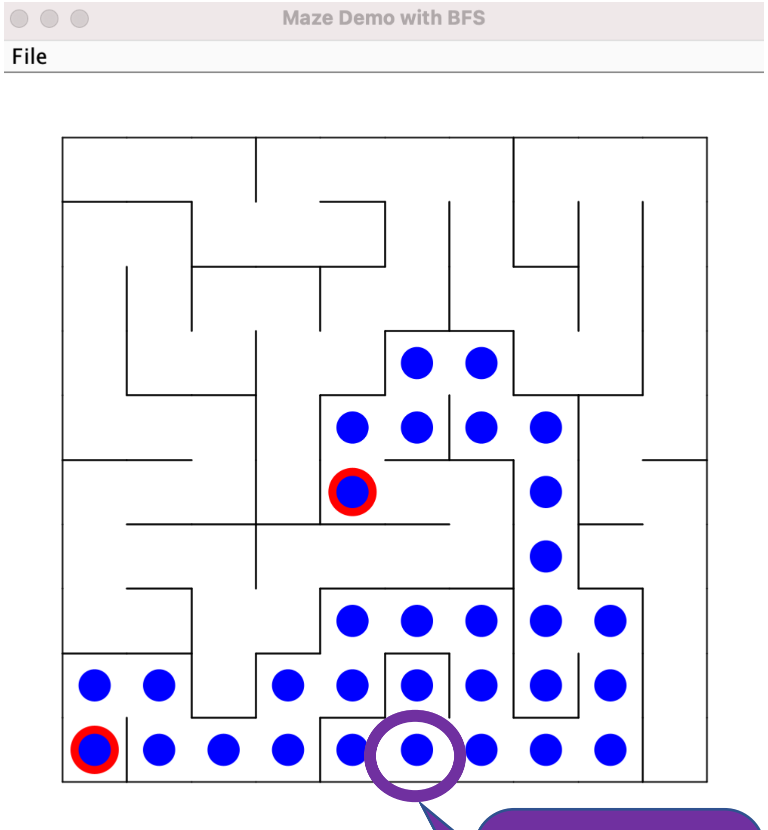
If impossible, **backtrack** to the most recent vertex adjacent to an unvisited vertex and continue.



Breadth-First Search for Solving Maze

Explore *all* your neighbors (adjacent vertices) before you visit any of your neighbors' neighbors.

Looking for the shortest path/solution.

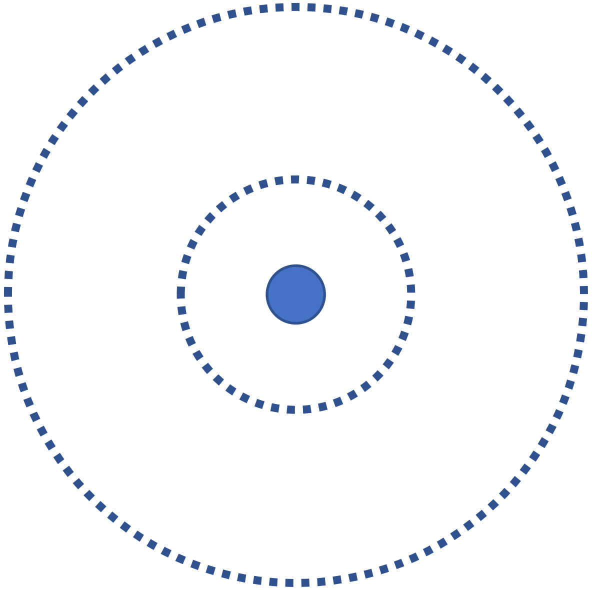


DFS never looked here!

