CompSci 201, L23: Iterative DFS BFS

Logistics, Coming up

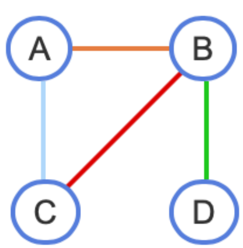
- APT10 (greedy) due today, Wed., 11/16

- APT Quiz 2: 2 hours, 3 problems
  - covers APT6-10, linked list and tree problems guaranteed
  - Release: This Thursday 11/17
  - Complete by: Monday 11/21
  - Quiz, not a hw, no late period

- Project 6: Route releasing this week, due Monday 12/5

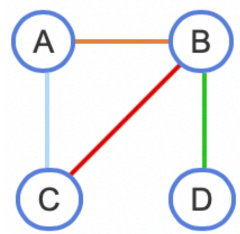
General data structures for graphs: Not necessarily a grid

Adjacency List



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

Adjacency Matrix

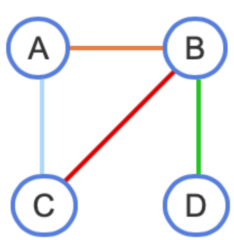


	A	B	C	D
A		1	1	
B	1		1	1
C	1	1		
D		1		

Zybook chapter 23

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

- Have an adjacency list for the graph
- Keep track of visited nodes in a set
- 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.

Iterative Depth-First Search (DFS)

Initializing Iterative DFS

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

```java
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…

If we haven't seen them before…

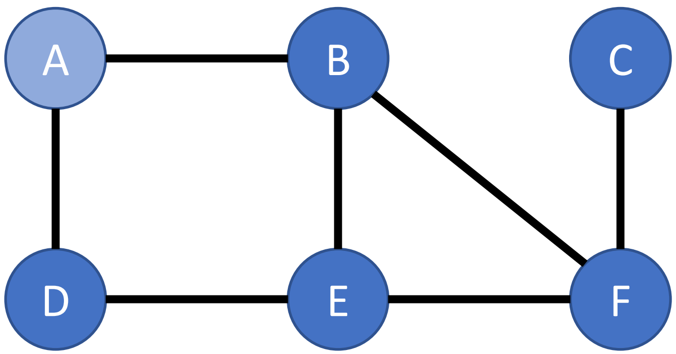
```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

Then:

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

Initialize search at A

start: A

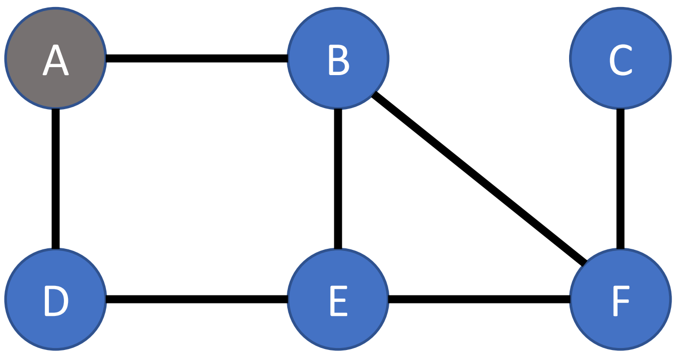


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Pop A off the stack

start: A



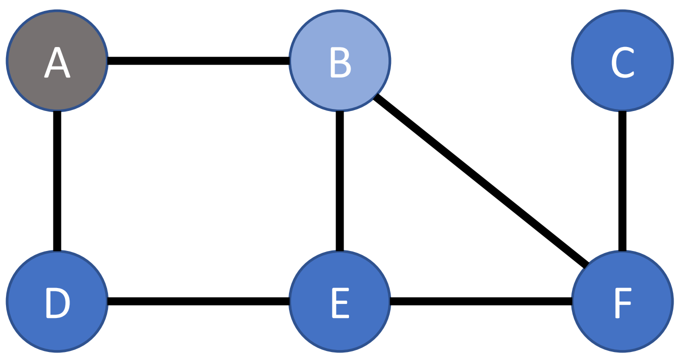
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

