BREADTH-FIRST SEARCH!

WHAT WE'VE COVERED SO FAR

So far we have covered Linear and Binary Search, and Bubble, Selection, Insertion, Merge, and Quick Sort!

WHAT WE ARE COVERING TODAY

Today we are covering Breadth-First Search, or BFS

BREADTH-FIRST SEARCH

BFS is a search algorithm for traversing a tree or graph data structure.

This is done one “level” at a time rather than one “branch” at a time.

WHAT?!

Think of it like your pet cat got out of the house and you are looking for it. You would ask your neighbors first, right?

And then you'd ask their neighbors, and then you'd ask their neighbors, and their neighbors... etc. until you found your cat!

Lets consider variables of a graph:

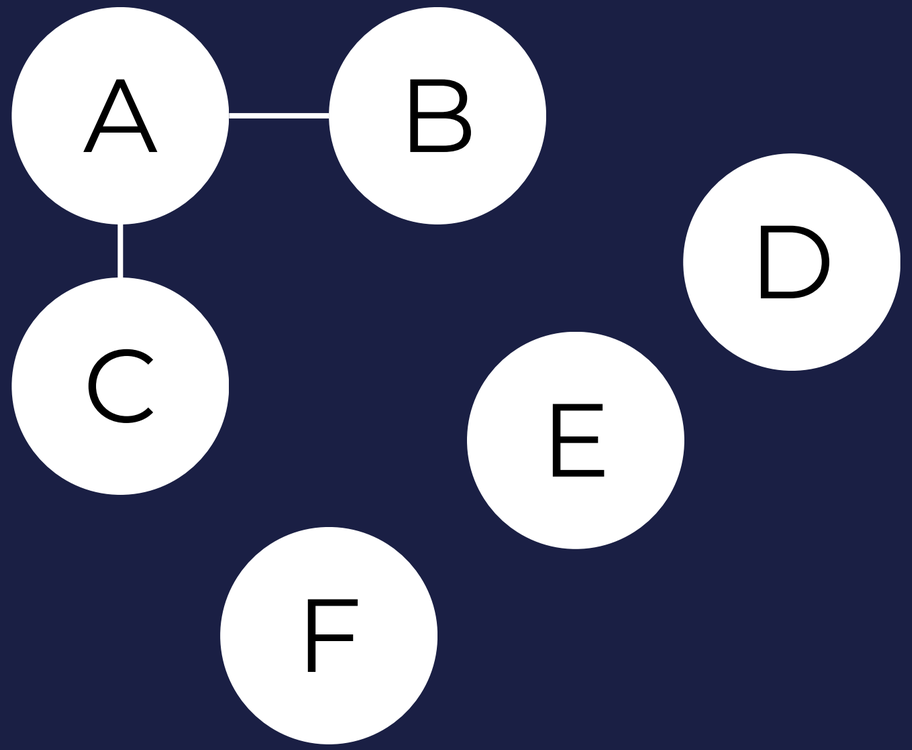
```
graph = {
 'A': {'B', 'C'},
 'B': {'A', 'D', 'E'},
 'C': {'A', 'F'},
 'D': {'B'},
 'E': {'B', 'F'},
 'F': {'C', 'E'}
}
```

But lets make it look like a graph

```
graph = {
 'A': {'B', 'C'},
 'B': {'A', 'D', 'E'},
 'C': {'A', 'F'},
 'D': {'B'},
 'E': {'B', 'F'},
 'F': {'C', 'E'}
}
```