Queue = BFS, Stack = DFS

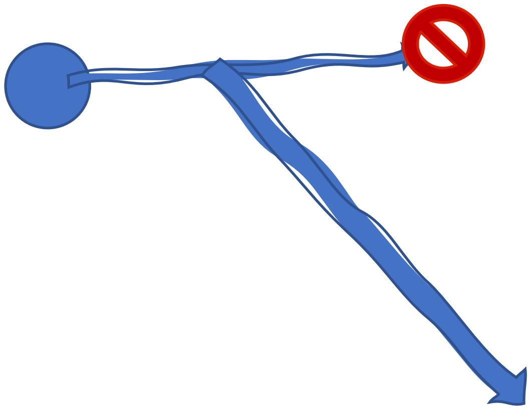
BFS: FIFO Exploration

search all locations one-away from start, then two-away, …



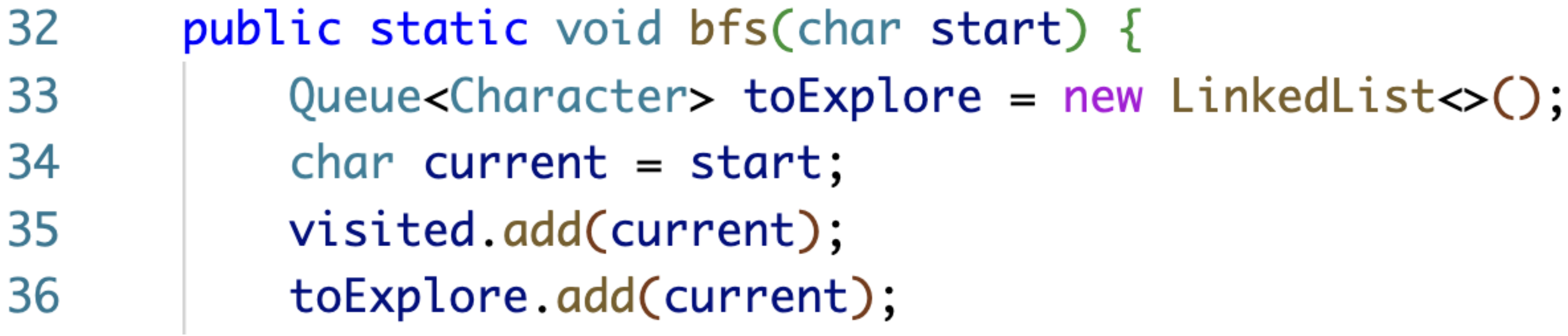
DFS: LIFO Exploration

Search path as far as possible, backtrack if need to another branch…



Initializing Iterative BFS

- **Queue** stores nodes we have *visited/discovered*, but not explored from yet.
- Explore from one *current* node at a time.



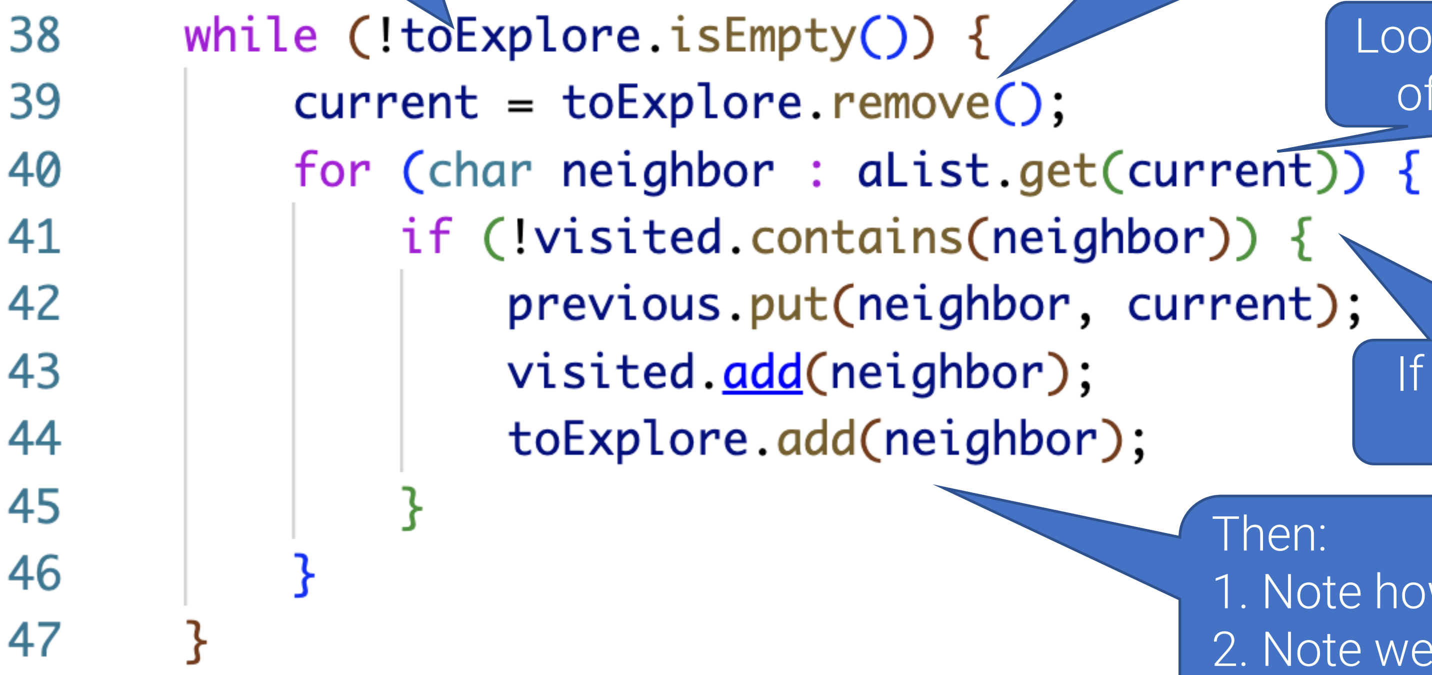
```
    public static void bfs(char start) {
        Queue<Character> toExplore = new LinkedList<>();
        char current = start;
        visited.add(current);
        toExplore.add(current);
```