{A}

Find B from A

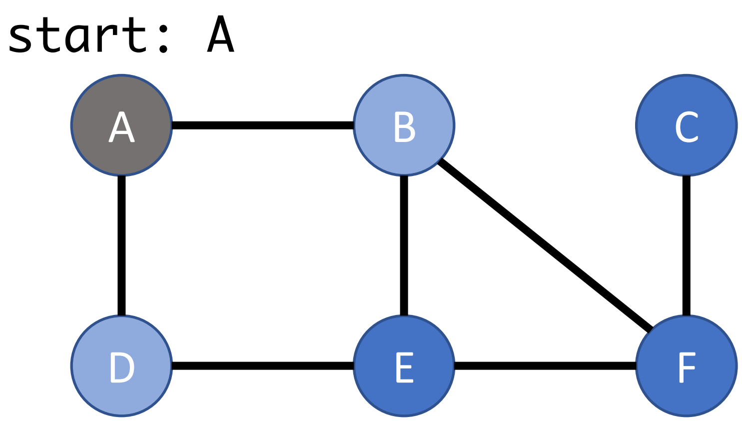
start: A



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

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

Find D from A

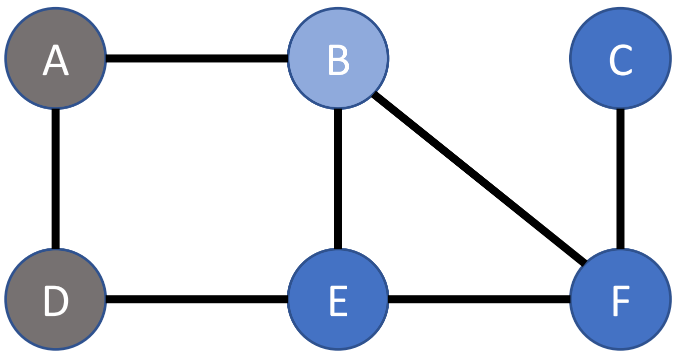


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Pop D off the stack

start: A

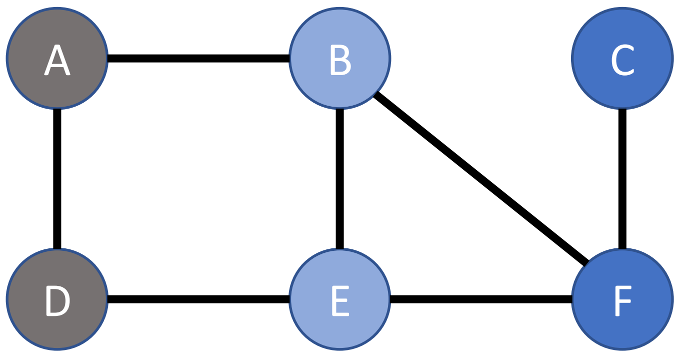


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Find E from D

start: A

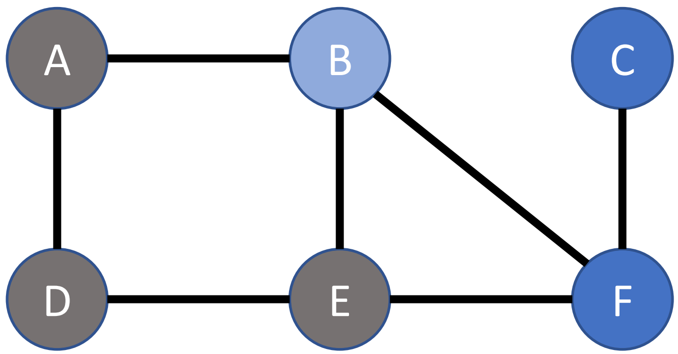


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Pop E off the stack

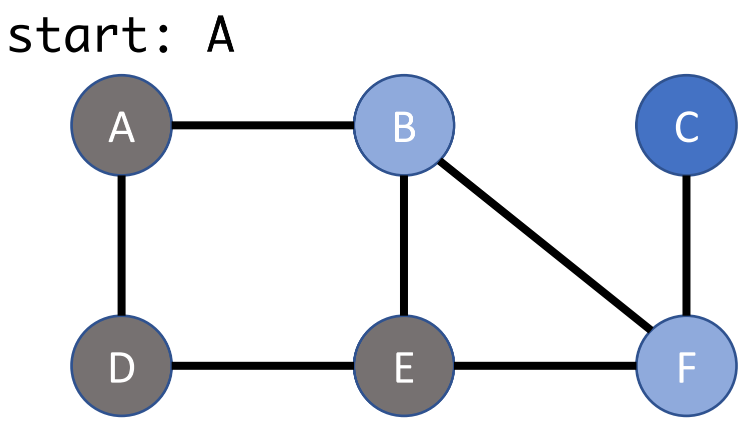
start: A



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}

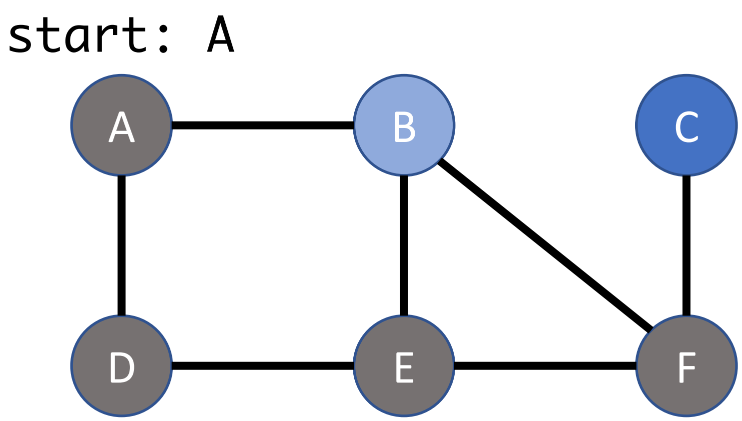
Find F from E



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	

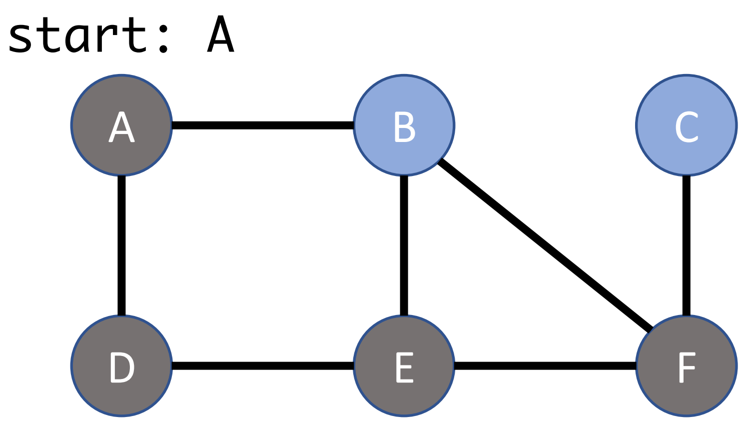
Pop F off the stack



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

