GRAB A BYTE

DEPTH-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!
And last week we covered Breadth-First Search

WHAT WE ARE COVERING TODAY

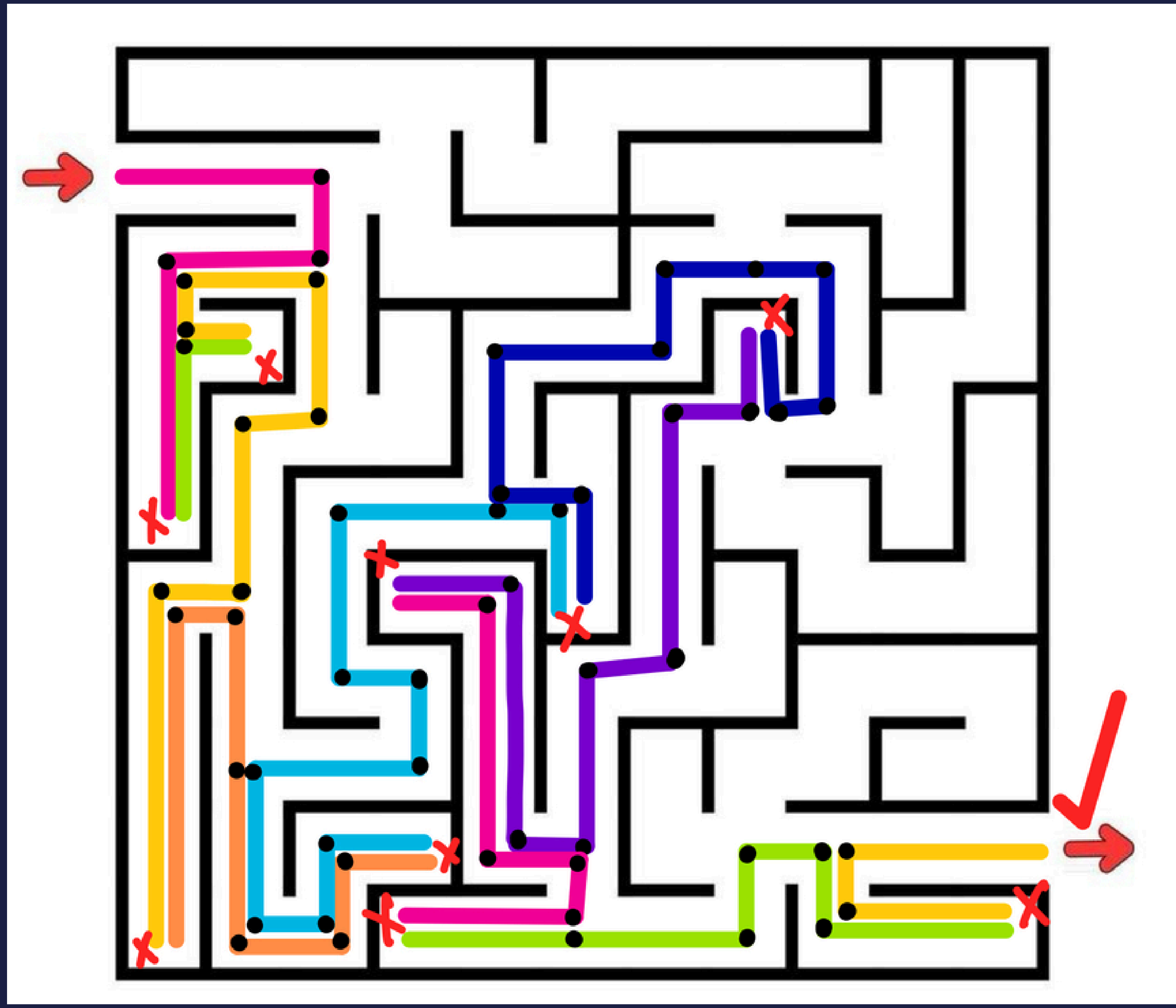
Today we are covering Depth-First Search, or DFS

DEPTH-FIRST SEARCH

A search algorithm for traversing a tree or graph data structure.

It explores as far along each branch as possible before backtracking and searching the next branch

WHAT?!



Think of it like a maze. You start going in one direction, and when you hit a dead end, you backtrack until you reach a intersection that has a path you haven't tried yet. Like a Maze.

Lets consider variables of a graph:

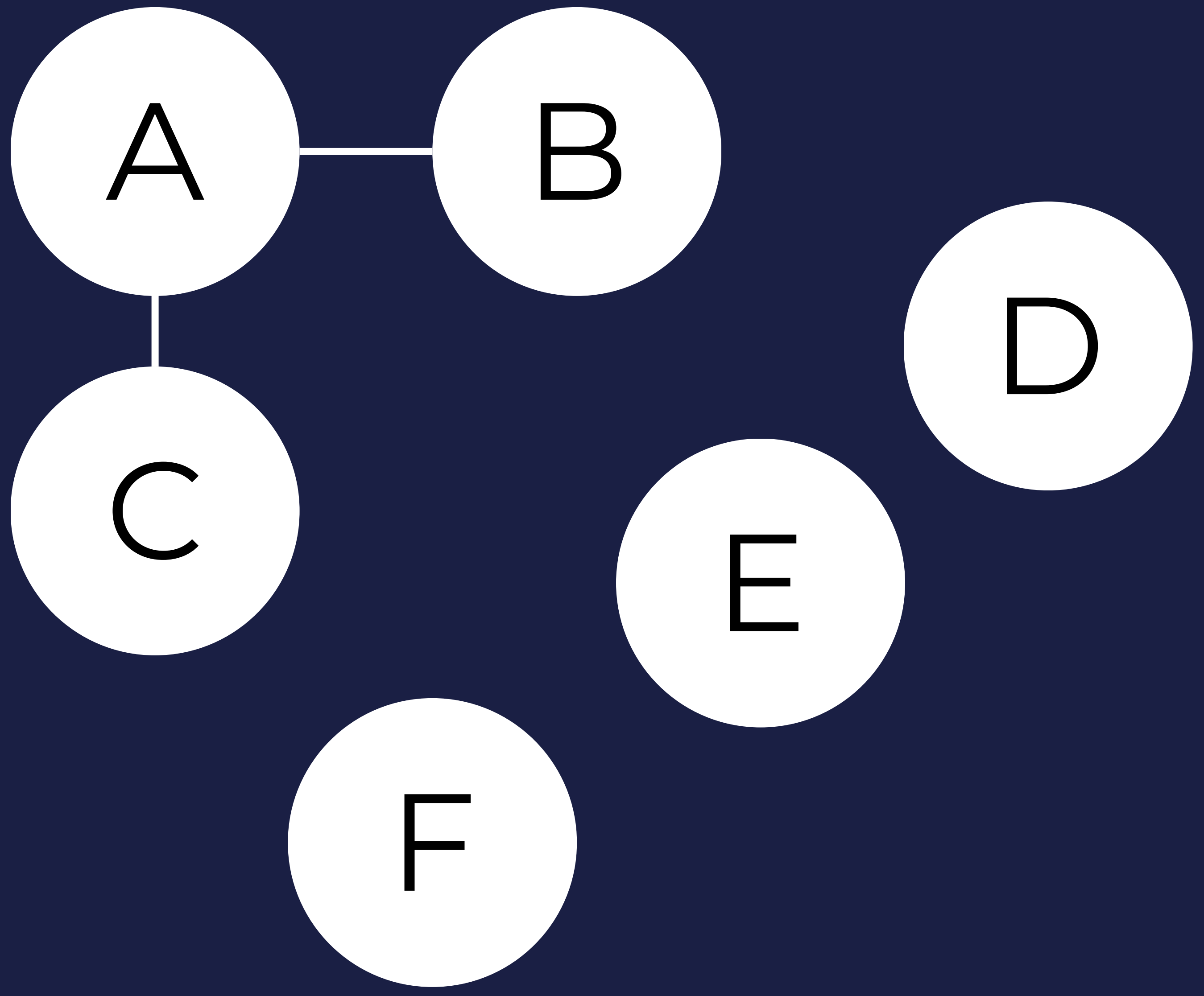
```
graph = {
 'A': {'B', 'C'},
 'B': {'A', 'D', 'E'},
 'C': {'A', 'F'},
 'D': {'B'},
 'E': {'B', 'F'},
 'F': {'C', 'E'}
}
```

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'}
}
```

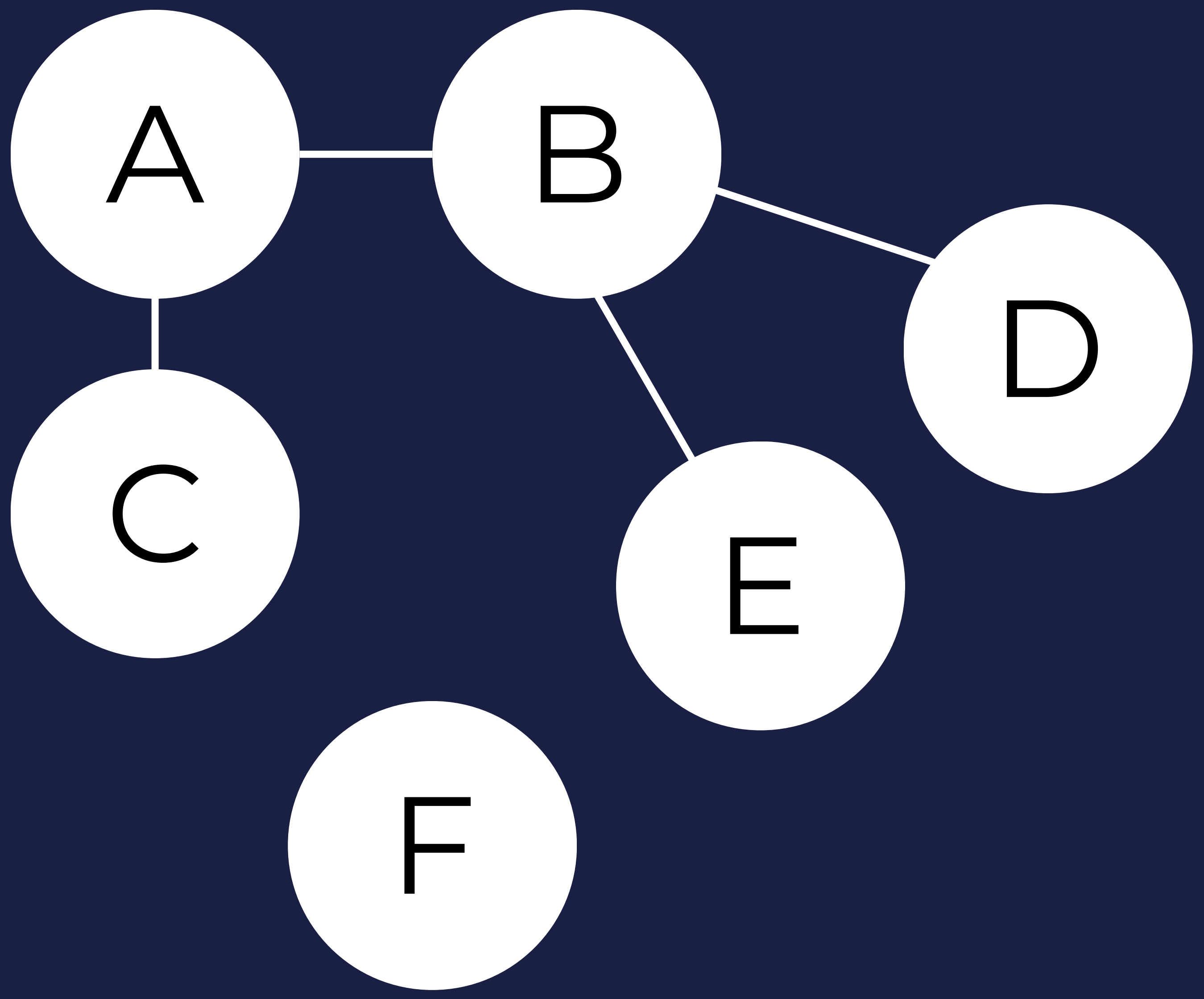
But lets make it look like a graph

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



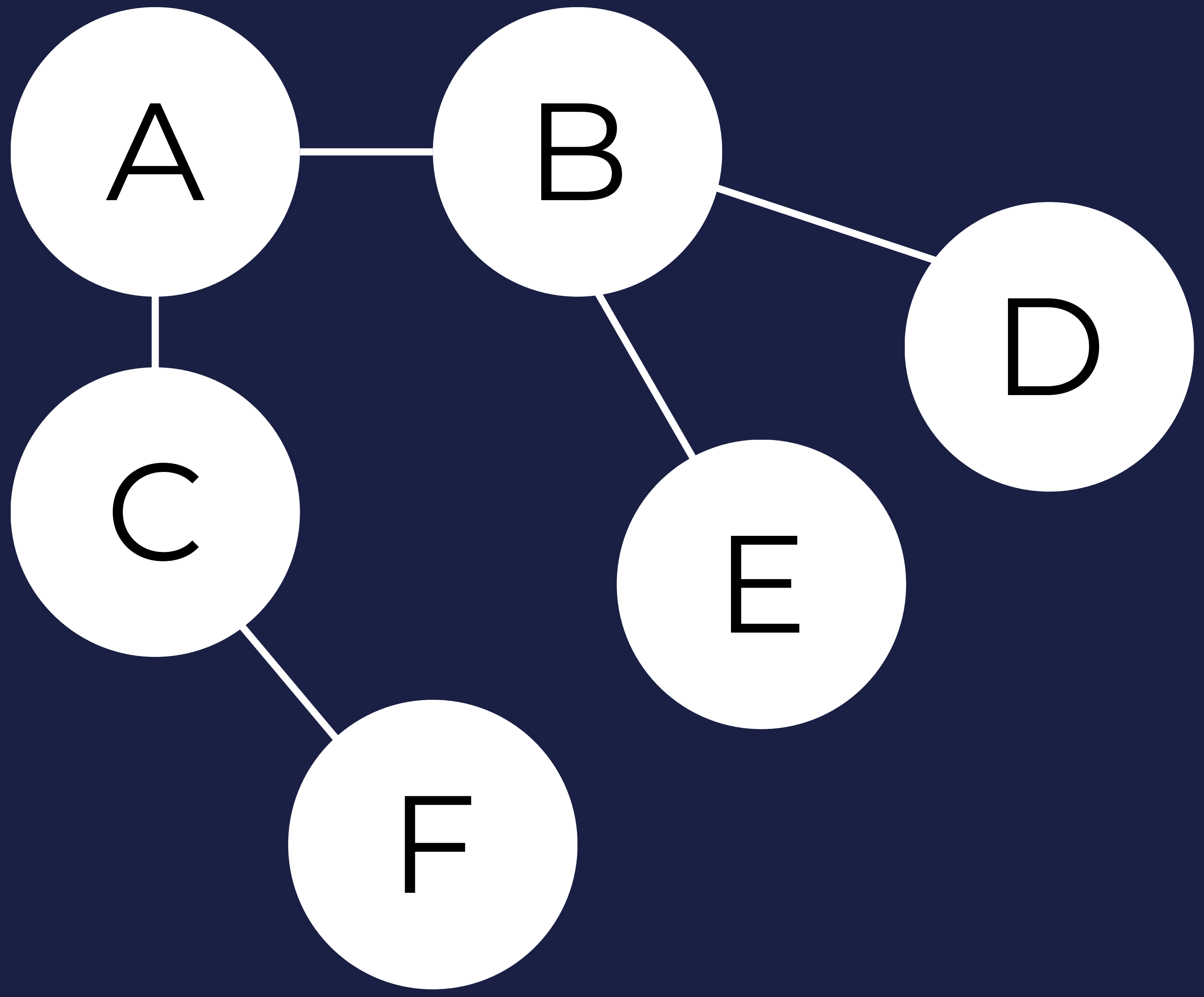
But lets make it look like a graph

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



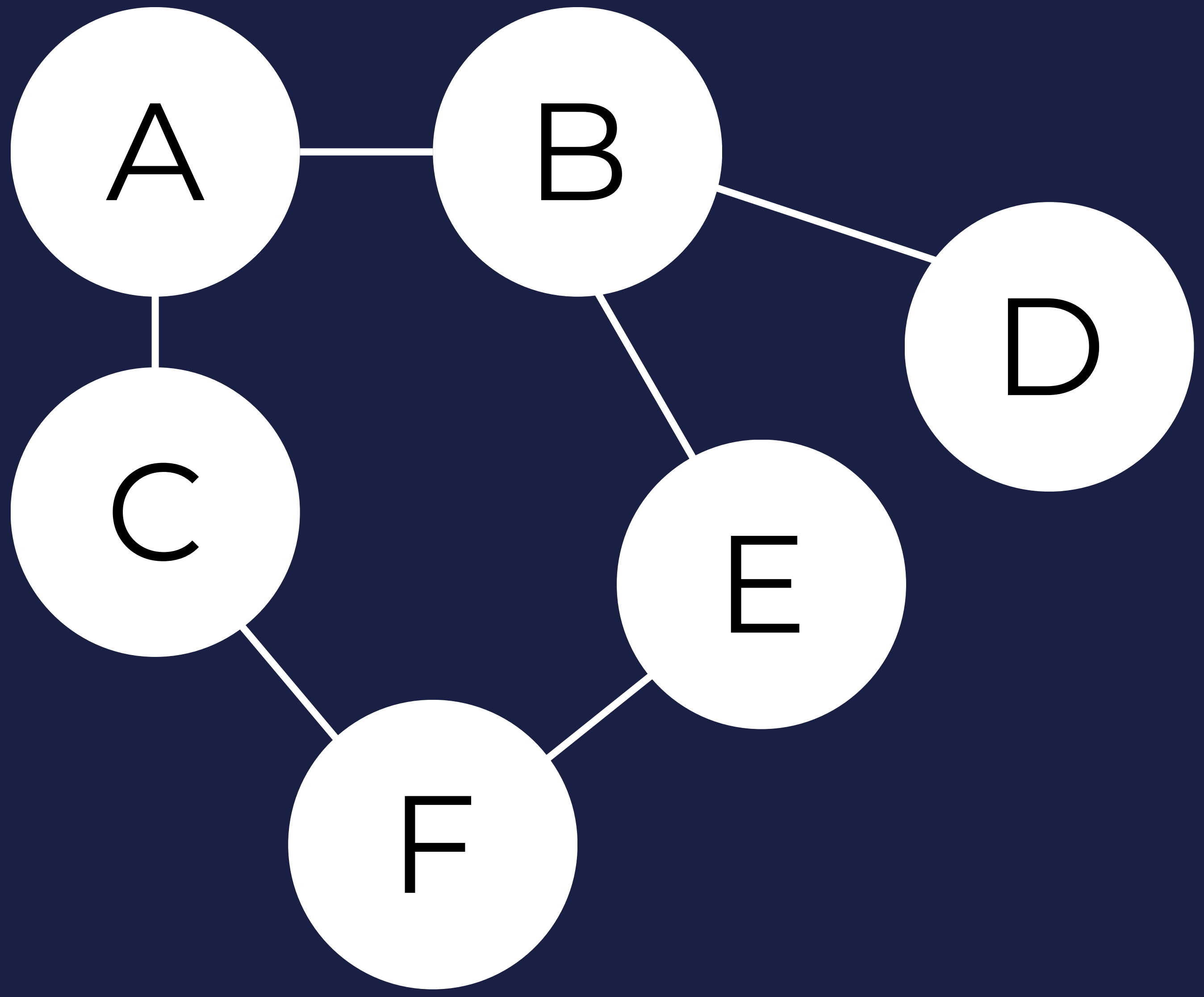
But lets make it look like a graph