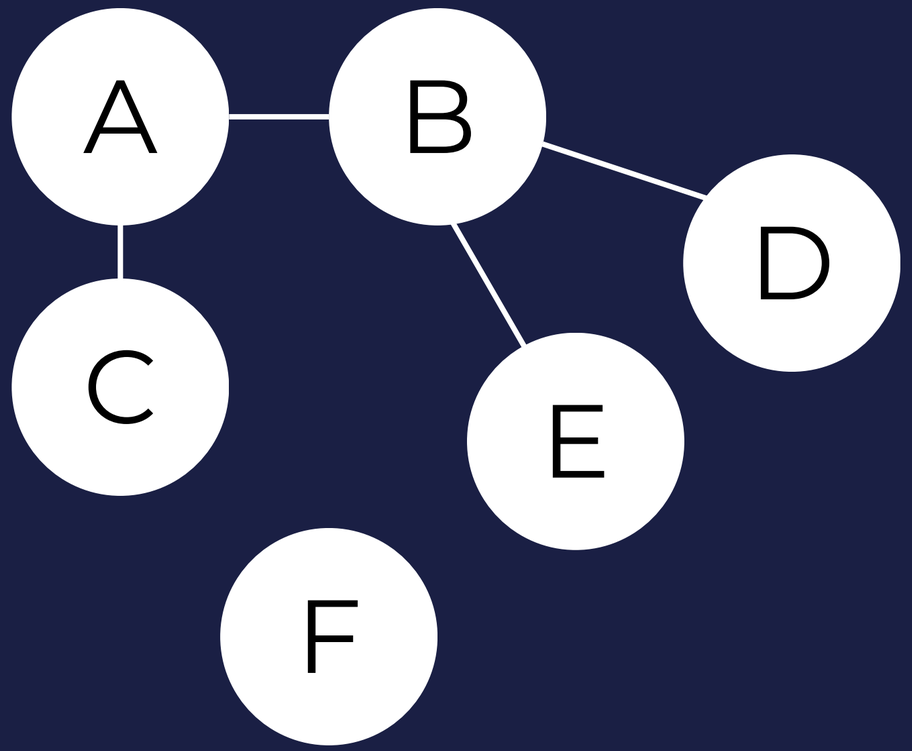
But lets make it look like a graph

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}
```



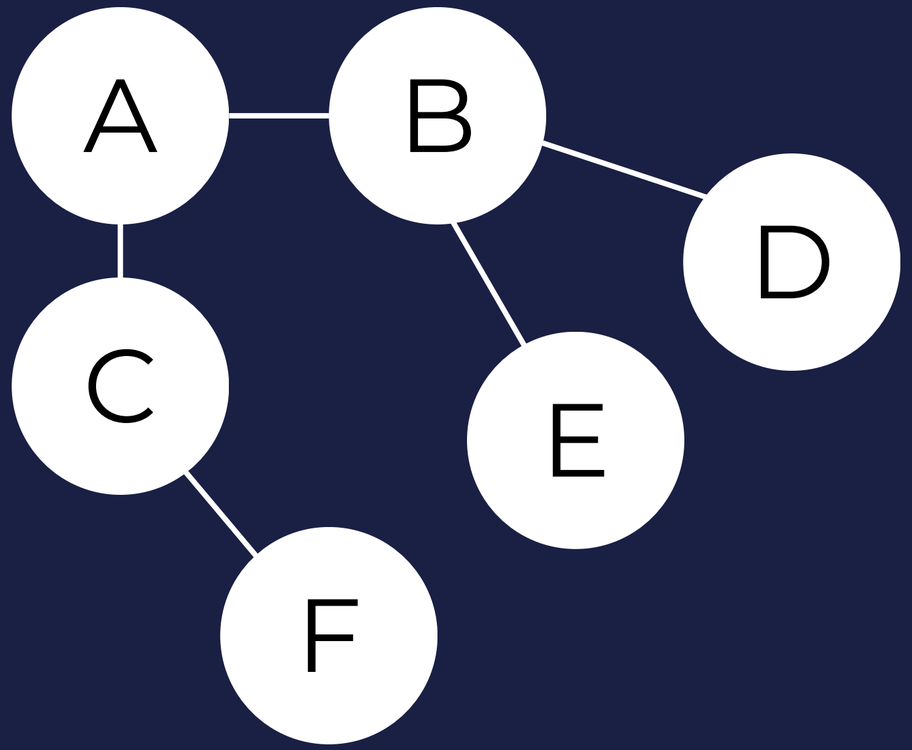
But lets make it look like a graph

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}
```