Find C from F

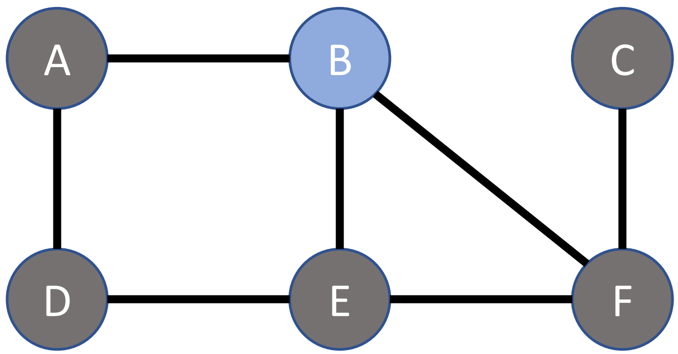


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

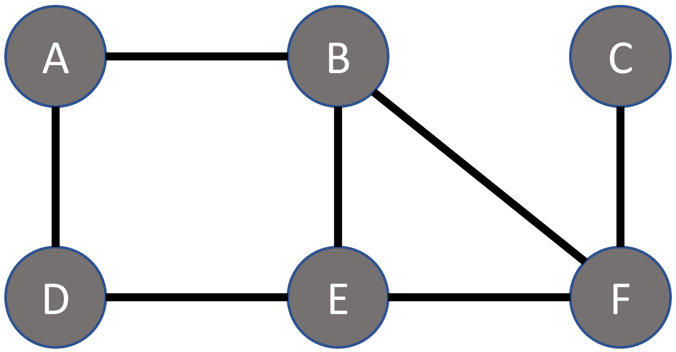


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

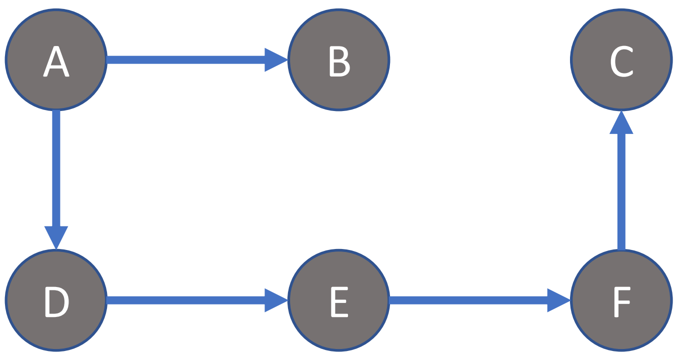
B <- A
D <- A
E <- D
F <- E
C <- F

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)**   **Visited (set)**

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

```
{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

WOTO

Go to duke.is/m467a

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

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?

```java
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).

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

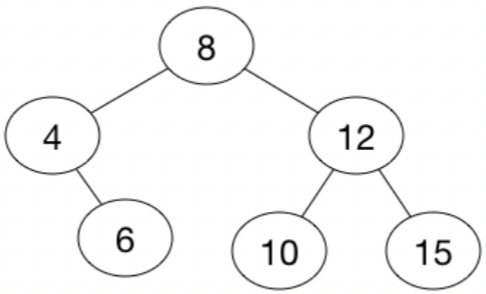
LinkedList implements the Queue interface.

```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
```