- Queue is FIFO (first-in first-out), so we always explore from the *first/closest (unvisited) node we discovered*, **breadth-first**!

Iterative BFS Loop

While there are nodes we have not explored from…

Explore from the **closest** discovered node…



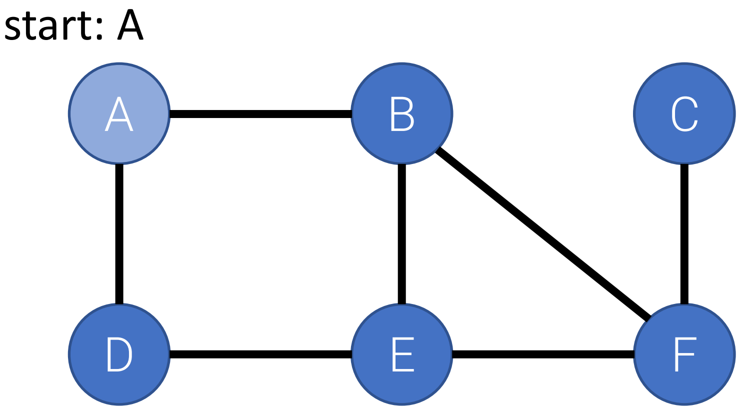
```
while (!toExplore.isEmpty()) {
    current = toExplore.remove();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.add(neighbor);
        }
    }
}
```

Look at all neighbors of current node…

If we haven't seen them before…

Then:
1. Note how we got here
2. Note we have seen
3. Mark to explore later

Initialize search at A

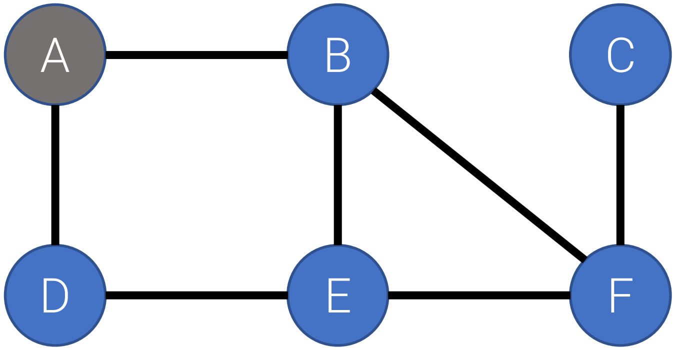


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
A		{A}

Remove A from the queue

start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

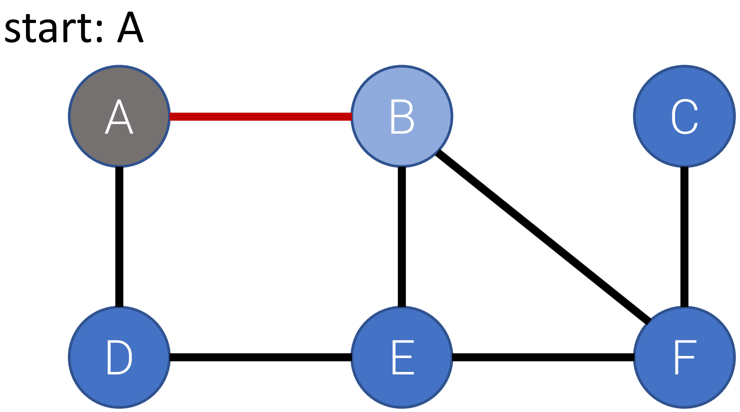
toExplore (queue)

previous (map)

Visited (set)