```
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'}
}
```

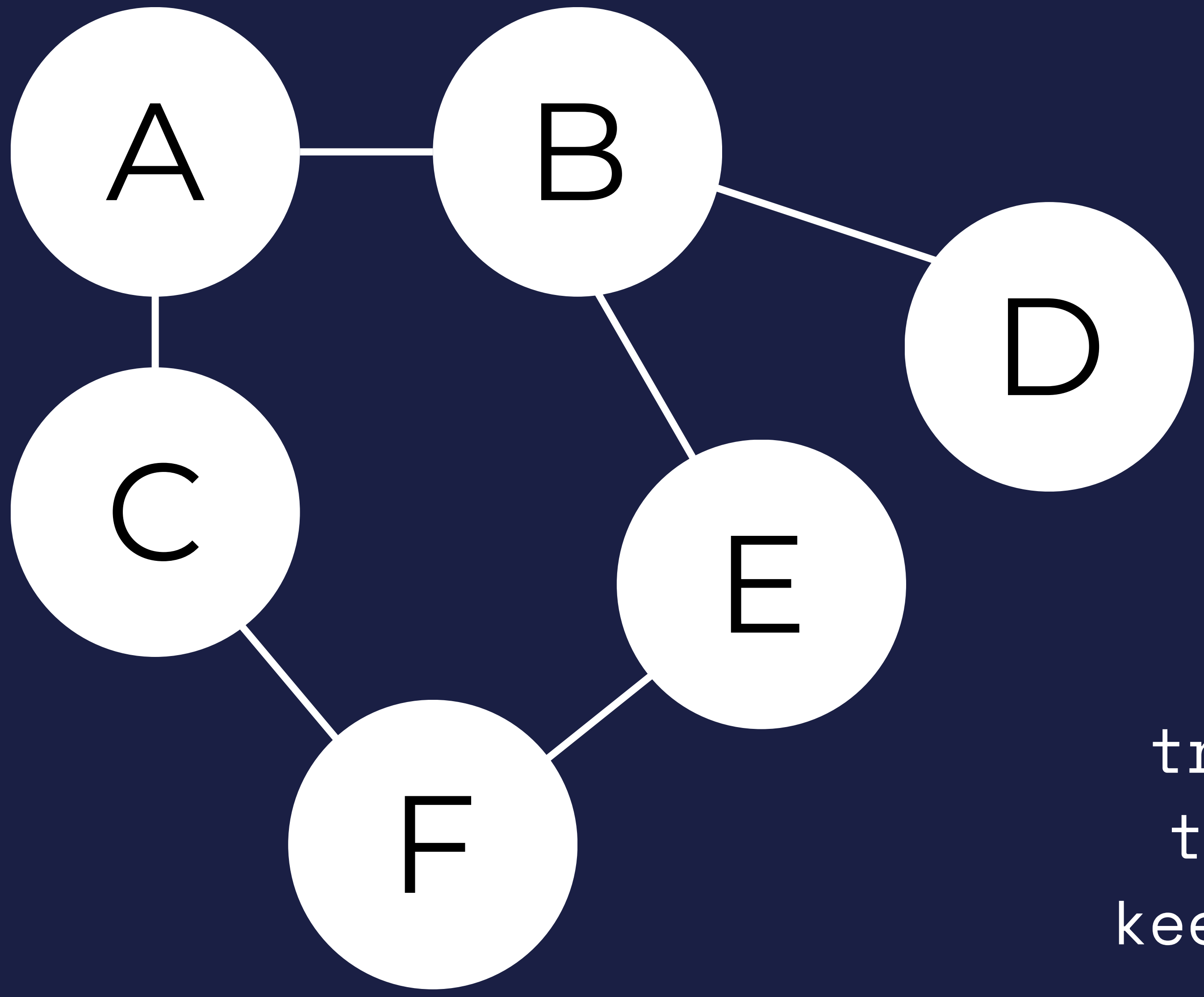


Then we choose a starting node. In our case, we will start with A, and we will track which items we have visited

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

start_node = "A"

visited = [ ]
```



DFS Uses a "Stack" to track what elements need to be searched. We will keep the stack here in the bottom right

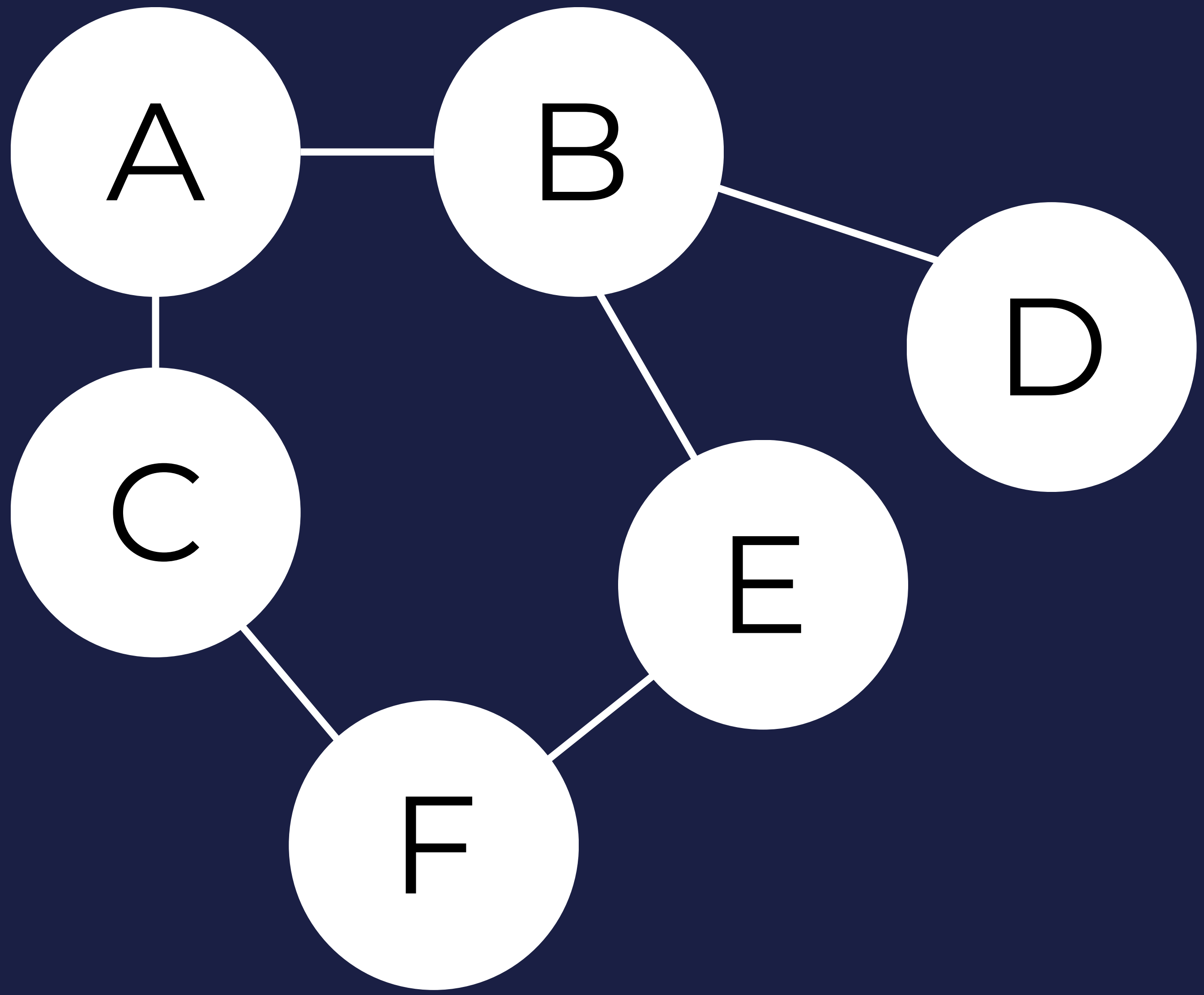
STACK

The first element we will check is “A” and we will add it to the stack!

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

start_node = “A”

visited = [ ]
```



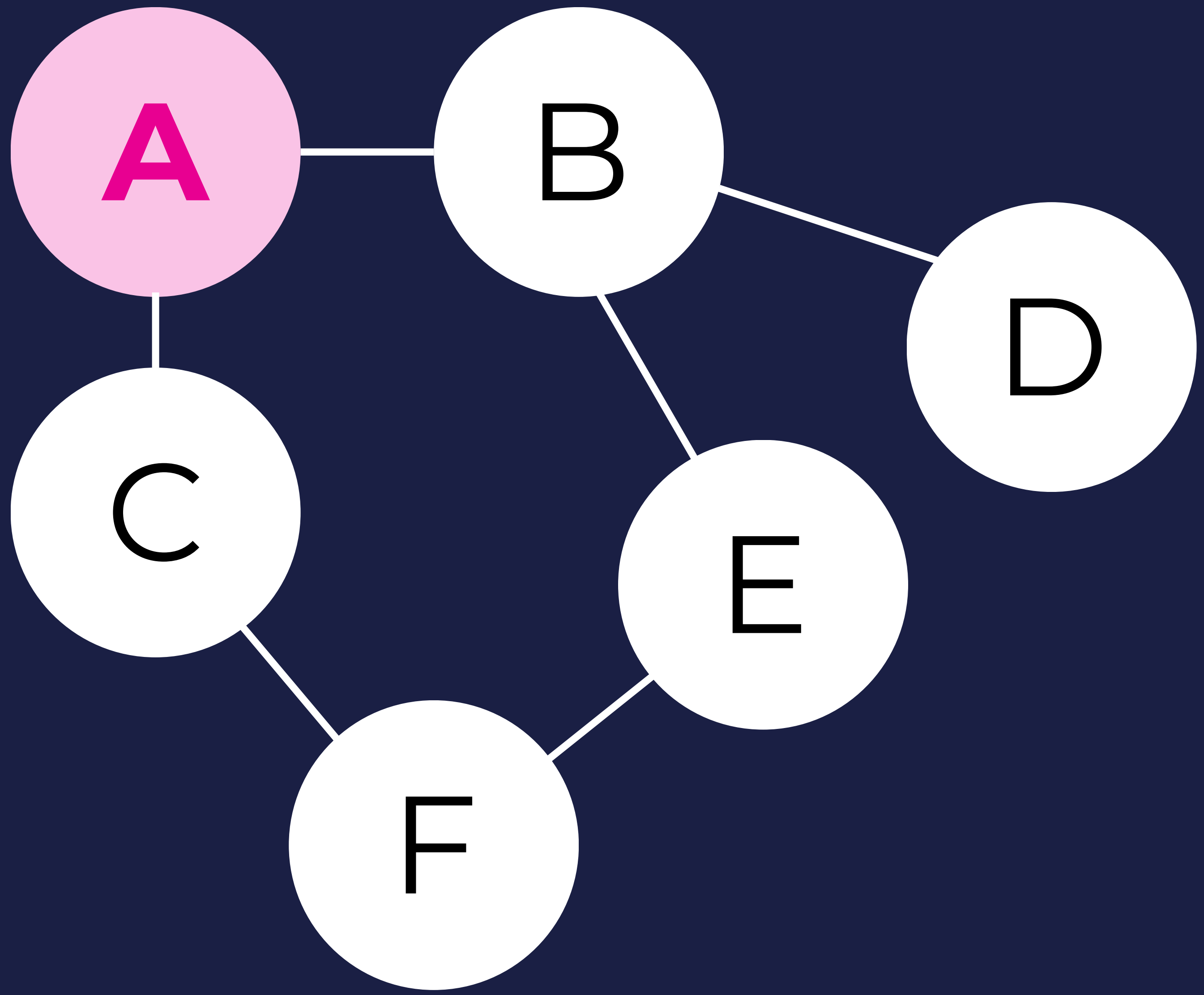
STACK

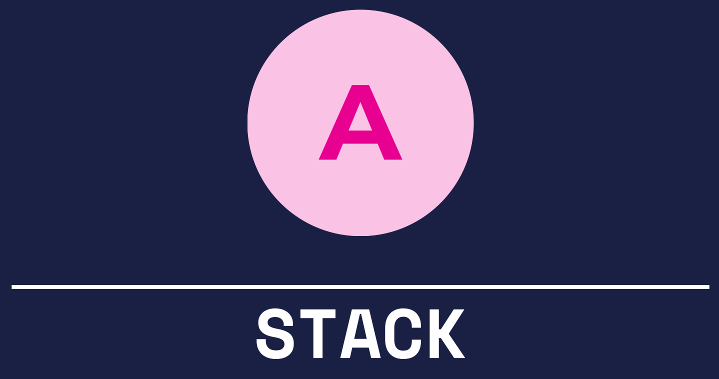
The first element we will check is “A” and we will add it to the stack!

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

start_node = “A”

visited = [ ]
```