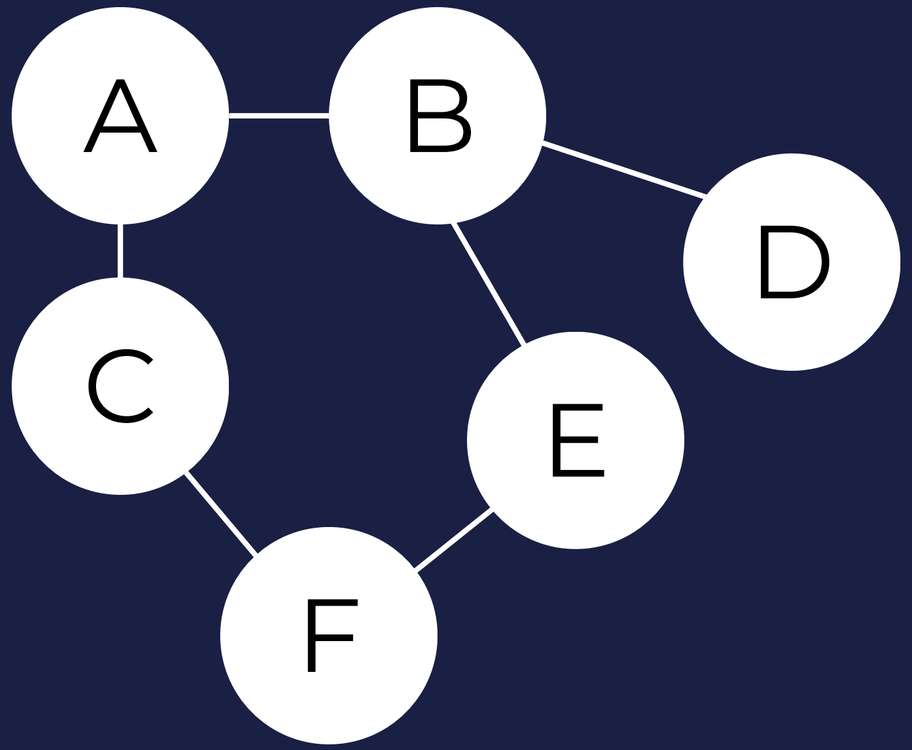
But lets make it look like a graph

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}
```



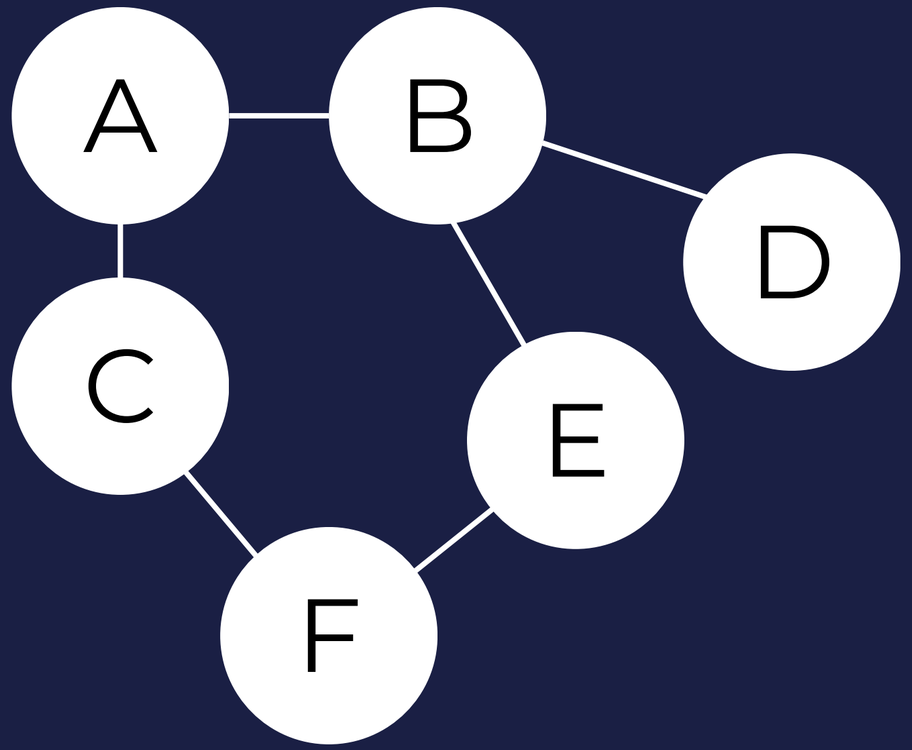
But lets make it look like a graph

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}
```



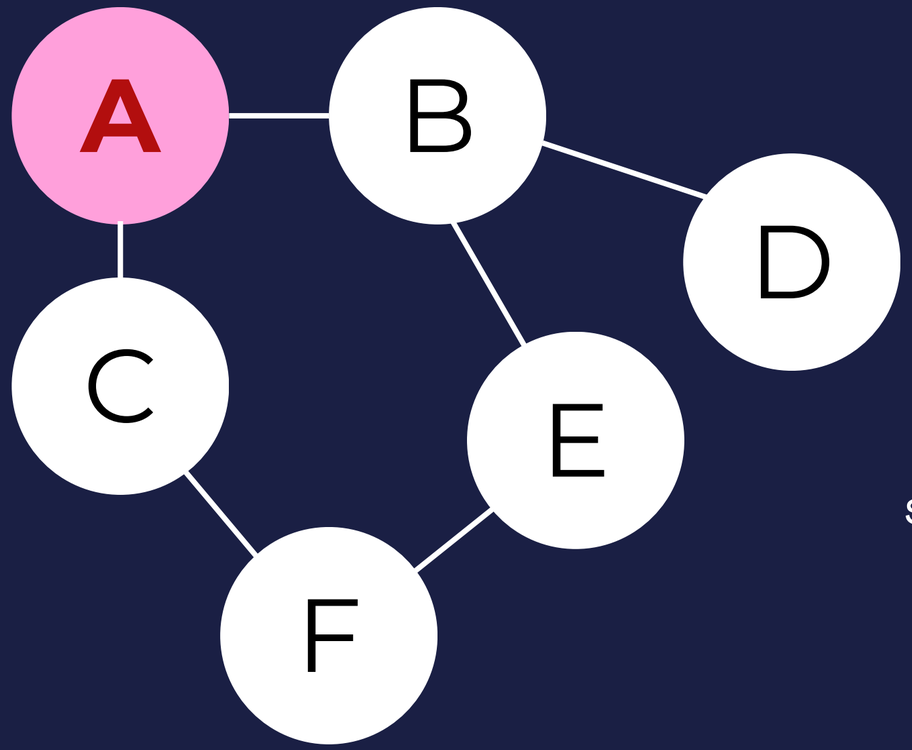
Then we choose a starting node. In our case, we will start with A

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”
```



Then we choose a starting node. In our case, we will start with A

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”
```