levelOrder Tree Traversal with 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
```

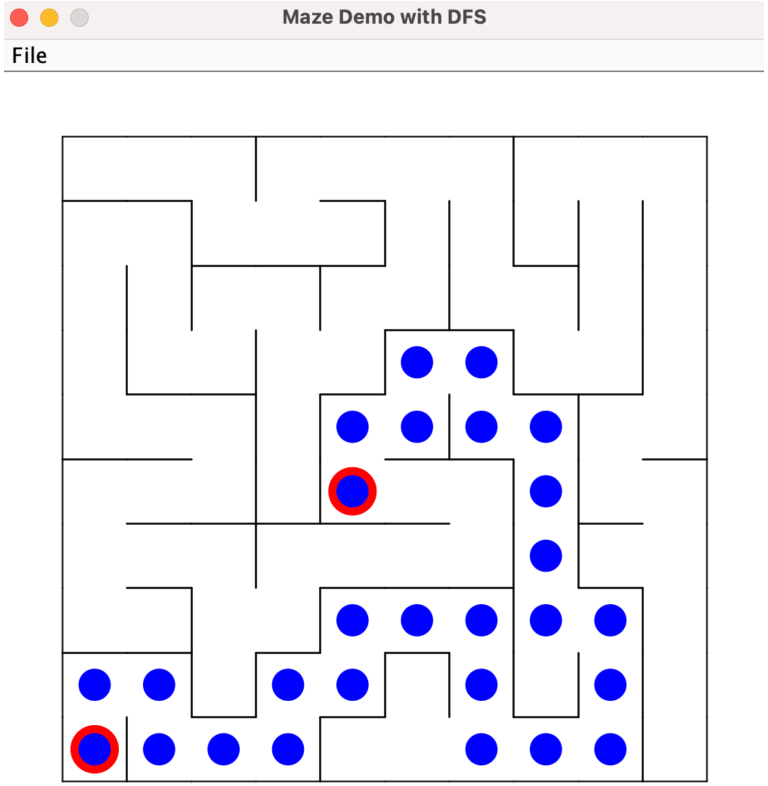
Use a queue to keep track of nodes
First in first out, nodes visited in level order

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.

coursework.cs.duke.edu/cs-201-fall-22/maze-demo

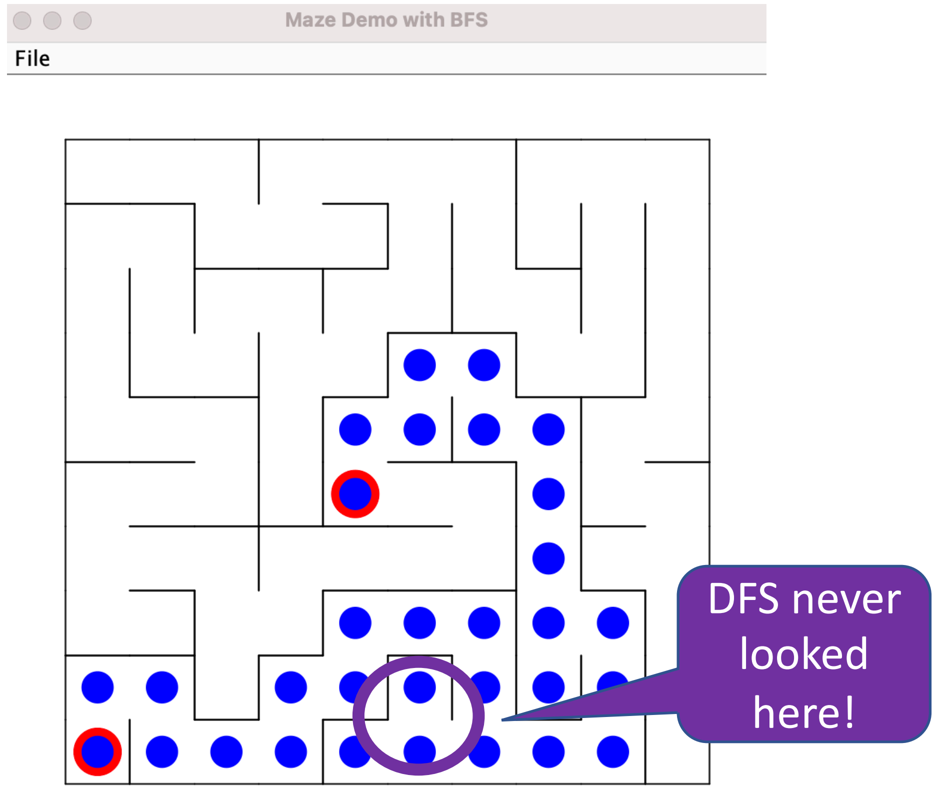


Breadth First Search for Solving Maze

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

Looking for the shortest path/solution.

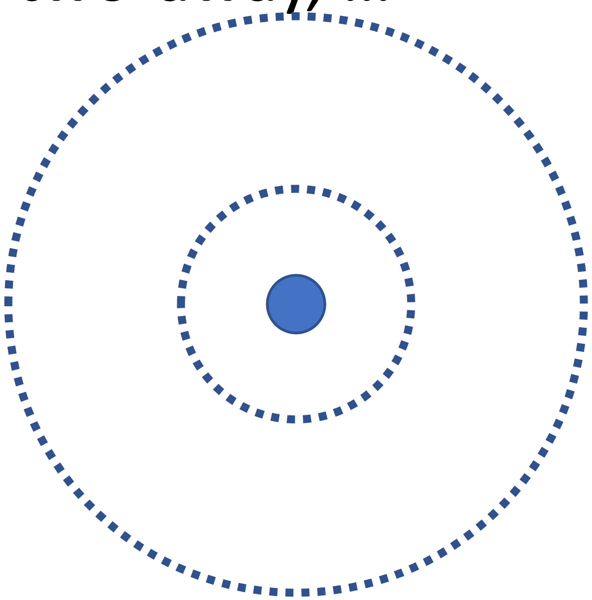
coursework.cs.duke.edu/cs-201-fall-22/maze-demo



Queue = BFS, Stack = DFS

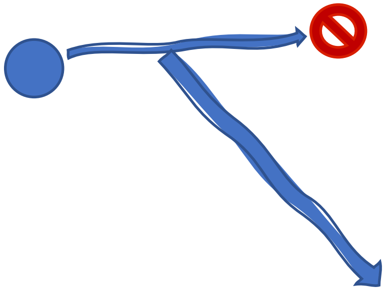
BFS: FIFO Exploration

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.