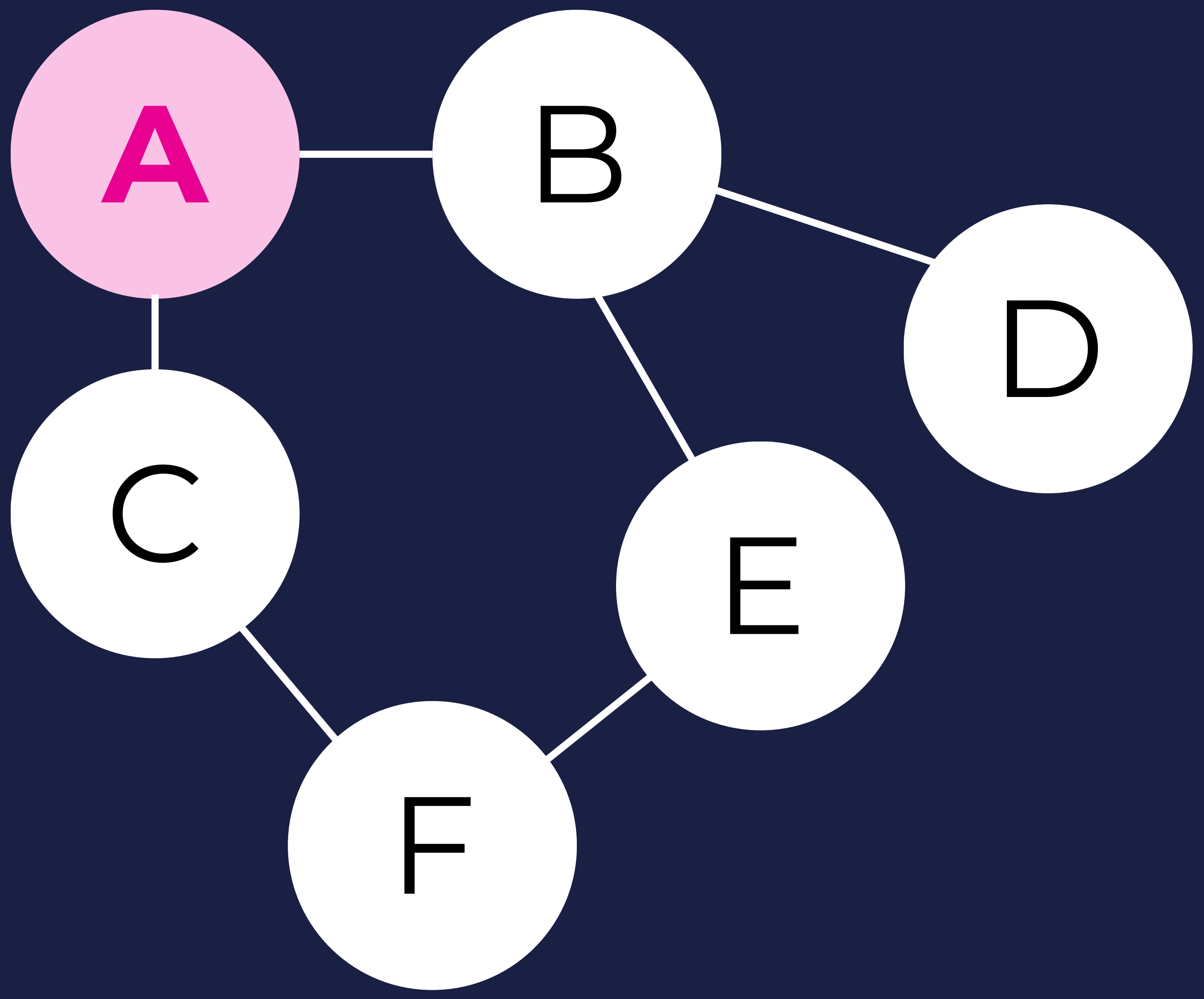


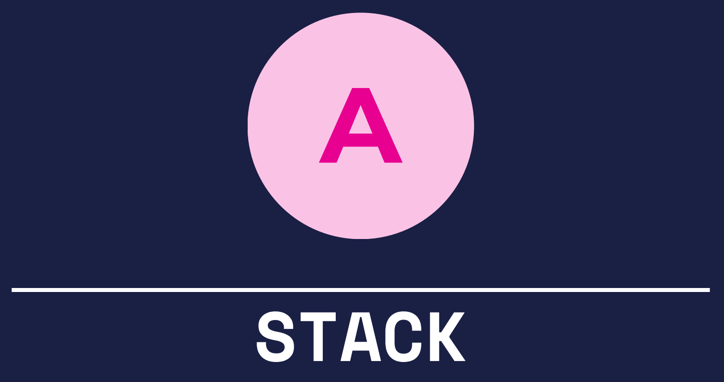
Once we check it, we pop “A” off the stack and mark it as visited.

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

start_node = “A”

visited = [ ]
```



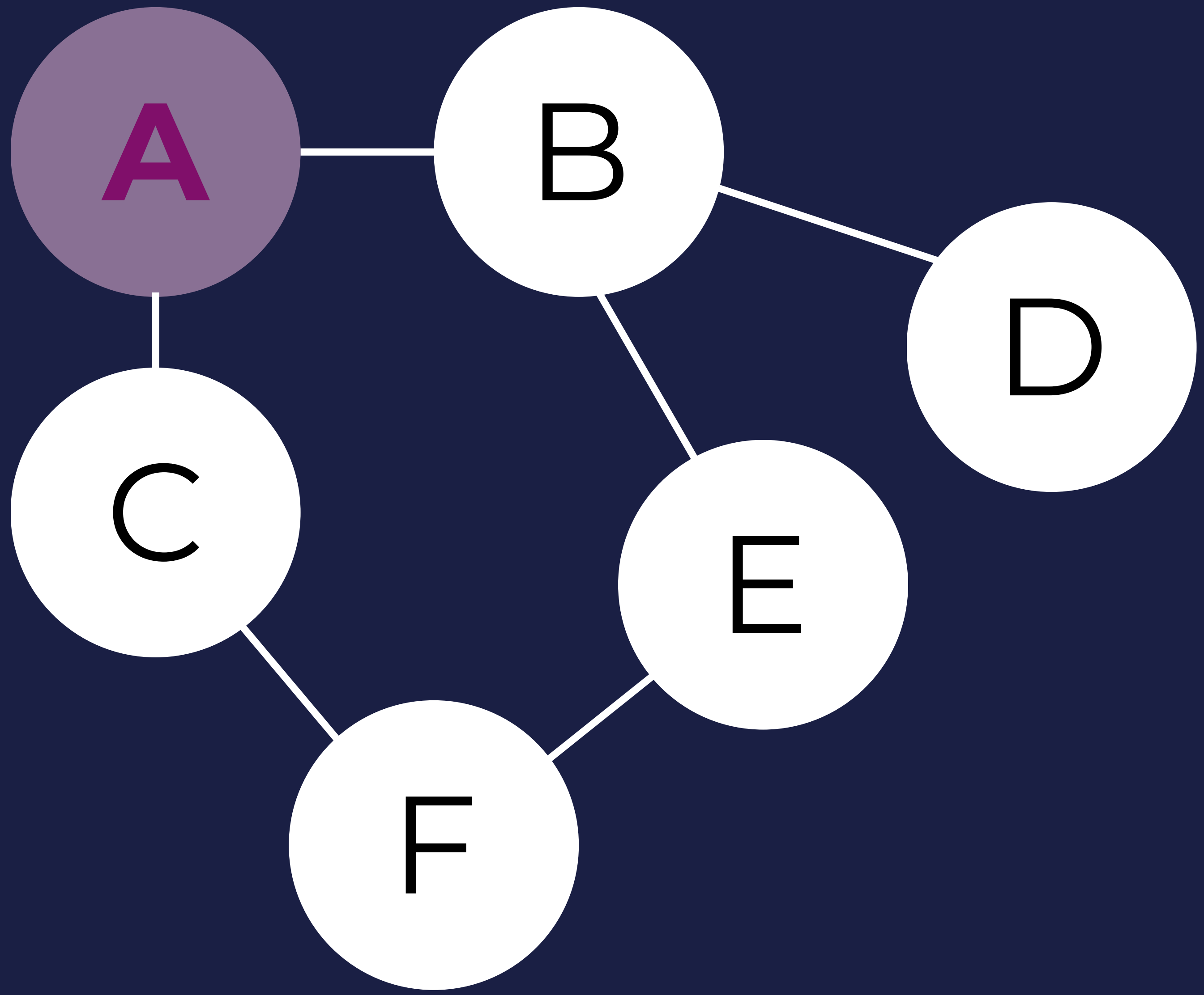


Once we check it, we pop “A” off the stack and mark it as visited.

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

start_node = “A”

visited = [
    “A”,
]
```



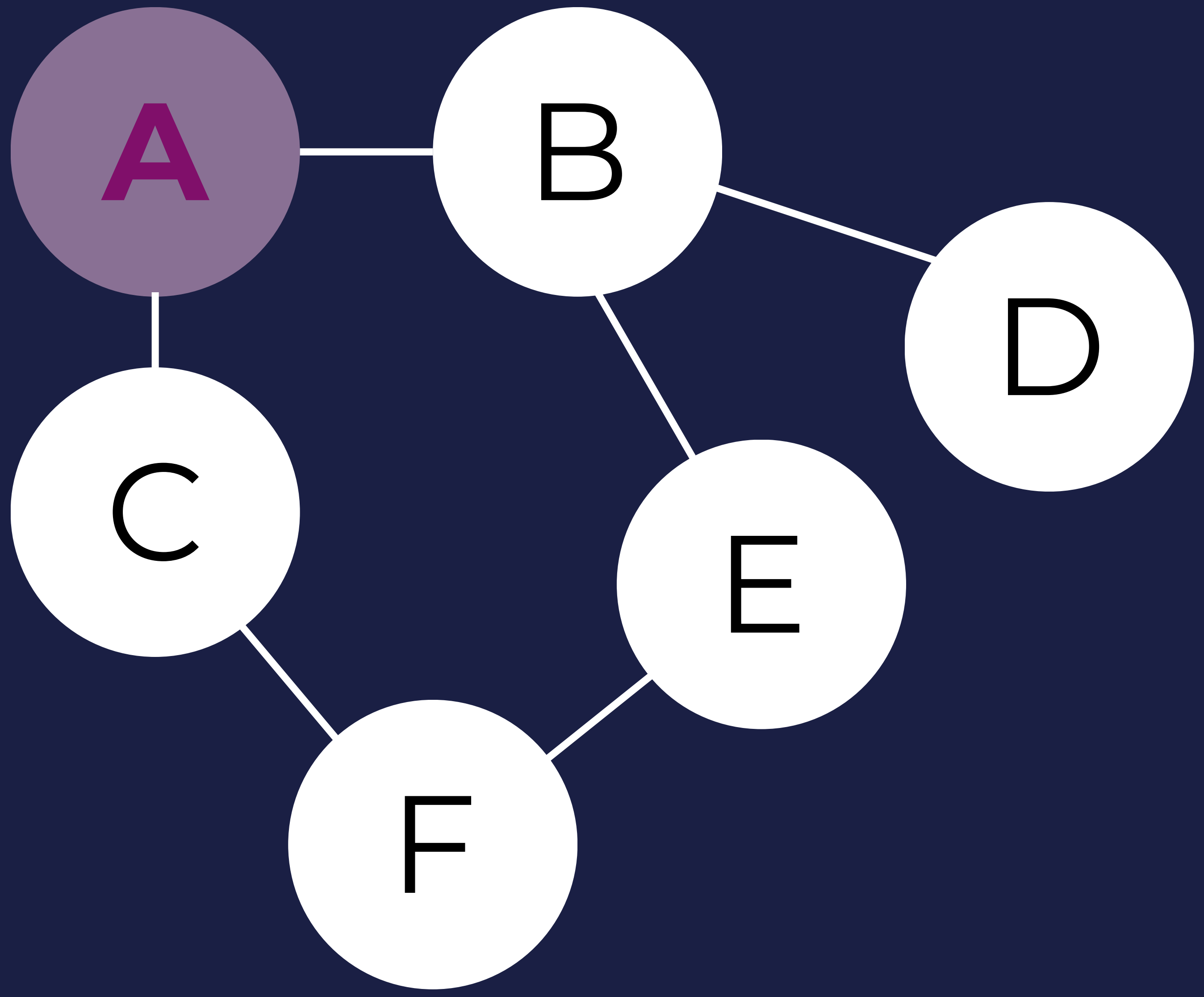
STACK

Then we add the elements that are neighbors of “A” to the Stack.