{A}

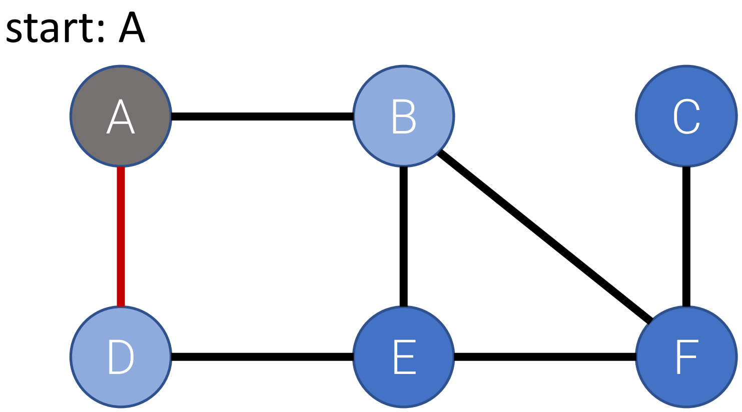
Find B from A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
B	B <- A	{A, B}

Find D from A

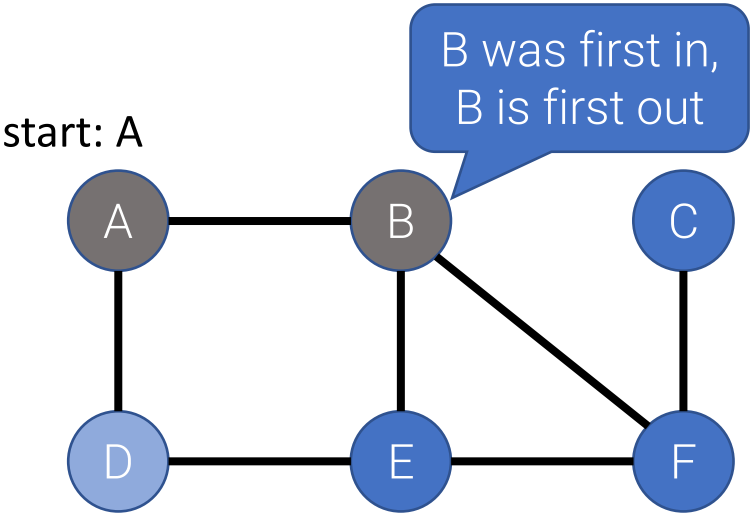


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
B	B <- A	{A, B, D}
D	D <- A	

Note the difference, add to end of queue!

Remove B from queue

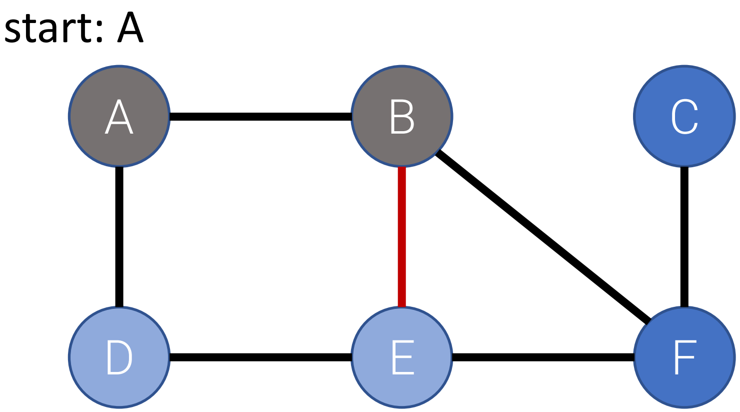


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)

D	B <- A
	D <- A
		{A, B, D}

Find E from B



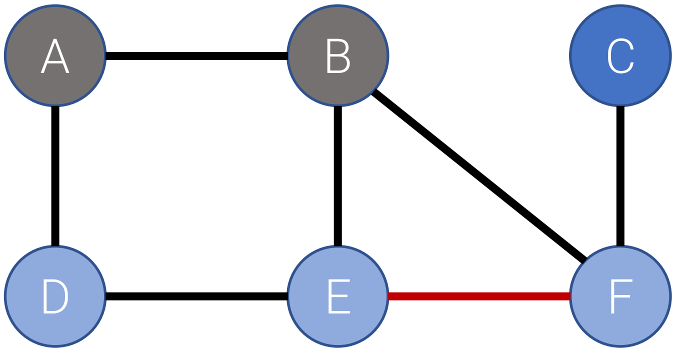
Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
D	B <- A	{A, B, D, E}
E	D <- A	
	E <- B	

Find F from B

start: A

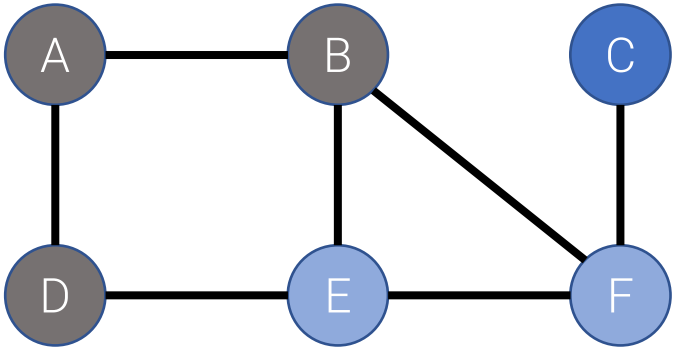


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
D	B <- A	{A, B, D, E, F}
E	D <- A	
F	E <- B	
	F <- B	