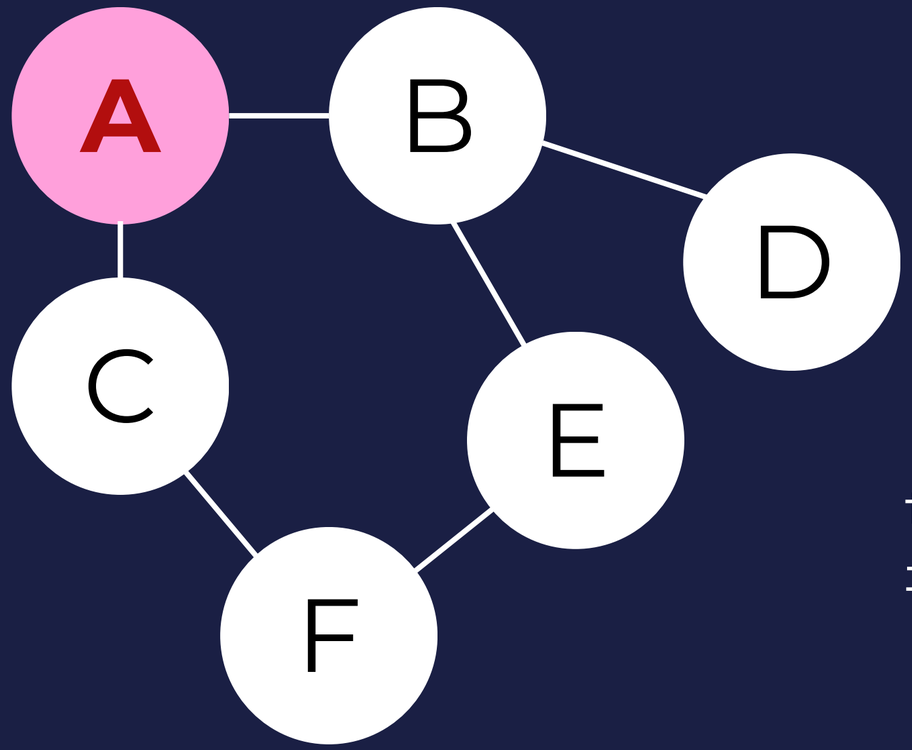


BFS uses a queue system. So we put A in the queue:

```
queue = [’A’]
```

And we'll track the traversal order.

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = "A"

traversal_order = [
    'A',
]
```

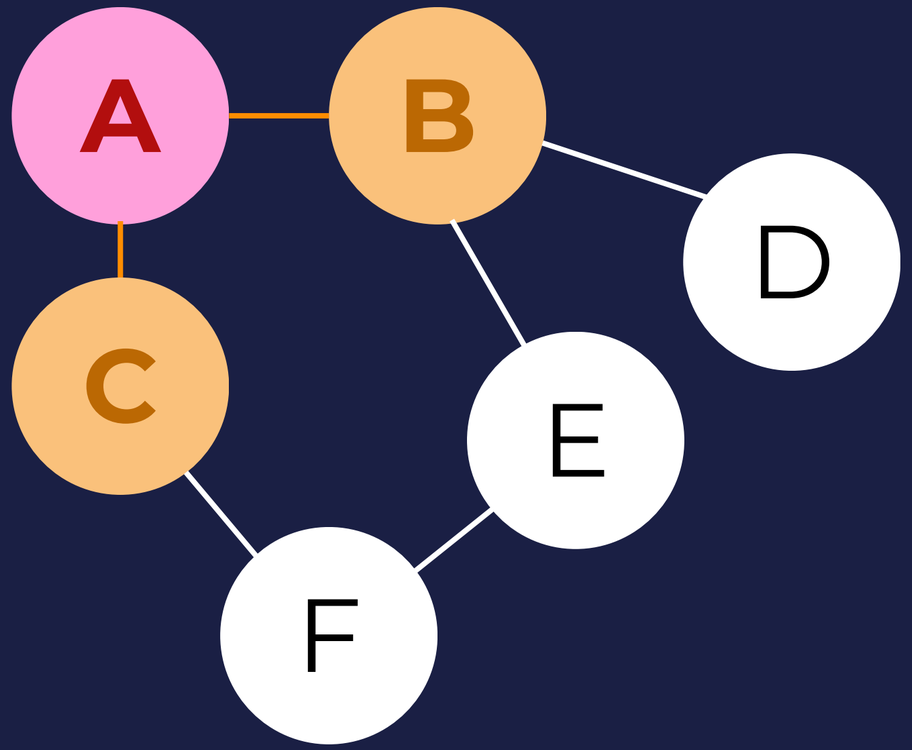


And we'll remove 'A' from the queue because it's been added to the traversal order

queue = [ ]

And we'll add A's immediate neighbors

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = "A"

traversal_order = [
    'A',
]
```