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

start_node = “A”

visited = [
    “A”,
]
```



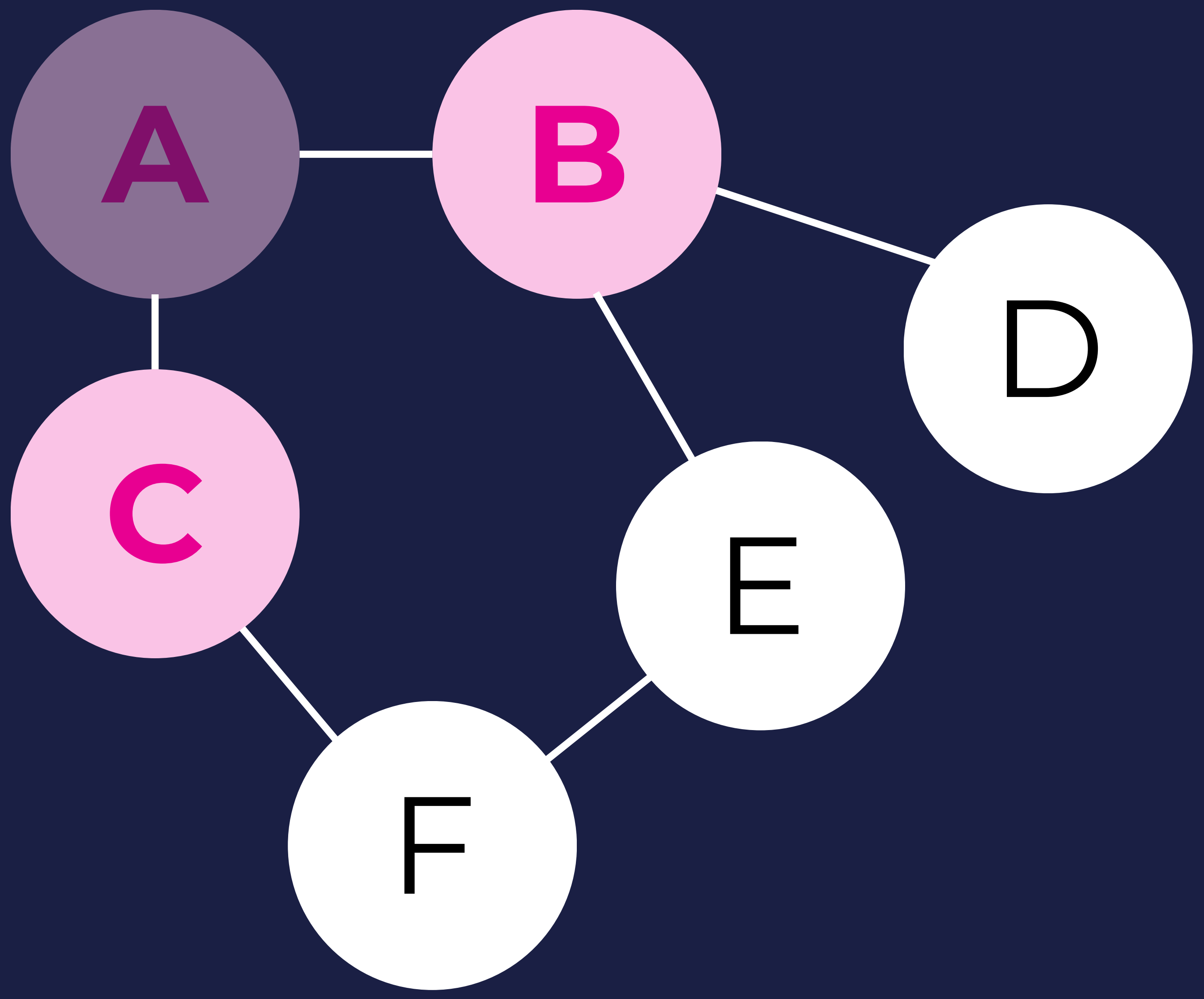
STACK

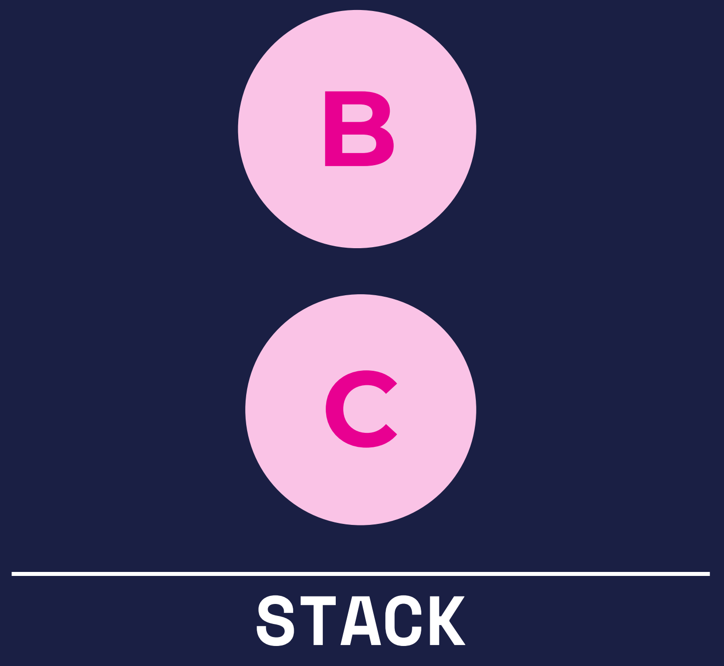
Then we add the elements that are neighbors of “A” to the Stack.

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

start_node = “A”

visited = [
    “A”,
]
```



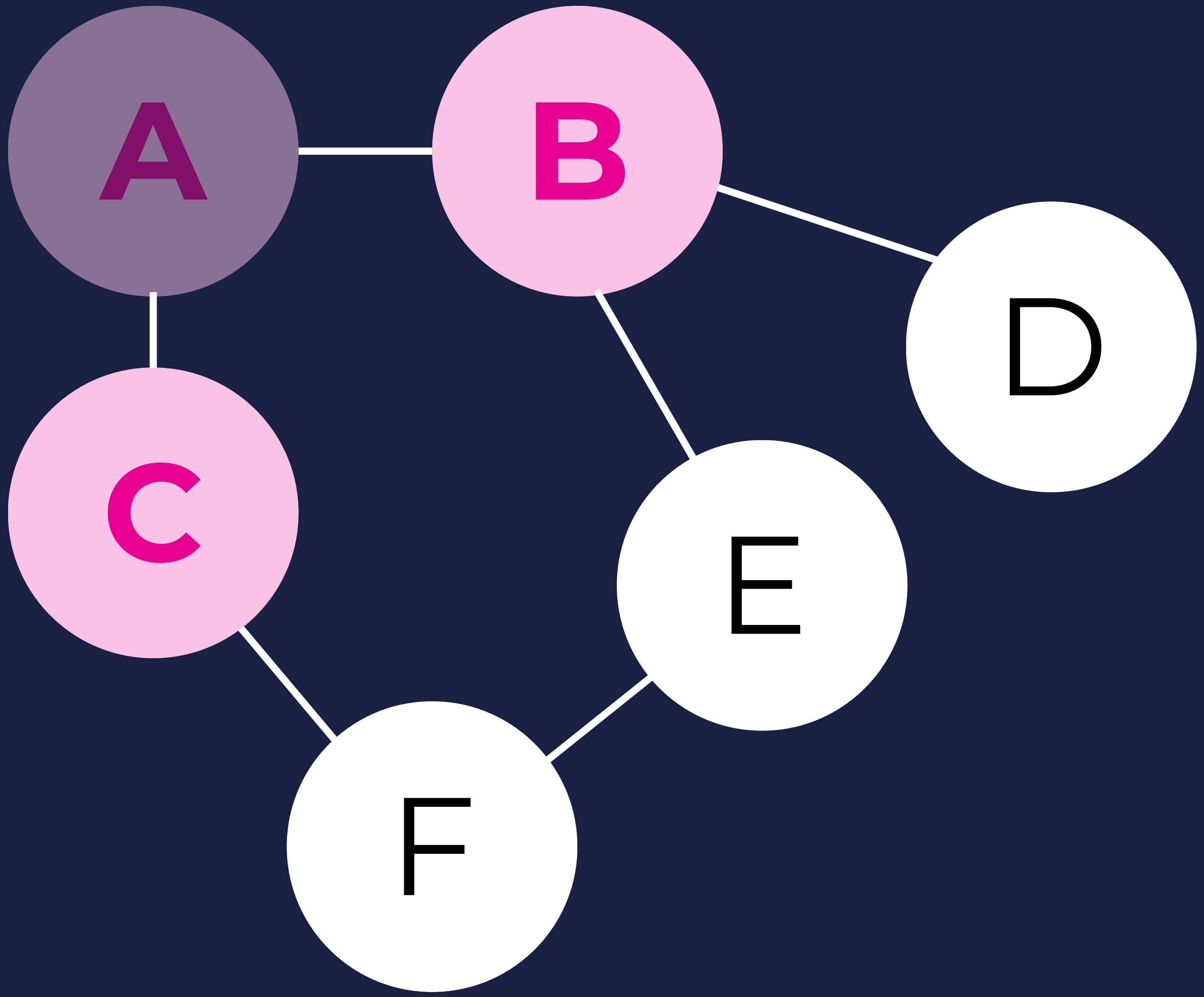


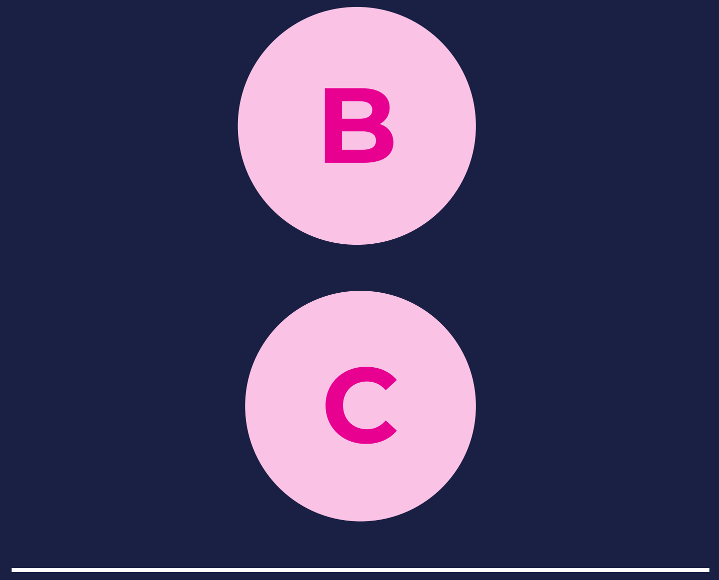
B is now the first item in the stack. We pop “B” off the stack and mark it as visited.

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

start_node = “A”

visited = [
    “A”,
]
```





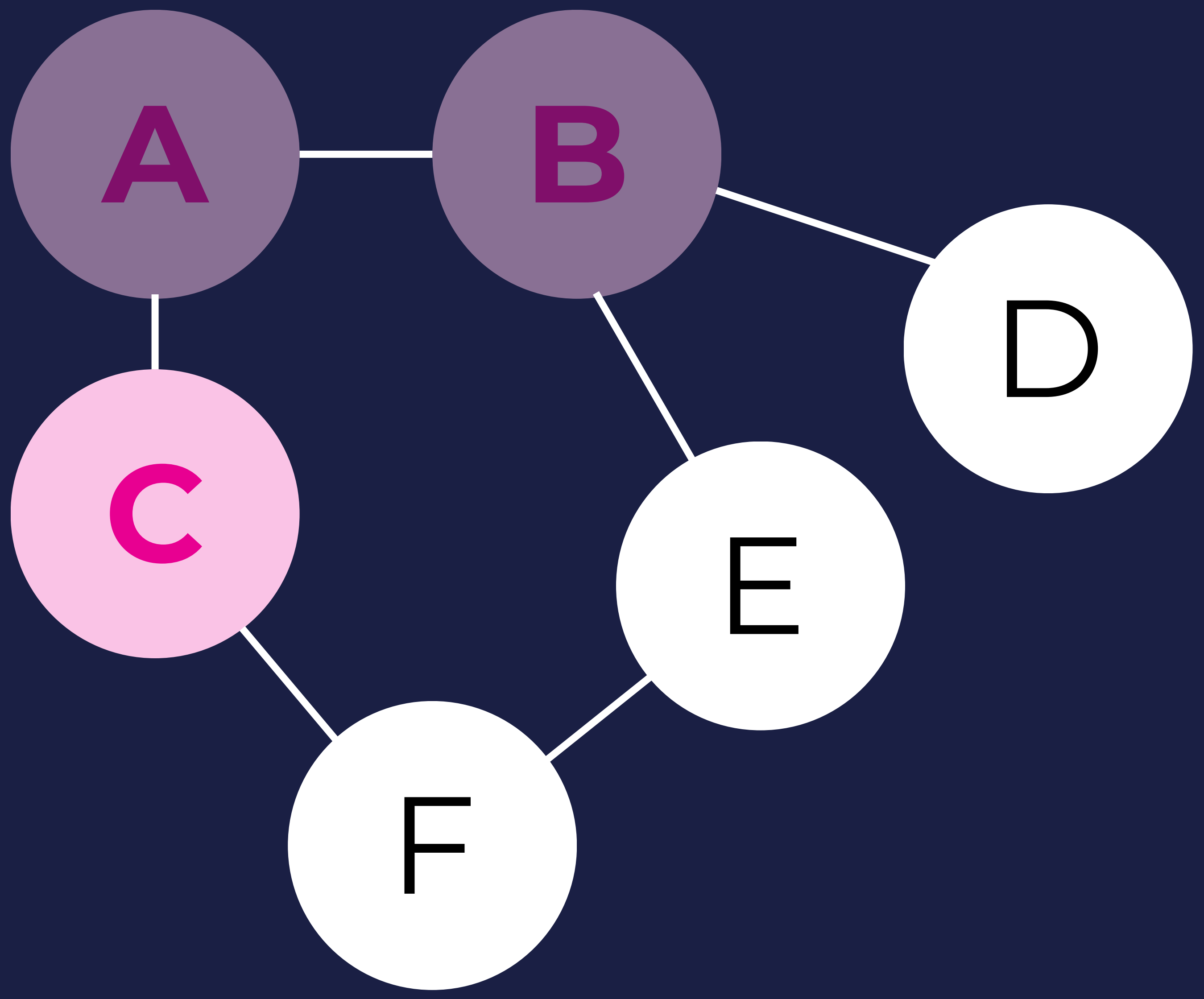
STACK

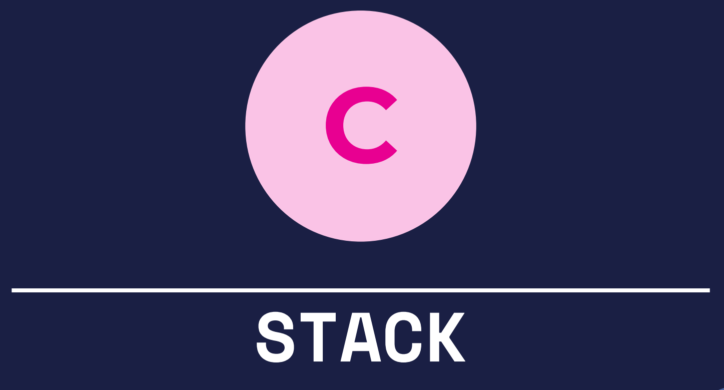
B is now the first item in the stack. We pop “B” off the stack and mark it as visited.

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

start_node = “A”

visited = [
    “A”, “B”,
]
```