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

- Queue is FIFI(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…

Look at all neighbors of current node…

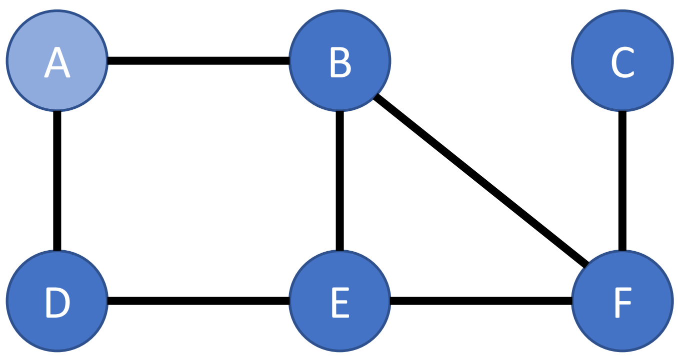
```
while (!toExplore.isEmpty()) {
    current = toExplore.remove();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.add(neighbor);
        }
    }
}
```

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

start: A

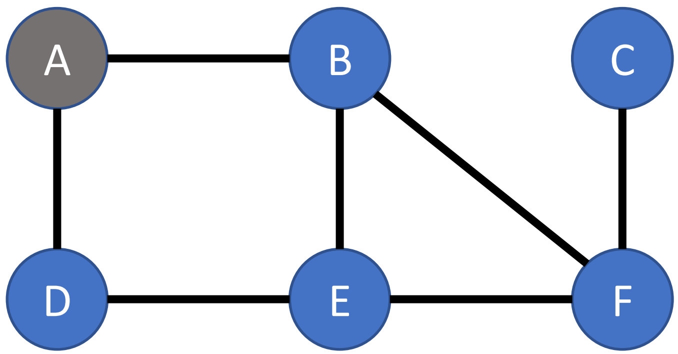


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Remove A from the queue

start: A



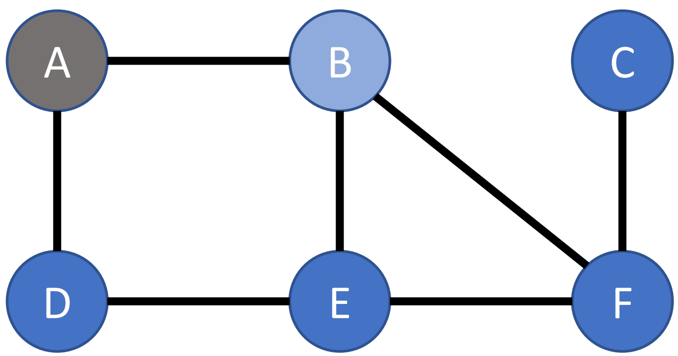
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

{A}

Find B from A

start: A



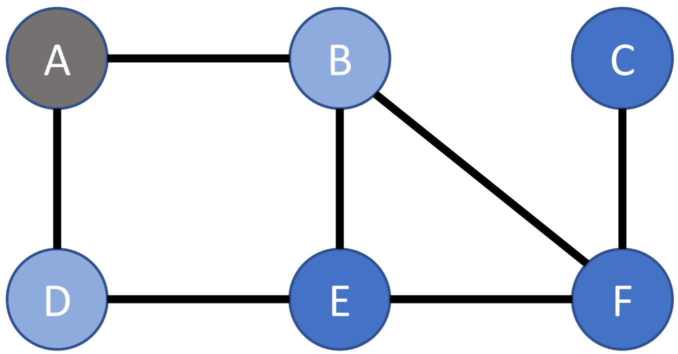
Adjacency List:

```
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	B <- A	{A, B}

Find D from A

```
start: A
```



Adjacency List:

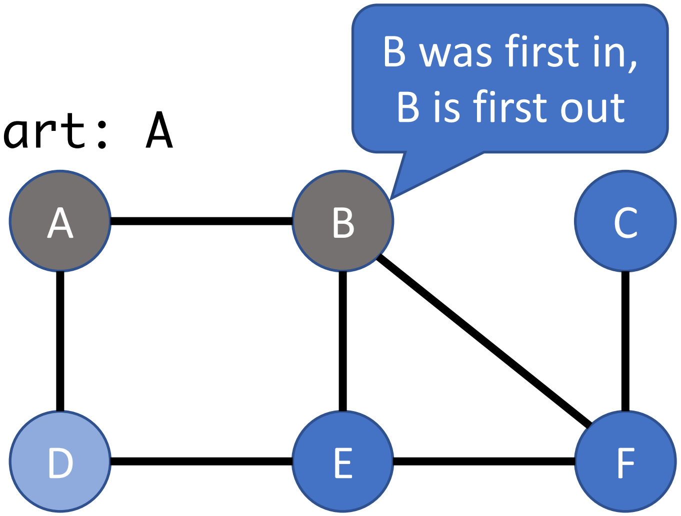
```
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]
```