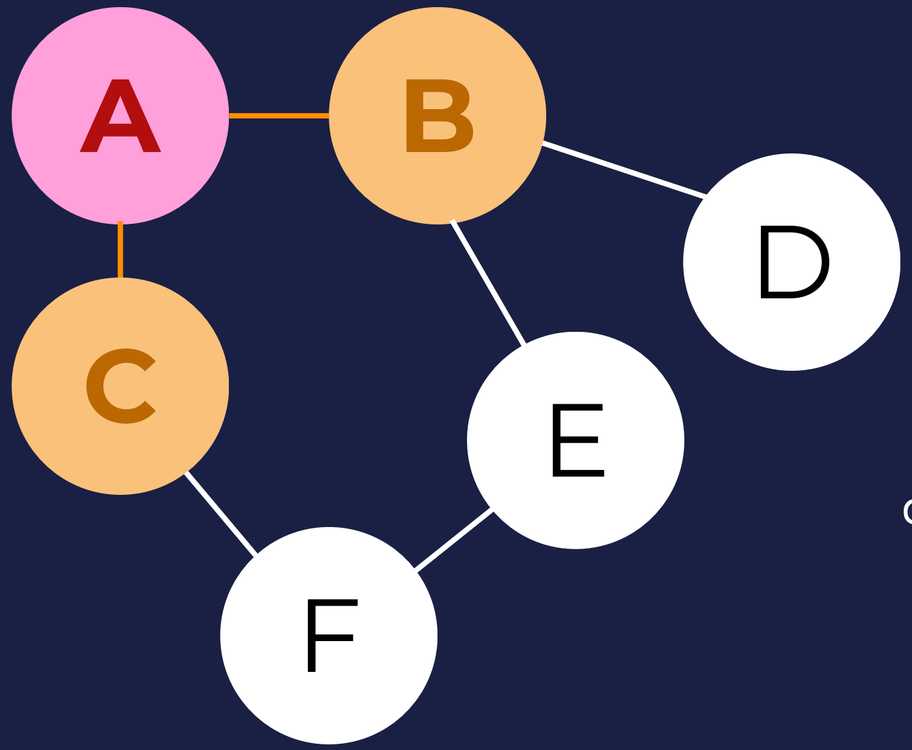


So that means we add 'B' and 'C' to the queue

queue = ['B', 'C']

And we'll add A's immediate neighbors

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = "A"

traversal_order = [
    'A',
    'B', 'C',
]
```

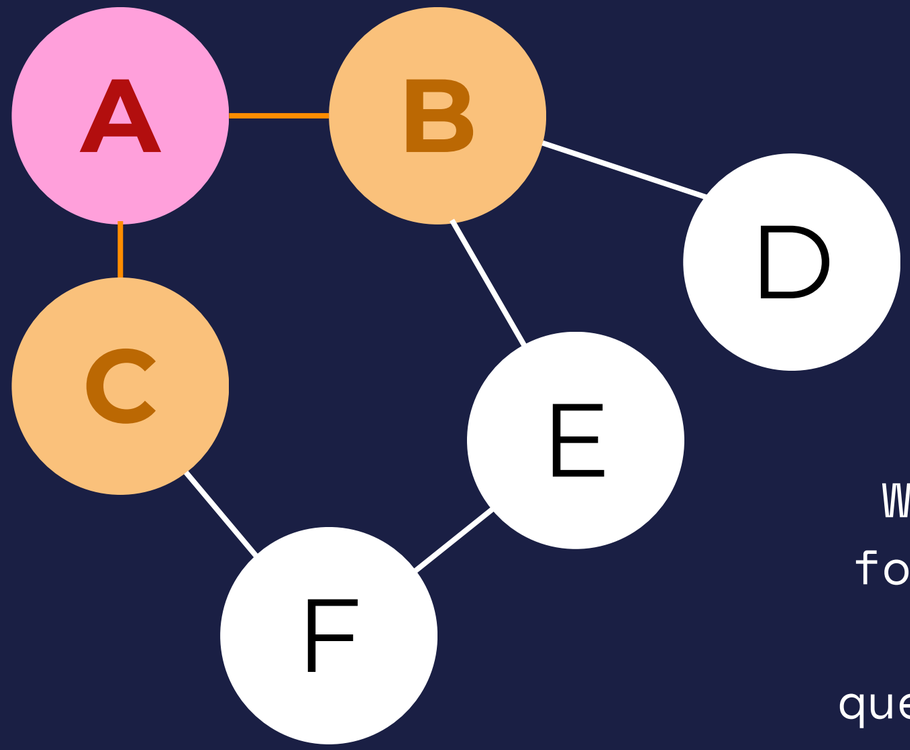


And we'll remove 'B' and 'C' from the queue

queue = [ ]

Then ‘B’s and ‘C’s immediate neighbors

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
    ‘A’,
    ‘B’, ‘C’,
]
```