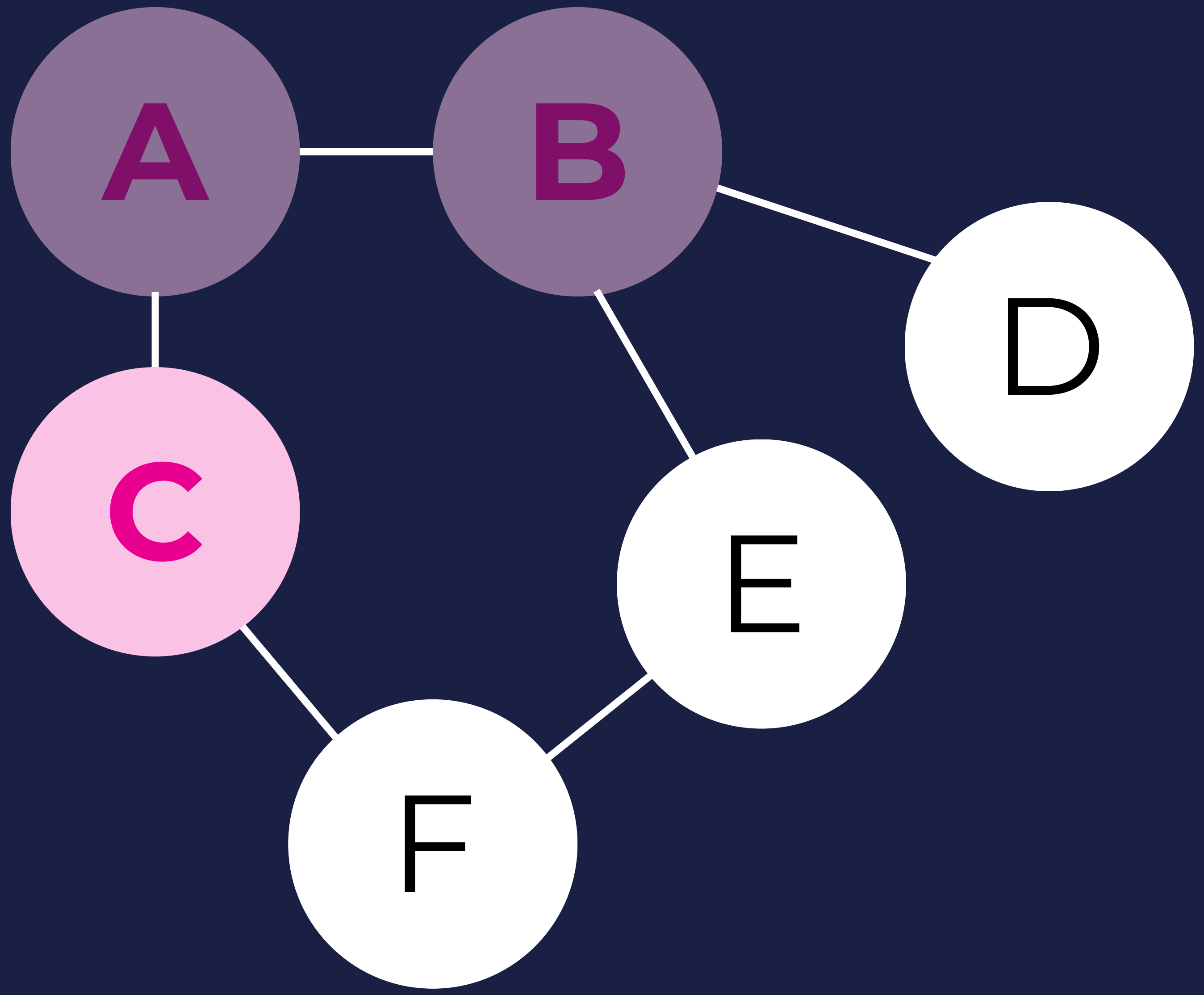


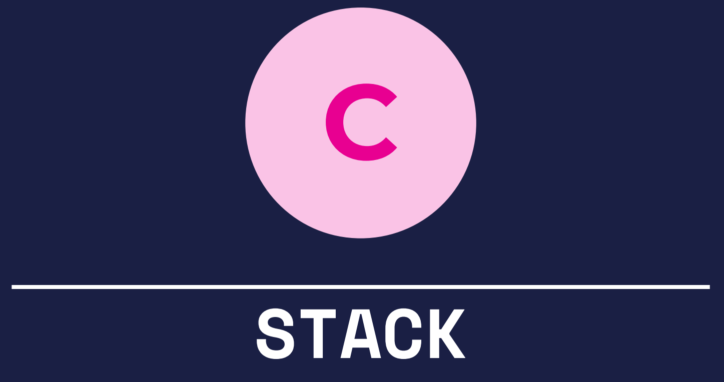
Now we add all of B's neighbors to the top of the stack

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

start_node = "A"

visited = [
    "A", "B",
]
```



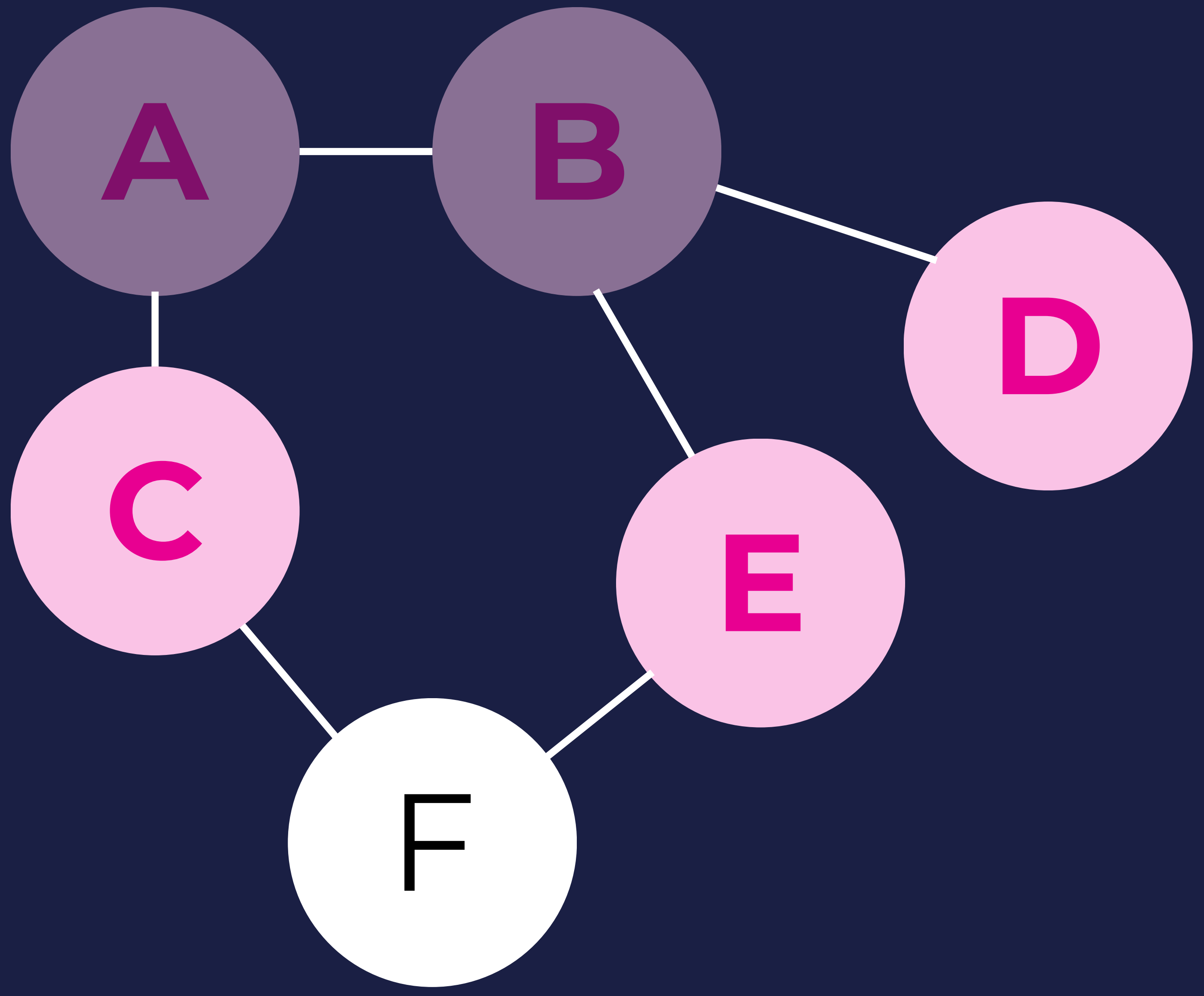


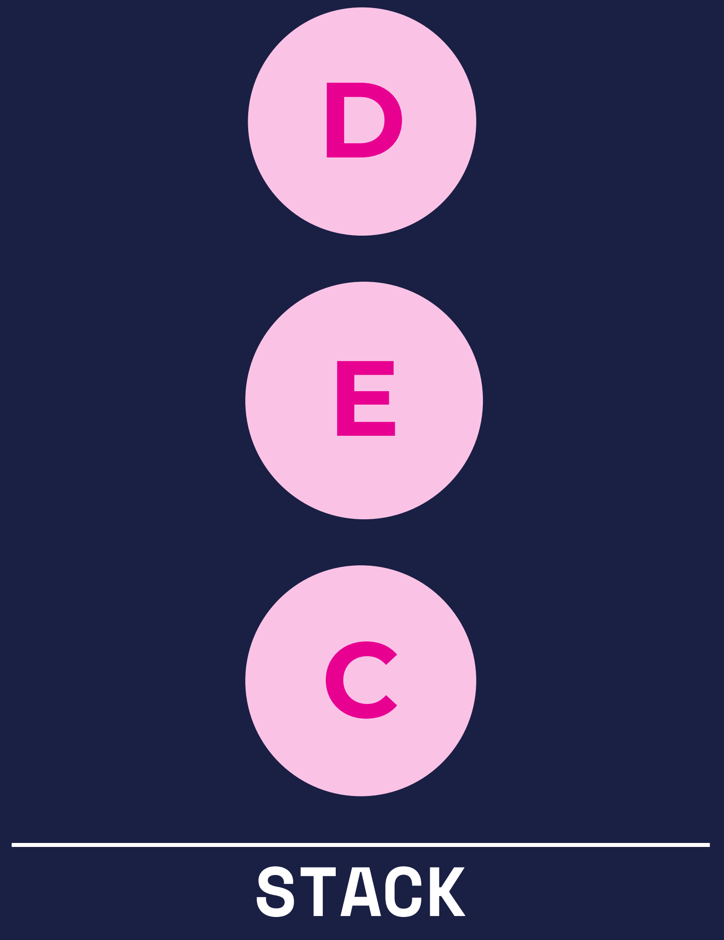
As you can see, C was the top of the stack, but now D is! Because we added B’s neighbors to the top.

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

start_node = “A”

visited = [
    “A”, “B”,
]
```



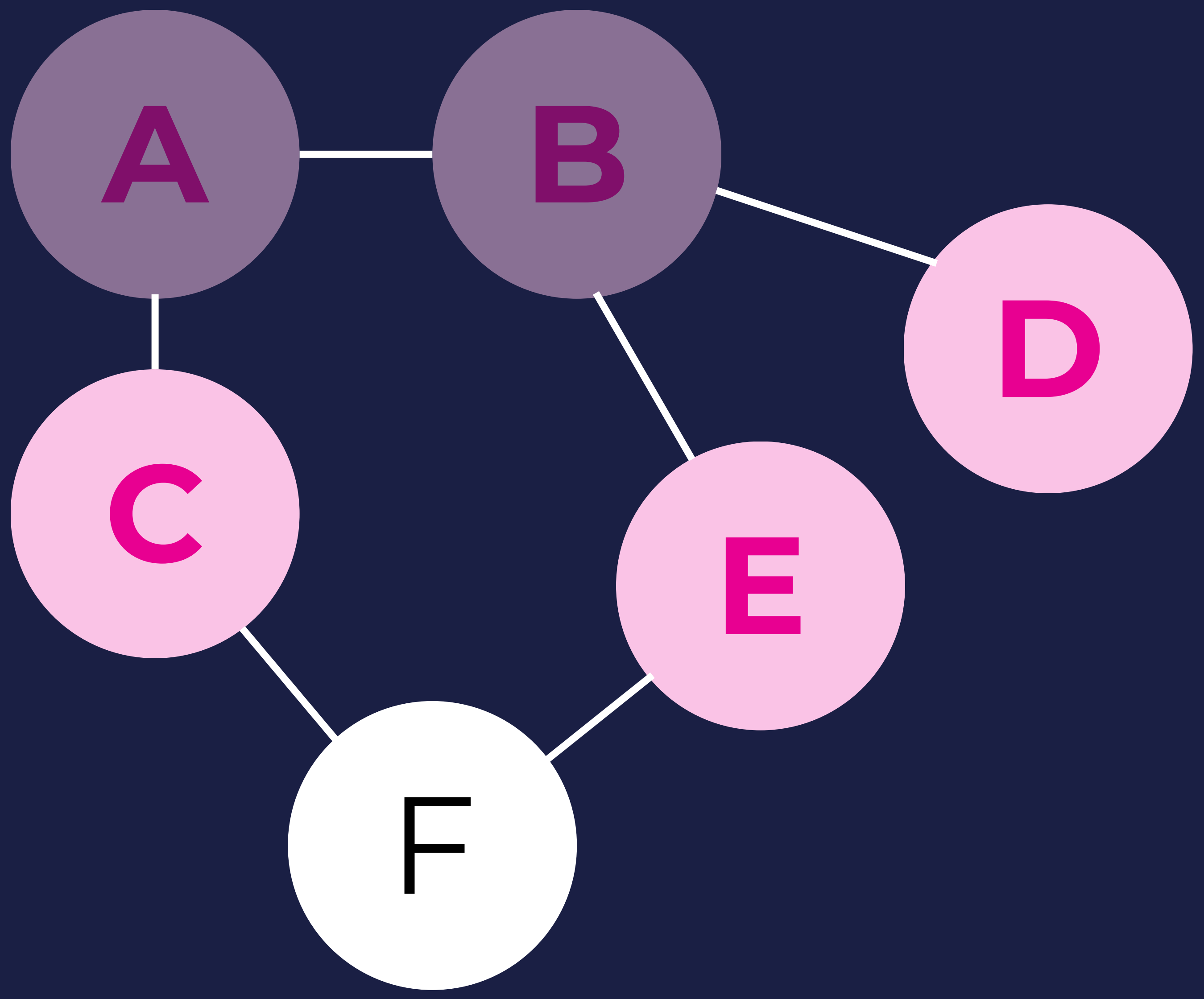


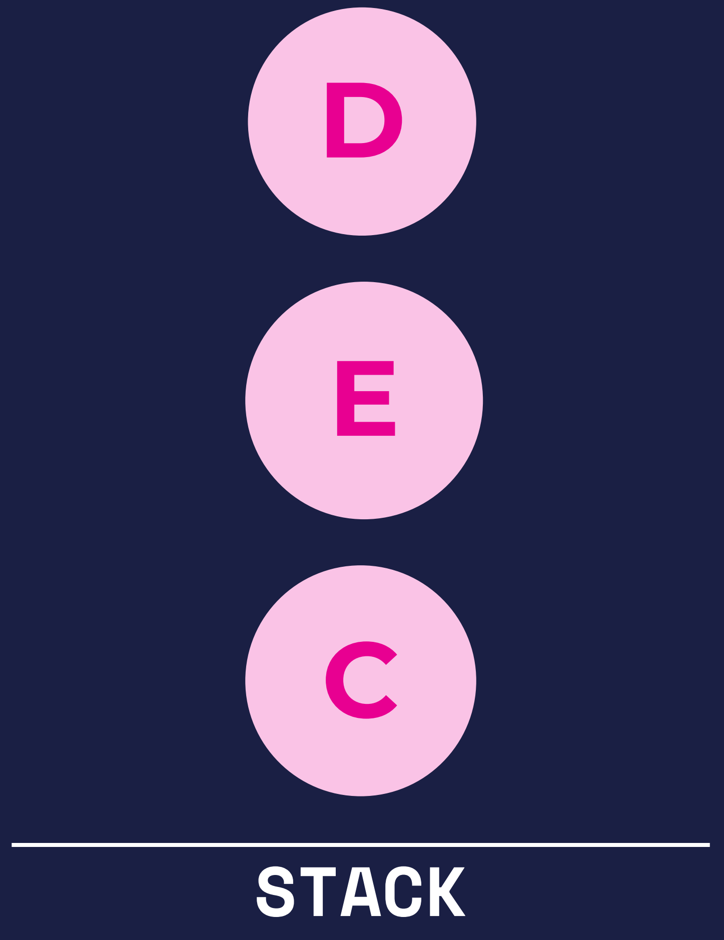
Now we pop D off the stack and mark as visited