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

start: A



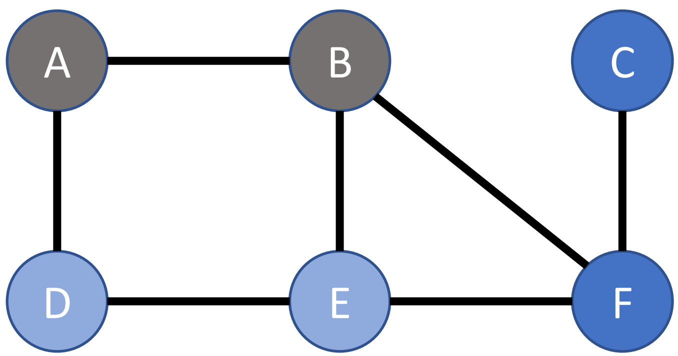
Adjacency List:

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

Find E from B

start: A

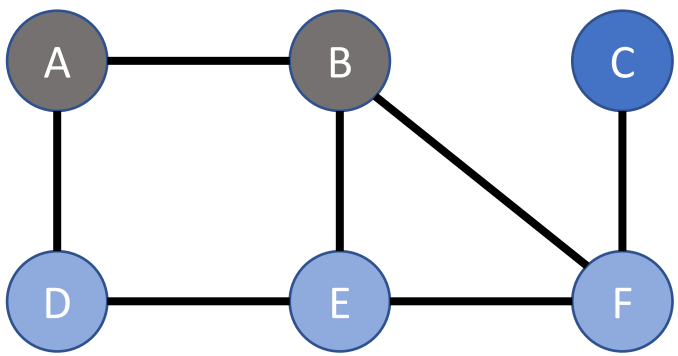


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Find F from B

start: A

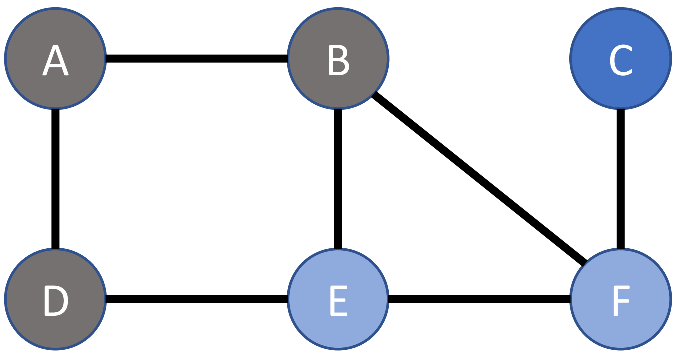


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

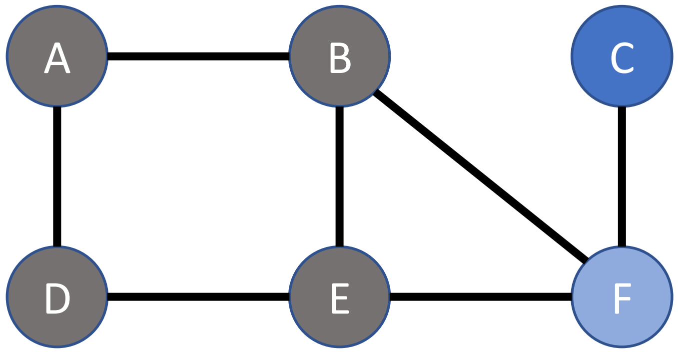


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

start: A



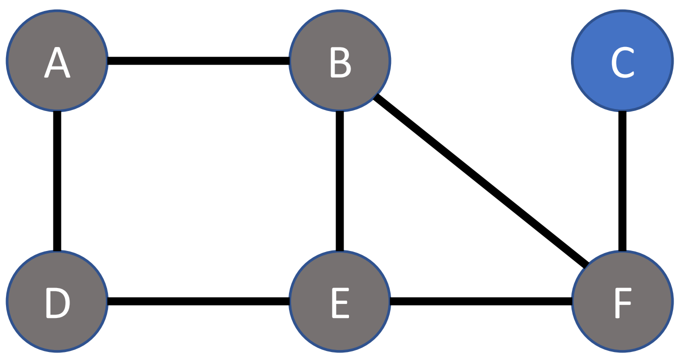
Adjacency List:

```
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)**
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]

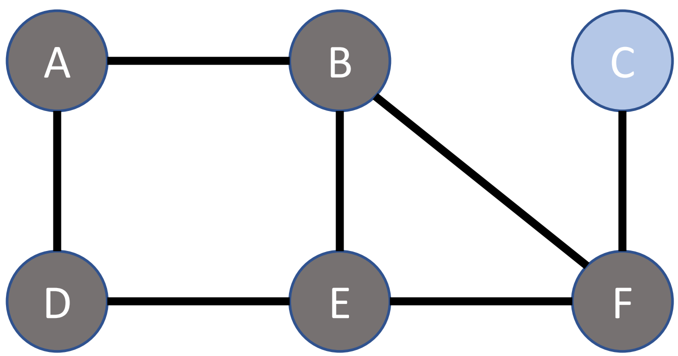
toExplore (queue) **previous (map)** **Visited (set)**