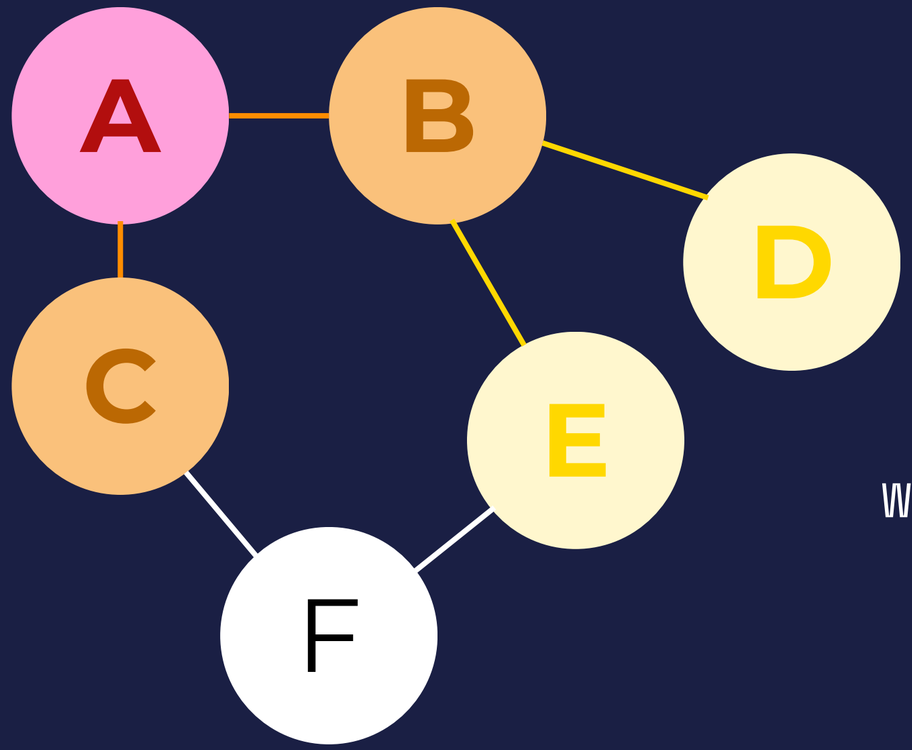


Which are ‘D’ and ‘E’
for ‘B’ and ‘F’ for ‘C’

queue = [’D’, ‘E’, ‘F’]

Then ‘B’s and ‘C’s immediate neighbors

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
     ‘A’,
     ‘B’, ‘C’,
     ‘D’, ‘E’,
]
```

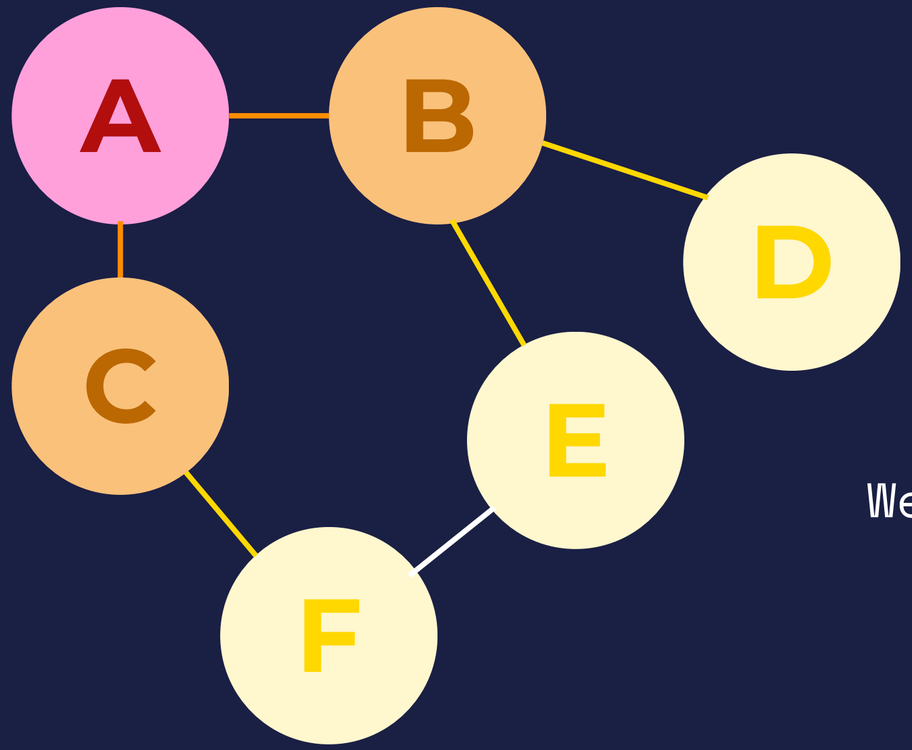


We remove ‘D’ and ‘E’ from the Queue

queue = [‘F’]

Then ‘B’s and ‘C’s immediate neighbors