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

start_node = "A"

visited = [
    "A", "B",
]
```



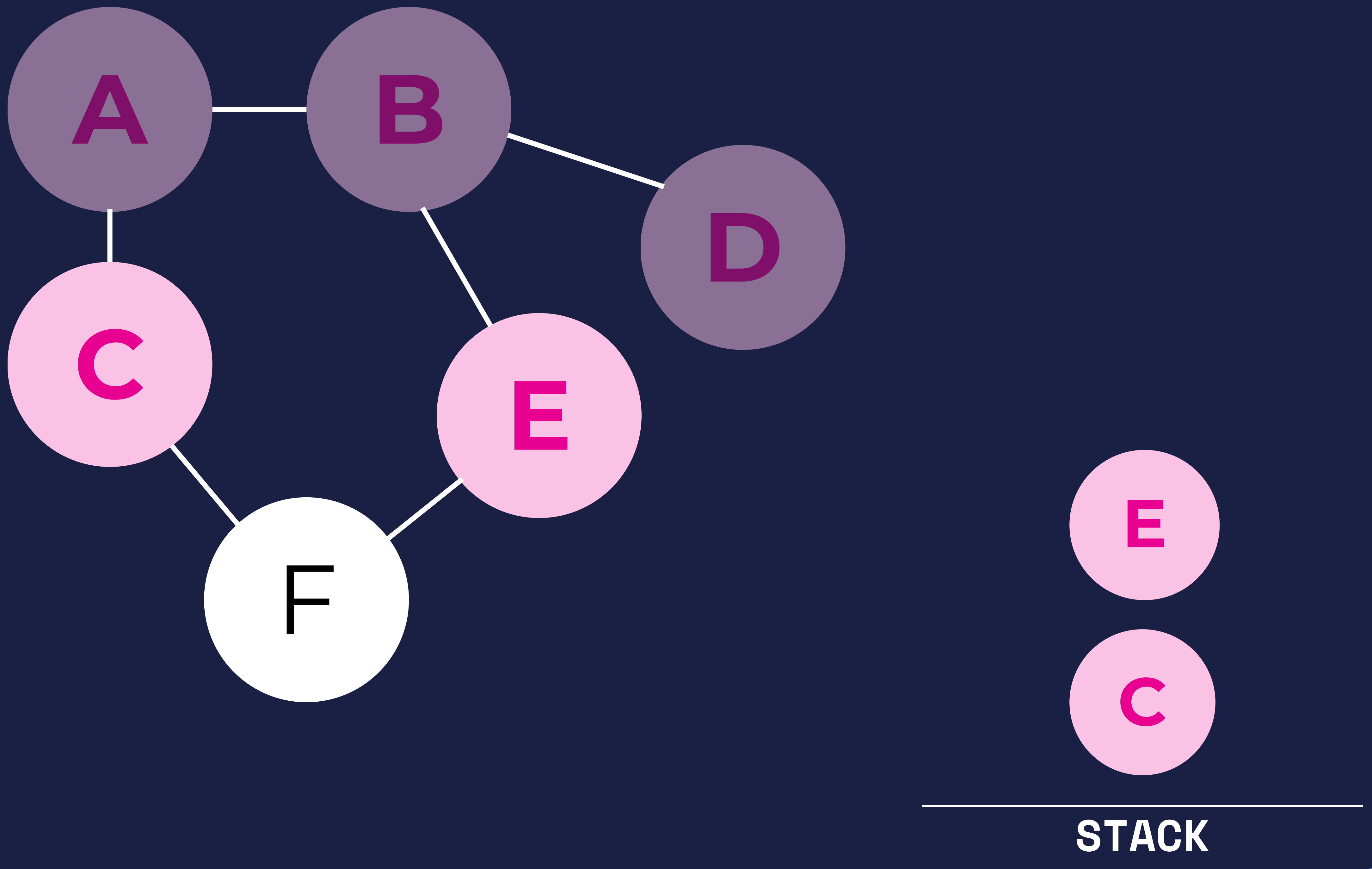


Now we pop D off the stack and mark as visited

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

start_node = "A"

visited = [
    "A", "B", "D",
]
```

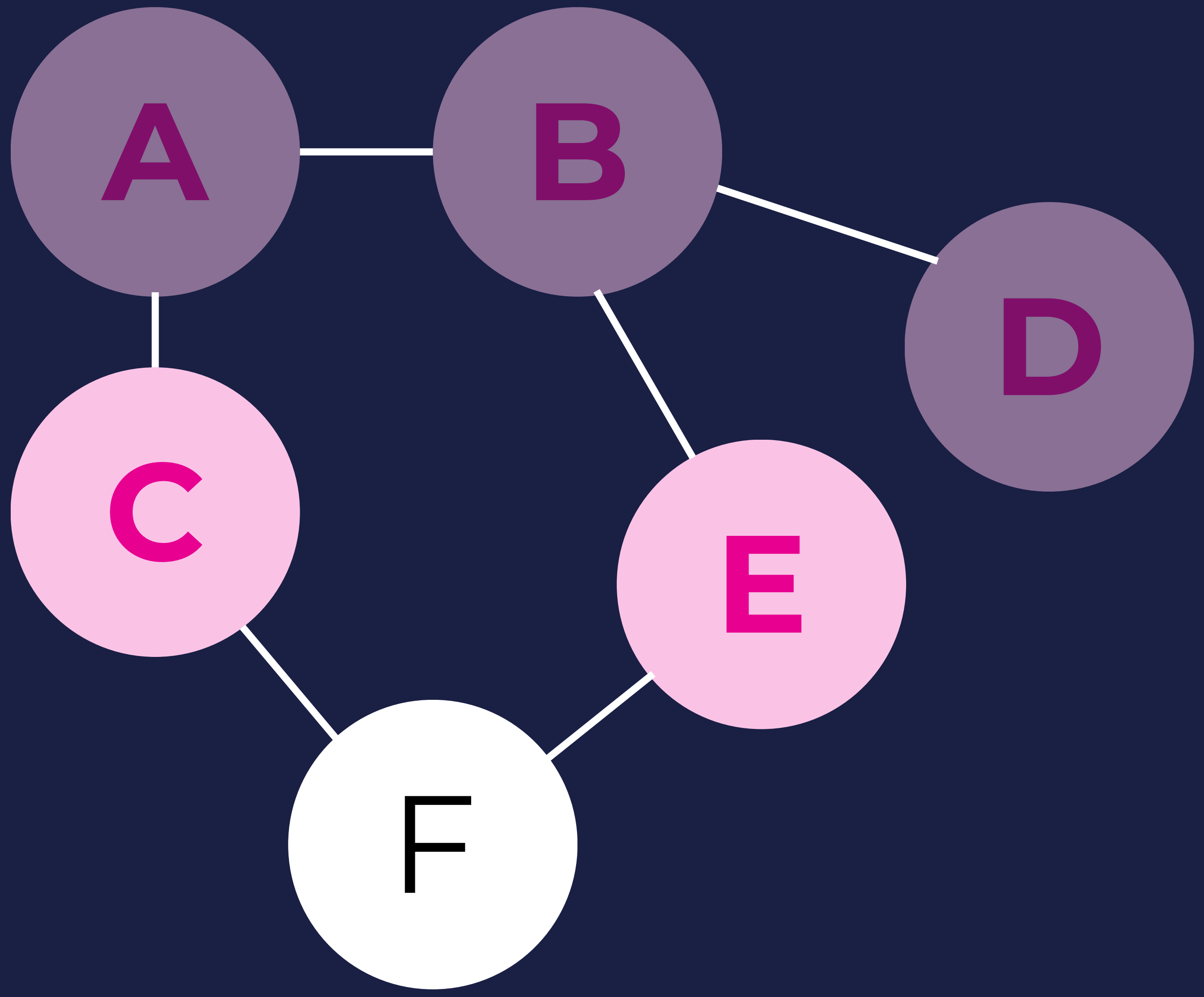


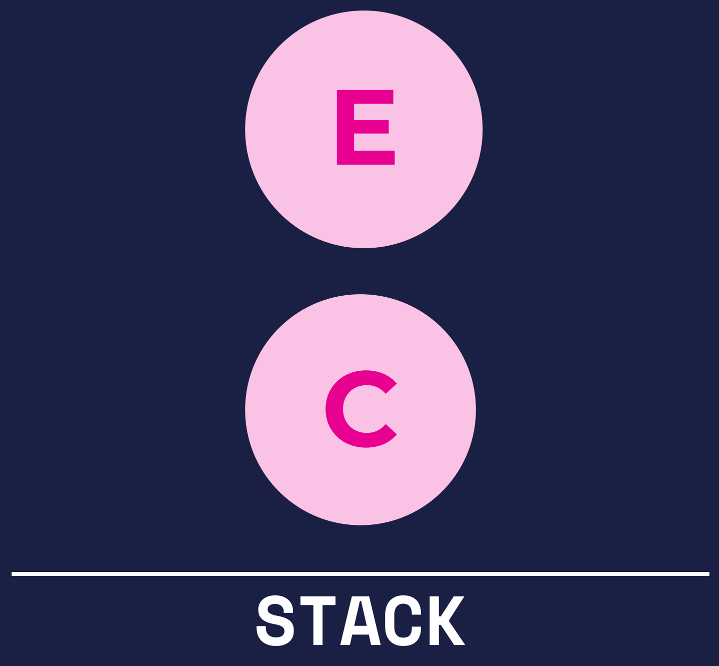
Since D doesn’t have any unvisited neighbors, we move on to E!

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

start_node = “A”

visited = [
    “A”, “B”, “D”,
]
```



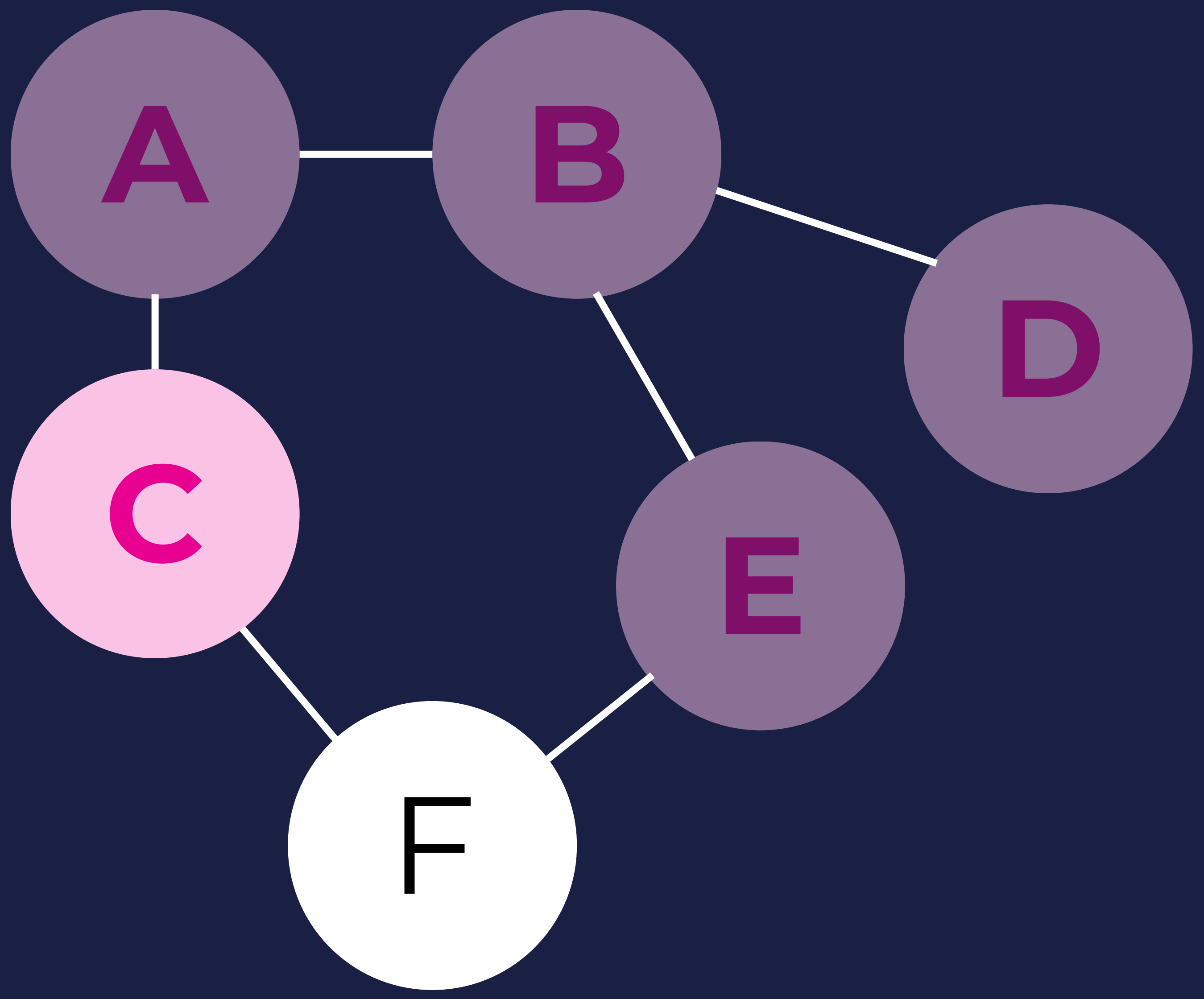


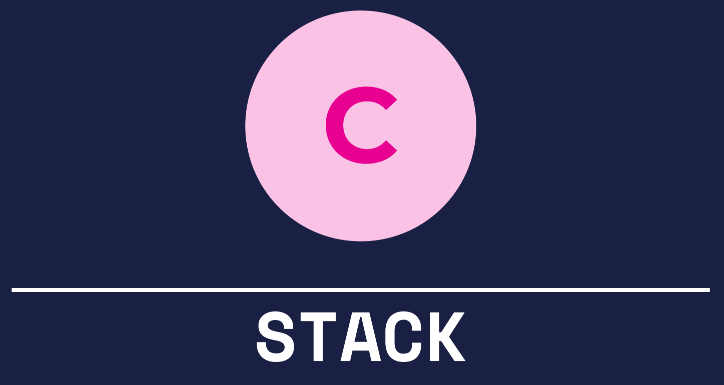
Since D doesn’t have any unvisited neighbors, we move on to E!

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

start_node = “A”

visited = [
    “A”, “B”, “D”,
    “E”,
]
```