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

{A, B, D, E, F}

Find C from F

start: A

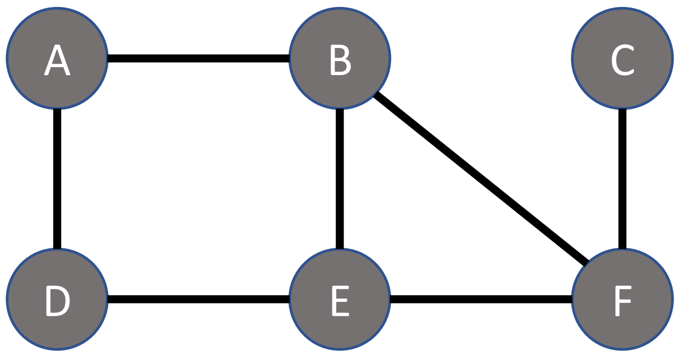


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Remove C from queue

start: A



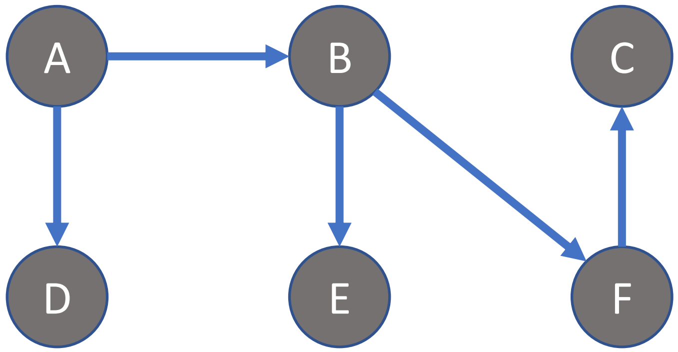
Adjacency List:

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

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

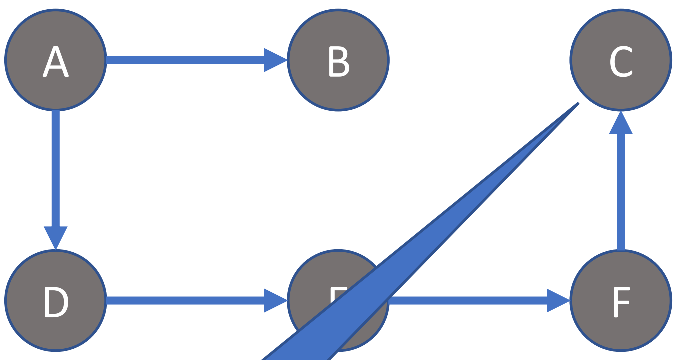
```
B <- A
D <- A
E <- B
F <- B
C <- F
```

Visited (set)

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

Comparing DFS and BFS Search Trees

`start: A`

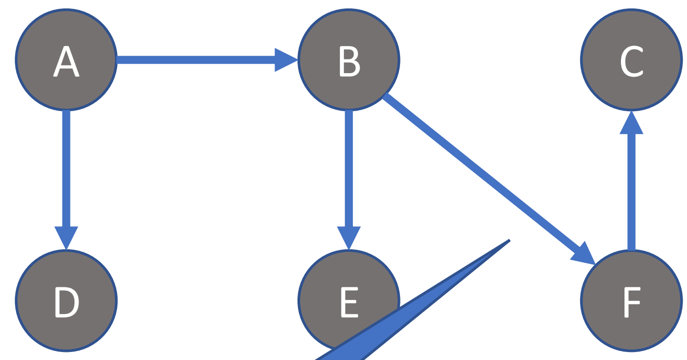


Length 4 path from A to C

`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.

WOTO

Go to duke.is/wjrfp

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!