```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
     ‘A’,
     ‘B’, ‘C’,
     ‘D’, ‘E’, ‘F’
]
```

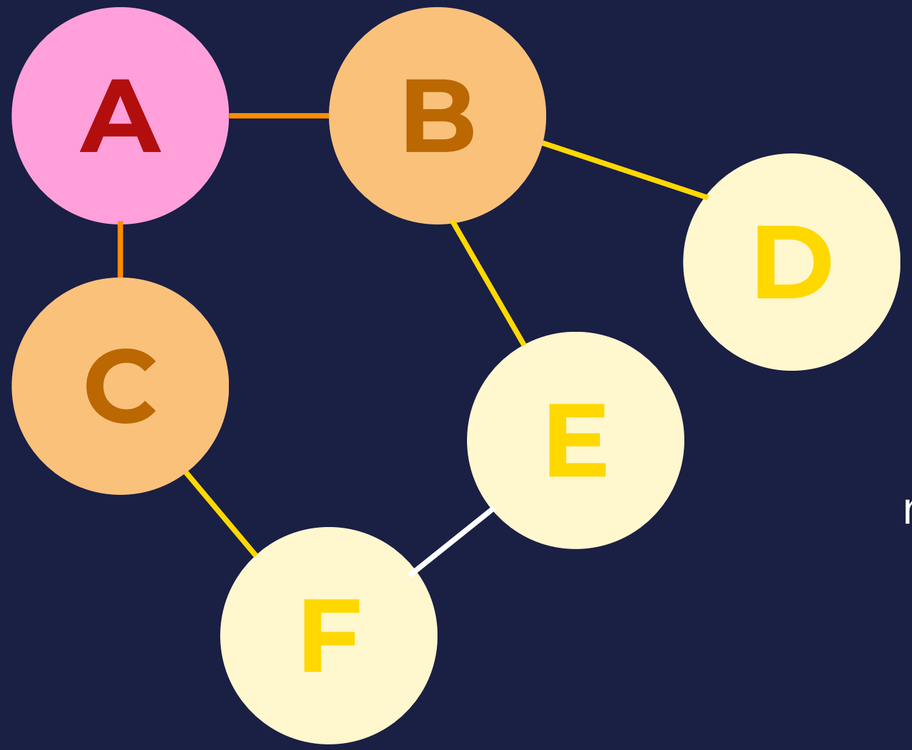


We remove ‘F’ from the Queue

queue = [ ]

And Then we check ‘D’, ‘E’, and ‘F’s neighbors

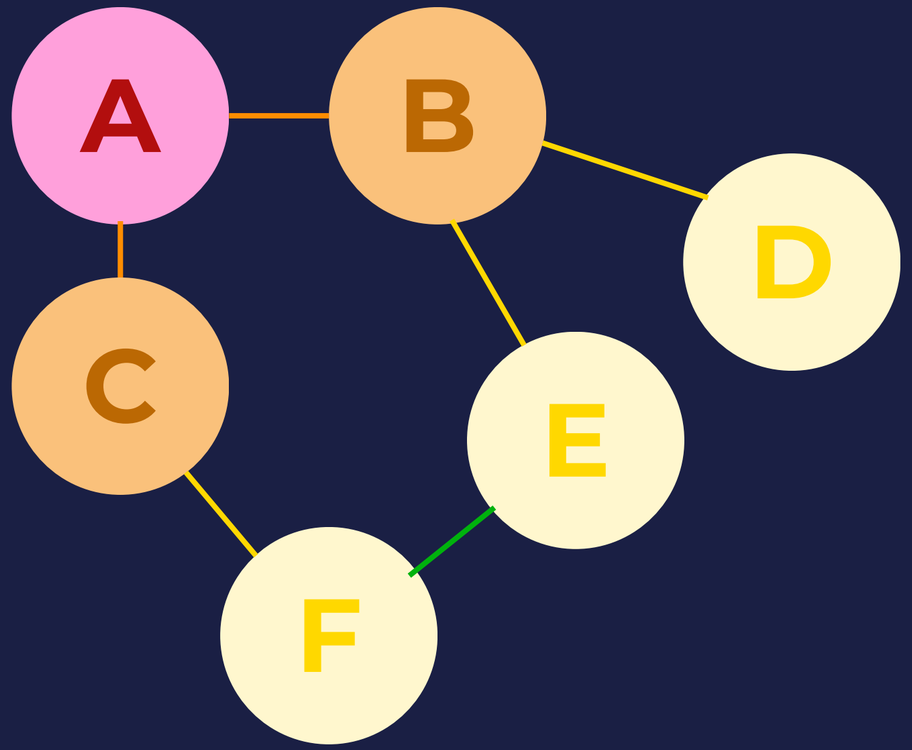
```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
     ‘A’,
     ‘B’, ‘C’,
     ‘D’, ‘E’, ‘F’
]
```



D doesn’t have any neighbors that haven’t already been checked.

And Then we check ‘D’, ‘E’, and ‘F’s neighbors

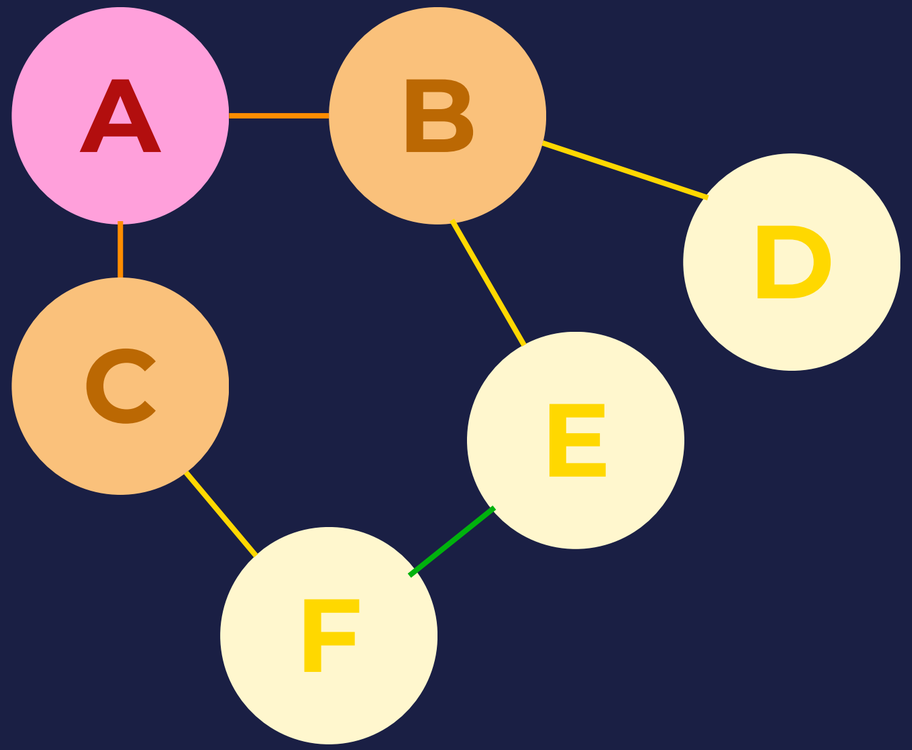
```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
    ‘A’,
    ‘B’, ‘C’,
    ‘D’, ‘E’, ‘F’
]
```



E’s neighbors have already been checked.

And Then we check ‘D’, ‘E’, and ‘F’s neighbors

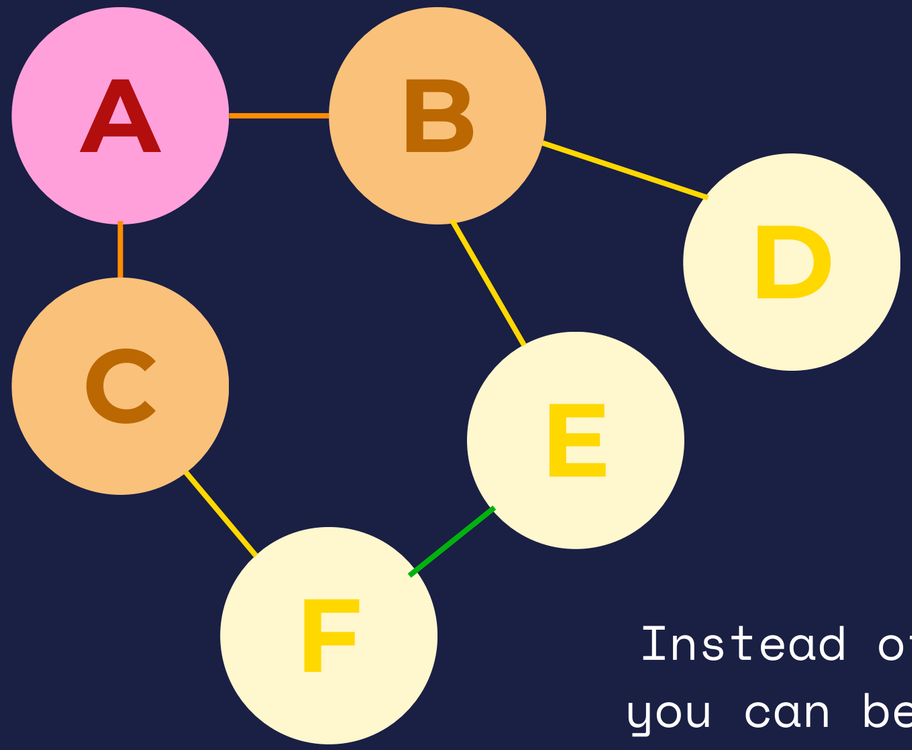
```
graph = {
  'A': {'B', 'C'},
  'B': {'A', 'D', 'E'},
  'C': {'A', 'F'},
  'D': {'B'},
  'E': {'B', 'F'},
  'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
    ‘A’,
    ‘B’, ‘C’,
    ‘D’, ‘E’, ‘F’
]
```



F’s neighbors have already been checked.

So the traversal order is [”A”, “B”, “C”, “D”, “E”, “F”]