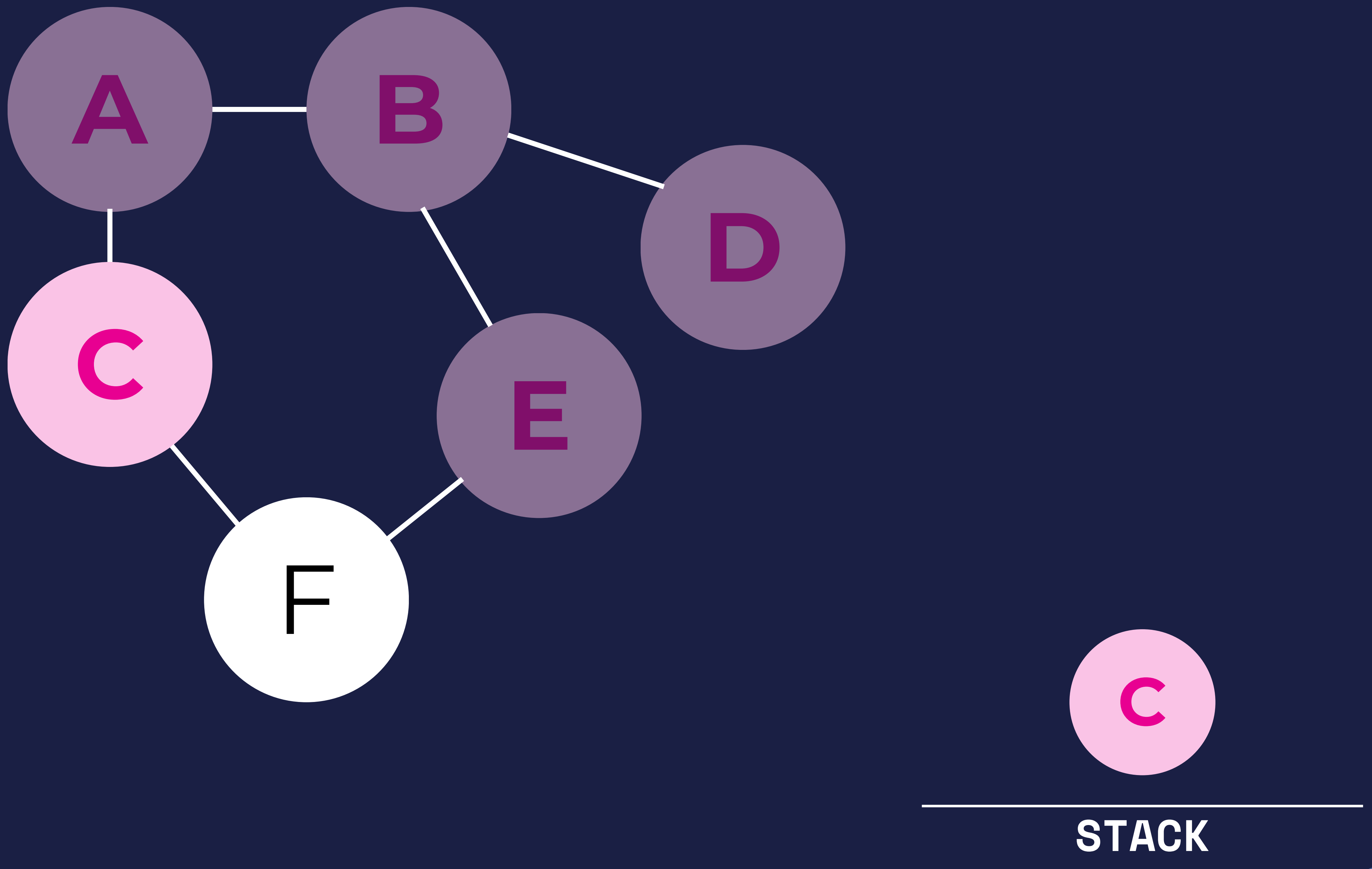


E has a neighbor that hasn't been visited yet, so we add it to the stack

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

start_node = "A"

visited = [
    "A", "B", "D",
    "E",
]
```

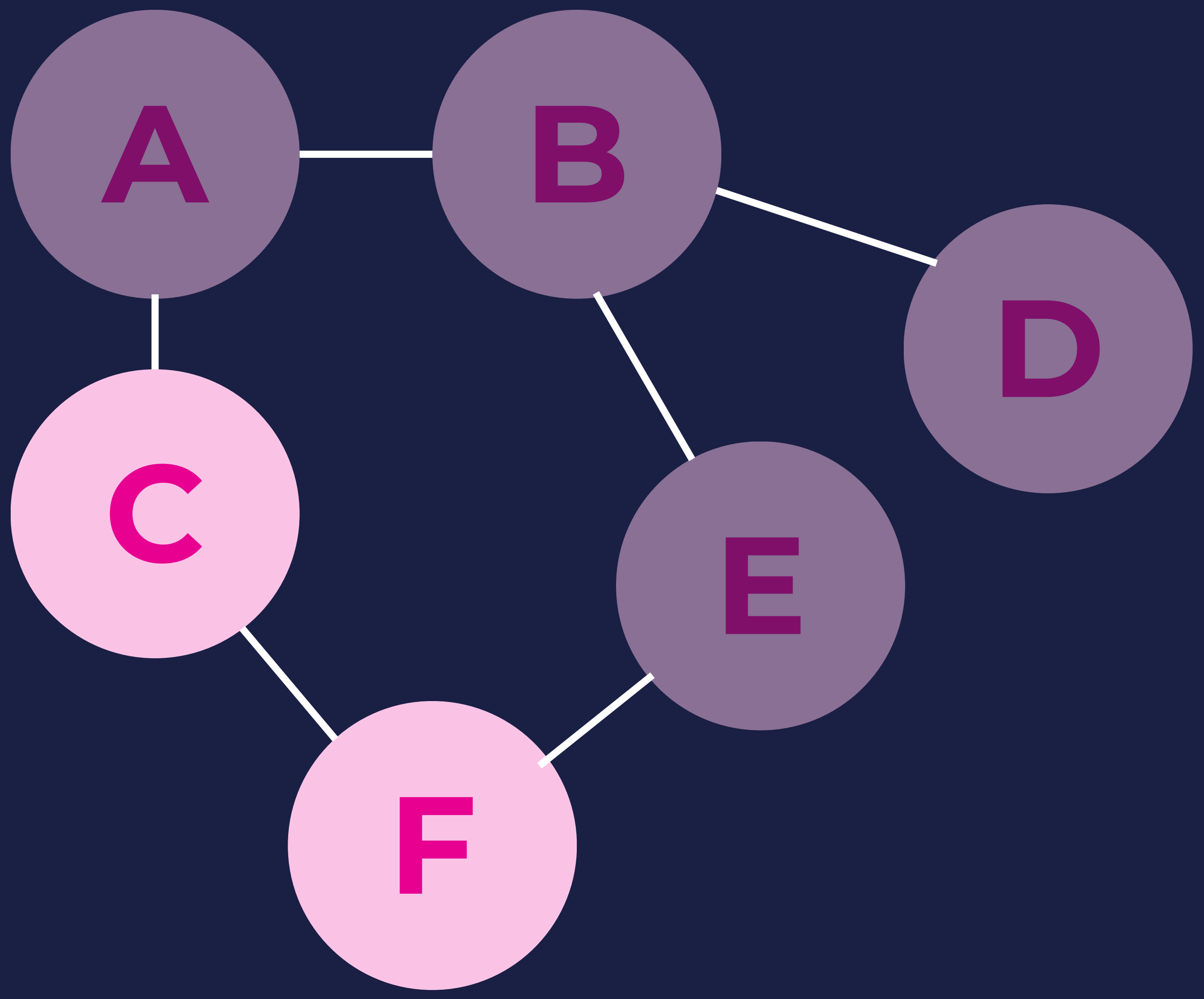


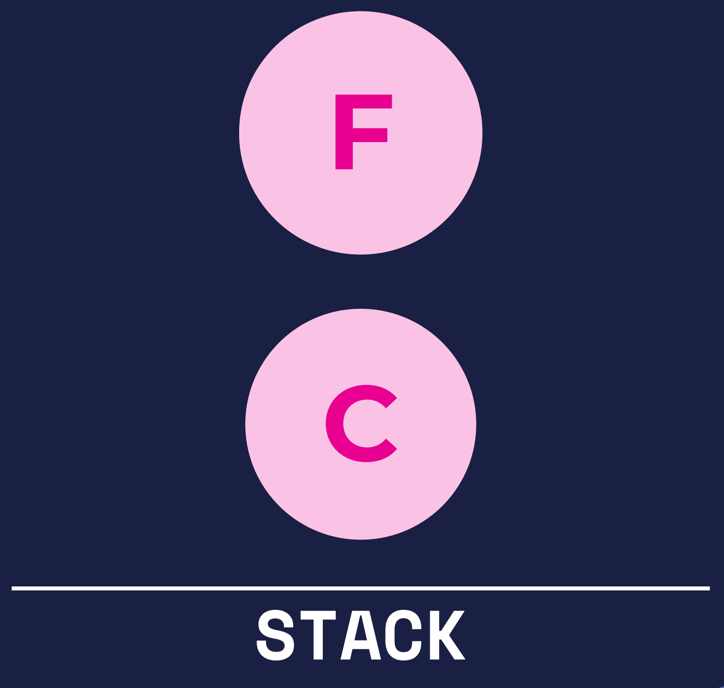
E has a neighbor that hasn’t been visited yet, so we add it to the stack

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

start_node = “A”

visited = [
    “A”, “B”, “D”,
    “E”,
]
```



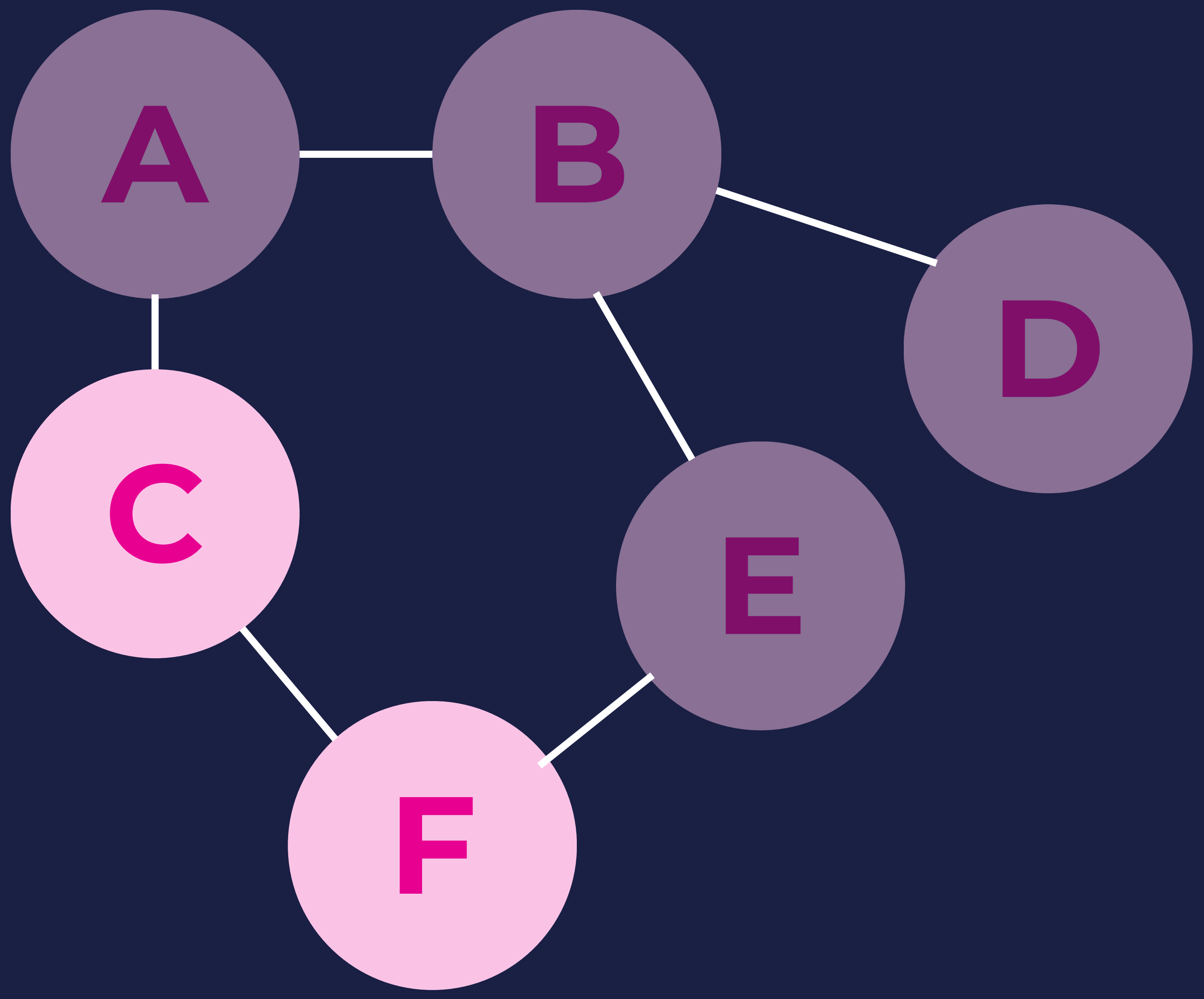


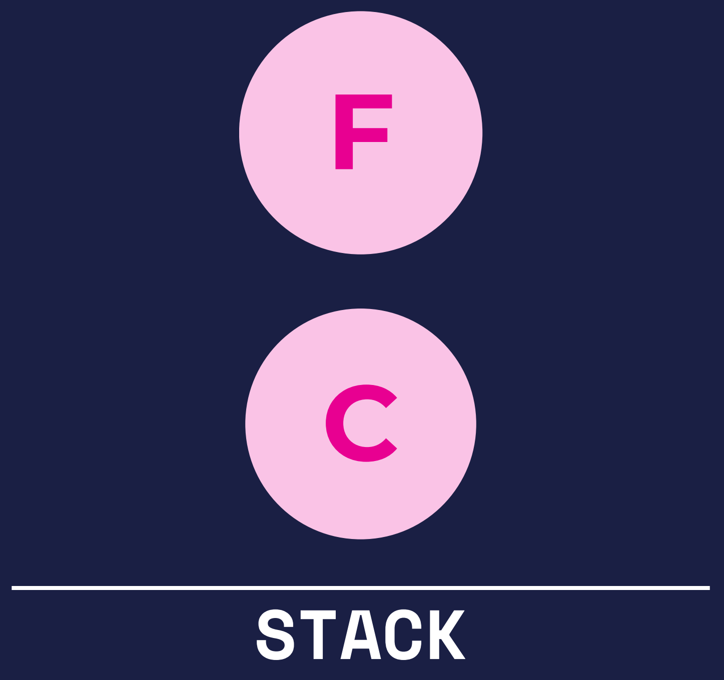
Now we’ll check F.