Remove D from queue

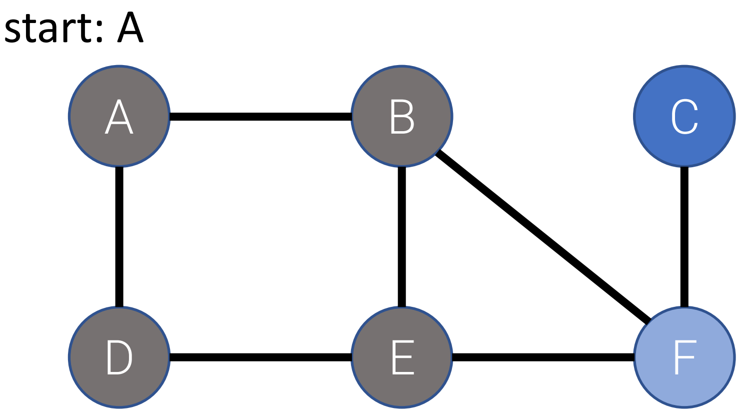
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)
E	B <- A	{A, B, D, E, F}
F	D <- A	
	E <- B	
	F <- B	

Remove E from queue



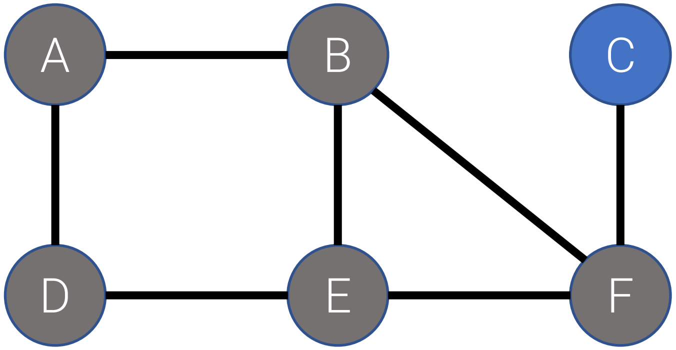
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)

F	B <- A
	D <- A
	E <- B
	F <- B
		{A, B, D, E, F}

Remove F from queue

start: A



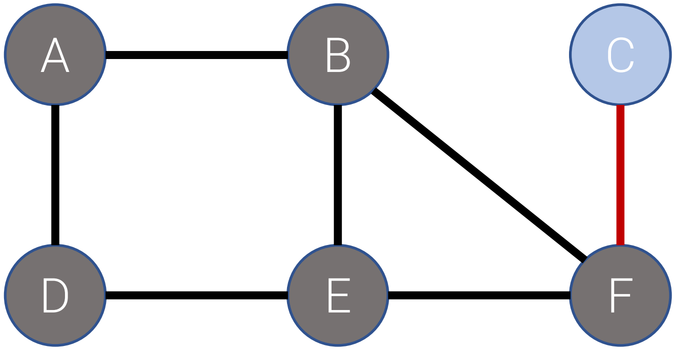
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)	**previous** (map)	**Visited** (set)

	B <- A
	D <- A
	E <- B
	F <- B
		{A, B, D, E, F}

Find C from F

start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)

C

previous (map)

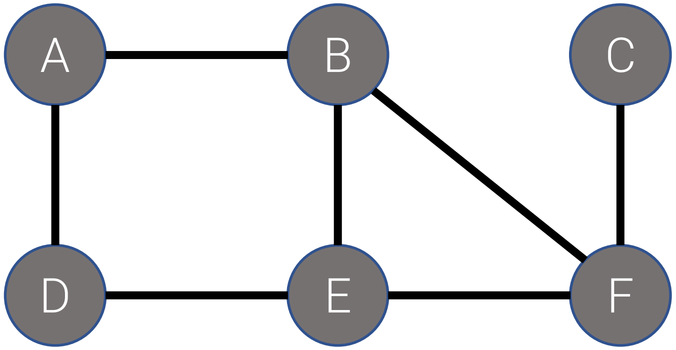
B <- A
D <- A
E <- B
F <- B
C <- F

Visited (set)

{A, B, D, E, F, C}

Remove C from queue

start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (queue)

previous (map)