```
graph = {
 'A': {'B', 'C'},
 'B': {'A', 'D', 'E'},
 'C': {'A', 'F'},
 'D': {'B'},
 'E': {'B', 'F'},
 'F': {'C', 'E'}
}

start_node = “A”

traversal_order = [
    ‘A’,
    ‘B’, ‘C’,
    ‘D’, ‘E’, ‘F’
]
```



Instead of a traversal order, you can be looking for values, searching for the shortest path, and more!

THE PSEUDOCODE

```
function bfs(graph, start):
    visited = empty set()
    queue = queue (start)
    traversal_order = []

    while queue is NOT empty:
        node = queue.popleft()

        if node is NOT in visited:
            visited.append(node)
            traversal_order.append(node)
            queue.append(node's neighbors)

    return traversal_order
```

EXAMPLES REPLIT AND GITHUB! PLEASE GO TO:

HTTPS://REPLIT.COM/@RIKKIEHRHART/GRABABYTE

HTTPS://GITHUB.COM/RIKKITOMIKOEHRHART/GRABABYTE

UP NEXT

Apr 2 - Depth-First Search (DFS)
Apr 9 Hashing
Apr 16 - Dijkstra's Algorithm
Apr 23 - Dynamic Programming (Knapsack Problem)
Apr 30 - Union-Find
May 7 - Kruskal's Algorithm
May 14 - Prim's Algorithm

Questions? - rikki.ehrhart@g.ausitncc.edu

If you'd like the opportunity to run a Grab a Byte algorithm workshop, please let me know!