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

start_node = “A”

visited = [
    “A”, “B”, “D”,
    “E”,
]
```



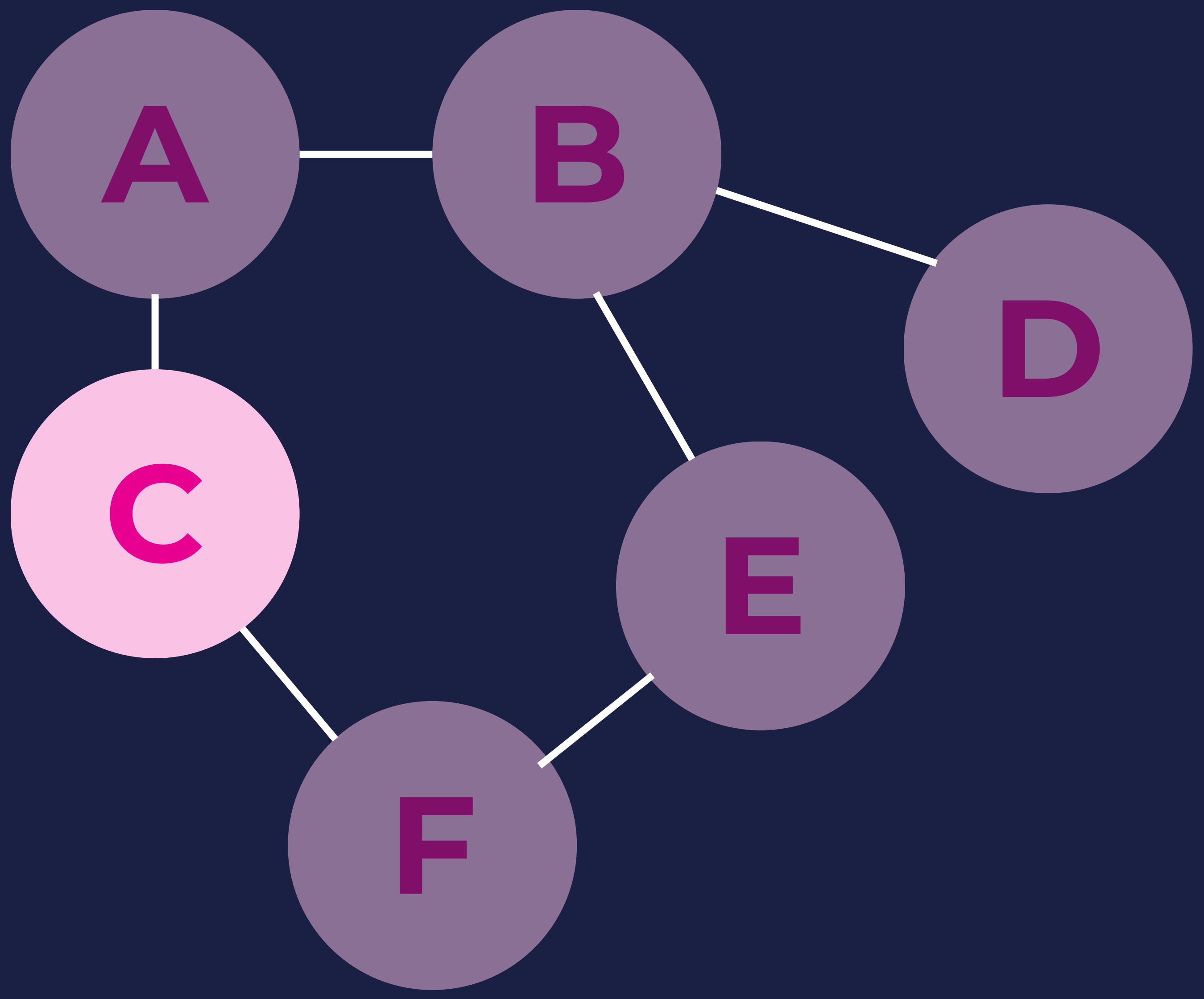


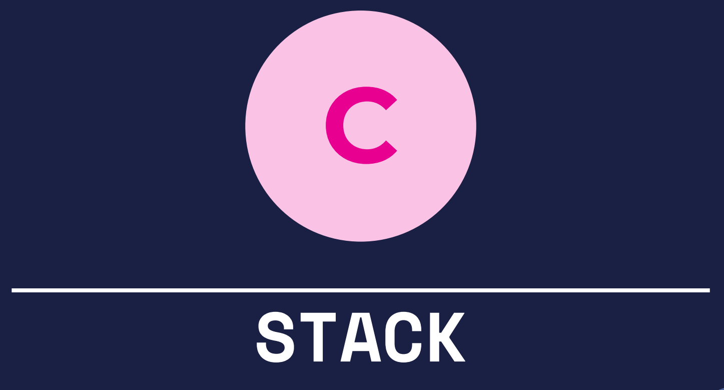
Now we'll check F.

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

start_node = "A"

visited = [
    "A", "B", "D",
    "E", "F",
]
```



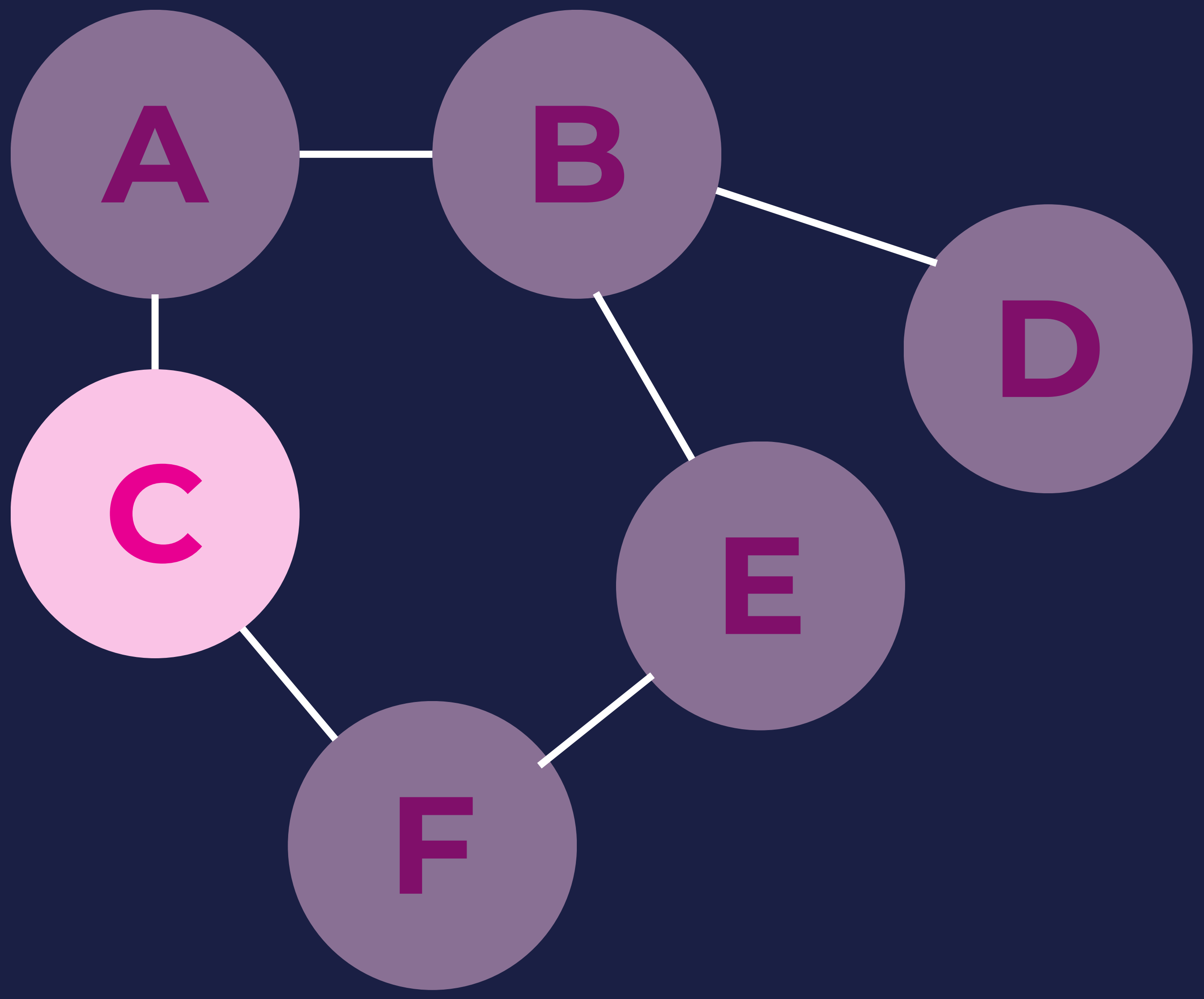


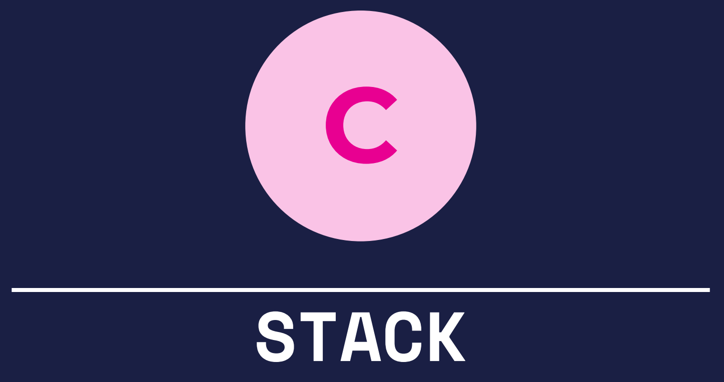
F doesn’t have any unvisited neighbors who aren’t on the stack already so we’ll move on to the last one in the stack.

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

start_node = “A”

visited = [
    “A”, “B”, “D”,
    “E”, “F”,
]
```



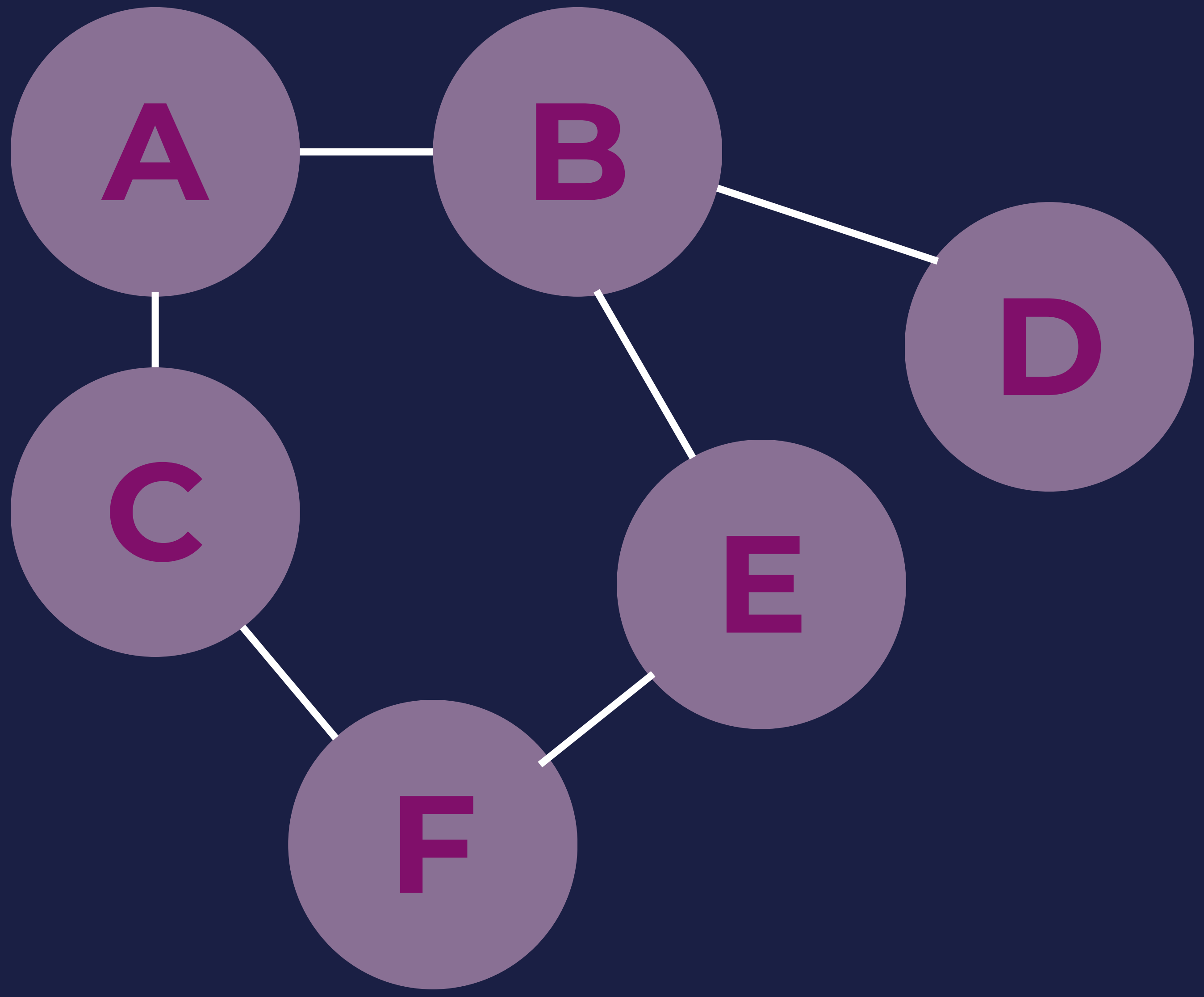


F doesn't have any unvisited neighbors who aren't on the stack already so we'll move on to the last one in the stack.

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

start_node = "A"

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



STACK

THE PSEUDOCODE

```
function dfs(graph, node, visited):
    if visited == null :
        visited = empty set()

    visited.add(node)

    // Recursively call the function
    for neighbor in (graph[node] != in visited)
       dfs(graph, neighbor, visited)

    return visited
```

EXAMPLES REPLIT AND GITHUB! PLEASE GO TO:

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

HTTPS://GITHUB.COM/RIKKITOMIKOEHRHART/GRABABYTE

UP NEXT

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!