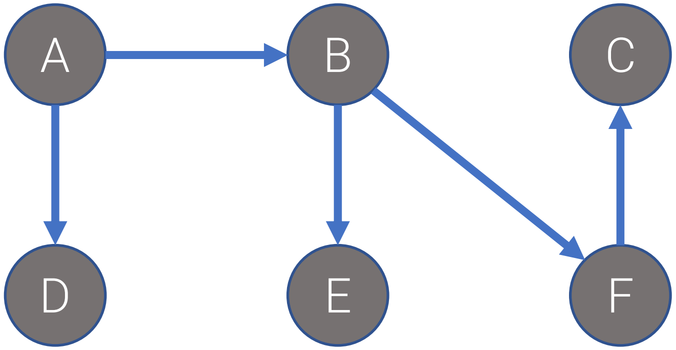
B <- A
D <- A
E <- B
F <- B
C <- F

Visited (set)

{A, B, D, E, F, C}

BFS Search Tree

start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

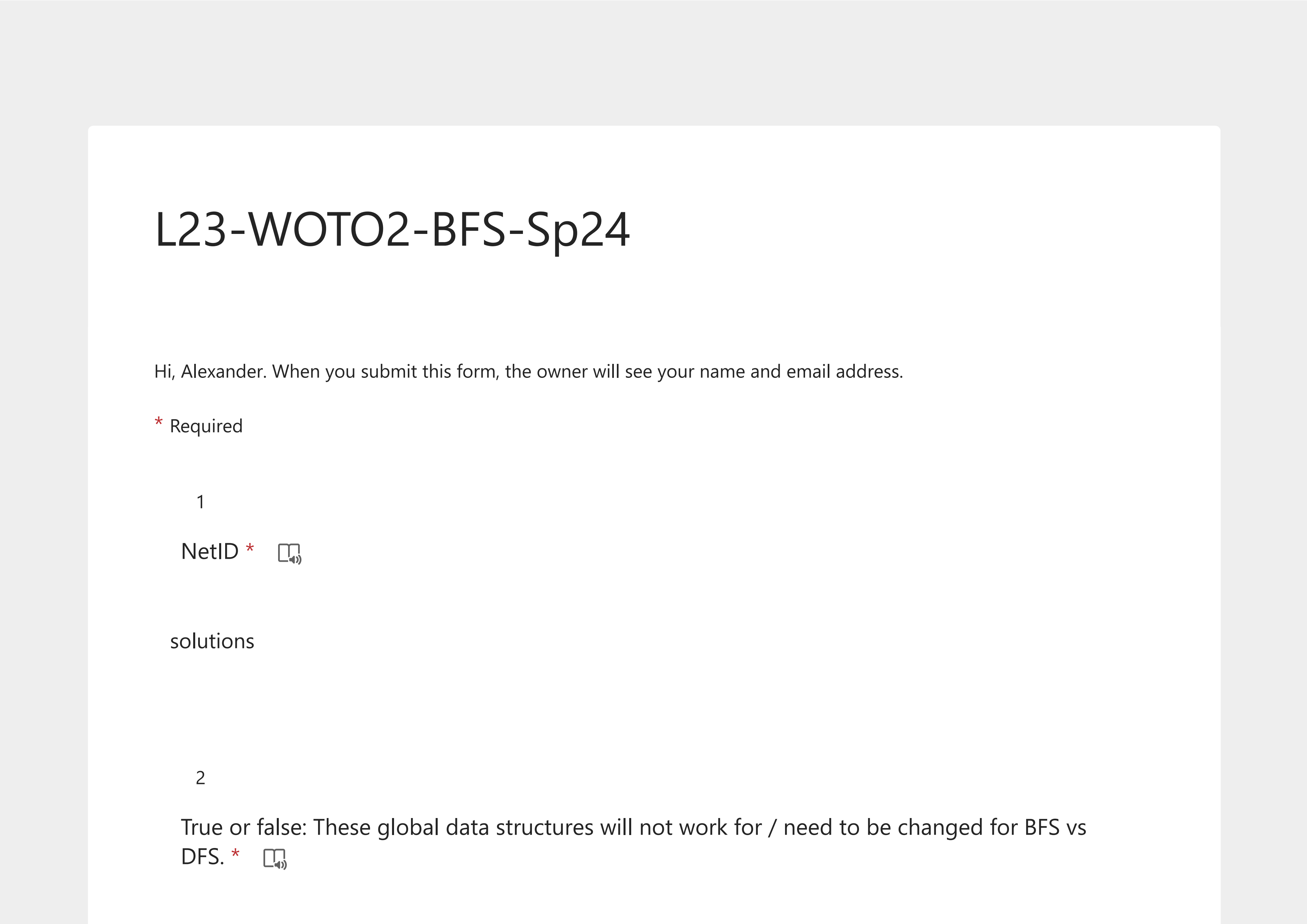
toExplore (queue)

previous (map)

B <- A
D <- A
E <- B
F <- B
C <- F

Visited (set)

{A, B, D, E, F, C}



L23-WOTO2-BFS-Sp24

Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

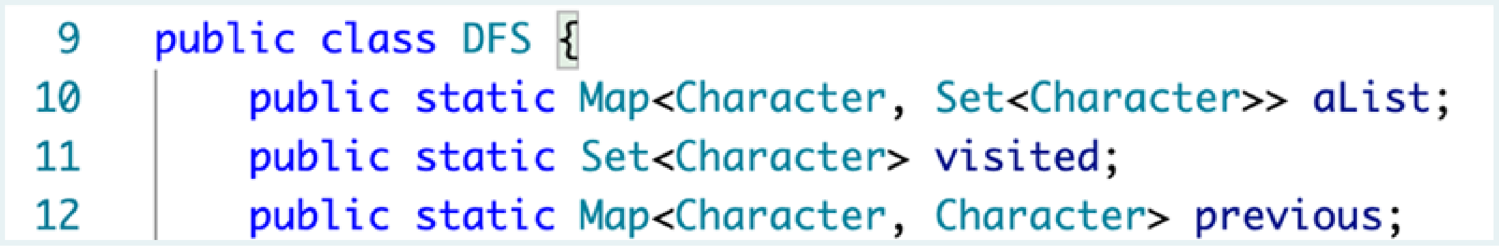
1

NetID *

solutions

2

True or false: These global data structures will not work for / need to be changed for BFS vs DFS. *



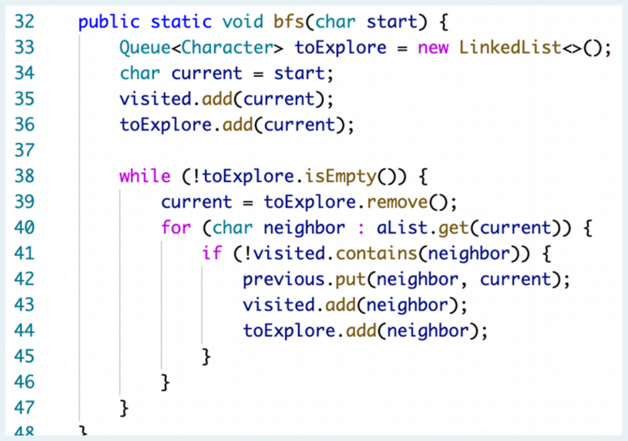
```java
public class DFS {
    public static Map<Character, Set<Character>> aList;
    public static Set<Character> visited;
    public static Map<Character, Character> previous;
```

○ True

● False

3

Which line of code best explains what is different about BFS vs. DFS algorithmically? *



```java
32  public static void bfs(char start) {
33      Queue<Character> toExplore = new LinkedList<>();
34      char current = start;
35      visited.add(current);
36      toExplore.add(current);
37
38      while (!toExplore.isEmpty()) {
39          current = toExplore.remove();
40          for (char neighbor : aList.get(current)) {
41              if (!visited.contains(neighbor)) {
42                  previous.put(neighbor, current);
43                  visited.add(neighbor);
44                  toExplore.add(neighbor);
45              }
46          }
47      }
48  }
```

● Line 33

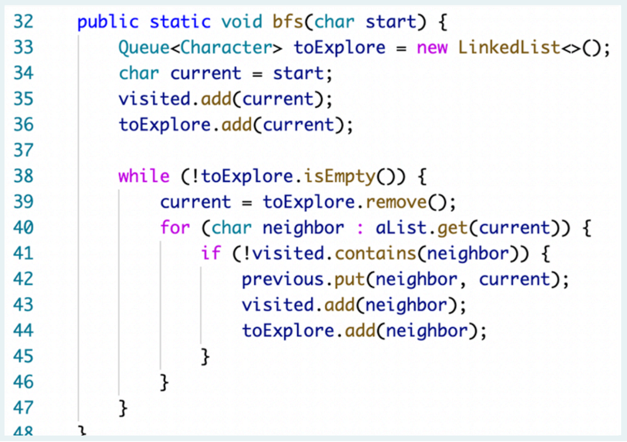
○ Line 38

○ Line 40

○ Line 41

4

What best explains why the while loop on line 38 only considers each node in the graph once / is O(N)? *



```
32  public static void bfs(char start) {
33      Queue<Character> toExplore = new LinkedList<>();
34      char current = start;
35      visited.add(current);
36      toExplore.add(current);
37
38      while (!toExplore.isEmpty()) {
39          current = toExplore.remove();
40          for (char neighbor : aList.get(current)) {
41              if (!visited.contains(neighbor)) {
42                  previous.put(neighbor, current);
43                  visited.add(neighbor);
44                  toExplore.add(neighbor);
45              }
46          }
47      }
48  }
```

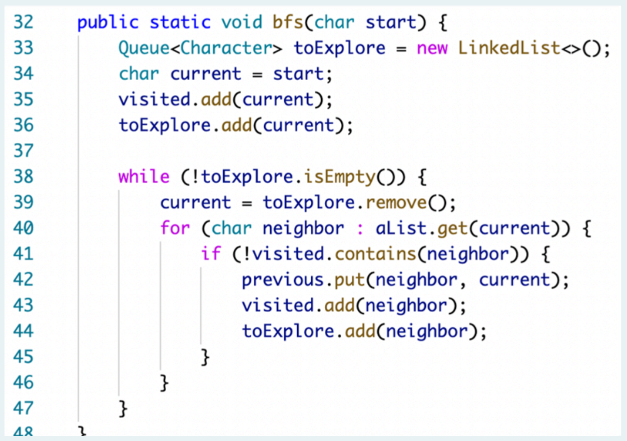
○ Because Queues do not store duplicates

○ Because we only consider each node as a "neighbor" once

● Because of the visited Set

5

If there are N nodes and M edges in the graph and the graph is connected, how many total times might line 41 be executed? *



```
32  public static void bfs(char start) {
33      Queue<Character> toExplore = new LinkedList<>();
34      char current = start;
35      visited.add(current);
36      toExplore.add(current);
37
38      while (!toExplore.isEmpty()) {
39          current = toExplore.remove();
40          for (char neighbor : aList.get(current)) {
41              if (!visited.contains(neighbor)) {
42                  previous.put(neighbor, current);
43                  visited.add(neighbor);
44                  toExplore.add(neighbor);
45              }
46          }
47      }
48  }
```

○ O(N)

● O(M)

○ O(NM)

6

True or false: BFS can find shortest paths from the start node to all other reachable nodes. *

● True

○ False

7

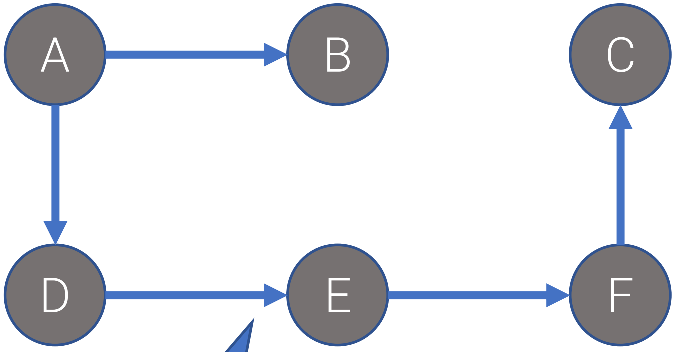
True or false: BFS explores all possible paths from the start node to all other reachable nodes. *

○ True

● False

Comparing DFS and BFS Search Trees

start: A

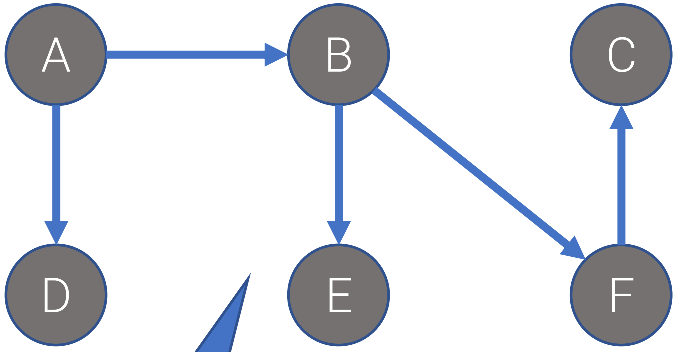


Length 4 path from A to C

previous (map)

B <- A
D <- A
E <- D
F <- E
C <- F

start: A



Length 3 path from A to C, shorter!

previous (map)

B <- A
D <- A
E <- B
F <- B
C <- F

Pathfinding Properties

- DFS and BFS **both** find valid paths to *all* nodes reachable from the start.
  - Can return early if you only want to find a path to a specific target node

- BFS finds the ***shortest path*** to every reachable node, DFS does *not* guarantee this.